Abstract
The importance of Operating System Security has been well known for the last twenty five years. While there have been great strides in the development of mechanisms for application space security mechanisms the area of operating system security has largely been neglected. Today’s trends of distributed computing and the increased possibility for malicious attacks have shown the need to implement mandatory control mechanisms and policy flexibility. This paper discusses the need for greater operating system security and mandatory access mechanisms. It further discusses the Flask Architecture Implementation of these mechanisms and also the SELinux implementation which is based on the Flask Architecture.

Introduction
Since the advent of the ability to connect one computer to another and be able to send and receive data across, there arose a need to be able to secure the application space and operating system space of one computer from the other. With the fast growth of distributed computing and the world wide web there is a greater urgency to solving the issues and problems related to computer security.

Current status of computer industry and advancements
Today there is a larger risk of malicious attacks from any where in the world on any computer as most of them are connected to the internet. Attackers can mount a large scale attack by subverting the resources of multiple computers world wide without the knowledge of the owners and launch sustained attacks against the computing resources of a government or country. With most of today’s commerce, health, utility systems and government systems heavily dependent on the use of computers, electronic wars are a possibility and a serious concern. Thus there is a greater understanding of the need to secure all aspects of a computer system like networking, operating systems, applications, memory, secondary storage as well as the actual computer itself. There have been great advances in the development of secure applications in order to secure many of the computing resources but all these application rely on the ability of the operating system to protect the application space mechanisms against tampering bypassing and spoofing attacks.

Operating system security responsibilities
The operating system provides and interface to the underlying hardware and data and is a platform on which various applications execute their operations. The security of the operating system is therefore a necessity for the overall system security. Today most commercially developed operating systems provide security through authentication of the users, maintenance of access control mechanisms, separation of kernel and user spaces and providing trusted applications to modify or manage system resources.

Vulnerabilities
However the above security features are inadequate to protect the operating system from attacks (either malicious or benign) in today’s environment.

Discretionary Access Controls
In many operating systems like UNIX, the access control mechanism is discretionary. That is the decision to restrict access to various user level objects is completely at the discretion of the individual user. Benign mistakes or malicious intrusion attacks can hijack the system by assuming the identity of a user and using his permissions to access other secure parts of the system. Thus there is no protection against malicious software. Decisions on granting access to an object are solely based on user identity and ownership. There is no way to verify the credentials or authenticity of the user beyond a simple password check. The system maintains only two categories of users; a superuser and the ordinary user. There is no scheme to restrict accesses to various levels of users based on their roles in an organization or their need to perform certain tasks. The superuser has complete authorization to access all the resources of the system. Subversion of the superuser credentials
by a malicious user can mean disaster to the system. In addition to this issue many of the system services and privileged programs run with coarse-grained privileges or as a superuser. Such applications can be made to perform tasks other than what they were originally started for. Such a hijack of a privileged process allows access not just to the resources that the process is currently seeking to access but also all the resources that it is authorized or has permission to access. There is thus a lack of adequate support to constrain trusted applications a lack of a trusted path through which applications proceed and the principle of least privilege for a process or user based on the task currently being performed is not supported. All these issues are like doors of a house without locks. All one needs to do is try to open the door and a large number of system resources become available.

What this paper talks about
This paper discusses the flaws in our current assumptions in the adequacy of today’s operating system security mechanisms, importance of further developing such security mechanisms, desired features and some example implementations with those features. Details like performance considerations of such systems are not discussed in detail as they have been covered extensively in the various publications associated with those examples.

Need for Enhancing Operating System Security
As discussed above the operating system security is a basic assumption on which a number of other security tools rely on in order to secure a computing system. This section discusses examples of how various application space mechanisms rely on basic operating system security.

Access Controls
Application space access control mechanisms have an enforcer and decider component. Based on a request by a subject to access an object the enforcer sends a request with the credentials of the subject and object and the type of access requested, to the decider who then consults its internal policy database or an external policy database to make a decision and communicate to the enforcer which grants or denies the request. The loopholes that may exist in this system are

1. A malicious agent may tamper with any of the inputs to the decider thus subverting the access control mechanism.
2. Even if the inputs reside locally in a file, the operating system has to provide adequate protection to the file. These two issues require integrity guarantees for data which can be provided by enforcing mandatory access control mechanisms.
3. The invocation of malicious software by an authorized opens the door for that software to modify the security attributes of an object or modify the policy database. This requires a trusted path to ensure explicit authorization for a process to be able to access various objects and their attributes.
4. The malicious software can also impersonate the decider to the enforcer or the subject or even the enforcer to the decider. This requires a protected path mechanism to ensure that source and destination are properly authenticated to each other and the transmission of credentials is secure.
5. A malicious agent may try to bypass the access control mechanisms and this requires the establishment of a mandatory security mechanism to ensure that all accesses to protected objects are regulated by the enforcer.

Cryptography
While using cryptographic applications there are two fundamental issues that determine the security of the system

a. The invoking the cryptographic mechanism
b. The functioning of the mechanism
1. The confidentiality and the integrity of the algorithms and keys used is dependent on the protection mechanisms of the operating system.
2. While invoking a cryptographic mechanism the application that invokes the mechanism may be bypassed.
3. A malicious program seeking to be authorized for access may impersonate the cryptographic tokens that contain keys to the application that invokes the cryptographic mechanism. All the above require a mandatory security mechanism as well as a protected path mechanism.
4. A malicious program may misuse an valid token to access any protected service, algorithm, session or key. Thus there is no way of verifying the identity of the user and limiting access based on that.

5. Since many applications run with coarse grained privileges it is necessary to enforce finer grained controls over the user of services algorithms and keys. Mandatory limiting of use of token to user who activated the token, limiting trusted path applications to highly sensitive actions.

Many of protection mechanisms like Kerberos, Network Security(IPSEC), Firewalls rely on the underlying assumption that the operating system will provide the necessary protection to the software and data that is needed for their operation.

Desired Features
Based on the discussion so far we can identify the following as the desired features of a trusted operating system. This list is by no means an exhaustive list and there are an number of avenues for further securing the operating system beyond these.

1. Mandatory Security Mechanisms
   a. Mandatory Access Controls
   b. Mandatory Authentication Mechanism – Protected Path
   c. Mandatory Cryptographic Usage
2. Support for principle of least privilege
   a. Limit trusted applications to least privilege – implement fine grained access controls
   b. Confine damage
3. Assured Pipelines – to assure integrity of subsystems
4. Control Dataflows in support of system security policy – implement domains
5. Limit Security policy administrative capability to a single or a small set of users
6. Flexible configuration of policy and support for multiple policies
   a. Revocation of permissions
7. Assurance

Policies
This section provides an outline of some of the security policies used in the Flask Architecture and later in the SELinux Implementation.

Type Enforcement
As system is a collection of active entities called subjects and passive entities called objects. A subject can be any process and an object can be any computing resource like a file, socket etc. Security attributes are associated to the subjects and objects to help in access control decisions. A subject has an attribute called domain assigned to it and an object is assigned a type. A domain definition table represents allowed interactions between domains and types (i.e subjects to objects). The rows correspond to domains and the columns correspond to the types. A decider checks the table to see if the cell in the table representing the domain and type interaction contains the requested access control (read, write, append, execute etc.). A domain interaction table is a representation for a domain to domain interaction which specifies allowable and automatic domain transitions. It also allows us to specify entry point and code execution restrictions for domains. In traditional TE, there is no distinction between types and domains and domains are treated just like any type.

Domain and Type Enforcement
Domain and Type enforcement is an enhanced version of type enforcement(TE). It provides language support for the specification of domain to type and domain to domain interactions through language constructs. It supports implicit typing. Type statements declare one or more types, domain statements define entry points, object access rights and subject access rights. Assign statements associate a path to the various types. Figure 1: shows the language implementation of domain and type enforcement through the statements discussed above.
Role Based Access Controls
Users and Processes have defined roles assigned to them. Each role has a set of defined domains assigned to it and it can only enter into those domains. Role transition from one role to another role have to take place explicitly and each role is associated with an initial domain in which it starts to function.

Identity Based Access Controls
Access control decisions are made based on the identity of the individual user. This scheme would work in conjunction with Role Based Access Controls.

Multi-Level Security Policy
This policy bases its access decisions on clearances for subjects and classifications for objects. But It does not provide adequate support for data and application integrity, separation of duty and least privilege. It also requires special trusted subjects that act outside of the access control model. It also fails to control the relationship between a subject and the code it executes.

Flask Architecture
The flask architecture is an attempt to implement a general architecture for Mandatory Access Controls. Its important design goal to support policy flexibility was implemented through the separation of the security policy logic from the enforcement mechanism. The enforcement of the security policy can be transparent to the applications.

The Flask Security Architecture is shown in Figure 1. It primarily consists of subsystem that enforces policy decisions and a subsystem that makes those decisions. The enforcers are object managers that manage a particular set of objects and the decider is a security server.

Requirements for Policy Flexibility
- Identify a portion of the set of system states that are security relevant and control operations that affect or are affected by that portion.
- Support a wide variety of security policies.
- Support atomicity of operations and revocability of previously granted permissions.
- Support revocation of migrated permissions
- Identify in progress operations and either safely abort or restart the operation or wait for the operation to complete safely.

Implementation of Desired Features for Operating System Security
Object Managers have three primary elements that help to implement the desired features for operating system security.

1. Interfaces between Object Manager and Security server to receive information from the security server on access control (ie. Whether a particular permission is granted) , labeling of objects (assignment of particular security attributes to objects and subjects) and polyinstantiation (determining which member of a polyinstantiated set of resources should be accessed for a particular request).
2. Provides and Access Vector Cache that allows the caching of previously made decisions to minimize impact on
performance due to implementation of mandatory and constant checks.

3. Object managers can register to receive notifications from the security server about changes to the security policy.

Object managers also have the following components defined to allow them to execute their operations effectively.

1. A mechanism to assign labels to their objects
2. A control policy specification to determine how services provided by the object manager are controlled
3. Handling routines that are called in response to policy changes
4. A mechanism to select a proper instantiation of a resource in the case of polyinstantiation.

**Object Labeling**

All objects are labeled to keep a track of their current security attributes and help the decider to make a decision on the access requests for that object. All objects controlled by a security policy are labeled by the security policy. The Flask Security Architecture provides two policy-independent data types for labeling objects. These are the security context and the security identification (SID). The security context is a variable length string that contains information about the object like user identification, classification level, role or a TE domain. All these are specifications for the kind of subjects that can access it. The security identification is a 32 bit integer that is assigned to an object when it is created. It is generated by a security server and is mapped to a security context. A security context of an object is referenced using this SID. The SID does not have any specific internal structure.

**Trust - Client and Server Identification**

The SID is used in the authentication of the client to the server and vice-versa. The SID of the client is provided to the server along with the request from the client. The client identifies the server by making a kernel call on the capability to be used for communication. Since a client would be using intermediate systems to communicate to the server there is a need to override its identity and provide an effective identity to the server. The decisions and the actual assignment is carried out by the security server and enforced by the kernel.

**Performance Improvement – Caching**

Every time a process makes a request to access an object an access control decision needs to be made. If the object manager requests a decision from the security server for every such access it could very easily overload the system. To prevent this and to minimize the impact of implementing mandatory access control the Flask architecture provides caching of security decisions within the object manager. The access vector cache module is a shared library among various object managers and helps coordinate policy implementation between the object managers and the security server. When a decision needs to be made whether a subject can access an object the access vector cache contains the information based on decisions previously made by the security server. To further minimize the performance impact, the security server can provide more decisions than requested to be stored in the AVC. When the security server receives a request it returns the current state of the policy for a set of permissions with an access vector which is a collection of related permissions for the pair of SIDs on which the security server has to take a decision.

**Restriction of Resource Sharing – Polyinstantiation Support**

Certain resources that are shared among clients are polyinstantiated and the clients are partitioned into sets that share the same instantiation of the resource based on the security level, role etc. of the client. Each instantiation is called a member. A dedicated interface allows decisions to be made on what
instantiation a client is allowed to access. The client and the instance are identified with SIDs.

### Revocation

In order to be able to implement policy flexibility and changes to policies it is necessary to be able to revoke previously granted permissions. Policy decisions are stored explicitly in an access vector cache and implicitly in the form of migrated permissions. It would therefore be necessary to also identify all migrated permissions that need to be revoked and revocation done. Once the revocation is done the security server needs to be intimated about it so it can proceed with the implementation of modifications to a policy or a new policy. The requirement of atomicity stated above is achieved through the establishment of two requirements on the system.

1. **Object Managers** must reflect policy change as soon as the change is complete.
2. The policy changes must be carried out in a timely manner by the object managers.

The revocation will have to be carried out with a well defined protocol. This protocol involves three steps.

1. The security server notifies the object managers who have been provided with a policy, whenever the policy changes.
2. The Object manager updates its internal state to reflect that policy change. This involves
   a. Updating the access vector cache of the object manager
   b. Invoke any callbacks registered by the object manager for revoking migrated permissions. This is done by examining the thread and memory states and performing the revocation as necessary.

Figure 2 shows this mechanism. The file server also supports revocation of permissions that have migrated to file description objects.

It should not be possible for the revocation request to be arbitrarily delayed by actions of untrusted software. The object managers must be able to handle situations where there is a possibility of a revocation request to be delayed.

### Security Server

The security server provides security policy decisions, maintains mapping between SIDs and security contexts, provides SIDs to newly created objects, member objects and manages the object manager access vector cache. It also provides functionality for loading and changing policies. A security server should also have its own cache to be able to store the results of access computations. The security server can use these results to improve its response time to access computation requests from clients.

The security server also has to enforce its policies on its own services. It must enforce the policy over the interfaces that it provides the various object managers and determine which subjects can access them. It may also limit the subjects that request policy information to maintain the confidentiality of the policy information.

In a distributed environment each node must have its own policy server. Each such policy server should coordinate with the others at an enterprise level. In addition the security server should easily be scalable and replicable. The security policy is defined in the server through a combination of code and a policy database. Any policy that can be expressed in the policy database language can be implemented. In addition one can also make limited modifications to the code to implement additional policies. If necessary the complete security server can be replaced to completely modify the policies supported.

The flask security server implements a security policy that is a combination of four sub-policies that have been presented before. Namely multilevel security, type enforcement, identity based access control and role based access controls. The server code primarily determines the MLS policy and the rest are determined by the policy database.
Security Enhanced Linux (SELinux)

The information assurance research group of the National Security Agency (NSA) of the United States Government has been working with the Secure Computing Corporation to develop a strong, flexible mandatory access control architecture based on type enforcement. A couple of Mach based prototypes the DT Mach and DTOS were developed. The NSA and the SCC then worked with the University of Utah’s Flux research group to transfer the architecture to the Fluke Operating system and the architecture was enhanced to provide better support for dynamic security policies. This enhanced architecture was named Flask. The NSA is now in the process of integrating the Flask architecture into the Linux operating system to transfer the technology to a larger developer and user community.

Architecture

The architecture of SELinux is based on the Flask architecture which is sufficiently general to be able to support various Mandatory Access Control Models. The architecture consists of a security server that makes access decisions and which is cleanly separated from the rest of the kernel behind a well-defined interface.

The objects are assigned security contexts upon which security decisions are based and only the security server has the authorization to interpret these security contexts. The security contexts contain all the security attributes associated with a particular labeled object. The security contexts are not bound to the objects. The security contexts maintain three relevant security attributes – identity, role and type. There is another policy independent data type as discussed in the flask architecture that is bound to each labeled object called the security identifier (SID). SIDs are mapped to the security contexts at run time and maintained by the security server.

In the SELinux implementation the object managers are kernel subsystems like process management, filesystem, socket IPC etc. Application object managers can also be supported. For example a database management system.

Model

The security model of SELinux implements a combination of Identity based access controls (IBAC), Role based access controls (RBAC) and Type enforcement(TE). However this is not a fixed model and can be replaced with other policy combinations.

Each portion of the security context is used by the server to compute a portion of the access decisions. Every process in the system has an identity associated with it and any changes to it are strictly controlled and limited to a small set of programs like login and crond.

The actions of a particular user are limited by the RBAC policy which determines what are the allowable user actions. This is dependent on the set of roles assigned to the user. Role transition is tightly controlled by the policy and role transitions occur only as a result of running programs that require user authentication. Thus role transitions can occur only through explicit user consent. Using the TE mechanisms and policy the RBAC policy can define allowable role transitions. The RBAC policy also defines domains that can be entered into by the roles and defers the assignment of permissions to the TE configuration.

The implementation of fine grained access controls is done through the implementation of the TE policy. Each object is assigned a type and an access matrix is used to determine the permissions granted for each subject and object class. Domains are basically just types that are assigned to processes. Thus the TE policy is based on type pairs only rather than implement separate Domain Definition Table (DDT) and a Domain Interaction Table (DIT). The TE policy allows the grouping of permissions based on the object class and allows permissions to be defined based on the services for that object. All these features allow us to implement the desired features discussed above for operating system security like clean separation of policy and enforcement mechanisms, policy flexibility and multi-policy support through the expression of integrity, separation, containment and invocation policies. There is no need to have trusted subjects that can violate the policy. Processes can be controlled using the principle of least privilege by enforcing fine grained access controls. The relationship between a subject and its executable is tightly controlled by basing the access decisions on the function and trustworthiness of the code. This offers protection against malicious code.

Policy Configuration

Security policy configuration is carried out through special security policy configuration files. These are text files and typically one
configuration file for each policy. Once a policy specification in the configuration file is completed it is easy to further modify it to customize the policy. These configuration files are written in a language specific to the security server. It is checked and compiled to binary representation and then loaded into the security server during the boot process. As and when it is deemed necessary by the policy currently implemented the binary configuration can also be loaded during run time.

The following discussion gives examples of how the policies are specified in the configuration files.

**The TE configuration File**
The TE policy assigns allowable permissions between pairs of types for each object class using the allow statement.

allow typeA typeB: class { permission_1 permission_2 .......};

Automatic transitions and default labels for files created by programs of certain types in directories of certain types can be specified using the type_transition statement

type_transition type_A type_B: file default_file_type;
type_transition type_C type_d: process default_process_domain;

**The RBAC configuration file**
This file defines a set of roles that can be modified as necessary. Since each process has an associated role, the configuration file specifies the types that can be associated with processes of a certain role through the role command. The role transitions are specified by the role_transition command.

role myroldename types { type_A type_B ...... }; role_transition current_role program_type new_role;

Since role transitions involve greater changes in privileges its implementation is very tightly controlled and allowed only through the execution of login or new_role programs. Instead of this domain transition is used more as it involves finer–grained access controls.

**The IBAC configuration file**

The roles assigned to each user are specified in the IBAC configuration file using the user statement.

User username roles { role_A ...... role_N}

**Implementing Policy Change Mechanism**
The Flask AVC provides and interface to make policy changes. This interface is accessed by the security server only. The policy change protocol is same as that implemented in the flask architecture. Many of the permissions in SELinux are revalidated upon use like read and write permissions. Thus the policy changes are automatically recognized and enforced. Object managers can efficiently revalidate permissions using references to AVC entries. However there is functionality yet to be implemented with respect to invalidating page cache entries when a policy change notification is necessary. The security_load_policy call is used to read a new policy configuration from a file. After the policy is loaded the Security Server updates its SID mapping, invalidates SIDs that are no longer authorized and resets the AVC. Figure 4: shows the AVC interface and an example call to check permissions.

```
Example AVC Interface and example call to check permission
extensible
int as_perm set
security_id_1 rolset, //pair of SIDs subject-subject, subject-object, object - object
security_id_2 rol0, //role
security_class_1 label, //role label
access_vector_1 requested, //set of requested permissions
use_entry_ref ("sentry", "user")

set as_perm set
context --old,
ss--old, sk--secure,
SUID_SETGID,
file--execute)
```

Figure 4: AVC interface and an example call to check permissions.

**Achieving Security Objectives**
The security objectives of data access control, kernel integrity protection, file system protection, implementing least privilege for processes, process separation and administrator domain protection are implemented mostly through the TE policy configuration after specifying the necessary roles and domains.
All system processes are assigned to the system_r role and two roles are defined for users user_r for ordinary users and system_r role for system administrators. The user_t domain is the initial domain for user_r role and the sysadm_t domain for the system_r role. Domain transitions occur later as needed.

The following example shows how system file integrity and confining privileged processes is maintained.

Let us consider the send mail program. This program must be able to modify the /etc/aliases and /etc/aliases.db files and the /etc/mail directory. However the /etc directory is protected and write access is strictly limited. The files and the directory that need to be modified have a type defined for it and the sendmail_t domain associated with the sendmail program is granted write access to objects of that type as shown below.

allow sendmail_t etc_aliases_t:file {read write };
allow sendmail_t etc_mail_t:dir { read search add_name remove_name};
allow sendmail_t etc_mail_t:file {create read write unlink };

Let us look at another example of implementing least privilege to privileged processes. The flaws in these processes are often exploited to subvert the security of a system. These processes can be confined using the features of SELinux by defining separate domains for the processes and restricting their accesses to only the absolutely necessary privileges.

allow sendmail_t smtp_port_t:tcp_socket name_bind; //allows sendmail to bind to smtp port
allow sendmail_t mail_spool_t:dir {read search add_name remove_name};
allow sendmail_t mail_spool_t:file {create read write unlink};

The second and third statements allow the sendmail program to manage the mail spool directory and the last two statements allow it to manage the mail queue directory. Even if any flaws in sendmail are sought to be exploited the set of accesses granted to the attacker are strictly limited to what have been specified above in the configuration.

Summary

The current trends in distributed computing and e-commerce over the internet highlight a very important need to secure operating systems and their mechanisms from being subverted by malicious attacks. This implementation of SELinux gives us a very good understanding of how mandatory access control mechanisms can be implemented and how desired operating security features can be implemented in actual systems. While there are a number of requirements that still need to be satisfied, they are only a matter of time and actual implementation. The mechanisms to implement those desired features are already in place.

The Flask architecture development so far as focused more on the enforcement mechanisms rather than on implementing multiple security policies. While the hooks for this are already in place further research into this area and in general in the area of security policies would be the logical next step.

The development of a distributed security server that coordinates between the security servers on individual nodes within a distributed environment is another important next step to ensure scalability of this system.

Other areas of further work are providing revocation capability to object managers for processes under execution while a policy changes, support for labeling and controls for NFS and IPSEC/ IKE integration and support for packet labeling.

References


