Lecture XII
Protection and Security

§1. Introduction

- Protection and Security are two related topics. In protection we study mechanisms for protecting processes and objects in the operating system from improper use by users. It is assumed that the users are not malicious but are liable to make mistakes.

  In security, we study methods for ensuring that the system is not vulnerable from malicious attacks. The attacks are usually from external sources, but not necessarily so. In contrast, protection is usually aimed an internal sources of error.

- This lecture will mainly focus on protection. We only briefly touch on security as this is a huge topic on its own. A particular aspect of security is crptography. That will be treated in another lecture.

- By objects, we mean abstract entities that can be software and hardware based.

  Hardware objects: printers, memory and CPU.

  Software objects: files, databased, semaphores.

- Each object has a set of meaningful actions:

  E.g., The actions of read, write, execute are meaninful on files.

  E.g., For semaphores, the actions are up and down.

- We achieve protection by allowing or disallowing actions on objects by processes.

  The set of all "who is allowed to do what to whom" is what we are trying to implement and maintain in this lecture.

- The setting for protection is similar to that on synchronization – we are assuming cooperating processes, not non-cooperating or malicious processes.

  It is a much harder problem to protect against non-cooperating processes. This would is studied under the rubric of security.

- As in scheduling and cache management, we want to distinguish between policy and mechanism. We view a mechanism as a method to achieve a certain policy. Policy may evolve over time. If the mechanism is general enough, slight changes in policy would not necessitate changes in mechanism.

  The basic policy in protection is the **principle of least previlege**: give the least permissions to each process necessary to accomplish a given task.

  As we see, most mechanisms can only approximate this policy.

§2. Abstract Framework for Protection

- We consider three abstract sets **domains** $\mathbb{D}$, **objects** $\mathbb{O}$ and **actions** $\mathbb{A}$.

  We said that the basic problem is "WHO can do WHAT to WHOM". In terms of this abstract setting, WHO is a domain, WHAT is an action, WHOM is an object.

  Note that the original idea of WHO is a process. But we find it useful to abstract this into a domain, or "protection domain" to be more accurate.

  Why do we want this abstraction? The primary example of a protection domain is a particular user of the system. This user might run many different processes, and we would want their permissions to be identical.

- There is a function $ACT : \mathbb{O} \to 2^{\mathbb{A}}$ where $ACT(O)$ is the set of **actions** for O.

  E.g., if O is a file, then $ACT(O) = \{read, write, execute\}$ (or R,W,X).

- A **(protection) domain** is just a function $D : \mathbb{O} \to 2^{\mathbb{A}}$ where $D(O) \subseteq ACT(O)$.

  E.g., D might be associated with a particular user in the system (this user might be the super-user or root).

- We can represent the set of domains by a matrix ACC, where each row represents a domain of projection, and each column represents an object. The (i,j)-th entry is denoted $ACC(i,j)$ and is equal to $D_i(O_j)$ where D_i and O_j are domains and objects.

  E.g., if D_i is the super-user, then for any j, the super-user can presumably set $D_i(O_j) = ACT(O_j)$ (if desired).

- Throughout the execution of a process, it will be associated with some domain; this domain may change over the lifetime of the process.

  Each action of the process can now be determined to be valid or invalid, depending on whether the current protection domain has the required permissions.

- Although objects are not necessarily files, we can associate every object with a unique one file.

  This is consistent with UNIX's approach of viewing devices and directories as special files. We just extend such special files to include other categories.

  For instance, protection domains is now a new category of files. This category require special treatment in our discussions.

§3. Implementation of Protection via ACL

- The matrix ACC is conceptual because we do not really want to implement it as a matrix: it is too large and cumbersome. There are two general ways to implement the matrix.

- Each column of ACC is called a **access control list** (ACL) for the corresponding object. Each row of ACC is called a **capability list**. The two standard implementations amounts to dividing the matrix either into rows or into columns.

- The first method, called **Access Control List Method**, amounts to storing ACL of the matrix with the corresponding object.

- In UNIX, there is a coarse-grained method to define a ACL using only 9 bits: 3 rwx bits for user, 3 rwx bits for group, 3 rwx bits for world.

  This approach is possible because every process is not just in some protection domain, but also has a user (UID) and group (GID) identity. UNIX regards the pair (UID,GID) as specifying the protection domain for this process.

  In addition, each file has two bits called the SETUID bit and SETGID bit. If the file is executable and the SETUID bit is on, then any process executing this file will have all the rights of the owner of that file. Similarly for the SETGID bit.

  Thus, these two bits allows a process to switch domains.

  SECURITY ISSUE: if one can create an executable file of his or her choice, with SETUID and SETGID bits on, and the ROOT as owner, then the system is compromised by that user. There are methods to avoid this but they are somewhat inconvenient to use.

- Discussion: GID as roles. A user can wear different hats at different times. For instance, the user might be a professor who has two research projects, involving different students and different file systems. To recognize this separation, we can set up two groups for this professor. These two GID's represents the two "roles" of the same "user".

- MULTICS (the precursor of UNIX) has the notion of protection rings, $R_0, R_1, R_2, \ldots, R_{N-1}$. Ring R_i has all the permissions of R_{i+1}. The implementation of this ring is somewhat awkward.

- Refinements of the UNIX approach: in addition to the 9 bits and SETUID bit, most modern implementation of unix (e.g., Solaris) also allows a more refined control of permission by allowing users to specify an ACL. The presence of this optional ACL list is indicated by the "+" indicator (following the 9 permission bits) when we list the file attributes:

```
1 -rw-r--r--+ 1 yap 1000 16 Oct 26 22:26 foo
```

There are functions `setfacl, getfacl` to manipulate this ACL list.

§4. Implementation via Capability List

- The second basic method of implementation is by distributing the capability lists.
- As noted above, a capability list can be implemented as a file, but it is specially tagged to be a **capability object**.
- Recall that each process executes in some protection domain. Therefore, we must allocate memory in the process space to store this capability object information.

  In segmented memory, this would be a specially protected segment which is accessible only by certain system programs.
- There is another approach called "lock and key mechanism" which can be viewed as a hybrid between ACL and capability list approaches. That is because the ACL is stored with objects can be viewed as a lock, and the capabilities stored with processes can be viewed as a key. In general, the lock and the key must match in order for actions to be valid.

§5. Protection Domains as objects

- We can refine the entries in the ACC matrix by including three additional bits: copy, owner and control bits.
- Each of the permissions (R,W,X) can have an associated copy bit that can be turned on. If turned on, we write the permissions as R*, W* or X* (respectively).

  If R* occurs in $ACC(i,j)$ it means that domain D_i can give the R* permission to any entry in the O_jth column. Similarly for R and W.
- There are two refinements for this copy bit.

  (1) It is called a **transfer bit** in case domain D_i can only give the R permission to one other domain D_k, and it loses its own $R*$ bit. Thus, D_i has transferred its permission to D_k.

  (2) it is called a **limited copy** if it can only give R permissions, but not R* permissions to other domains.
- If the owner bit is turned on in $ACC(i,j)$ it means that domain D_i can confer and remove RWX permissions for object O_j to any other domains. THIS IS LIKE THE owner of a file.
- Finally, the control bit is only meaningful in $ACC(i,j)$ where O_j is a capability object. If this bit is turned on, it means that D_i can modify the permissions in domain D_j.

§6. ADDITIONAL ISSUES

Revocation. We need to consider the revocation of previleges. The revocation might be temporary.

Consider revocation of previleges for a given object.

- This is easy under ACL-based protection: just go through the ACL list for the given object and modify as needed.
- Under the capability-based protection, this is more difficult. One way is to maintain a list of back pointers from each object to the domains with previleges for that object.
-

Trusted Computing Base (TCB) Ultimately, we need a core set of routines that can be trusted.

- Basic operations:

§7. Security Issues: Buffer Attack

- Security is the term for external attacks on the integrity of a system. In contrast, protection is reserved for errors caused by internal sources (e.g., legitimate users who make mistakes).
- Some general remarks about the varieties of security issues. Consider the case of a file: there are three ways it's security can be compromised.

  It could lose **confidentiality** if an unauthorized user reads its contents; it could lose **integrity** if an unauthorized user modifies its contents; it could lose **availability** if an unauthorized user deletes it.
- Here, we will go into the details of a widely known security problem known as buffer overflow.
- Consider the following program:

```
#include <stdio.h>
#include <string.h>
#define N 128

int main( int argc, char ** argv)
{
    char buff[N];
    strcpy( buff, argv[1] );
    return 0;
}
```

The procedure `strcpy` works by copying the string `argv[1]` to the buffer area pointed to by `buff`. It copies one byte at a time until it encounters the NULL byte (0) in the `argv[1]`. The problem is that it does not check if the buffer area is large enough.

- A more secure procedure to use would be `strncpy( buff, N, argv[1] )`, which copies at most `N` bytes. As this version has an additional comparison to make for each copied byte, it is slightly less efficient. Programmers may choose `strcpy` for efficiency when they know that no buffer overflow can occur. Unfortunately, in the present case, we do not know this since `argv[1]` comes from the caller of the program.

- To understand how this buffer overflow can be exploited, we must understand how local variables are allocated and how procedure calls are implemented in modern programming languages.

  See lecture 5 for the discussion in the context of THE Machine and our STM language.

Exercises

Exercise 7.1: Consider the Unix approach for achieving protection.
(i) Briefly describe the Unix approach to implementing the abstract ACC matrix.
(ii) What are the properties of files and processes that allows Unix to achieve this protection?
(iii) What are ways that a process can modify its protection domain at run time? ♢

End Exercises