Problems with Information flow :

The increasing usage of information flow based policies may lead to many objects becoming sensitive, making it difficult for the system to be used by lower clearance objects. This is called “Label Creep”.

We need to enforce legitimate exceptions in the policies. If trusted programs are allowed to be exempted from many of the policies, then there always exist chances of misuse. It is also difficult to configure the whole system and be confident about the strength of the security.

So, we require alternative and hybrid approaches that can overcome the main concerns like:

- Many programs being tagged as trusted
- Trusted programs themselves having hidden vulnerabilities

1.Domain and type enforcement:

1.1 Introduction

Domain and type enforcement (DTE) is an access control mechanism which helps in controlling the accesses granted to specific programs.

DTE defines “domains” as a set of subjects and “types” as a set of objects. You may recall that subjects are entities like processes that act on behalf of a user and objects are entities like files, sockets etc.. on which subjects operate on.

The enforcement of policies in DTE is such that a system administrator knows exactly what domains access what types. The policies basically need to handle:

- Restrictions of accesses from domains to types
- Restrictions of transitions from one domain to other domains

DTE can help enforce the principle of least privilege to processes, typically based on the functionality and usage of a particular process. For instance, consider the encryption example discussed before in the context of MLS. We had to make that program fully trusted. In this context of DTE, we can grant it more rights than other programs, but still don't have to give complete access. For instance, if the encryption program is used mainly to transmit sensitive data over a public network, then the program can be given access to write to the network, but it need not be given the ability to overwrite local files.

Another example: you can have a “viewer” program that can be used by multiple applications (programs) which may want to provide some interface for the users to view output data. When the viewer program is invoked, DTE policies can ensure that the viewer can read various files, but is very limited in terms of the files it can write, or other types of resources that it can access. (For instance, it may be permitted to write its own preferences files.) We can arrange for a domain transition at the point of invocation of a viewer program so as to ensure that these restrictions can be selectively applied to viewers without having to be applied to the programs that invoke them.

1.2 : Domain Transitions:

As was mentioned in the introduction to DTE, policies should enforce restrictions not only on domains accessing types, but also on domain transitions.

Example DTE configuration

Note that after booting, a number of processes are started up initially. These processes, called daemons, are responsible for all subsequent operations of the system, including the startup of network services, allowing user logins, and so on. As such, many of these daemon processes need to run with root privileges. In the absence of mechanisms like DTE, this need for root-privilege can be exploited by attackers. In particular, suppose that a network server runs with root privilege. If an attacker can exploit a vulnerability on this server to inject code, then this code can do a lot of damage. For instance, the attacker can install a “rootkit,” which can be thought of as a set of Trojan programs that allow attackers future access to the system (e.g., they can remotely login into the system) but hide their presence as well as the presence of attacker-launched processes. Installing a rootkit requires the attacker to overwrite important system binaries, which is possible because the exploited processes were running with root privileges.

The figure below shows how we could configure DTE to eliminate the possibility of rootkits being installed as a result of exploiting daemons. The figure shows 5 domains. The daemon domain is not permitted to install new executables/libraries or overwrite existing ones. This is permitted only in the “admin” domain --- when a system administrator wants to install new programs, she needs to go into that domain. Guarding access to this domain is a new login program that performs user authentication. It can perform stronger authentication before transitioning to the admin domain, as compared to the transition to the user domain.

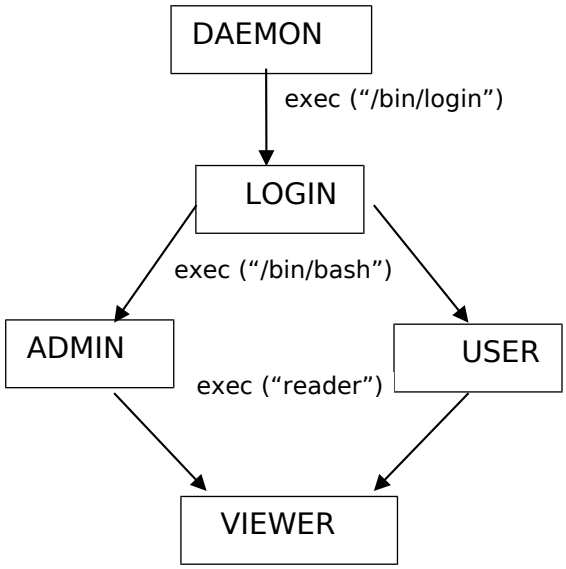

Figure: 1.1: A highly simplified example of DTE domains

A transition from the daemon domain to login domain (a domain that contains a single application, namely, the new login program) takes place when the login program is executed. The login program can transition to admin domain or user domain --- it has to specify which --- by executing the shell program. In the former case, a shell with full access, including the ability to install or upgrade software, is launched. In the latter case, the rights can be restricted so that normal user activities can be carried out but not tasks that require overwriting system programs. We can also limit access to the login program (ie prevent its execution) from user domain. In addition, we can ensure that the only way to transition into admin domain is using the login program. As a result, no matter what happens, it

becomes impossible to install or upgrade binaries without first going through strong authentication incorporated into the login program for access to the admin domain.

The figure also shows a “viewer” domain, where the accesses are further restricted so that file writes are not possible. As a result, even if a viewer program is attacked and compromised, it will not be possible for an attacker to create or modify any files on the system.

It is clear from this discussion that DTE can help in practicing the principle of least privilege. It allows for privilege amplification as well as attenuation. While all this flexibility is good when DTE is viewed as a mechanism, it does complicate policy development. For instance, DTE policies in SELinux are quite large and complex. They are difficult to develop, understand, or administer.

Another problem with DTE is that in reality, we end up creating one domain for each application. Part of the problem is that the basic DTE does not provide much in terms of parameterization, which makes it difficult to reuse policies across different applications. In SELinux, they attempt to mitigate this by using macros, but macros dont provide particularly clean abstractions, so in the end, the policies become difficult to understand or reuse.

Finally, while DTE allows each domain to be confined in a fine-grained fashion, it provides no help in understanding the overall effect on a system. Compare this with MLS, where we can assert overall properties such as the absence of information leaks from highly sensitive data to low-sensitivity data.

2. DTE and SELinux:

SELinux stands for “Security-Enhanced Linux”.

As the name suggests, they are talking about a Linux system with inbuilt enhanced security features.

SELinux combines the standard UNIX DAC and DTE. (It can also support MLS and a whole slew of other policies, but in practice, its essence is DAC and DTE.)

3. DTE and MLS:

In MLS, you can either not trust a program or completely trust it. Whereas, in DTE, you can limit the access rights such that the rights granted are just enough for the program to run as it is intended to.

DAC and DTE are mechanisms whereas MLS can be thought of as a high-level policy.

In DTE, the security is dependent on the configuration of the system and classifying domains, whereas in MLS, policy mostly involves object labeling (attributing access rights).

4. Linux (POSIX) capabilities:

The main idea here is to decompose the root privilege into several components, so that instead of giving root privilege, only subset of these privileges are available. This enables better adherence to the least privilege principle and hence lead to more secure systems. can be provided to the legitimate user and process to reduce the damage due to compromise. Following are some of the capabilities to which root privilege can be decomposed:

CAP_CHOWN: Capability to change the owner of any file.

CAP_DAC_OVERRIDE: Capability to override the DAC permissions.

CAP_NET_BIND_SERVICE: Capability to bind to low numbered ports (typically below 1024).

CAP_SETUID: Capability to change the uid of the process.

CAP_SYS_MODULE: Capability for the process to load kernel.

You can learn more by doing “man capabilities” on a POSIX-compatible OS like Linux or by searching on the same string on the Internet.

4.1 Effective, Permitted and Inheritable Capabilities for processes:

There are 3 types of capabilities associated with a process:

Effective capability is a set of capabilities that signify whether the current process has a particular privilege. This is similar to effective uid in DAC.

Permitted capability is the set of capabilities that the process can have. Effective capability can be changed only if the new set of capabilities is a subset of permitted capability for the process. This is similar to the real uid.

Inheritable capability is the set of capabilities that can be retained by child process of current process. When a child process is created using execve, capabilities of the child is determined by calculating permitted capabilities intersected with the effective capabilities and then masked with inheritable capabilities.

4.2 Attaching capabilities to executable files:

There are 3 types of capabilities associated with executable files:

Allowed capability is a set of maximum capabilities that an executable file can have. This is the superset of all the allowed capabilities for the file.

Forced capability is the set of capabilities that the executable file is given that did not belong to the parent process executing it. This is similar to setuid and is a mechanism for privilege amplification.

Effective capability is the set of capabilities that can be retained by the child process executing this file from the effective capabilities of the parent process.

5. Commercial policies:

High-level policies in commercial environment (like banks) are different from those enforced in military environments. There is a difference of degree of confidentiality expected in military setting and degree of data integrity expected in commercial settings.

Example policies:

- Clark-Wilson policy

- Chinese wall policy

5.1 Clark-Wilson policy:

In this policy, emphasis is on data integrity. Data integrity is a major concern in commercial deployments: in these setting, errors and fraud correspond to attacks on integrity, while confidentiality threats can largely be addressed using processes, procedures, regulations, and laws.

To come up with integrity policies in a domain-independent way, C/W is phrased in terms of transactions. C/W does not constrain the semantics of these “Well-formed transactions” (WFTs) but requires that they maintain system integrity, i.e., each WFT takes the system from one consistent state to another. An integrity verification procedure (IVP) may be used periodically to verify consistency. (Note: consistency and integrity are being used synonymously here.)

These transactions are assumed to be aware of the data integrity requirements and hence preserve them. However, there is the issue of integrity of the WFTs themselves --- i.e., we need to be sure that a fraudulent WFT was not performed. For instance, consider a transaction that transfers money from one customer account to another. This WFT maintains overall data consistency, since the total amount of monies across all accounts in unchanged in the absence of money coming into or going out of the bank. However, it is still possible for a fraudulent bank clerk to initiate such a transaction to transfer money from customer accounts to her own account. To prevent such fraud:

1. WFTs could be launched only by authorized users. Proper authentication should be used. Moreover, mechanisms must be in place to restrict access to WFTs, e.g., we may want to limit a person to performing travel voucher approvals, and another to process equipment purchase vouchers.

2. Critical functions need to be performed by multiple users. This is called “separation of duty.” For instance, in a transaction involving the processing of travel vouchers, the person approving the voucher should be different from the one making the claim. Similarly, in the money transfer example, the person performing the transfer should be different from the ones that own the accounts in question.

3. Auditing to verify integrity of transactions. Auditing is used to check if various processes and policies set up to protect the integrity of WFTs were in fact followed. The purpose of auditing is to ensure actions can be traced and attributed. (Such attribution serves as a powerful deterrent against misuse and fraud.)

4. Maintain adequate logs so that if errors/fraud were discoverd, the WFTs in question can be undone.

5.2 Chinese Wall policy:

In this policy, emphasis is on conflict of interest when dealing with confidential information.

This policy is defined in terms of:

- CD (Company Data) – objects related to a single company.

- COI (Conflict Of Interest) – sets of competing companies

The policy specifies that no person can have access to two or more CDs which come under the same COI class. I.e. a consultant cannot provide service to companies X and Y if X and Y are competitors. And this applies to past, present or future access, i.e., if a user has had past access to the data of company X, they can never work for company Y. (In the real-world, there is often a time limit, or finer granularity separation among the data belonging to the same company.)

6. Delegation:

Delegation refers to the ability to transfer certain rights to an entity to operate on behalf of the initiator. For instance, if a principal A requests a service from another principal B, then A must be willing to transfer any rights that B may need in order to service A's request.

Example: Requesting a print server to print a file, where (a) the file is present on a file server, and (b) the file server will only allow the user to access his files. In this case, the user has to grant access rights to the print server to read files from the file server on his behalf. As you may have observed this can be accomplished by public key cryptography.

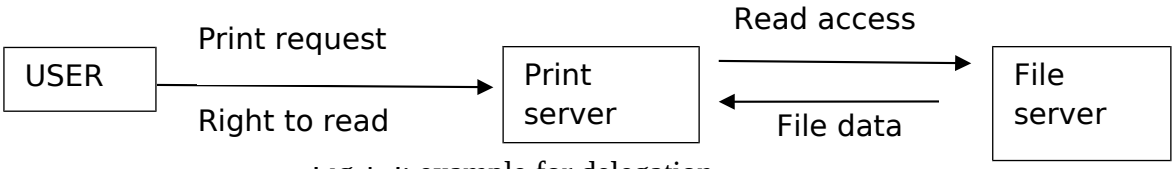

Fig 1.2: example for delegation.

It can also be done using symmetric key cryptography, in a manner similar to the printer example we used with Kerberos in one of the previous lectures.

Trust management is concerned with specification and enforcement of security policies in a distributed setting that is characterized by explicit statements regarding trust. A principal can say which principals it trusts for what information. It is convenient to think of these in terms of attributes, where policies specify which principals can vouch for which attributes. For instance, an instructor can state that a course page is can be accessed by a user provided the registrar determines that the user is registered for the course. In this example, the instructor's policy delegates the responsibility of determining course registration information to the registrar.

Operating System security:

The system is layered and each layer is supposed to cater to the requests of the upper layers.

This means that the user, and user applications need to be at the topmost layer followed by the kernel and then at the bottom most layer by the hardware.

In ensuring security of such a system, we do not assume that the higher layers may not attempt to bypass all or some of the lower layers. An attacker can try to attack at any layer whichever exhibits vulnerability and is easier to attack.

Memory Protection:

One of the important aspects of Operating system security is Memory Protection. Without memory protection, other protection mechanisms may be bypassed. For instance, suppose that process A is permitted access to a file F, while process B is not. Process B can bypass this policy by attempting to read F from A's memory. Alternatively, B may attempt to modify the policy that is stored in the OS memory so that B is allowed access to F.