Paper Summary for ”The Protection of Information in Computer Systems” by Jerome H. Saltzer and Michael D. Schroeder

Austin J. Heath
CS6210: Advanced Operating Systems

College of Computing, Georgia Institute of Technology

March 25, 2020

1 Basic Principles of Information Protection

1.1 Considerations Surrounding the Study of Protection

1.1.1 General Observations

This paper covers the study of security issues that are introduced by applications that involve both storing information and simultaneous use of that information by several individuals. Examples provided include:

- Airline reservation systems, and authorities for individuals employed by the airline to access reservation information.
- An online warehouse inventory management system and controlling access on who can view reports generated by an application about the current status of the inventory.
- Credit bureau data banks.
- Law enforcement information systems.
- Time-sharing service bureaus.
- Online medical information systems.

All of these applications have one thing in common, they are used by multiple users to gain access to information, therefore, mechanisms need to be implemented to enforce the correct authority structure.

The authors go on to define two terms that will be used throughout the rest of the paper:

- **Privacy** - denotes a socially defined ability of an individual to determine whether, when, and to whom personal information is to be released.
- **Security** - describes the techniques that control who may use or modify the computer or the information contained in it.

Potential security violations can be placed into three categories:

- **Unauthorized information release** - an unauthorized person is able to read and take advantage of information stored in the computer.
- **Unauthorized information modification** - an unauthorized person is able to make changes in stored information - a form of sabotage.
- **Unauthorized detail of use** - an intruder can prevent an authorized user from referring to or modifying information, even though the intruder may not be able to refer to or modify the information.

Unauthorized in these categories usually denotes that the security violation occurs contrary to the desire of the person who controls the information, however, complications arise when potential security violations are generated from the actions of legitimate users of the computer system.

Examples of security techniques sometimes applied to computer systems are as follows:

1. Labelling files with lists of authorized users.
2. Verifying the identity of a prospective user by demanding a password.
3. Shielding the computer to prevent interception and subsequent interpretation of electromagnetic radiation.
4. Enciphering information transmitted between two computing systems.
5. Locking the room containing the computer.
6. Controlling who is allowed to make changes to the computing system (both hardware and software).
7. Using redundant circuits or programmed cross-checks that maintain security in the face of hardware or software failures.
8. Certifying that the hardware and software are actually implemented as intended.

The paper defines the terms **protection** and **authentication**:

- **Protection** - security techniques that control the access of executing programs to stored information.

- **Authentication** - security techniques that verify the identity of a person making a request to a computer system.

The objective of a secure system is to prevent all unauthorized use of information, however, it is hard to prove that this has been achieved for a computing system. We must avoid viewing security in a narrow lens to help ensure that no gaps appear in our security strategy. A narrow concentration on protection may lead to false confidence in the computing system as a whole.

1.1.2 Functional Levels of Information Protection

The paper defines protection schemes according to their functional properties:

- **Unprotected systems** - computing systems that provide no security.
- **All-or-nothing systems** - systems that provide isolation of users, each user might as well be using their own private computer. The systems might contain public libraries that users can contribute to if a mechanism exists in which the library will accept user contributions. All users have equal access.
- **Controlled sharing** - explicitly controls who may access each data item stored in the system.
- **User-programmed sharing controls** - A user can define how an object is shared or protected through two abstractions, *protected objects* and *subsystems*:
    - **Protected subsystems** - a collection of programs and data with the property that only the programs of the subsystem have direct access to the data. Access to these programs are limited by calling specific entry points. Through the use of **protected subsystems**, a user can develop any programmable form of access control to the objects he creates.
- **Putting strings on information** - a protection scheme concerned with maintain control over the use of the information *after* it has been released. This prevents authorized users from accessing data and releasing it to unauthorized users.

The **dynamics of use** refers to how one establishes and changes the specification of who may access what - the need to change access authorization dynamically and the need for such changes to be requested by executing programs introduces complexity. How do we implement a system in which access to information can be revoked from user while the information is in use? How do we prevent users from gaining authorized access to information and then modifying the permissions for other users to also be able to access the information?

1.1.3 Design Principles

Producing a system that actually does prevent unauthorized acts has proved to be extremely difficult. This difficulty arises from the negative quality of the requirement that a system must be designed to prevent *all* unauthorized actions. The paper presents eight examples of design principles that apply to protection mechanisms:

- **Economy of mechanism** - keep the design as simple and small as possible. This prevents the implementation of unwanted access paths.

- **Fail-safe defaults** - base access decisions on permission rather than exclusion. The default situation is a lack of access, and the protection scheme identifies conditions under which access is permitted.

- **Complete mediation** - every access to every object must be checked for authority.

- **Open design** - the design should not be secret. The mechanisms should not depend on the ignorance of potential attackers, but rather, the possession of specific, easily protected keys or passwords.

- **Separation of privilege** - a protection mechanism that requires two keys to unlock it is more robust and flexible than one that allows access to the presenter of only a single key.

- **Least privilege** - every program and user of the system should operate using the least set of privileges necessary.

- **Least common mechanism** - minimize the amount of mechanism common to more than one user and depended upon by all users. Every shared mechanism represents a potential information path between users - these must be designed with care to ensure security is not compromised.

- **Psychological acceptability** - the human interface must be designed for ease of use s o that users will routinely and automatically apply the protection mechanisms correctly.

- **Work factor** - compare the cost of circumventing the mechanism with the resources of a potential attacker.

- **Compromise recording** - implement mechanisms that reliably record that a compromise of information has occurred.

1.2 Technical Underpinnings

1.2.1 The Development Plan

The paper defines two ways to approach a development of the technical basis of information protection in modern computing systems:

- **Top-down** - emphasizing the abstract concepts involved in information protection.
- **Bottom-up** - identifying insights by studying example systems.

The authors pursue the bottom-up approach, introducing a series of models of systems as they could be built in real life:

1. A multi-user system that completely isolates its users from one another.
2. A capability-based system implementing mechanisms for sharing.
3. An access control list system implementing mechanisms for sharing.
4. A system leveraging protected objects and protected subsystems which allow arbitrary modes of sharing that are unanticipated by the system designer.

1.2.2 The Essentials of Information Protection

The information stored in a computer system is not a single object. Information is divided into partitions, the collections of information in the partitions are the fundamentals objects to be protected.

An analogy that is presented is that each of these objects needs to be guarded and separately protected. A guard stands outside the door to each of these objects and demands users identify themselves to the guard - an authority check. The user must present something the guard knows and something the user possesses. Protection and authentication can be viewed in terms of this model.

1.2.3 An Isolated Virtual Machine

To allow virtual processors to be unaware of the existence of others, an isolation mechanism must be provided - enter the **descriptor register**.

- **Descriptor register** - a special hardware register that controls exactly which part of memory is accessible for a virtual processor. This register contains two components, a **base value** and a **bound value**.
  - **Base value** - the lowest numbered address the program may use.
  - **Bound value** - the number of locations beyond the base that may be used.
  - **Descriptor** - the value that resides within the **descriptor register**; describes an object in memory.

The program controlling the processor has full access to everything in the base-bound range, by virtue of possession of its descriptor. So who loads the descriptor? If any arbitrary program could load any descriptor, there would be no protection.

Enter the **privileged state** bit - all attempts to load the descriptor register are checked against the value of the privileged state bit. One program, the **supervisor**, runs with the **privileged state** bit on and controls the simulation of the virtual processors for the other programs.

With these mechanisms implemented correctly, **descriptors**, **memory bounds**, a **privileged bit**, and a **supervisor**, we completely isolate users from one another. This brings us to the requirement of having to verify the identity of the user when a user attempts to use a terminal.

1.2.4 Authentication Mechanisms

This section encompasses a system that verifies a user's claimed identity. Usually a user provides an identity they claim to be and a password, something that the user has and the guard of the system knows. The passwords are transformed before being stored on the system, making it difficult to invert if compromised. The user provides a password, the system conducts a conversion and attempts to match the two transformations (the user's input and the stored password).

Passwords are generally weak as the defect lies in the user's choice of a password - providing something that can be easily guessed. Another defect is that the password must be exposed to be used - the password must be sent through some sort of communication and an observer may be able to intercept it.

An alternative is **unforgeability**. The user is given a key or some other unique object; the terminal receives the key through some input device and the key transmits its unique identifier to the computing system. The primary weakness of this scheme is that, if an intruder has access to the computer's device reader, an intruder can transmit any sequence of bits he chooses.

A problem with both of these approaches is that they are both **one-way** authentication schemes - an intruder could masquerade as the system the user is authenticating to and acquire the user's credentials once the user is fooled into entering them.

A remedy for this is to conduct a key exchange. Prior to completing authentication, the computer provides the user with a key. The user checks the key against its own key stored for the computer he is authenticating to and confirms the key matches. The user terminal and the computer are then able to conduct a key exchange, handshake, and encrypted credentials can now be passed across the network.

1.2.5 Shared Information

The virtual machines described earlier were completely isolated, however, users may want to permit sharing of data. Implementations of protection mechanisms fall into two general categories:

- **List-oriented** implementations - the guard holds a list of identifiers of authorized users, and the user carries a unique unforgeable identifier for access to be permitted.
- **Ticket-oriented** implementations - the guard holds the description of a single identifier, and each user has a collection of unforgeable identifiers or tickets, corresponding to the objects to which he has been authorized access.

Most real systems contain both kinds of sharing implementations - a **list-oriented** system for human interface and a **ticket-oriented** system for the underlying hardware implementation.

In a **list-oriented** system, the guard knows whose virtual processor is attempting to make an access. The virtual processor is marked with the identifier of the user accountable for its actions. In a **ticket-oriented** system, the guard only cares that a virtual processor presents the appropriate ticket when attempting an access.

This requirement to track the ownership of processes creates the need for the **principal** abstraction - the entity accountable for the activities of a virtual processor. For each **principal** we can identify all the objects in the system the **principal** has been authorized to use - the **domain**.

In order to share libraries, we utilize the previous mechanism of a **supervisor** and the **privileged bit** to globally define the location of a library for multiple users to be able to access. User's **principals** can be modified to be able to access the library, however, without the **privileged bit** they cannot modify the library or the pointer to its location. Multiple implications arise when we discuss these mechanisms:

1. If virtual processor P1 can overwrite the shared math routine, then it could disrupt the work of virtual processor P2.
2. The shared math routine must be careful about making modifications to itself and about where in memory it writes temporary results, since it is to be used by independent computations, perhaps simultaneously.
3. The scheme needs to be expanded and generalized to cover the possibility that more than one program or data base is to be shared.
4. The supervisor needs to be informed about which principals are authorized to use the shared math routine (unless it happens to be completely public with no restrictions).

In terms of a shared procedure, to answer the first implication listed above, we could extend the descriptor registers and the descriptors themselves to include **accessing permission**. This would denote whether or not a running process has the ability to read or write data to the shared procedure.

To answer the second implication in terms of a shared procedure, preventing a program from writing into the shared routine also prohibits the shared routine from writing into itself.

To answer the third implication in terms of a shared procedure, we could generalize our example by increasing the number of descriptor registers to match the increasing number of shared items. There are two forms of this generalization. Most implementations use a combination of these two forms:

- **Capability systems** - ticket-oriented; fast
- **Access control list systems** - list-oriented; used for human interfaces

2 Descriptor-based Protection Systems

2.1 Separation of Addressing and Protection

This portion of the paper discusses attempts to separate memory locations and authorization by organizations using **descriptors** and **segments**. **Segments** represent uniquely identified storage areas with a distinct addressing descriptor.

The mechanism works as such:

- A processor owns a **protection descriptor**.
  - **Protection descriptor** - a descriptor that contains a unique segment identifier and a permissions for read or write set.
- A processor can address a **map entry** with the same unique segment identifier as contained within the **protection descriptor** the processor holds.
- The **map entry** contains the base-bound addressing descriptor identifying the area of memory the process has read or write access to.

These mechanisms help us separate global address spaces - the processor address space is different for different users while the system address space remains the same for all users.

2.2 The Capability System

2.2.1 The Concept of Capabilities

A **capability system** is one in which it lets the user place protection **descriptor values** in memory addresses that are convenient to him. A **capability** is a memory word that contains a protection **descriptor value**.

The example provided shows us a processor that contains **protection descriptor registers** and, when executing programs, said programs can request that the processor acquire some capability located in a segment in memory. With these capabilities loaded into the processor's **protection descriptor registers**, the processor executing the program is able to access shared routines within memory, or private databases tied directly to an acquired capability.

Capabilities can contain separate read and write permission bits, and we can use capabilities to explicitly define what a user does and doesn't have access to by tying the use of that resource to a capability.

Adding **authentication** into the example, a user could provide their credentials to a **supervisor** in order to execute some program. The **supervisor** would verify the credentials, load the user's necessary capabilities, and the destroy any capabilities that don't belong to the user residing on the processor. This ties together two protection systems:

- Authentication that controls access of users to named catalog capabilities.
- A general capability system that controls access of the holder of the catalog capability to other objects stored in the system.

The catalog capability becomes the principal identifier for the actions taken by the processor, however, we lose some accountability because we can no longer inspect the registers of a running processor to determine who is accountable.

The scheme described is useful for static arrangement of access and authorization, however, it does not address the possibility that some user might want to create a segment and provide authorization to another user - **dynamic authorization**.

2.2.2 Dynamic Authorization of Sharing

For dynamic authorization of sharing within a computer, there must be some previous communication from the recipient to the sender, external to the computer system. The concept is that there is some shared communication segment between two users, and each user has a capability that allows them to address that capability. In order to know which capability and user to communicate with, however, the programmer has to know the name of the person in which they intend to share a segment for communication - this has to be done outside of the computing system. Here is the complete protocol for dynamically authorizing sharing of a new object:

- *Sender's part*:
    1. Sender learns receiver's principal identifier via a communication path outside the system.
    2. Sender transmits receiver's principal identifier to some programming running inside the system under the accountability of the sender.
    3. Sender's program uses receiver's principal identifier to ensure that only virtual processors operating under the accountability of the receiver will be able to obtain the capability being transmitted.
- *Receiver's part*:

1. Receiver learns sender's principal identifier, via a communication path outside the system.
2. Receiver transmits sender's principal identifier to some program running inside the system under the accountability of the receiver.
3. Receiver's program uses the sender's principal identifier to ensure that only a virtual processor operating under the accountability of the sender could have sent the capability being received.

2.2.3 Revocation and Control of Propagation

The capability system's chief virtues are **efficiency**, **simplicity**, and **flexibility**. So what if a user changes their mind and no longer wants other people to use a capability to access private information? This is **revocation**. What about when a user is granted a capability, and then they distribute that access to their friends? This is **propagation**.

Methods that have been explored to combat this are the addition of a **copy bit**, preventing the capability from being copied once shared - this hinders flexibility but prevents **propagation**. Another method is to designate segments as **capability-holding** segments, and only those segments can be targets of load and store instructions for **descriptor registers**. This makes **revocation** possible.

A third method is to implement a **depth counter** with each protection descriptor register - the supervisor leads the register with a depth of 1, and when the processor acquires the capability the depth increases to 2. If the processor attempts to share with others, the depth increases again, however, if it goes past 3 a fault occurs - preventing **propagation**.

To tackle **revocation**, proposals have been made that there exists some form of indirection for capability references - extending the mechanism by creating an independently addressable object and anyone with the appropriate capability could destroy the object, revoking access to anyone else who had been given a capability for that indirect object. These objects are closely related to **access controllers**.

At the time of this paper being written, capabilities were best served to implement higher level authorization descriptions, like an access control list system.

2.3 The Access Control List System

2.3.1 Access Controllers

The **access control list system** provides the ability to reverse bindings by delaying an authorization check to the last possible point. An **access controller** contains two pieces of information:

- **Addressing descriptor** - a reference to an segment; unique identifier.

- **Access control list** - a kind of indirect address.

In order to access a segment, the processor must supply the unique identifier of that segment's access controller. A data reference by the processor proceeds in the following steps:

1. The program encounters an instruction that would write in a segment.
2. The processor uses the unique identifier found in the register to address access controller. The processor at the same time presents to the memory system the user's principal identify, a request to write, and the offset.
3. The memory system searches the access control list in the access controller to see if the user's principal identifier is recorded there.
4. If the principal identifier is found, the memory system examines the permission bits associated with that entry of the access control list to see if writing is permitted.
5. If writing is permitted, the addressing descriptor of the segment, stored in the access controller, and the original offset are used to generate a write request inside the memory system.

This organization differs in several ways from the pure capability system:

1. The decision to allow access to a segment has known, audit-able consequences.
2. The access control list directly implements the sender's third step of the dynamic sharing protocol - verifying that the requester is authorized to use the object.
3. Revocation of access has become manageable - change to an access control list immediately preludes all future attempts by users to access a segment.
4. The question of "who may access this segment?" is answered directly by examining the access controller for the segment.
5. All unnecessary association between data organization and authorization has been broken.

2.3.2 Protection Groups

Protection groups are principals that may be used by more than one user. When a user logs in, he can specify the set of principal identifiers he proposes to use. His right to use his personal principal identifier is authenticated, for example, by a password. His right to use the remaining principal identifiers can then be authenticated by looking up the now-authenticated personal identifier on each named protection group list.

2.3.3 Implementation Considerations

There are three key areas in which a direct implementation of access control lists might encounter practical problems:

1. Every reference to an object in memory requires several steps, These steps are serial, and because of the number of references, fast memory access would be needed.
2. An access control list search with multiple principal identifiers is likely to be complex and slow.
3. Allocation of space for access control lists can be a formidable implementation problem.

To attack these problems, the authors proposed these solutions:

1. Implement a shadow capability register that is invisible to virtual processors - once initial authentication is done no further checks need to be made. We can revoke access by changing the shadow capability register.
2. Restrict the number of entries for all access control lists.
3. Leverage protection groups to decrease the specificity and number of entries.
4. Implement access control lists in interpretive software in the path to secondary storage or the filesystem, removing access control list checking mechanisms out of memory.

2.3.4 Authority to Change Access Control Lists

How do we control who may modify the access control information? Two different authority controlling policies are proposed:

- **Self control** - This scheme extends the earlier concept of defining read and write permissions by extending these permissions to who can modify the access control list itself. The biggest objection to **self control** is that it is so absolute.
- **Hierarchical control** - Whenever a new object is created, the creator must specify some previously existing access controller to regulate future changes to the access control list of the new object. The primary objection to **hierarchical control** is that system administrators are *too* powerful - the use and possible abuse of higher level authority is completely unchecked.
  - The **prescript** field is proposed - whenever an attempt is made to modify an access control list, the accessing modifying permission of the higher level access controller is checked. Some possible actions might be triggered by the **prescript value**:
    1. No action.
    2. Identifier of principal making change is logged.
    3. Change is delayed.

4. Change is delayed until some *other* principal attempts the same change.
5. Change is delayed until signal is received from some specific principal.

- The goal of all these policies is to ensure that some independent judgement moderates unfettered use of authority.

2.3.5 Discretionary and Non-discretionary Controls

Discretionary access control implies that the user may, at his own discretion, determine who is authorized to access the objects he creates. This goes against the applications of most organizations, as system administrators may have security requirements - users shouldn't be able to modify the access control of objects they create to be able to provide information to other users who aren't intended to access it. This is called **non-discretionary** access control.

Similar constraints are imposed in a military context with **sensitivity levels**, information generated at a sensitivity level cannot be disseminated to lower levels of sensitivity.

To prevent disclosure of classified information, the concept of **confinement** is introduced - programs will run within a domain containing all necessary data but should be constrained so that it cannot authorize sharing of anything found or created in that domain with other domains. **Confinement** requires that the virtual processor be constrained to write only into objects that have a compartment set identical to that of the virtual processor itself. Writing classified information to unclassified locations can be considered **declassification** and a violation of the security policy.

2.4 Protecting Objects Other Than Segments

To begin protecting different objects of a computing system, we must first establish the kind of operations that can be performed on an object of interest and work out what permissions would be appropriate. An example would be a queue that provides **enqueue** and **dequeue** operations - these are special operations that we can design protection for. All of this is happening to registers within memory, however, with no context as to what memory the registers contains.

With this, the authors introduce the concept of **types** in a protection system. The **protection descriptor registers** are expanded for processors to contain a type field, and each segment can be identified by its type to determine the protection mechanisms that can be applied to its operations. The concept of **typing** isn't restricted to just segments within memory. Any data structure can be distinguished and protected from misuse, e.g. input and output streams for devices.

2.5 Protected Objects and Domains

The controlled sharing presented by the access control list system and the capability system has two important limitations:

1. The first limitation is that only those that access restrictions provided by the standard system facilities can be enforced.
2. The second limitation concerns users who borrow programs constructed by other users. Execution of a borrowed program in the borrower's domain can present a real danger to the borrower - the borrowed program can exercise all the capabilities in the domain of the borrower. A user must have a certain amount of faith in the provider of a program before execution within his own domain.

The authors propose a solution to these limitations, the **protected subsystem**. A **protected subsystem** is a collection of program and data segments that is "encapsulated" so that other executing programs cannot read or write the program and data segments and cannot disrupt the intended operation of the component programs, but can invoke the programs by calling designated entry points. The encapsulated data segments in this context are the **protected objects**.

Programs within a **protected subsystem** can protect the **protected objects** and enforce complex controls on access to them. Programs outside the **protected subsystem** can manipulate **protected objects** only by invoking caretaker programs. This provides mutual protection for multiple programs cooperating in the same computing system.

Things that must be considered in the implementation of a practical system that implements **protected objects** are briefly discussed as follows:

1. The principle of "separation of privilege" - the internal structure of some data object is accessible to processor A, but only when the virtual processor is executing in program B.
2. The switching of protection domains by a virtual processor should be carefully coordinated with the mechanisms that provide for dynamic activation records and static variable storage - the storage of one protection domain must be distinct from that of another.
3. The passing of arguments between domains must be carefully controlled to ensure that the called domain will be able to access its arguments without violating its own protection intentions.

3 The State of the Art

3.1 Implementations of Protection Mechanisms

As you can imagine, this portion just covers protection mechanisms that were currently in use at the time this paper was written and published. The content here is not greatly relevant - the authors were expounding upon organizations and technologies that were currently in the market at the time. I would recommend you quickly glance at this section, but it's nothing more than a listing of technologies by the author.

3.2 Current Research Directions

A research problem that the authors described was attracting attention is how to certify the correctness of the design and implementation of hardware and software protection mechanisms. Funnily, enough this is still an issue today and has only gotten harder to prove as software systems have increased in size and complexity. The sub-problems the authors identify are:

- One must have a precise model of the protection goals of a system against which to measure the design and implementation.
- Given a precise model of the protection goals of a system and a working implementation of that system, the next challenge is to verify somehow that the presented implementation does what it claims.
- User interfaces that more closely match the mental models people have of information protection are needed.
- In most operating systems, at the time this was published, had an unreasonably large quantity of "system" software running without protection constraints.
- The use to which information may be put after its release to an executing program - prevention of dissemination once information is provided.
- Development of encipherment techniques for random access to data, especially for small pieces of data. How can we justify executing expensive-crypto algorithms for small pieces of data, and how do we protect small pieces of data from being crypt-analyzed?

3.3 Concluding Remarks

Matching protection goals to a protection architecture by setting the bits and locations of access control lists or capabilities or by devising protected subsystems is a matter of programming the architecture. It is not surprising that users of the current first crop of protection mechanisms have found them clumsy to program and not well-matched to the users' image of the problem to be solved, even if the mechanisms are sufficient. In the future, it will be necessary for protection systems to be used and analyzed by users - these users will propose better views of the necessary and sufficient semantics to support information protection.

4 Works Cited

Saltzer, J.H. and Schroeder, M.D., " Protection Information in Computer Systems ", Proceedings of the IEEE, 63(9):1278-1308, Sept. 1975. https://web.mit.edu/Saltzer/www/publications/protection/