Reading 15: Security

Security For Operating Systems

- One role of the operating system is to provide useful abstractions for application programs to build on.
- These applications must rely on the OS implementations of the abstractions to work as they are defined
- Ex: We expect that the OS's file system will enforce the access guarantees to ensure that a file they have specified as unwritable does not get altered

What are we protecting?

- At a high level, we are trying to protect everything
- OS has control over all of the hardware on the machine and is able to do literally anything the hardware permits
    - Can control the processor, read and write all registers, examine any main memory location, and perform any operation of its peripherals supports
- OS can
    - examine or alter process' memory
    - read, write, delete or corrupt any file/writeable persistent storage medium (hard disks and flash drives)
    - change scheduling and halt execution
    - send any message anywhere
    - enable or disable peripheral devices
    - give any process access to any other process' resources
    - arbitrarily take away any resource a process controls
    - respond to any system call with a maximally harmful lie

Security Goals

- What does it mean when we want an OS to be secure?
    - what we really mean is that there are things we would like to happen in the system and things we don't want to happen and we want a high degree of assurance
- **Confidentiality:** Keep your secrets. If some information is supposed to be hidden from others, don't allow them to find out
- **Integrity:** If some piece of info or component of a system is supposed to be in a particular state, don't allow any adversary to change it
- **Availability:** If some information or service is supposed to be available for your own or others' use, make sure an attacker can't prevent this
- In essence, processes are the memory of the operating system
- It is nearly impossible for a process to "protect" any part of itself from a malicious OS
- Ultimately, the OS software must do its best to enforce those flexible security goals, which implies we'll need to encode those goals in forms that software can understand

Designing Secure Systems

- There are certain design principles that are helpful in building systems with security requirements
- **Economy of mechanism:** Basically means to keep your system as small and simple as possible
- **Fault safe defaults:** default to security, not insecurity -> if policies can be set to determine the behavior of a system, have the default of those policies be secure. Setting a firewall to only allow certain good traffic through
- **Complete mediation:** Security term meaning that you should check if an action to be performed meets security policies every single time the action is taken
- **Open design:** Assume your adversary knows every detail of your design. Base your security on the assumption that your clients know every detail about the system
- **Separation of privilege:** Require separate parties or credentials to perform critical sections. For example, two-factor authentication, where you use both a password and possession of a piece of hardware to determine identity
- **Least privilege:** Give a user or a process the minimum privileges required to perform the actions you wish to allow
- **Least common mechanism:** for different users of processes, use separate data structures or mechanisms to handle them
- **Acceptability:** a critical property not dear to the hearts of many programmers. If your users won't use it, your system is worthless. Do not ask too much of your users

The Basics of OS Security

- If the OS can maintain a clean separation of processes that can only be broken with the OS's help, then neither shared hardware nor OS services can be used to subvert our security goals
- OS needs to be careful about allowing use of resources and hardware
- System calls offer the OS another opportunity to provide protection

Authentication for Operating Systems

- Analogy: If your significant other asks for you to pick up a gallon of milk, you'd probably do so while if a stranger asks you probably won't
    - Similarly, if a system admin asks the OS to do something, it probably should, while a random script from the web does it probably shouldn't
- Therefore, knowing who is requesting an OS is crucial in meeting our security goals
- OS services are most commonly requested by system calls made by particular processes, which trap from user code into the OS
    - The OS has an OS-controlled data structure to determine the identity of the process.
    - Based on the identity, the OS has the opportunity to make a policy-based decision on whether to perform the requested operation
- If the OS permits a process to access a certain resource (object), then we can remember that resource

    - Any form of data created and managed by the OS that keeps track of access decisions for future reference is called a **credential** (e.g.. page tables)
- Regardless of the kind of identity we use to make our security decisions, we must have some way of attaching that identity to a particular process
- Crux: How can we securely identify processes? For systems that support process belonging to multiple principals, how can we ensure that each process has the corect identity attached

Attaching Identities to Processes

- One simple way to copy the identity is to attach an identity to a new process , then copy the identity of the process that created it
    - ex: fork() consults the parent's process control block to determine its identity, then creates a new control block with the same identity as the parent
- Ex: multi-user system
    - assign identities to processed based on which human user they belong to
- If processes have a security-relevant identity, like a user ID, we're going to have to set the proper user ID for a new process
- When a user first starts interacting with a system, the OS can start a process up for them
- The OS must be able to query identity from human users and verify that they are who they claim to be, so we can attach reliable identities to processes -> so we can implement our security policies

How to Authenticate Users?

- Classically, authenticating the identity of human beings has worked in one of three ways
    - authentication based on what you know
    - authentication based on what you have
    - authentication based on what you are

Authentication By What You Know

- Most commonly performed by using passwords (a secret known only to the party to be authenticated)
- The system does not need to know the password
    - Since the system is simply checking whether the user knows the password or not, it's not checking to see what it actually is
- We can store a **hash of the password**
    - By nature, you can't reverse hashing algorithms, so the adversary can't use the stolen hash to obtain the password
- If the stored copy is leaked to an attacker, he doesn't know the passwords themselves
    - We want to ensure that, whatever we store, the attacker gets no help in trying to figure out what the password is
- If an attacker steals the hashed password, he should not be able to get any clues about the password itself

    - **cryptographic hashes** make it infeasible to use the hash to figure out what the password is -> other than guessing at it
- Tip: Never store secrets like plaintext passwords or cryptographic keys
- The longer the password, the more difficult it is to guess
    - over time, password systems have expanded the possible characters in a password from alphabetical characters to numeric characters to symbols, etc
- You can prevent the account from attacks through guessing by limiting the number of guesses a user can make
- If the cryptographic hash is well-known, there is a chance that the attacker can steal the password file and use the cryptographic hash to generate the password
    - fix: before hashing a new password and storing it in your password file, generate a big random number (32/64 bits) and concatenate it to the password -> hash and store that result
    - You also must store that random number and run through the hashing algorithm

Authentication by What You Have

- Real life analogy: we need a ticket to get in somewhere
- ex: ATM machine -> the device has special hardware to read our ATM card.
    - The hardware allows it to determine that we have the card and asks for further proof for asking for your PIN
- If we have something that plugs into one of the ports on a computer, such as a hardware token that uses USB, then, with suitable software support, the OS can tell whether the user trying to login has the proper device or not
- We can make use of the human's capabilities to transfer information from whatever it has to the authentication system itself

Authentication By What You Are

- **In addition to properties of the human body such as DNA, there are characteristics of human behavior that are unique**
- ex: using facial detection to open a phone
    - challenges: the camera is going to take a picture of someone who is presumably holding a phone. What if the owner is someone who looks like the user? What if he's wearing a mask?
    - Lighting could easily affect the quality of the photo taken and therefore the ability to authenticate
- Where each authentication method fails, it can be backed up by a mechanism that doesn't fail in some cases

Authenticating Non-Humans

- **Ex: A web server**
    - There really isn't some human user logged in whose identity should be able to reach the server
- Mechanically, the OS need not have problems with the identities of such processes

    - simply set up a user called webserver (or something else) on the system in question and attach the identity of that user to the processes that are associated with running the web server
- One approach is to use passwords for these non-human users as well
    - assign password to the web server user
    - The system admin could log in as the web server user, creating a command shell and using it to generate the actual processes the server needs to do its business
    - As usual the processes created by this shell process would inherit their parent's identity -> webserver, in this case
- Alternatively, we can provide a mechanism whereby the privileged user is permitted to create processes that belong not to them, but some other user webserver
- Allow a temporary change of process identity while keeping the original process identity
- ex: linux systems use sudo for superuser capabilities
- Sometimes we wish to identify not the individual users, but groups of users as well
    - we might have four or five system admins, any of who is allowed to start up the web server
    - Instead of assigning privilege to each individual user, it would be easier to have a meaningful group of users with that privilege
    - We can either associate a group membership with each process, or use the process' individual identity information as an index into a list of groups that people belong to

Important Aspects of the Access Control Problem

- **Figure out if the request fits within our security policy**
- **If it does, perform the operation, if not make sure it isn't done**
    - First part is generally referred to as **access control**
- open("/var/foo", O_RDWR)
    - How should the system handle this request from the process, making sure that the file is not opened if the security policy to be enforced forbids it?
- Crux: How to determine if an access request should be granted?
    - How can the OS decide if a particular file request made by a process belonging to a user should or should not be granted?
- The OS will run some kind of algorithm to make this decision -> will take certain inputs and a binary output, a yes - or - no decision on granting access
- **Subject:** the entity that wants to perform the access, perhaps a user or process
- **Object:** thing the subject wants to access like a file or device
- **access:** some mode of dealing with the object such as reading or overwriting it
- Code that implements the algorithm is called the **reference monitor**
- Give subjects objects that belong only to them
    - If the object is inherently thiers, by its nature and unchangeably so, the system can let the subject (a process) access it freely

    - Ex: Virtual memory: A process is allowed to access its virtual memory freely with no special operating system access control check at the moment the process tries to use it
        - Otherwise we would need to run the algorithm to check for access each time -> time consuming
        - A process might be granted control of a GPU based on initial access control decision, after which the process can write to the GPU's memory or issue instructions
        - Merely relying on virtualization to ensure proper access just pushes the problem down to protecting the virtualization functionality of the OS
- **Two main approaches in allowing access**
    - **access control lists:** pretty much a list of who gets access
    - **capabilities:** kind of like a lock and key system to have access to a room.

Using ACLs for Access Control

- Each file has its own access control list, resulting in simpler, shorter lists and quicker access control checks
    - open() call in an ACL system will examine a list of /tmp/foo
    - This ACL is more like metadata, so is likely to be stored with or near the rest of the metadata for the file
- ACL simply checks whether the user has the write permissions and access over being able to access the object
- ACL needs to be stored in some persistent storage such as somewhere on the disk
    - Unless it's cached, we'd need to read it off the disk every time someone tries to open the file
    - Where do we store the ACL to minimize the disk reads/seeks?
    - Might be better if we store it in metadata like the inode
- We don't want to reserve enough space in the ACL for all users because that would be a waste of space since the people accessing a file will probably be fairly limited
    - If we need to constantly search through a large list, even if we are able to do it quickly with modern computers, it can add a considerable overhead to the system
- Unix designers used 9 bits for each file's ACL, and realized that there were effectively three modes that we cared about
    - read, write, and execute
    - Cleverly partitioned the bits into three groups
    - Owner of the file, the members of a particular group/group ID (could be stored in the inode), and everybody else (no need to use bits since it's just the complement of the group)
    - Solved the problem of the amount of storage eaten up + the cost of accessing and checking them
- **Advantages of ACL**

    - If you want to answer who is allowed to access a resource, we can just access the ACL itself
    - If you want to change the set of subjects who can access a resource, you can just change the ACL
    - Since the ACL is kept near or with the file itself, you can get to the file -> and get to all the relevant access control information
- **Disadvantages**
    - Having to store the ACL near the file and dealing with potentially expensive searches of long lists
    - If you want to figure out the entire set of resources some principal (process or user) is permitted to access?
        - You'll need to check every single ACL in the system
    - In a distributed environment, you need to have a common view of identity across all the machines for ACLs to be effective

Using Capabilities for Access Control

- Option that is more like keys and tickets as opposed to an access list
- How to perform an open() call in a pure capability system
    - When the call is made, either your application will provide a capability permitting your process to open the file in question as a parameter, or the OS would find the capability for you
- **Capabilities are bunches of bits**
    - A bunch of bits does not seem safe but if we never let anybody touch the capabilities, then we wouldn't have the problem
    - Processes can perform various operations on capabilities, but only with the mediation of the OS
    - Ex: if process A wants to give process B read/write access to file /tmp/foo using capabilities, it wouldn't merely just give them the appropriate bit pattern. A would make a request to the OS to give B the proper capability
- If we want to rely on capabilities, the OS will need to maintain its own protected capability list for each process
- Capabilities need not be stored in the OS -> it could be cryptographically protected
- **Advantages**
    - Easy to determine which system resources a given principal can access
        - Look through the capability list
    - If you have the capability readily available in memory, it can be quite cheap to check it, particularly since the capability can itself contain a pointer to the data or software associated with the resource it protects
    - Offer a good way to create processes with limited privileges
    - With ACL's the process just inherits the parent's privileges
- **DIsadvantages**
    - **Determining the entire set of principals whos can access a resource is expensive**

        - Any principal might have a capability for the resource must check all principal's capability lists to tell
        - System must be able to create, store, and retrieve capabilities in a way that overcomes the forgery problem

Mandatory and Discretionary Access Control

- **Who gets to decide what the access control on a computer resource should be?**
    - In case of a user's file, the user himself should determine access control settings. In the case of a system resource, the system administrator, or perhaps the owner of the computer, should determine them
- Example: military
    - Even if you're allowed to see top secret information, you are not allowed to let others see it
- **Discretionary Access Control:** Whether almost anyone or almost no one is given access to a resource is at the discretion of the owning user
- **Mandatory Access Control:** Some elements of the access control decision in each system are mandated by authority, who can override the desired information

Practicalities of Access Control Mechanisms

- Most systems expose either a simple or more powerful access control list mechanism to their users, and most of them use discretionary access control.
    - However, because it is generally infeasible to have the user set individual permissions on thousands of millions of files, generally the system allows each user to establish a default access permission for each file they create
- However, while many will never touch access controls on their resources, for an important set of users and systems these controls are of vital important to achieve their security goals
- Even if you rely on defaults, many software installation packages use some degree of care in setting access controls on executables and configurable files they create
- One practical issue that large institutions discovered when trying to use standard access control to implement security method
    - people performing different roles within the organization require different privileges
    - Organizing access control on the basis of particular roles is difficult
    - This is particularly valuable if certain users are allowed to switch roles
- **Role-based access control:** you can merely remove the role label off a certain position when a user switches position
- One can build a minimal RBAC system under Linux and similar OSes using ACLs and groups
    - **Privilege escalation:** allows careful extension of privileges, typically by allowing a particular program to run with a set of privileges beyond those of the user who invokes them

        - Feature (in Unix and Linux) is called SetUID and allows a program to run with privileges associated with a different user
        - A carefully written SetUID program will only perform a limited set of operations using those privileges, to ensure that the privileges cannot be abused
    - Linux sudo command offers a general approach to allow designated users to run certain programs under identity