Department of Computer Science

Jörgen Sigvardsson
<Jorgen.Sigvardsson@kau.se>

Execution Access Control Model (EAC) - A Model for Controlling Software Access and Execution Based on Functionality and Origin

Master's Thesis
2001:02

Execution Access Control Model (EAC) - A Model for Controlling Software Access and Execution Based on Functionality and Origin

Jörgen Sigvardsson
<**Jorgen.Sigvardsson@kau.se**>

This thesis is submitted in partial fulfillment of the requirements for the Masters degree in Computer Science. All material in this thesis which is not my own work has been identified and no material is included for which a degree has previously been conferred.

Jörgen Sigvardsson
<Jorgen.Sigvardsson@kau.se>

Approved, Date of defense: TBD

Opponent: TBD

Advisor: Simone Fischer-Hübner

Examiner: TBD

Abstract

This thesis presents a security model, EAC, for monitoring and controlling executable content. In this model, controlling software is based on its origin and functionality. Software must meet two criteria; the software must be trusted in order to be executed and it must adhere to program specific access rules. The model is described informally as well as formally. The thesis also describes and compares three existing models and products. These are the Java, ActiveX and Tripwire model. An implementation specification is given. The implementation takes advantage of the reference monitor concept which is a common approach for implementing security models. What sets this model apart from other security model is that the programs are subjects for security monitoring rather than user processes which is the common approach in many security models.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

The traditional UNIX operating system[1] implements protection mechanisms using *discretionary access control*. This means that users of a system has full access rights to owned objects, such as files. Processes executed by a user executes on behalf of the user, i.e., the process inherits the access rights of the user. In essence, the process acts as an extension of the user. This means that a program may for example delete files, modify contents of files, add new files in the name of the user. Processes usually do what the user expects it to do. These processes are called *trusted* processes, the user can trust that the process does *the right thing*.

In a perfect world all processes do the right thing, and are thereby trustworthy. However, since the world is far from perfect, some processes fail to do the right thing. Some processes fail due to a bug in the program. Some processes fail due to a failure in its environment. Some processes fail because they were intended to *the wrong thing*. The latter case is often called malicious code and are also known as trojans, virii, trap doors, back doors and then some.

A process which fails to do the right thing can cause serious security, privacy and integrity problems for a user which executes the process. The standard discretionary access control mechanisms found in UNIX allows this, so there exists no simple remedy for this, unless one writes the program 100% bug free (never going to happen). My contribution to this problem is a better protection mechanism model.

1.2 Goal

The goal with this thesis is to define a new model for handling the protection mechanisms in UNIX. My model which I call Execution Access Control, or EAC, adds a new level of protection. EAC defines two domains of protection;

- What programs are considered trustworthy?
- What can a trustworthy program do?

Upon execution of any executable code, EAC tests whether the code is trustworthy or not. This is done using digital signatures and certificates. Before some code can be executed it must first have been signed by some person and that person must be considered trustworthy. Once

[1] Meaning all typical *flavours* of UNIX

a test verifies that the signature is correct and that the signer is trustworthy, the program is allowed to execute a process.

After a program has been allowed to execute a process, its operations are controlled by a set of access control rules. These rules define explicitly what the process may do. The rules are there to protect the user from program bugs.

EAC is a mandatory access control model. This means that the protection mechanisms always apply for all users and there is no way of changing the protection. No user may add a new trustworthy signer to the system. No user may change an access control rule for a program, except for a special user called the *Security Officer*. The role of the security officer is to administrate the access control rules and trustworthy signers.

EAC works in parallel with the ordinary protection mechanisms. For an operation to be granted, both EAC and the ordinary mechanisms must grant the request.

1.3 A Use Case

BIND (Berkeley Internet Name Domain) is a software package which implements a DNS (Domain Name System) server. DNS is used for translating symbolic names such as `www.domain.com` to IP addresses which are used for communication over IP (Internet Protocol).

In 1998 a bug was discovered in BIND which allows a remote attacker to execute arbitrary code on the BIND hosted machine. The attack is based on a technique called *buffer overrun*. The effect of this bug was that any remote attacker could execute commands on the server machine. Since the BIND server executed as root (super user), all commands would be accepted. In effect, a remote attacker would then have full control of the server host. A technical explanation about the attack can be found at `http://www.insecure.org/sploits/bind.multiple.vuln.html`.

This bug could have been avoided if there were some access control rules disallowing BIND to execute commands. These sort of attacks are quite common which a quick browse through `http://www.cert.org` would confirm. Other types of attacks such as worms, trojans and virii can also limited by having access control rules specifying what a program may do.

1.4 Organization

This thesis is organized as follows:

Chapter 2 discusses some concepts needed to understand the details of the EAC model. Cryptography, hash functions, digital signatures, certificates, reference monitors and security models are discussed.

Chapter 3 covers three existing models or technologies which are widely used. These models capture some security aspects which EAC shares or potentiall could share. They are examined for advantages and disadvantages which aids in creating the EAC model itself.

Chapter 4 describes the EAC model both informally and formally using mathematical notation. The informal description explains the fundamental ideas behind the model, the methods of achieving the model and how it works from a practical point of view. The formal description defines a finite state machine along with its invariants and state transformation functions. No

proof is provided, the description serves more as an exact "recipe" for the model rather than means for a proof of its correctness. This chapter also compares this model against the three models or technologies discussed in chapter 3.

Chapter 5 specifies how the model should be implemented in Linux, a UNIX variant. It specifies cryptography schemes (including hashing, encryption, signatures), key and certificate management, and which operating system objects are to be protected. Operating system objects may be viewed upon as abstract datatypes whose operations are to be protected by the access control rules. In short, this chapter is about how the model should be implemented.

1.5 Summary

This thesis describes a model which adds rules and boundaries for programs as opposed to the approach of setting rules and boundaries for users. The thesis also loosely specify how the model could be implemented in a traditional UNIX system as well as a specialized UNIX derivative.

Chapter 2

Concepts

2.1 Cryptography

Cryptography is the practice and study of encryption and decryption of information. *Encryption* of information, also known as *plaintext*, is to transform it into data which cannot be interpreted, also known as *ciphertext*. The reverse transformation, *decryption*, transforms uninterpretable ciphertext into plaintext again. Cryptography is generally used for sharing information between two or more parties without disclosing the information to an outside party. There exists many schemes for using cryptography for this purpose. Some of these are RSA, DES, AES, Blowfish, Twofish, RC4, RC5 and ElGamal. These schemes are divided into two categories; *symmetric* and *asymmetric* cryptography schemes.

2.1.1 Symmetric Cryptography Schemes

Symmetric cryptography schemes encrypts plaintext and decrypts ciphertext using the same key or two equivalent keys[1]. That is; a ciphertext which was encrypted using key k can only be decrypted into plaintext by using the key k. This is the symmetry property of symmetric cryptography schemes.

Mathematically, encryption and decryption can be viewed upon as functions:

$$encrypt : plaintext \times key \mapsto ciphertext$$
$$decrypt : ciphertext \times key \mapsto plaintext$$

Figure 2.1: Encryption and decryption in terms of functions

And of course, the following *symmetry* property must hold:

$$decrypt(encrypt(plaintext, key), key) = plaintext$$

Figure 2.2: Symmetry property

It is obvious that a third party should not be able to decrypt encrypted information shared between two parties. The security of the *cryptography scheme* used by the two parties depends on several points:

[1] It may be possible to deduce the decryption key from the encryption key and vice versa

- The key shared between the two parties must be fully secret. If a third party knows the key, then it is possible for the third party to decrypt the ciphertext.
- The strength of the encryption algorithm. If an unauthorized third party wants to decrypt the ciphertext, it should not be an easier way to do so other than trying all possible keys. A strong encryption algorithm should not make it possible to reduce the key space[2] for an attacker.
- The key length. Generally the longer key length, the better encryption scheme. The longer the key is, the bigger key space, and the harder it is to try all possible keys.
- Given ciphertext and its corresponding plaintext, it should not be possible to deduce the key used to encrypt the plaintext.

Symmetric cryptography schemes use one key for both encryption and decryption as explained above. For each pair of parties which share information, one key is needed. Thus, for n securely communicating parties within a system, $\frac{n \cdot (n-1)}{2}$ keys are needed, as can be seen in figure 2.3. Effectively this means one key per communication channel. This means that if a key for two communicating parties is compromised, then only that communication channel is insecure. The other communication channels are not compromised, given that all keys in the system are unique. A weakness using this scheme however is that if a key has been compromised, a new key must be used. In order to setup a new key for a communication channel, some sort of protocol must be used for the key exchange. It is not possible to just use another key, since it has to be known in advance prior to communication with the other party. And it is not wise to just send the key in plaintext to the other side.

2.1.2 Asymmetric Cryptography Schemes

Asymmetric cryptography schemes differs from symmetric cryptography schemes in that key pairs are used instead of a single key. One key is called the *private key* and the other is called the *public key*. Asymmetric cryptography is also known as *public-key* cryptography. The public key is used to encrypt plaintext into ciphertext, which can only be decrypted by the private key.

The encryption and decryption functions are similar to that of symmetric cryptography, they transform plaintext into ciphertext and vice versa. However the relation between the functions are different:

$$decrypt(encrypt(plaintext, key_{pub}), key_{priv}) = plaintext$$

Figure 2.4: Asymmetry property

It is also important that it is not practically possible[3] to deduce the private key by knowing the public key:

$$key_{pub} \nRightarrow key_{priv}$$

Figure 2.5: Public and private key relationship

In order for party A to communicate with party B, A must encrypt the information with B's public key. B can then decrypt the ciphertext using its private key. Public keys can be

[2] Key space is the set of all possible keys
[3] The critiera is that it should be very hard in terms of time complexity

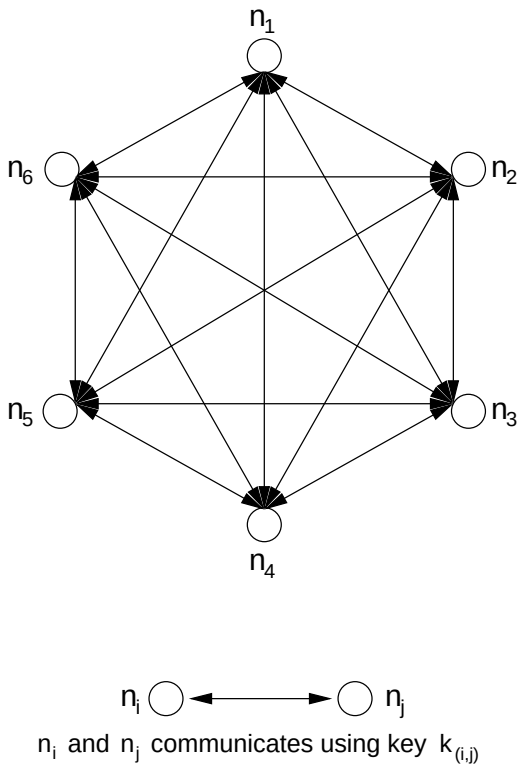


Figure 2.3: Six securely communicating parties

distributed openly to any party. An important property of the key pairs is that the private key cannot practically be deduced by the public key. The private key must always remain secret. Figure 2.6 shows how two parties communicate securely.

Unlike symmetric cryptography schemes, keys are not bound to a communication channel. Keys are bound to the communicating party. Therefore among n communicating parties, only n key pairs are needed. The number of keys have been greatly reduced, but it comes with a price. If the private key of party A has been compromised, then no information can be securely sent to A using A's public key. However, if A's key has been compromised, A can easily generate a new key pair.

2.2 Cryptographic One-Way Hash Functions

Cryptographic one-way hash functions are functions that have the following properties

1. maps an infinite domain to a finite range
2. small changes in input yields big changes in result
3. there exists no inverse function
4. it is very hard to find a pair of input data which yields the same result

The first important property is that the one-way hash function maps an infinite domain to a finite range. This means that the function applied to some data is reduced into a hash code,

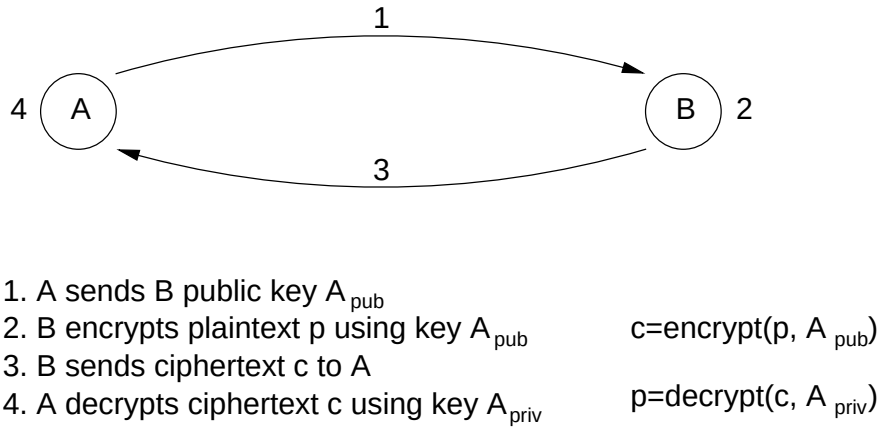


Figure 2.6: A and B communicating using asymmetric cryptography

also known as *digest*. The function should distribute the digest evenly over the range so that $x \neq y \wedge hash(x) = hash(y)$ occurs as seldom as possible.

The second property is that small changes in input yields big changes in digest. This is to ensure that given only digest codes, it should be very hard to deduce the input data.

The third property of a one-way hash function is that there exist no inverse hash funcion. I.e., for $hash(x) = y$ there may not exist a function such that $hash^{-1}(y) = x$.

The fourth property of a hash function is that it should be very hard to find a pair of input data x_1 and x_2 such that $hash(x_1) = hash(x_2)$. If a function fulfills this property, it is said to withstand *birthday attacks*. The name birthday attack stems from a statistical paradox called the *birthday paradox*. The paradox asks two questions:

1. How many persons are required to be in a room with *you* so that the probability is fifty percent or more that anyone in the room has the same birthday as *you*?
2. How many persons are required to be in the room so that the probability is fifty percent or more that any two of the persons in the room share the same birthday?

The answer to the first question is 253. The answer to the second question is as low as 23. The reason for this is that in the first case one is given a fixed starting point. In the latter case no fixed starting point is given, it is basically the problem of finding a pair of persons which fulfill the probability criterion. If 253 persons are needed to find another person in the room with same birthday with a 50 percent probability, how many persons can be paired in 253 combinations? The answer is 23 which is roughly the square root of 253. A more mathematical explanation behind this paradox is available in [Gar00] pp 28 and [BS96]. This paradox is important not to overlook as we shall see in section 2.3.

2.3 Digital Signatures

A digital signature is used to ensure the integrity and authenticity of some information.

Generation of signatures are generally done by first creating a digest using some hash function, of the information that is to be signed. This digest is then encrypted with a private key using some asymmetric cryptography scheme. Note that the roles of the keys in the key pair are switched. The public key is used to decrypt ciphertext encrypted using the private key.

Verification of digital signatures is done by decrypting the encrypted digest (the ciphertext). Then a new digest is generated by applying the same hash function on the information that was used for the generation of the signature. Then the digests are compared. If the digests are not equal, then the signature cannot have been generated for the claimed information or it could not have been signed by the claimed signer. If the digests are equal, then it is possible to say that the information was signed by the claimed signer and that the information has not been modified since it was signed.

It must have been the claimed signer who signed the information since it was the public key of the signer which decrypted the encrypted digest. The digest was encrypted using the private key of the signer. This makes the document authentic. The document may not have been modified since it was signed either, since the digest was equal to the decrypted digest.

When generating digital signatures it is important that the hash function used to create digests is not vulnerable to birthday attacks. Suppose that a manager wants to fire someone in the department he manages. He lets the secretary prepare a document for this. The person being fired is a friend of the secretary. The secretary does not wish that his friend is to be fired. Instead the secretary writes two documents which yields the same digest. He writes one document which meaning is that his friend is to be fired, while the other document is to be given a salary raise. The secretary then hands over the first document to his manager so that the manager can signed the document so that it will be clear that this document is authentic. The manager then gives the signed document to the secretary for deliverance. The secretary then copies the signature from the “fired” document and simply places it in the “salary raise” document. The secretary does not need the managers private key to do this, since the digests are equal for the two documents. An encryption of the two equal digests would result in two equal ciphertexts. The second document is then handed over to the salary division of the firm. The result is that the friend of the secretary gets a salary raise, since the salary division verified that the signature was correct.

The example above is a simplified version of an example given in [KPS95] chapter 4.

2.4 Certificates

2.4.1 Introduction

Certificate is the digital equivalence of identification tokens such as passport and drivers license. It constitutes attributes about a subject which have been certified by an authority also known as *certificate authority*. Anyone who trusts the certificate authority for a subject, may trust the authenticity of the subjects certificate. Certificates are digitally signed by the issuing certificate authority, so the validity of a certificate can be determined.

In the context of cryptographical applications, at least one of the signed attributes is a public key of the subjects key pairs. This public key can then be used for secure communication with the subject using some asymmetric cryptography scheme, as well as signing certificates.

There are mainly two ways of organizing certificates and certificate authorities in terms of

trust. Either the organization is tree like (hierarchical) or graph like (web). Since these organizations are *key-centric*, they are also known as *Public-Key Infrastructures* or *PKI*.

2.4.2 Tree Organization

In a tree organized PKI, any certificate holder may issue a new certificate using its private key which belongs to the public key of the issuers certificate. Certificate holders which have signed their own certificates are called root certificate authorities (CA). Any other certificate issuing entity is called a certificate authority.

The reason for this hierarchical divisioning is delegation. Instead of making the root CA responsible for all certificates, the root CA may delegate the responsibility to CA's below itself. Below means that these CA's holds certificates signed by the root CA. This way the root CA is relieved from issuing certificates for "end users". The CA's below the root CA may now issue certificates for "end users".

The result of this organization is a tree where the root node is the root CA, all internal nodes are CA's and the leaves are certificate holders which are not issuing certificates as can be seen in figure 2.7.

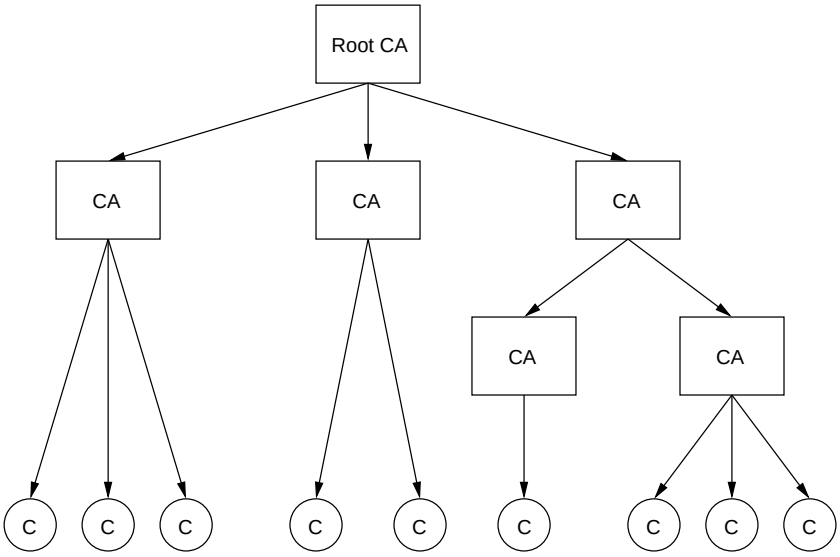


Figure 2.7: Certificates in tree organization

This is how the certificates are organized in terms of trust. A CA which signs a certificate assumes the certificate to be correct.

To trust the correctness of a certificate, some CA above the certificate must be trusted. This means that for each trustworthy certificate there must be a trusted path, or *chain of trust*, from a trusted CA to the certificate itself. Trust in this sense basically means that it is possible to verify the path by some means. If CA itself is trusted, then all certificates below it may be trusted. If the CA becomes untrusted for some reason[4], then no certificate below it can be trusted.

Large companies may for instance create their own tree PKI for internal and external use. When two companies wish to establish trust with eachother, it is possible to do *cross certification*. The root CA of company X and the root CA of company Y can sign eachothers root

[4]The CA itself or any CA above it may have been compromised

CA certificates. This cross certification yielded a forrest (set of graphs). The effect of it is that employees in company X may trust the certificates from company Y.

A benefit of organizing certificates hierarchical is that verification of a certificate is quite easy. A drawback is when a CA becomes untrusted for some reason. That means that all certificates issued by the untrusted CA may no longer be trusted. This is because all certificates below it was signed using the untrusted CA's private key. The certificates below the untrusted CA can then not be reused since their signatures are coupled to a defunct key. The certificate holders below the untrusted CA must request new certificates from some other trusted CA. This may be very impractical if the untrusted CA has issued many certificates. The worst case scenario is when the root CA becomes untrusted. Then the whole PKI must be rebuilt.

An well known implementation of a tree organized PKI is the X.509 standard. For more information about X.509 see [Sta00] pp 101 – 110 and [CCI89].

2.4.3 Graph Organization

Graph organized PKI's does not imply a root CA. There is no authority at all. Certificate holders themselves keep track of which certificate can be trusted or not. The trust is also weighted from *does not trust* to *trusts all that the certificate holder trusts*. Figure 2.8 is an example of graph organized certificates. This graph is also known as *web of trust*.

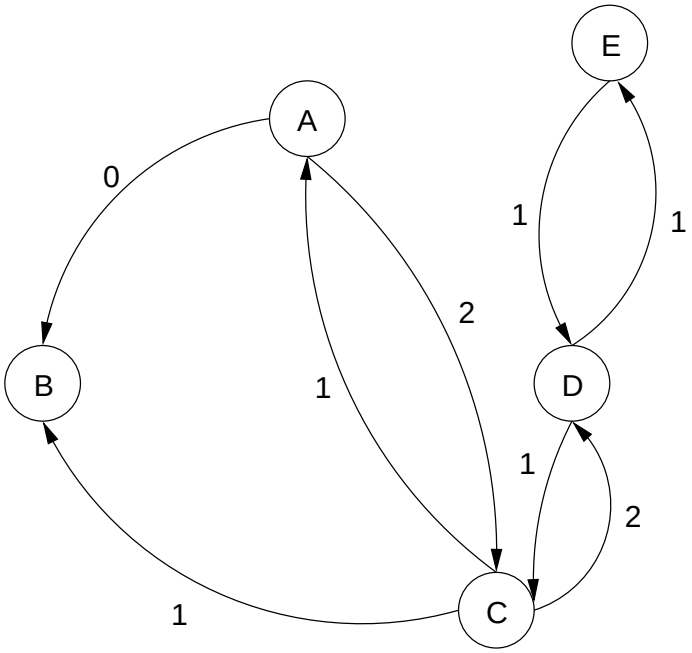


Figure 2.8: Certificates in graph organization

The example in figure 2.8 shows a simple web of trust. It is easy to see that C trusts the identity of B and B trusts the identity of C. It is also easy to see that trust relationships are not bidirectional. One can see that C trusts whatever D trusts, but D trusts only the identity of C. Also C trusts the identity of B, but B does not trust C. What is less obvious is that A trusts the identity of E, despite that there exists no edge directly from A to E. A trusts whatever C trusts. C trusts whatever D trusts. D trusts the identity of E. So A trusts the identity of E transitively. There are dangers with transitive trust. There may be cases where A trusts the identity of B both by a direct edge and by transitive trust via C. If C becomes less trustworthy, A can still trust the identity of B because of the direct edge. However, if B becomes less trustworthy, i.e. not trustworthy at all, then B is still trustworthy since there is still transitive trust via C! Therefore "no trust"-edges are needed as can be seen in figure 2.8. Eventhough A trusts B transitively via C, the direct "no trust"-edge from A to B overrides. If there exist no edge (direct or transitive) between two nodes in the graph, then the trust between those two nodes is undefined, which is to be interpreted as "Does not trust".

This organization is quite complex. It is up to the individual to decide who is more or less trustworthy.

At the same time, the web of trust is very powerful in the way that a subject must not rely on a fix point in the organization, such as a root authority as in the hierarchical organization. A compromised node in the graph will only affect the edges connecting to that node.

An implementation of a graph organization is PGP. For more information about PGP see [Sta00] pp 118 – 136 and the PGP website [PGP].

2.5 Reference Monitors

2.5.1 Description

A reference monitor is a part of the kernel which control accesses to operating system objects. It may inspect the process which attempts to access the object, it may inspect the object itself and the access operation (system call). These parameters are then compared against either a set of static rules, or a set of dynamic rules, to accept or deny the operation and perhaps log the operation. Static rules are also known as *access control rules* and sometimes *access control lists*. Dynamic rules are rules that may adapt themselves over time. An *Intrusion Detection System* may for instance learn the usage pattern of a particular user and then perform some action when the user actions does not conform to the expected pattern.

A logical overview is found in figure 2.9.

2.5.2 Criteria

A reference monitor must have the following criteria:

Tamperproofness No process may change the state of the reference monitor in an unauthorized way

Always Invoked The reference monitor should be started when the kernel boots, and be active throughout the entire life time of the kernel

Small and Compact The reference monitor shall be as small and compact as possible

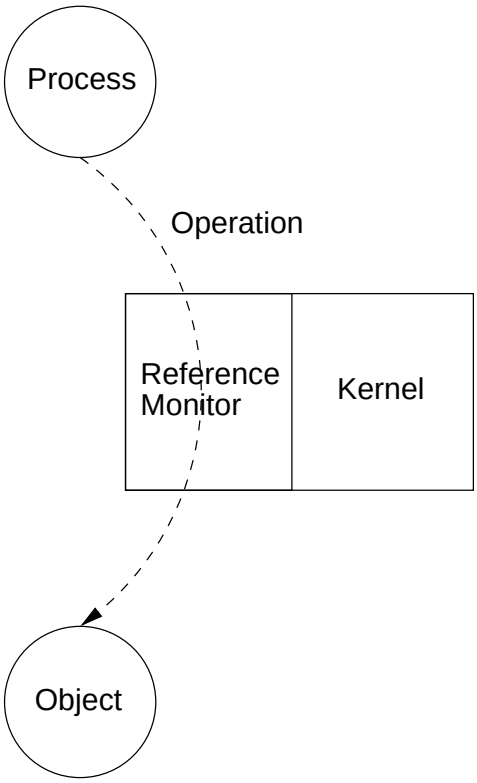


Figure 2.9: Logical overview of a Reference Monitor

2.5.2.1 Tamperproofness

No process may change the state of the reference monitor in an unauthorized way. If such state changes were possible, then circumvention could be possible. All object accesses must be monitored by the reference montor, and appropriate action must always be performed. If not, there would not be any point in having a reference monitor in the first place.

2.5.2.2 Always Invoked

The reference monitor must always be invoked so that all object accesses are monitored. No object access may be performed without having been monitored.

2.5.2.3 Small and Compact

The reference monitor must be small and compact to reduce the complexity of the reference monitor. With reduced complexity it becomes easier to analyze and test the reference monitor to ensure that it is complete to meet the above criteria.

2.6 State Machine Security Models

2.6.1 Description

A state machine security model is a *system model* which defines a finite state machine. The state at any given time is an aggregated state over one or more *state variables*. State variables denote sets of elements. These sets have at least one security attribute which plays some role in the state machine. These state variables may only change in such ways that the aggregated state is always a *secure state*. Therefore operations on the state variables are guarded by *state transition functions* which make sure that the system never enters an insecure state. The secure state is defined by one or more *invariants* which the state transition functions must adhere to. A state machine model may also include functions which transform some values to some other values.

A state machine security model must like a reference model be small and compact. It should be easy to comprehend so that it can be proven, and that it can be implemented with full confidence that the implementation really implements the security model.

It is important to understand that the model does not define any design or implementation details. While state machine models usually view the state variables as *sets*, it is not necessary to implement the state variables using the *abstract data type* set. For instance a model may view all available files in a computer system as a set of files. The operating system however (at implementation level) views all available files as a tree (the directory tree structure). The model says nothing about the implementation. However, the implementation says everything about the model.

2.6.2 An Example

I shall now present a very simple state machine security model. I will first give an *informal description* and then a *formal description*. Informal descriptions are good means of presenting the idea of the model in human language. It helps the reader to get a broad picture of the means and goals of the model, and it also gives an indication whether the model is sound or not. The formal description is a more exact rewrite of the informal description. It is commonly written using some form of formal notation. Formal notations are useful since they usually offer the ability to prove the correctness of the model.

2.6.2.1 Informal Description

This example will model a simple computer system. A computer system is a computer which users can log onto. The computer system holds a catalog of all users which are allowed access to the system. Each entry in the catalog is a tuple consisting of a user id and a hashed password.

When a user logs on, that user must first claim its identity and then a password. The system then looks up the user in the catalog. If the user does not exist in the catalog, then the login is rejected. If the user does exist in the catalog, the provided password is hashed and then compared to the hashed password in the catalog entry. If these two hashed passwords does not match, the login is rejected. Otherwise, the login is accepted.

An already logged in user may at any time logout without any restrictions.

Note that this model does not take any action on commands issued by the users after they have logged in. This example only model the authentication procedure.

At any point in time, all logged in users must be referenced in the catalog of users with access to the system. This is the system invariant.

2.6.2.2 Formal Description

System state is denoted as $\Sigma = \{C, U, uid, hpasswd\}$

System State Variables

Σ **Components**	
C	The set of all authorized users
U	The set of all logged in users

Table 2.1: System State Variables

The C variable denotes all authorized users. This set is corresponds to the catalog of users mentioned in the informal description. Each member of the set is a tuple on the form $(uid, hpasswd)$ where uid is a unique id for a user and $hpasswd$ is the hashed password for the user.

The U variable denotes all logged in users. All users in this set must have been properly authenticated. Each member is a unique id for a user - uid.

Security Attributes

Security Attributes	
$uid(c)$	Unique user id of catalog entry c
$hpasswd(c)$	Hashed password of catalog entry c

Table 2.2: Security Attributes For System Variables

Functions

Functions in the System	
$hash : p \mapsto hp$	Hashes a clear text password p into a hashed password hp
$isvaliduser : uid \mapsto bool$	Predicate for determining whether a user with unique id uid is a member of C
$lookupuser : uid \mapsto c$	Looks up a catalog entry for user with unique user id uid. The preconditions for this function is that there must exist such a user.

Table 2.3: Functions

The system invariant: $\forall u \in U : isvaliduser(u)$

The system invariant says that for all logged in users, each user must be represented in the user catalog C.

proc *Logon*$(uid, passwd) \equiv$
  if *isvaliduser*$(uid) \wedge$ *hpasswd*(*lookupuser*$(uid)) =$ *hash*$(passwd)$
    then
      $U \leftarrow U \cup \{uid\}$

Figure 2.10: State Transition Function: $Login$

State transition function *Logon* In order to logon, a user must exist in the user catalog, and the password which the user gave in the login process must be equal to the one in the catalog. Once these criteria are met, the user is considered logged on.

Chapter 3

Existing Models and Technologies

3.1 Overview

This chapter examines models and technologies which are in many ways similar to the model which this thesis is about.

3.2 Java

3.2.1 Introduction

Java is an application development platform developed by Sun Microsystems. It is comprised by:

language **[GJSB00]** the java language and associated compiler

distribution mechanisms special distribution format and tools which allow binding of meta data to an application

virtual machine [LY00] the java interpreter

set of services class libraries and methods for communication with the outside of the virtual machine

These points provide the basis for the security mechanism found in the java platform. The aspects of this security mechanism are of interest for this thesis.

3.2.1.1 The Java Language

The java language is used to describe algorithms and data structures in a formal notation, which allows execution in an automated environment. The language is an object oriented language - it has the elements which one would expect from such a language. The most significant element of the java language is the class concept. Classes are instantiated in run time to form objects, which is a normal concept in object oriented languages. What is specific to the java language is that the compiler generates a single class file for each declared class. This means that for an application which declares X classes, X class files will be generated by the compiler.

3.2.1.2 Distribution Mechanisms

When distributing an application written for the java platform, it is distributed in a special format. This format is called *java archives*, or *jar* for short. A jar is a single file. After an application has been compiled into class files, these files are then put into the jar file. In common programming languages such as C or C++, it is common to compile the individual files into object files, which are then linked together to form a binary program file. The process of creating a jar file can be compared to the process of creating a binary program file for C or C++. The main difference is that the jar file is not a native binary program file.

3.2.1.3 The Virtual Machine

When an application in a jar file is executed, the jar file is read by the java interpreter. The java interpeter is a native program in the operating system. It runs like an ordinary process in the operating system. It uses the operating system services to implement a new java specific environment, in which the java programs are executed.

The java interpeter is given the jar file and a class name which is used for boot strapping the execution. The java intepreter then unarchives the referenced class and executes a boot strap method (a method called main()) in that class. From there, the execution continues. The interpreter acts as an operating system loader. The interpreter also defines a virtual machine, which the java application executes in.

3.2.1.4 The Services

When an application wishes to communicate with the environment outside the virtual machine, it must do so through the interpreter. This is achieved by using the services provided by class library that comes with the java platform. There are java services which allows utilizing the services provided by the host operating system as well as general programming services such as abstract data types and etc.

3.2.1.5 The Security Mechanisms

What is interesting within the scope of this thesis, is the security mechanism in the java platform. The security mechanism includes properties such as digital signatures and access control.

The java platform introduces a concept called *Security Manager*. The security manager is however not a new concept. The concept is generally known by the name *Reference Monitor* [Sta00] pp 332-33. The role of the security manager is to monitor the requests made by applications. Such a request may be to read a file, write to a file, establish network connections and so on. Based on some rule, a request is either accepted or denied. This is exactly what the security manager in the java platform does. An illustrated example is given in figure 3.1. The figure depicts a scenario where the rules are simple; the java application may read the file *file.ext*, but it may not write to it.

This approach has also been called “the sand box model”. The idea is to execute the application in a well defined environment. Communication with the environment outside the sand box is constrained.

In the java platform, the granularity of the access control rules is very small. It is possible to apply rules for single applications and single operations. This makes it possible to apply *the principle of least privilege* for individual applications.

A default security manager ships with the Java platform. However, it is possible to write a new security manager if need be. The default security manager can handle policies for signed as well as unsigned jar files. When invoking java applets, the default security manager is always invoked. Applets downloaded and executed directly from the web, are always given the most restrictive security policy. For instance, it may not open files on the local file system, it may not communicate with other computers on the Internet, other than the one it was downloaded from. Applications do not activate the security manager by default.

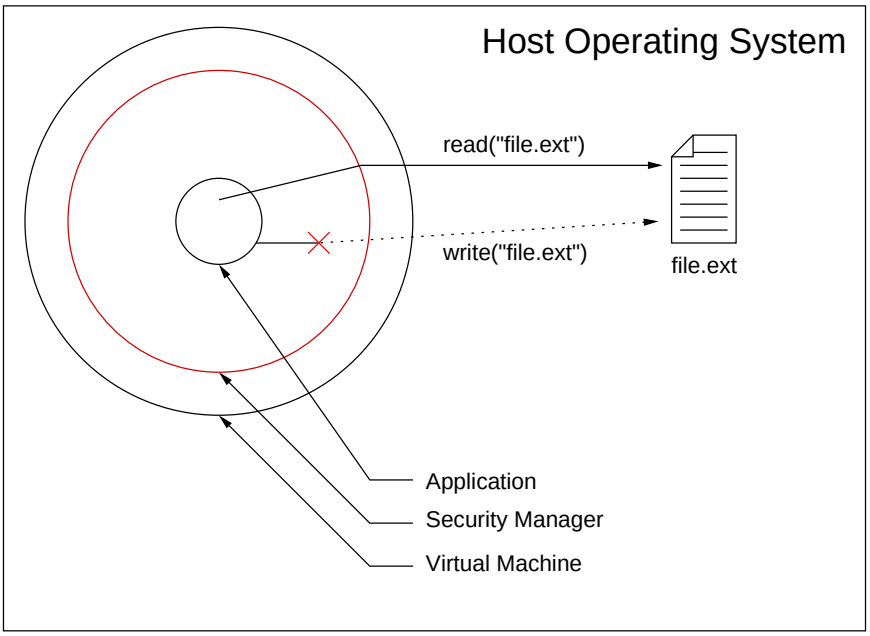


Figure 3.1: An application is denied write access to a file

3.2.2 Security Mechanisms in Detail

3.2.2.1 Signatures

Signatures in the java platform are created using public-key cryptography and one-way hash functions. The jar files are signed using the private key in the development environment. The signed jar is then shipped to the receiver. What the receiver needs in order to establish authenticity of the jar file, is the public key associated with the private key which was used for signing the file. The receiver may have received the public key earlier or it may be delivered inside the jar file.

3.2.2.2 Certificates

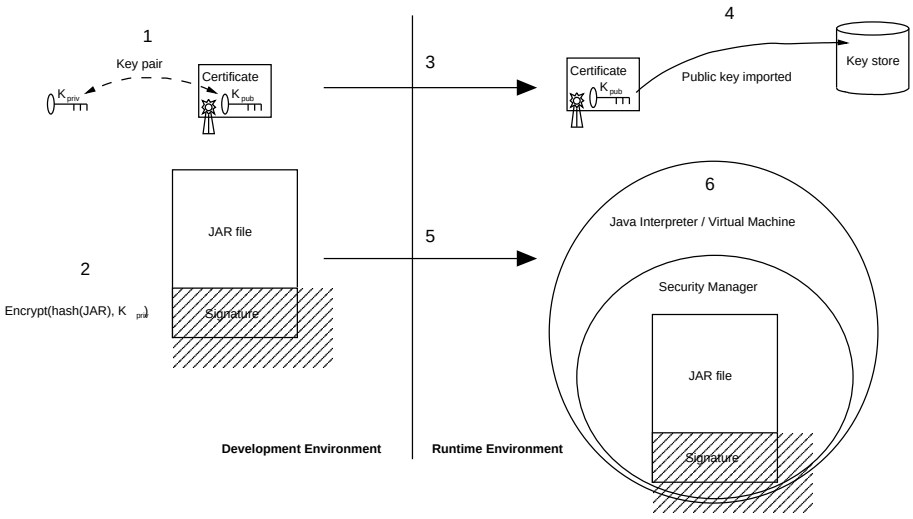
The public key which is used to authenticate jar files are packaged within certificates. In the java platform the X.509 format is used. The public key associated the certificates are stored in so called *key stores*. Before the java interpreter loads the jar file and starts the requested application, the security manager uses the public key associated with the certificate, found in the key store to test the integrity and authenticity of the jar file.

X.509 also standardizes revocation of certificates. This means that the validity of a certificate may be revoked because of various reasons. For example, a certificate holder accidently exposes the private key for a certificate, then it is possible to impersonate the certificate holder.

3.2.2.3 Policies

After the java interpreter has authenticated a jar file, and loaded the requested application, the security manager continously tests whether the application behaves according to predefined rules - *policies*. Policies can be specified per *signer*, *operation* and *target*. For instance; Applications written and signed by signer *X*, may *write* to the file *Y*.

Figure 3.2 shows how certificates, signatures, jar files, key stores, java virtual machines and security managers work together to establish authenticity and integrity.



1) Generate key pair (public and private) along with certificate
2) Sign JAR file
3) Distribute certificate
4) Import the certificate into key store if certifcate holder is found trust worthy
5) Distribute signed JAR file
6) Test JAR for authenticity and integrity using the key store and attached signature
7) Execute according to security policies

Figure 3.2: Certificates, signatures, jar files, key stores, java virtual machines, and security managers establishing authenticity and integrity

3.2.3 Advantages

Authenticity The java platform provides the framework for authenticity, which can be used to establish trust between user of an application and the creator of an application. The platform makes use of PKI standards such as X.509 which is an open standard, thus it has been publically scrutinized.

Integrity The java platform ensures that integrity can be tested along with authenticity. Since the authentication of jar files are based on the contents, the effect is that integrity can be tested.

Policies The java platform enforce security policies. It is possible to restrict individual applications within jar files.

3.2.4 Disadvantages

Security Manager The security manager is not enabled by default for java applications. It is only enabled by default for java applets.

3.2.5 Summary

The java platform defines an environment where java applications may execute. The defined environment, also known as the java virtual machine, execute as a program inside a host environment. It is possible to restrict execution of java applications by using signatures and policies.

3.3 ActiveX

3.3.1 Introduction

ActiveX is a component object model based on Microsofts COM[1] technology. A component is native code intended for execution within a host program. The Microsoft Internet Explorer is such a host program. It allows for embedding ActiveX components into HTML pages. Normally when downloading HTML pages, embedded parts such as images are downloaded automatically without notifying the user. The images are seen as parts of the HTML page, thus it is natural that the user does not have to request that images shall be transferred. ActiveX components on the other hand, contain code that is intended to execute within the web browser. There is an immediate danger in automatically downloading and executing code. A rogue web master could create ActiveX components which performs some malicious action when executing. Because of this, Microsoft extended the ActiveX model with a technology called *Authenticode*. The role of authenticode is to "label and shrink wrap" software on the Internet. The motivation for labeling software is to ensure accountability and authenticity. With accountability, Microsoft means that a user shall be able to hold the vendor accountable for the actions performed by the component. Authenticity ensures that a component really comes from whom it claims. It also ensures that no third party has tampered with the component since it was signed by the vendor.

3.3.2 Authenticode

Authenticode signatures are attached to the file which a component is distributed within. It can be an executable file (EXE), a dynamically linked library (DLL), object control (OCX), java class file, cabinet file (CAB). In fact, the authenticode signature does not only cover ActiveX components exlusively, it can also be used to sign ordinary applications, function libraries and other codes that can be put into these sorts of files.

Authenticode signed files constitutes three parts:

- Content
- A certificate
- A cryptographic signature

[1] Component Object Model

Content can be executable code or resources.

A certificate is a cryptographical identification tag. It contains information about the certificate holder, e.g. the creator of the signed code. The crypographical signing methods makes use of assymmetric cryptography involving public key cryptography. In order to verify the signature, a public key is needed. This key is information contained within the certificate. The certificate also contains a reference to a certificate authority. This authority has vouched for the identity of the certificate holder. To protect the certificate itself from tampering, it has been signed by the certificate authority.

A cryptographic signature is an encrypted digest of the content. The digest has been encrypted using the private key of the signer. To decrypt the encrypted digest, the public key must be used.

When an ActiveX host program is about to download and execute a component, it first looks at the file at hand to see if it has been signed. If it has not been signed at all, the host program warns the user that the host program is about to download and execute an unsigned component. The user can then choose to accept or reject download and execution. If the component contains signing information, but it is not properly signed (the file carrying the component was modified after it was signed), the user is notified. The user can then choose to accept or reject download and execution. If the signature is valid, then the certificate is presented to the user. The presentation of the certificate include information about the signer, whether the signer is a corporation or an individual, when the certificate expires and what certificate authority vouches for the identity of the signer. The user is not only given the opportunity to reject download and execution of the component, but it is also the opportunity to always accept future components from the signer and the opportunity to always accept any component created by any signer vouched for by the certificate authority.

The steps for verifying a signature are as follows:

1. Inspect the certificate - test the certificate against the known certficate authorities. Information needed for performing this test comes with the operating system (e.g. Windows 98). The actual testing steps are basically the same as the following steps - a signature on the certificate is verified using the public key of the certificate authority.

2. Generate a digest of the component code using a one way hash function.

3. Decrypt the signature code using the public key found in the certificate.

4. Compare the digest with the decrypted signature code. If they are equal, then the code was signed by the certificate holder, and the code has not been modified since.

The steps described above can be seen in figure 3.3.

3.3.3 Advantages

Authenticity Authenticity is possible. It is always possible to determine where the code comes from.

Integrity It is always possible to determine whether a third party has modified code after signing.

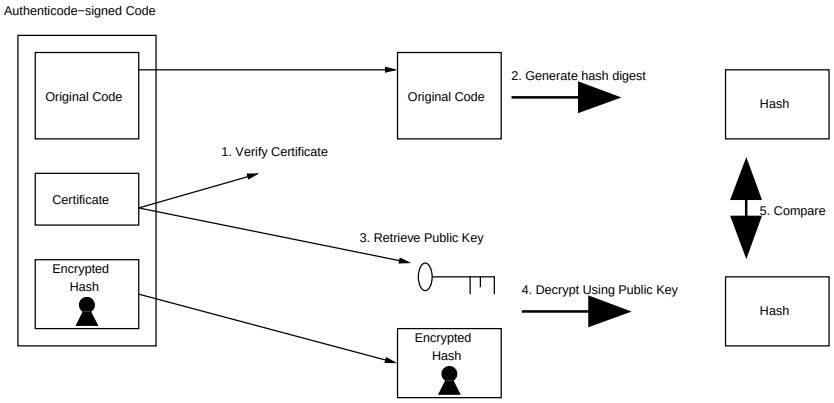


Figure 3.3: Authenticode Verification Process

Standards Compliant The technology uses well known standard mechanisms for signatures and certificates.

3.3.4 Disadvantages

Discretionary Control It is up to the user to accept or reject code. This is not good for any organization with security policies, since a single user may break the policy by accepting untrusted ActiveX components.

Weak Mandatory Control It is possible to apply system policies that rejects downloading executable code by any user. However, these policies are weak. Decisions on what is executable or not is based on the file extension. Furthermore, it is possible to download executable code using the floppy drive and then execute the code. The system policies only applies to the Internet Explorer program.

Accountability This model claims that it gives the user accountability. If a vendor creates a component which is malicious, the user can hold the vendor accountable for the actions performed by the component. This does generally not hold for any piece of software. It is common that the license which the user must accept for using the software, contains disclaimers basically saying that the software may or may not cause damage to the users computer or information therein. Therefore, accountability is as weak as the license for which the software is distributed under.

3.3.5 Summary

The ActiveX security model, Authenticode, is good for ensuring authenticity and integrity in software. However, it does not provide any means of access control. There is no way of specifying what an ActiveX control can and cannot do witin the system. For more information about Authenticode, see [MS96].

3.4 Tripwire

3.4.1 Introduction

Tripwire is a *file system integrity checker* program written and maintained by Tripwire Inc. The main purpose of this product is to monitor file system changes and alarm when such occurs. Tripwires performs integrity checks on files and directories including special devices, symbolic links and so on.

Tripwire monitors changes by computing signatures of file system objects. Various ways of computing signatures are described in [GE94]. The signatures of the monitored objects are then put into a database which is the main component of the tripwire product. The database serves as a point of reference for determining file system changes. Obviously this database must be kept secure at all times to prevent attackers from modifying it to hide their file system changes. The database is preferably stored in on a read-only device and is updated by other means than "traditional" file system access. The database contains no information which an attacker can use for malicious purposes.

Another vital component of the tripwire product is the configuration database. This database contains directives for the tripwire administration tools on how signatures are to be computed, which file system objects are to be monitored and so on. Some files in a system changes often, especially system logs which may change every second. This database must also be kept secure at all times so that an attacker cannot change it to hide their file system changes.

The administrator of a system uses supplied tools for creating and maintaining the signature database and for detecting file system changes. These tools must, just like the signature and configuration database, be kept secured. If an attacker can modify the tools, the tools can no longer be trusted.

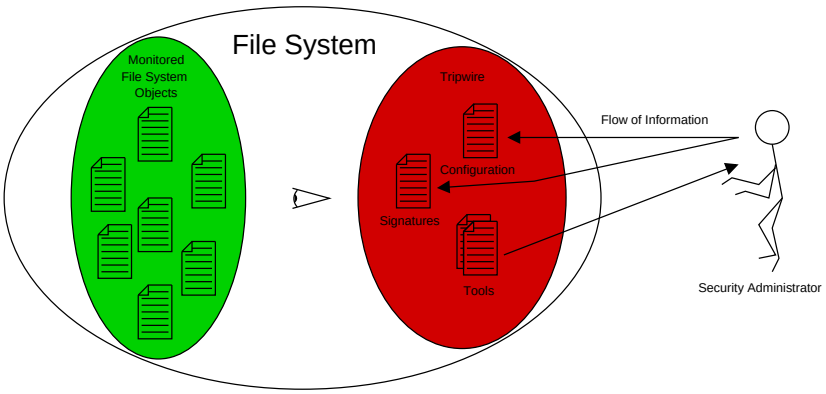


Figure 3.4: The tripwire product

To maintain a tripwire installation, one must first create a configuration database. Based on the configuration database, an initial signature database is created using the supplied user tools. From this point, the file system is now guarded by tripwire. To detect unauthorized changes in the file system, the tripwire tools can be used to test the file system integrity. If the system administrator needs to do an authorized file system change, then the signature and configuration database must be rebuilt to reflect the change.

3.4.2 Advantages

The advantages and disadvantages discussed here are at the conceptual level, rather than the technical level. Tripwire as a product, has many technical advantages such as letting the system security administrator interface with programs for various purposes. E.g. if the signature computation algorithms provided with tripwire are not satisfying, a custom routine can be hooked into the tripwire product. Tripwire Inc. also provides impressive GUI tools for log acquisition and such. These features does not add anything to the product at a conceptual level, they are merely simplified tools.

General file system protection Any file system object can be monitored.

Small overhead The file system integrity check can be performed during low activity periods thus achieving a small overhead and a small performance impact on the computer system.

Self contained Given that the signature database and the administrator tools are immutable, the tripwire system is self contained. Nor do the tools depend on other libraries or programs in the operating system, which otherwise could be used to undermine the security of the tripwire system.

Denial of denial of service If an attacker manages to penetrate a tripwire guarded system and manages to modify files, the system does not halt or stop functioning.

3.4.3 Disadvantages

Integrity checks are done on command Integrity checks are done periodically on command. Depending on the period time, the attacker is given an open time window to work in. If the period time is big, say 24 hours, then the attacker has 24 hours to "do the job". On the other hand, making the period too small, say 5 minutes, then there is a high performance impact on the system - verifying the integrity requires intensive calculations. This is a trade off which needs careful considerations - higher security of higher performance?

Integrity checks must be done on command The tripwire tools are applications which are run in user space. This means that tripwire have no access to kernel internal functions and data structures. Thus it is not possible to perform integrity checks on a monitored file directly after it has been accessed.

Weak trust The trust model is based on the fact that tripwire will perform its task correctly. If the files themselves cannot be trusted to begin with, the model fails, even if tripwire performs its task correctly.

3.4.4 Summary

Tripwire does a great job of monitoring any kind of file system object. However, the system has a trade off: high security/low performance or low security/high performance.

Chapter 4

Proposed Model

4.1 What does it do?

4.1.1 Overview

This model considers two stages of program execution; *before execution* and *during execution*. Before execution means after a BEC (Binary Executable Content) has been requested to execute but before it has been loaded into memory by the operating system kernel. During execution means after the BEC has been loaded into memory. Before execution will later be referred to as *stage 1* and during execution will later be referred to as *stage 2*.

4.1.1.1 Stage 1 - Before Execution

The standard text editor that comes with the operating system is used to produce text documents. Users of the operating system know that the text editor edits text and nothing else. It is considered a *known* BEC assuming that there is no trojan code in the editor. If a user of the operating system has second thoughts about the standard text editor, the user may acquire a different one from the Internet or perhaps from a friend. The newly acquired text editor is not a *known* BEC, since it has not been used on a daily basis, in fact it has not been used at all. Because of that, the BEC is considered *unknown*.

The terms known and unknown are weak. A user may know of a program in advance, a friend may have talked about it or the user may have read a good review, but has still not run it once. The user cannot know in advance how the program is going to execute. Therefore a stronger term needs to be introduced; *trust*.

How can the user trust that the program will do what it claims? The user cannot do that, unless given the source code of the program, which could then be analyzed. But this is seldom the case, programs are usually distributed in binary form which are much harder to analyze. It is easy to back this fact up just by visiting the most popular shareware sites on the Internet or by reviewing the distribution of any proprietary software. To analyze proprietary programs, it has to be reverse engineered, which is not always legal depending on the laws for which the user must abide. Because of this, the users trust must be directed to something other than the program. If the user trusts the vendor which produced the program, then the user can transitively trust the program.

In this model, trust for a BEC is established iff the BEC has been signed by a trusted entity.

A signature in this context is conceptually equal to the old way of sealing documents. After a document had been written, it was folded up and sealed with a wax mold. This would assure

that no one had read or modified the document before it was opened again. And finally a stamp was pressed into the wax mold, making the document authentic. A BEC is signed so that its origin is possible to determine, and if it has been modified since it was signed.

It is important that, after trust has been established with a BEC, the trust is not misused. It should not be possible to modify the BEC after it has been signed without breaking the seal. Nor should it be possible to forge a signature to produce a new seal. This model addresses this by use of cryptographic methods, which makes it very hard to break.

This stage of the model does not make a distinction between programs and other BEC files. The reasoning is simple; Most operating systems today use dynamic linking and loading. This means that BEC files such as function libraries are not aggregated with the program file. On the contrary, such BEC are put into files of their own. These BEC files are then loaded dynamically when the program which uses them starts. Function libraries are not aggregated with the program BEC until it has been activated - i.e., it has become a process. Since BEC files such as function libraries are separated from the program, the behaviour of a program may change if the function library is modified.

What has been described above constitutes the first stage of this model; Only trusted BEC files are allowed to execute. Stage 1 is depicted in figure 4.1. It acts as a guard against executing untrusted BEC files.

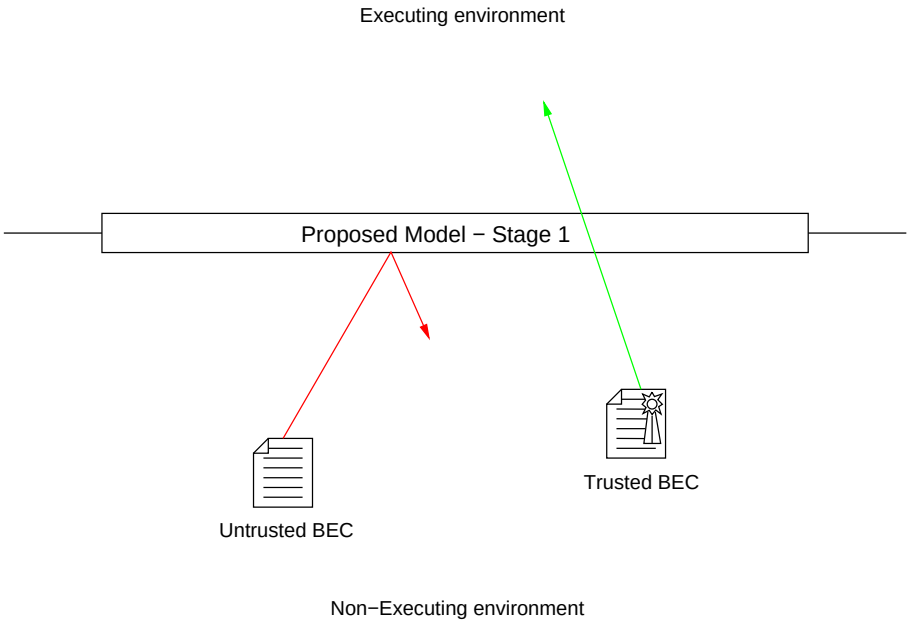


Figure 4.1: A simple view of the proposed models first stage

4.1.1.2 Stage 2 - During Execution

When a program is executing, it has access to the operating system services. These services include means of communication with the outside of the computer system, manipulation of file system objects, execution of other BEC files, interprocess communication and etc. From a security perspective, all these services should not be available to all programs. Again, the principle of least privilege rules. A program whose functionality is to count words in a text file,

should never have the access to open network connections, executing other programs etc. It should only be given access to open and read files.

Stage 2 of this model restricts a programs access to operating system services. This is achieved by introducing a concept called *access control rule*. An access control rule is a rule that *grants access* for a program, to an *operating system object* with an access right. An operating system object may be a file, process, network connection etc. Access rights may for instance be *read, write* or *execute*. This allows for fine grain access control of programs which promotes the principle of least privilege. In short; stage 2 of the model is a reference monitor. Conventional reference monitors view processes as "extensions" of the user. The programs acts on behalf on users, thus the subject being monitored is the process using the priveliges specified for the user who started it. This model views the process as an extension of the program, rather than the user. Thus the subject being monitored is the process using the priveliges specified by the program.

Figure 4.2 shows a simple scenario where a program is granted read access to a specific file.

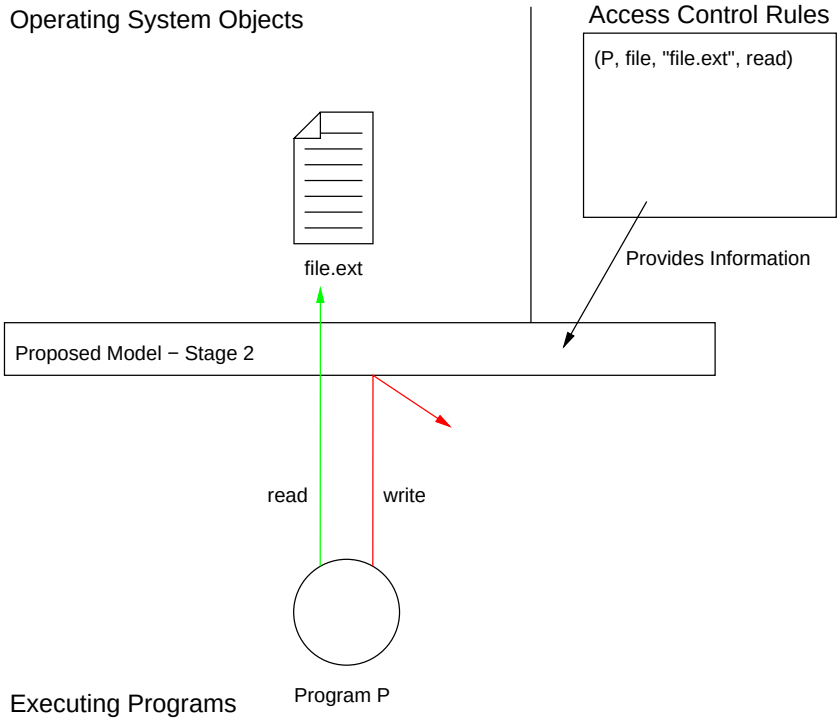


Figure 4.2: A simple view of the proposed models second stage

4.1.1.3 Summary

In the conventional user access control model, users must log on, or authenticate themselves, to the system in order to use it. While being logged on, users are restricted by the access control mechanisms in the operating system, such as file access bits or access control lists. The proposed model treats BEC files similarly to that of users in the above model. Before BEC files are allowed to execute in the system, they must first authenticate themselves, i.e. "log on". During execution in the system, the BEC's must adhere to the access control mechanisms. A program

executed by a user, must not only adhere to the users access rights, but it must also adhere to the access rights given to the BEC file which constitutes the program.

4.1.2 Facets of Stage 1

There are mainly five facets of stage 1:

- Authenticity
- Integrity
- Trusted signers
- Defence against attackers
- Virus protection

4.1.2.1 Authenticity

Authenticity in this model is important. If the origin of a BEC cannot be established, then trust cannot be established.

4.1.2.2 Integrity

Integrity is equally important as authenticity. If a BEC has been signed by a trusted signer, then the BEC is trusted to begin execution. If the BEC is modified by an untrusted entity (attackers, viruses and etc.) after signing, it is important that the model does not establish trust with the BEC. Trust is established under the premise that the BEC was produced and signed by the trusted entity.

4.1.2.3 Trusted Signers

In order to enforce denial of execution of untrusted BEC files, the system must maintain a set of trusted signers. The computer system cannot compute if the signer of a BEC is trustworthy or not. Trust is a concept based on human factors such as ethics and morals. Therefore the organization of humans which owns and maintains the system, must decide what signers can be trusted enough to run their BEC files. When a signer is decided to be trusted, information about the signer is added to the set of trusted signers.

This model assumes that the decision is ultimately correct - a trusted signer will not act untrustworthy. Preferrably, there should be some kind of legal binding with the signer. If a signer acts in an untrustworthy way, then it should be possible to revoke the trust earlier established. And then, of course, take some legal action against the signer based upon the “trust agreement”.

4.1.2.4 Defense Against Attackers

This model offers great defense against system attackers. Attackers are individuals who exploit weaknesses in system software to escalate their privileges to such extent that they can recover secret information, spread disinformation (modification of information), use the system for their own purposes, or perform some malicious act upon the system (removal of files, hard disk corruption and etc.). Attackers are never welcome, whether their actions are benign or not.

Attackers use tools and techniques to hide their presence, escalating privileges, sniff networks for password or install back doors. This involves downloading source codes to the attacked system, or writing source code on the system using the standard text editor. The source code is then compiled into programs (BEC files). By requiring that BEC files must have been signed by a trusted signer, the attacker runs into problems. The attacker cannot execute his or her tools. This makes the "job" for a hacker extremely difficult.

4.1.2.5 Virus Protecion

This model offers minimal defense against computer viruses. Viruses are programs that "infect" other programs. When a program is infected by a virus, it generally performs some sort of malicious action. The most common is to delete files, alter partition tables, or perhaps perform a practical joke on the behalf of the program user. In some cases the virus does not perform any action. What all viruses have in common is that they infect other programs in the system. Even if a virus does not perform malicious actions, they do degrade the system as a whole. Infection of a program means that code is injected into a program file, i.e. a modification of the program file. This gives extra overhead information in the file system, thus the file system is degraded. The virus is also consuming CPU cycles and primary memory when executing, thus degrading memory and CPU utilization. Any kind of degradation is not acceptable.

If a BEC is infected with a virus and then signed by a trusted signer, the virus is then part of the trusted BEC. Thus it is not possible to detect the virus. However, vendors generally do not release virus infected programs. Programs are scanned for viruses before put on media. Virus infected code which have been signed may not be detectable by this model, but the model slow down the spread of the virus. If it infects a BEC in the system, that BEC will not be able to execute later on. The virus itself may not be detected by the model, but its effects will quickly be detected when BEC's start to show up as "defect".

It could be argued that this minimal virus protection becomes a denial of service weakness. However, viruses are only given access if they are part of a trusted BEC. Then it is questionable if one should sign a BEC which has not been cleared by a virus scanning tool. There is a plethora of anti virus toolkits, which can scan BEC files for viruses. This model does not try to eliminate the need for these tools. On the contrary, these tools complement the model. Although new viruses not known by any scanner may go undetected into a BEC, their spread is slowed down by the model.

4.1.3 Facets of Stage 2

There are mainly three facets of stage 2:

- Software Exploitation Protection
- Trojan Protection

4.1.3.1 Software Exploitation Protection

If access control rules are set up correctly, it becomes harder to exploit program faults. A common attack on computer systems connected to the internet, is to exploit some fault in a server program. The goal of the exploit is to change the state of the computer system so that security mechanisms, such as authentication, are crippled. A common angle of attack is to make the server program overrun buffers so that it is forced to execute system calls that it was not intended to do. These system calls generally operate on operating system objects which the

server program was not designed to operate upon. If these objects are excluded from the access domain of the program, then it becomes considerably harder to exploit the faults in the server program.

4.1.3.2 Trojan Protection

Programs are restricted to what objects they can access and with what access right. Given correct access control rules, it is hard for a trojan to perform operations which the program was not intended to do.

4.2 How Does It Work?

4.2.1 Stage 1

4.2.1.1 Signature Information

Signature information is attached to BEC files as an attribute. The signature information contains information about the signer and information about the BEC files state during signing. All signed BEC files have this information. Unsigned BEC files do not. Unsigned BEC files are always considered not trustworthy.

The signature itself must be very hard to forge. There are several techniques for digital signing and authentication which will be covered in detail in section 5.1.

4.2.1.2 Trusted Signers

The proposed model assumes that all signers are initially not trustworthy. When the organization, which owns the system, confirms that a signer entity (individual or organization) is trustworthy, information about the trusted signer is added to the system. This means that only BEC files signed by a trustworthy signer may be allowed to execute in the system. In this respect the model is very defensive. It could be argued that it should be the other way around, where rules are rejection rules, or a mix thereof for greater flexibility. This however is in conflict with the principal rule of secure systems; *The principal of least privilege*. If there is a need to reject some signer, the damage may already have been done at the time of revocation of the trust. Therefore it is better to take a more defensive approach.

Trusted signer information constitutes an identifier for the signer and a public key. The identifier uniquely identifies a signer. The public key is to be used for signature checking mechanisms.

4.2.1.3 Handling of Trusted Signer Information

When information about a trusted signer is transferred to the system, certain measures have to be taken into account. An arbitrary user of the system may not add new information about trusted signers. By giving all users in a system the privilege to do so, it becomes harder to restrict what can be executed or not. Users shall be given minimal privileges to perform their duty - the principle of least privilege. Therefore it is wise to appoint a single user in the system whose duty is to maintain the policies, and let all other users perform other duties, such as text editing.

This model assumes that each user has a *security class attribute*. This attribute can have two values; *Security Officer* or *Ordinary User*. The security officer is the user whose duty is to maintain the trusted signer information. The security officer role is given to a user by the organization which owns the computer system. The decisions on what signer is to be trusted

or not are taken by the organization. When decisions have been made, the security officer is notified to perform necessary system updates to reflect the decisions. Figure 4.3 gives a more detailed view of the first stage, showing how information flows from the organization to the computer system via the security officer.

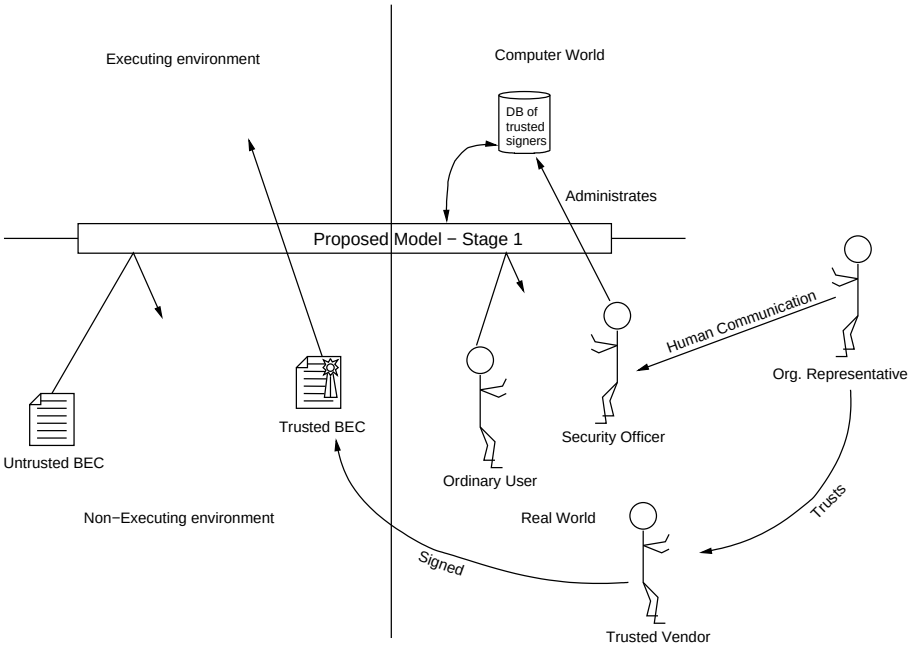


Figure 4.3: A detailed view of the proposed models first stage

4.2.1.4 Execution of BEC files

When a BEC is requested to be executed, the following algorithm must be performed:

1. Test to see if it was signed at all. If not, then the BEC is not trustworthy.

2. Acquire the id of the signer found inside the signed BEC.

3. Lookup signer information in the set of trusted signers, using the id from the BEC file. If there is no trusted signer with that id, then the BEC is not trustworthy.

4. Use the public key associated with the trusted signer id to decrypt the signature found inside the BEC.

5. Calculate a digest of the BEC.

6. Compare the decrypted signature with the calculated digest. If equal, then the BEC is trustworthy.

4.2.2 Stage 2

4.2.2.1 Processes

Stage 2 of this model is a reference monitor. It monitors and regulates processes in the system. Processes are BEC files which have been executed. One of the executed BEC which constitutes a process is called the program BEC, and it is the subject of monitoring. This is the BEC which was requested to execute. If this BEC also dynamically links and loads BEC files such as function libraries, the dynamically linked BEC files are also executed. The dynamically linked BECs will then adhere to the rules defined for the program BEC. The view on a process in this model is a set of BEC's where one BEC is distinct from all other BEC's.

4.2.2.2 Access Control Rules

Access control rules describe what access a program has on a given operating system object. If there is a rule r for a program BEC p, then all processes initiated by p must adhere to the rule r.

4.2.2.3 Access Control

When a process p requests r access to an operating system object o, the following algorithm is performed:

1. Acquire a reference to the program BEC b which initiated process p.
2. Find an access control rule which matches b, o and r. If no such rule is found, reject the request.
3. If the found rule has been revoked, reject the request.
4. Accept the request.

4.3 Comparisons

The main points addressed by the proposed model are:

origin of BEC files Only BEC files with a known and trusted origin may begin execution.

functionality of BEC files Programs shall be constrained based on its intended functionality.

Tripwire is a tool which can monitor any file system object. It is conceivable that it could be used to monitor BEC files. However, tripwire cannot distinguish between trusted BEC files and ordinary BEC files. There is no concept of origin in the tripwire model. It can only detect changes within a file system. Tripwire does not provide any means for access control, only integrity verification. Another drawback with tripwire is that it is not actively check file modifying operations on each access. There is always a time window where the system is in an unknown state. Therefore tripwire does not really address any of the main points.

ActiveX's Authenticode is used to determine the origin of a BEC file. However, there is no concept of trust in the same manner as in the proposed model. Trust in the Authenticode model is based on identity - *is the signers identity trustworthy?* Trust in the proposed model is based on the signer rather than the identity of the signer - *is the signer trustworthy?*. Based on this reasoning, Authenticode cannot address the origin of BEC files as is needed. Authenticode does not attempt to regulate the functionality of BEC files at all. Therefore the Authenticode model cannot address any of the main points.

Java has the concept of security manager which comes close to the proposed model. However, there is a major difference in what is done when the origin of a BEC file is not determined or is not considered trusted; Java continues execution of the BEC within a "default sandbox". The proposed model does not allow such execution. Therefore, the java security model is not compatible with the proposed model and can therefore not adress all of the main points.

4.4 Formal Model

4.4.1 State Variables

4.4.1.1 System State Σ

The system state is denoted as $\Sigma = \{E, B, D, U, sign, role, origin, id, pkey, deleted\}$

Σ **Components**	
B	The set of available BEC files
D	The set of trusted signer information
E	The set of BEC's in execution
U	The set of users in the system
A	The set of access control rules
P	The set of processes
CA	The set of current accesses
R	The set of access rights
BR	The set of removed BEC files

Table 4.1: System State Variables

4.4.1.2 B Variable

B is a set variable which represents all available BEC files.

4.4.1.3 D Variable

D is a set variable which represents trusted signers. A trusted signer is represented with a triple containing: an *id,* a *public key* and a *revocation flag* - $(id, pkey, flag)$. The id identifies the trusted signer. The public key is the signers public key. The revocation flag tells if the signers trust has been revoked.

4.4.1.4 E Variable

E is a set variable which represents activated BEC's.

4.4.1.5 U Variable

U is a set variable which represents the users in the system.

4.4.1.6 A Variable

A is a set variable which represents access control rules. Each rule is represented with a quadruple containing a *BEC reference,* an *object reference,* a *set of access rights* and a *revocation flag* - $(b, o, r, flag)$. Note that $r \in R$.

The meaning of the quadruple is; An executing instance of a BEC (BEC reference) may access an object (object reference) with a given access right.

The revocation flag indicates that the access rule has been revoked.

4.4.1.7 P Variable

P is a set variable which represents the processes in the system. A process is a tuple of executing BECs - $(e_0, ..., e_n)$. One of the executing BECs within a process tuple, e_0, originates from a program BEC. Remaining executing BECs, $e_1 ... e_n$, are function libraries. Any access to an object from any of the executing BECs must adhere to the access control rule defined for e_0.

4.4.1.8 CA Variable

CA is the set of current accesses, or currently accepted accesses. Each CA element is a triple (p, o, r) where p is a process, o is an object and r is a set of access rights. Its meaning is p has access rights r to object o.

4.4.1.9 R Variable

R is a set variable which represents available access rights for any object in the system. These values are:

Access Right Values	
r	Read access
w	Write access
a	Append access
e	Execute access

Table 4.2: Access Rights

4.4.1.10 BR Variable

BR is a set of removed BEC files. Whenever a BEC is removed from the system, it is moved from the B set to the BR set. The use of this set is important as we shall see in section 4.4.5.3.

4.4.2 Security Attributes

Security Attributes	
$sign(b)$	Signature of BEC b
$role(u)$	The role of the user u.

$origin(e)$	The origin of an executing BEC e which is a BEC. The BEC may originate from either the B set or the BR set.
$id(b)$	The signers id of a BEC
$revoked(d)$	Revocation flag value of a signer information triple
$pkey(d)$	The public key of a signer information triple
$program(p)$	The program BEC file of process p
$access\_right(a)$	The access right of an access control rule

Table 4.3: Security Attributes For System Variables - Stage 1

4.4.3 Functions

Functions in the System	
$digest : b \mapsto dig$	Calculates a digest dig of a BEC b
$decrypt : k \times d \mapsto d'$	Decrypts data d using key k
$activate : b \mapsto e$	Axiomatic function which transforms a BEC b to an executing BEC e.
$lookup : id \mapsto d$	Looks up signer information triples based on the signer id. Lookup is performed over the D set. It is assumed that id is valid.

Table 4.4: Functions

4.4.4 System Invariants

A BEC is properly signed if it follows the rule

$$\textbf{properly\_signed}(b) = b \in B \wedge decrypt(pkey(lookup(id(b))), sign(b)) = digest(b) \tag{4.1}$$

The signature found in a BEC file is a digest of the BEC, encrypted using the private key of the signer. If a decryption of the signature using the signers public key results in a digest that is equal to a recalculated digest, then the following can be concluded;

authenticity The private key was used to encrypt the signature. The private key is only known by the signer, thus the signer and no one else must have encrypted the signature.

integrity The signature is an encrypted digest of the BEC file. If the decryption of the signature yields a digest equal to a digest recomputation, then the BEC file cannot have been changed after it was signed.

Further more, a properly signed BEC must have been signed by a trusted signer who is represented in the D set. The function $lookup()$ ensures that the public key used comes from the D set.

At any given point in time, the following invariant must hold for the system

$$\textbf{System Invariant: } \forall e \in E : properly\_signed(origin(e)) \tag{4.2}$$

The system invariant says that for all executing BECs, the origin of the executing BEC must be properly signed. The origin of an executing BEC is a BEC file.

Another invariant in the system is that no process shall have more accesses than what is allowed by the set of access control rules. Therefore, the CA must adhere to the following invariant:

$$\textbf{Access Invariant: } (p, o, r) \in CA \Rightarrow (program(p), o, r, True \vee False) \in A \tag{4.3}$$

As one can see is that the access invariant does not respect the revocation flag for a particular access control rule. This means that a process will not be affected by a access control rule revocation while it is running. A revocation will only affect new processes which are executed after the revocation. This is to ensure that data integrity is preserved for any process. If a process is holding a access rights to an object and the access control rule has been revoked, the process must be terminated in order to release the access rights. Preferably such a process should be gracefully terminated so that it can preserve data integrity.

4.4.5 State Transition Functions

4.4.5.1 System Variable D

Since the system invariant indirectly depends on the set of trusted signers, the D set, all modifying operations on the set must be constrained. Operations are requested by users. Not all users may be granted requests to modify the set of trusted signers. Only trusted users may do so. In this context, a trusted user is a user which have been appointed the *Security Officer* role. All users have a *role* attribute. The attribute may have one of two values:

SecOfficer The user has the Security Officer role

OrdUser The user has the Ordinary User role

To accept a new signer as a trusted signer, the user issuing the request must have been appointed the security officer role.

proc *accept_signer*(u, i, pk) $\equiv$
  if *role*(u) = *SecOfficer*
    then
      $D \leftarrow D \cup \{(i, pk, False)\}$

Figure 4.4: State Transition Function: $accept\_signer$

Note that the revocation flag is false by default.

To revoke a trusted signer, the user issuing the request must be a security officer.

proc *revoke_signer*(i, u) $\equiv$
  if *role*(u) = *SecOfficer*
    then
      entry = *lookup*(i)
      $D \leftarrow (D - \{$*entry*$\}) \cup \{(i,$ *pkey*(*entry*), *True*$)\}$

Figure 4.5: State Transition Function: $revoke\_signer$

Note that the effect of this state transition rule has only changed the revocation flag in the signer information. Also note that the id i must be a valid signer id.

4.4.5.2 System Variable E

Before a BEC can be activated, it must be properly signed. The trust for the signer of the BEC must also not have been revoked.

$$\begin{array}{l}\underline{\textbf{proc}}\ \mathit{add\_to\_execution}(b)\ \equiv \\ \quad \underline{\textbf{if}}\ \mathit{properly\_signed}(b) \wedge \neg \mathit{revoked}(\mathit{lookup}(\mathit{id}(b))) \\ \quad\quad \textbf{then} \\ \quad\quad\quad E \leftarrow E \cup \{\mathit{activate}(b)\}\end{array}$$

Figure 4.6: State Transition Function: $add\_to\_execution$

To remove an active BEC from memory, there are no constraints. Removing an active BEC from memory could not break the system invariant.

4.4.5.3 System Variable B

When removing a BEC it must be done so that the system invariant is preserved. The only way to do so is by removing the BEC from the B set and put it in the RB set (the set of deleted BEC files). Also, when adding a BEC file to the system, there may not be a BEC file in both B and BR, because this would violate the semantics of these sets. Therefore there is a constraint on adding and removing BEC files to and from the B set.

$$\begin{array}{l}\underline{\textbf{proc}}\ \mathit{add\_BEC}(b)\ \equiv \\ \quad B \leftarrow B \cup \{b\} \\ \quad BR \leftarrow BR - \{b\}\end{array}$$

Figure 4.7: State Transition Function: add_BEC

$$\begin{array}{l}\underline{\textbf{proc}}\ \mathit{rem\_BEC}(b)\ \equiv \\ \quad B \leftarrow B - \{b\} \\ \quad BR \leftarrow BR \cup \{b\}\end{array}$$

Figure 4.8: State Transition Function: rem_BEC

4.4.5.4 System Variable U

Ordinary users shall not be able to escalate its privileges by changing its role. Therefore only security officers may change the role attribute of a user.

$$\begin{array}{l}\underline{\textbf{proc}}\ \mathit{set\_role}(u, targ\_u, r)\ \equiv \\ \quad \underline{\textbf{if}}\ \mathit{role}(u) = \mathit{SecOfficer} \\ \quad\quad \textbf{then} \\ \quad\quad\quad \mathit{role}(targ\_u) \leftarrow r\end{array}$$

Figure 4.9: State Transition Function: set_role

4.4.5.5 System Variable CA

When a process requests access to an object, the following state transition rule must be adhered:

proc *get_access*(p, o, r) $\equiv$
  if $(\textit{program}(b), o, r, \textit{False}) \in A)$
    then
      $CA \leftarrow CA \cup (p, o, r)$

Figure 4.10: State Transition Function: get_access

This rule protects the access invariant and it also considers whether the access rule has been revoked or not. If the rule is satisfied, then the access rights r on object o for process p is added to the set of current access rights CA.

4.4.5.6 System Variable A

Access control rules may only be manipulated by the security officer.

proc *add_rule*(u, b, o, r) $\equiv$
  if $\textit{role}(u) = \textit{SecOfficer}$
    then
      $A \leftarrow A \cup \{(b, o, r, \textit{False})\}$

Figure 4.11: System Transition Function: add_rule

Note that the revocation flag is initially false.

proc *revoke_rule*(u, b, o, r) $\equiv$
  if $\textit{role}(u) = \textit{SecOfficer} \wedge (b, o, r, \textit{False}) \in A$
    then
      $A \leftarrow (A - \{(b, o, r, False)\}) \cup \{(b, o, r, True)\}$

Figure 4.12: System Transition Function: $revoke\_rule$

Note that the effect of this state transition rule is that the access control rule has only changed its revocation flag. This means that currently executing processes are not affected, only processes started after this transition.

4.4.6 Initial Secure State Σ_0

The initial state, denoted Σ_0, must be a secure state - i.e., it must fulfill the invariants.

Variable	Value
P	$\emptyset$
B	$\emptyset$
D	$\emptyset$
E	$\emptyset$
U	$\{u\}, role(u) = \textit{SecOfficer}$
A	$\emptyset$
CA	$\emptyset$
R	$read, write, execute$

BR	$\emptyset$

Table 4.5: Initial Secure System State Σ_0

Of course, if the invariants are fulfilled, then all succeeding states are also secure.

Chapter 5

Implementation Issues

5.1 Signature Cryptography

Digital signatures can be devised in primarily two ways;

- by the use of an explicit digital signature scheme
- by the use of a public-key encryption scheme combined with a secure hash algorithm

In [BS96] several schemes for digital signatures are presented. These are listed in table 5.1. One of the listed scheme will be selected for the implementation of the proposed model.

Public-Key Crypto Schemes	
Scheme	**Patent**
RSA	no
Pohlig-Hellman	yes
Rabin	no
ElGamal	no
McEliece	no
LUC	yes
Public-Key Signature Schemes	
DSA	yes - royalty free
GOST R 34.10-94	no
Ong-Schnorr-Shamir	yes
ESIGN	yes
ElGamal	no

Table 5.1: Digital Signature and Public-Key Crypto Schemes

Patented technology should not, in the authors oppinion, be used in academic research for mainly one reason: research results should be available and usable to and by anyone pursuing knowledge. Based on this statement, all patented algorithms listed above are disregarded for use in an implementation of this model.

5.1.1 Encryption Schemes

RSA is a public-key encryption scheme developed by Rivest, Shamir, Adleman [RSA78], hence the name RSA. This scheme used to be protected by a patent in the USA, but expired the 20th of september 2000.

The RSA scheme includes algorithms for creation of keys, decryption and encryption of data. What is interesting for the implementation of the proposed model is the decryption algorithm which is used during authentication of BEC files.

The security of the scheme is based on the problem of factoring large numbers into prime factors. The problem of factoring large numbers into primes is a hard problem and is thus not computationally feasible. The large numbers referred to are the three components in the scheme; public key, private key and a modulo. The modulo component is the component which determines the hardness of the factoring problem. This modulo component is generally a number of size 2^{512} and larger. This number determines the key-length in bits for the scheme. It is the exponent, 512 in this case, which is called the key-length.

The security of the RSA scheme depends on the speed of todays computers. It has been shown that the RSA scheme using key-length of less than 768 bits, is not secure [RSA] because it can be broken by very fast computers. As computers become faster and faster, larger problems can be solved within shorter times. In essence, the scheme is as secure as the key-lengths used in combination with available computing power. Since the scheme does not put an upper limit on the key-length, the scheme scales well as computing power increase.

The scheme has received great acceptance, and is one of the most widely used public-key cryptography scheme, it is *the de facto standard* [RSA]. It has undergone major scrutiny since it was first introduced. So far, breaking the RSA scheme is not feasible for strong keys. Strong keys are such keys that fulfill certain criterias listed in [BS96], [Moo92]. Considering that the weaknesses of RSA are well known, it is relatively fast compared to other public-key schemes [BS96], and that RSA scales, RSA is a strong candidate for the implementation of the proposed model.

A mathematical description on the RSA scheme is given in [BS96] pp 466 - 474, [Gar00] pp 161 - 171 as well as the original paper [RSA78].

Rabin [Rab79] is a public-key cryptography scheme which is based on the problem of finding square roots modulo of a composite number. According to [BS96], this problem is equivalent to factoring (RSA problem).

Just like RSA, the scheme includes algorithms for creation of keys (public and private) as well as encryption and decryption of messages. What is interesting for the implementation of the proposed model is the decryption algorithm which is used during authentication of BEC files.

This scheme has one shortcoming in the respect on how decryption works. The decryption algorithm gives four distinct messages, where only one is the original message. As stated in [BS96], this becomes a problem if the message is not in a structured format such as English text. One solution to the problem is to add a *message marker* to each message which is known by the decrypting party. The scheme has since it was published, been redefined [Wil80], [Moo85], [Moo86], so that the “four messages shortcoming” has been solved. However, as pointed out in [BS96], these redefinitions have made the scheme vulnerable by chosen ciphertext attacks. Therefore Rabin is not a very strong candidate for the implementation of the proposed model.

A mathematical description of Rabin is given in [BS96] pp 475 as well as in the original paper [Rab79].

ElGamal is a scheme which can do both encryption and digital signatures without the need of hashing. The scheme [ElG86] is based on the problem of calculating discrete logarithms in a finite field.

The scheme specifies how keys should be generated and how encryption, decryption and signing is performed. What is interesting for the implementation of the proposed model is the decryption algorithm which is used during authentication of BEC files.

When using the ElGamal scheme for signatures and encryption/decryption, signatures and ciphertext expand to twice the size of the original message due to the nature of the scheme. This adds much overhead to storage needed for encrypted messages and signatures.

When messages are signed, a random value k is picked. This k makes the scheme work. However, it is also the weakness of the scheme. The value k may never be reused. If two messages have been signed using the same k, it is possible to deduce the private key used for ciphertext. If the value k is recovered by a third party, then the private key may be recovered. Thus, k must be chosen carefully and carefully discarded.

Given that keys are generated carefully, ElGamal is a strong candidate for the implementation of the proposed model.

A mathematical description of ElGamal is available in [BS96] pp 476 - 477 as well as the original paper [ElG86].

McEliece is a public-key cryptography scheme which is based on algebraic coding theory [McE78].

Despite the fact that the scheme has withstood cryptanalytic examination and is quite fast compared to other successful schemes such as RSA, it has not gained acceptance. This is mainly because of the fact that encrypted messages are double the size of the original messages, and that the public key grows very large (2^{19} bits long!). Eventhough this scheme is efficient in terms of time, it has major drawbacks in storage efficiency which renders the scheme as a weak candidate for the implementation of the proposed model.

A mathematical description of McEliece is available in [BS96] pp 479 as well as the original paper [McE78].

5.1.2 Digital Signature Schemes

DSA is a scheme explicitly tailored for digital signatures [NIS91]. DSA is an acronym for *Digital Signature Algorithm*. It is patented royalty free by NIST in the USA. Because DSA is royalty free, it has not been disregarded.

DSA is not an atomic algorithm as the name implies, it is composed by variants of Schnorr and ElGamal signature algorithms and SHA1 hashing.

DSA has been highly criticized since it was published by NIST. The main arguments against DSA is that signature verification is slower than RSA (by a magnitude 10-40), the key sizes specified by the standard are too small [BS96] to guarantee security for the future. Key sizes are not scalable either, so this scheme does not scale well in the future, unless it is revised. Therefore, this scheme makes it a weak candidate for the implementation of the proposed model.

A mathematical description of DSA is available in [BS96] pp 486 - 487 as well as the original paper [NIS91].

GOST R 34.10-94 is the digital signature algorithm defined by the former USSR [Fed94a].

GOST is according to [BS96] very similar to DSA. It is not easy to read the standards since they are published in russian. Given that the source of information for this scheme is very limited, this scheme is a weak candidate for the implementation of the proposed model.

A mathematical description of DSA is available in [BS96] pp 495 - 496 as well as the original paper [Fed94a].

5.1.3 Summary of Signature Algorithms

McEliece suffers from great storage expansions, it is not feasible for an implementation of the proposed model. The data base of n trusted signers would be approximately $n \cdot \frac{2^{19}}{8} = n \cdot 2^{16}$ bytes large! This is unmanagable in terms of primary and secondary memory capacity.

Rabin in its original form has the "four messages shortcoming" thus it is less managable. Redefinitions of Rabin has led to weaknesses against chosen ciphertext attacks which is not managable from a security point of view. Therefore Rabin is not chosen for an implementation.

DSA is a scheme which does not scale, it has an upper limit on key size. It may be secure today, but may not necessarily be secure in the future. It is also very slow compared to other solutions, e.g. RSA + hashing. Therefore DSA is not chosen for an implementation.

GOST R 34.10-94 is a scheme which little is known about. The algorithms used are well known, but it has not been scrutinized like other algorithms, e.g. ElGamal and RSA. Therefore GOST R 34.10-94 is not chosen for an implementation.

ElGamal and RSA are the algorithms remaining to chose from. Both algorithms are well known, and have been scrutinized. Many of the schemes weaknesses have been revealed and can thus be avoided. The choice is not easy, both schemes scales well into the future in terms of security. However, ElGamal seems to be the least efficient one - see [BS96] table 19.4 and 19.7. Therefore, based on efficiency, RSA is chosen for the implementation.

Since RSA is chosen for the implementation, a hashing algorithm is needed since "pure" RSA cannot be used to produce digital signatures, see section 5.1.4.

5.1.4 Hashing Algorithms

The hash algorithm which this model shall use in an implementation must fulfill two criterias; it has to be *efficient* and *secure*. The efficiency of an algorithm is measured by the time needed to compute a hash value over a given data size. In this thesis, several hash algorithms have been evaluated based on efficiency. No security evaluations have been performed, since that is beyond the scope of this thesis.

The evaluated algorithms are as follows:

SHA-1 Secure Hash Algorithm 1 [NIS92]

GOST GOST R 34.11-94 [Fed94b]

RIPEMD-160 Hashing algorithm developed in the EU RIPE framework [RAC92]

HAVAL 3/160 A one-way hashing algorithm with variable length of output, using 3 rounds and 160 bit digest output [ZPS93]

HAVAL 4/160 Using 4 rounds and 160 bit digest output

HAVAL 5/160 Using 5 rounds and 160 bit digest output

These algorithms were chosen primarily based on available implementations. The used implementations of these algorithms come from a software package called *Mhash* [MS].

The evaluation tests were conducted as follows:

- Each algorithm was tested 20 times. The experiments were conducted on a machine which only ran the absolutely necessary processes. These processes are idle processes and only request CPU time when a user is interacting with the computer. Since no user had access to the machine during the tests, there was a minimum of interference with the testing process.
- Each algorithm test computed a hash value upon several different data sizes. The test algorithm is described in figure 5.2.
- The data sizes were [1000, 2500, 5000, 7500, 10000, 25000, 50000, ..., 1000000000] bytes. The range may seem quite large as the maximum data size is one billion bytes. The size of BEC files on a Linux system typically range from around 3 kilobytes upto 10 megabytes. The reasoning behind the large test sizes is to put a bigger perspective on the algorithms. It may also predict the scalability of the algorithms as BEC file sizes may increase in the future.

The results of the conducted experiment are somewhat interesting. The diagram in figure 5.1 shows a non-linear curve for each algorithm for data sizes less than approximately 100 kilobytes, and linear for data sizes larger than 100 kilobytes. It was anticipated that the graphs would be linear for all data sizes, thus this result is a bit surprising. It seems as if the algorithms are more effective for small data sizes than large data sizes. The reasons for this lies probably in the hardware architecture such as CPU caching. It is possible that this curve depends on the buffer size used in the testing algorithm presented in figure 5.2. However, since the curve does not affect the effectiveness in a negative way, no further investigations were pursued.

The diagram in figure 5.1 shows that RIPEMD-160 is the most efficient hashing algorithm. For one gigabyte of data, RIPEMD-160 has a mean time of approximately 53.9 seconds while the runner up algorithm, HAVAL 3/160 has a mean time of approximately 71.0 seconds which is approximately 31.7 percent slower. Based on these experiments and observations, RIPEMD-160 should be used as the hashing algorithm for the implementation of this model.

5.2 BEC File Format

5.2.1 BEC files

The layout of a BEC file is very straightforward; signature information is simply appended at the end of the file, see figure 5.3. The major BEC file formats used on the Linux platform are the ELF - *Executable and Linkable Format* [TIS93], [Hau98] and the *a.out* file format. These file formats do accept arbitrary data at the end of the file. These types of files are parsed based on the syntactic rules defined for the file type. When the parser is finished, it has parsed the file up to its logical end, it pays no respect to arbitrary data at the end.

By simply appending the signature information to a BEC, the internal structure of the BEC file does not have to be addressed. This makes it easy to extend the system whenever a new BEC file format becomes available. In special cases where a new BEC file format does not accept arbitrary data appended, then the system needs to be extended to care of the format. In such cases, the signature information must be carried within some syntactically and semantically correct construct of the BEC file format.

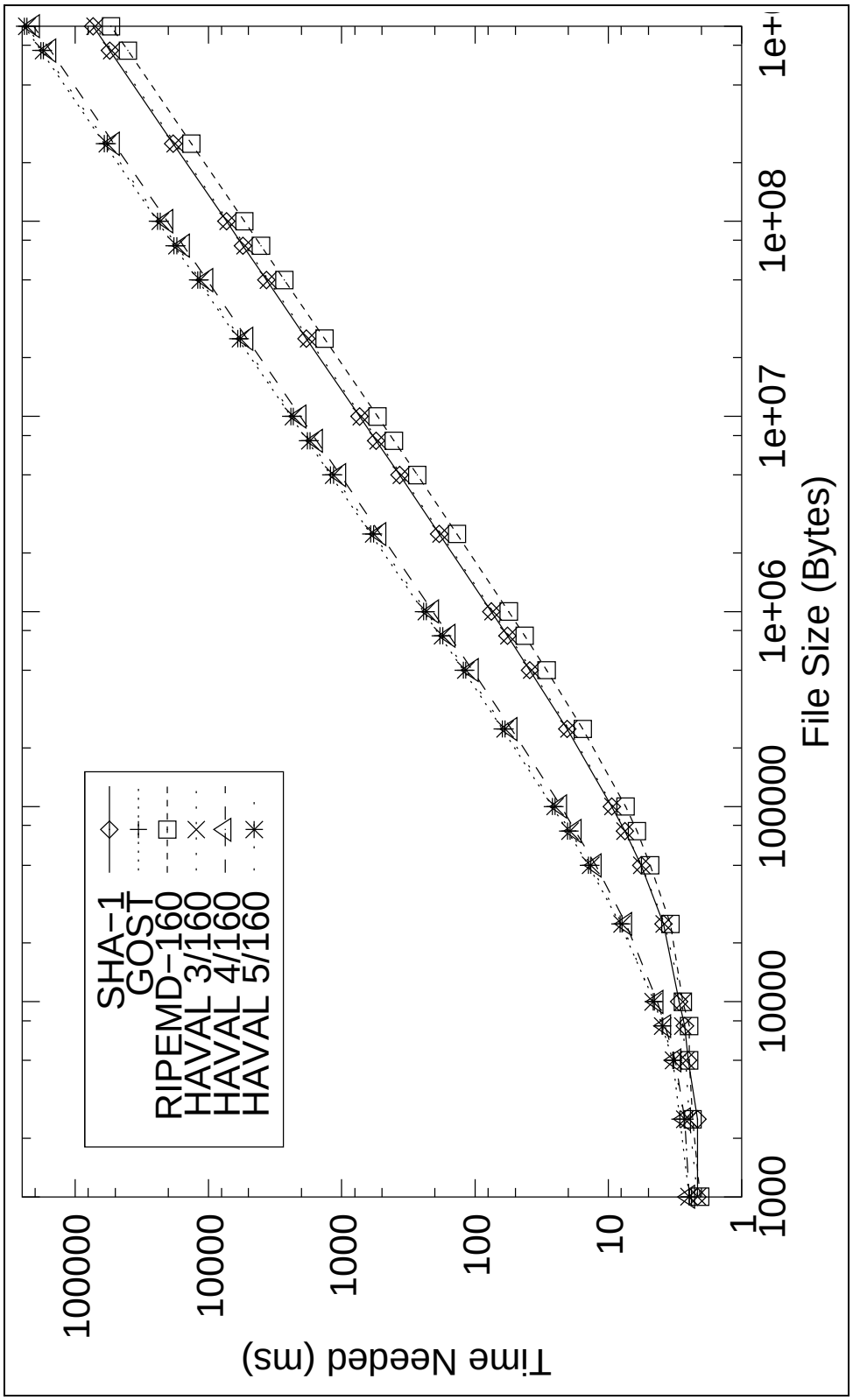


Figure 5.1: Hash Algorithm Efficiency Chart / Mean Values

```
proc test_hash(size) ≡
  buffer = allocate_bytes(10000);                    Allocate a scratchpad
  hash_init();                                       Initialize hash function
  sum = 0;
  while sum < size do                                Hash buffer until size bytes have been processed
      length = MIN(10000, size − sum);               Test for boundary condition
      hash_accumulate(buffer, length);               Accumulate hash
      sum = sum + length;
  od
end                                                  size bytes have now been hashed
```

Figure 5.2: Generic Algorithm Test Procedure

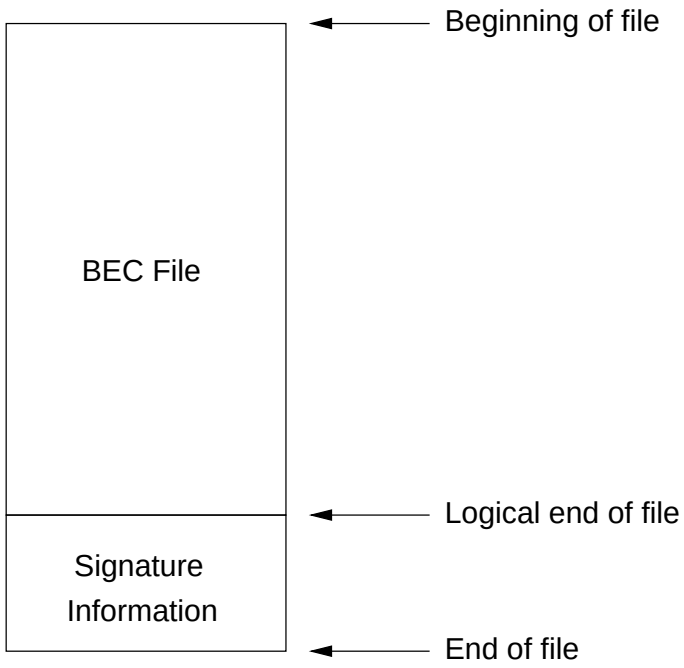


Figure 5.3: BEC File Format Layout

5.3 Signatures and Key Handling

5.3.1 Signatures

Signatures shall be computed over the whole BEC file. The signature is then appended to the file.

5.3.2 Key Handling

The data base of trusted signers contains identifying information and the public key associated with the signer. X.509 version 3 [CCI89], [Sta00] pp 101-110, [Pfl97] pp 135-140, is to be used for holding this information. From the models point of view, the tuple $(id, pubkey)$ is an X.509 certificate.

The main motivation for this choice is:

Suitable X.509 version 3 can hold enough information which the model requires

Open Standard X.509 is an open standard

Certificate Authorities are trusted organizations which issue certificates

Revocation It is possible to use the revocation mechanism found in the X.509 standard.

Open Standard is clearly defined in standard texts and has been scrutinized. It does not leak information which could jeopardize the security.

Certificate Authority When the organization receives a certificate from a signer and the certificate was issued by a certificate authority, then the certificate is correct.

Revocation of a certificate is possible if the certificate has become invalid after it has been issued for some reason. It is thus possible to see if a certificate has been revoked since it was issued and it is possible to remove the signer associated with the certificate from the data base of trusted signers. This revocation shall not be confused with the revocation of a trusted signer. The revocation of a certificate is equal to not trusting the identity of the signer while revocation of a trusted signer is equal to not trusting the signer. In mathematical terms: $revocation(certificate) \Rightarrow revocation(trustedsigner)$.

5.4 System Environment

5.4.1 Operating System Objects

This section lists and describes the operating system objects which programs may manipulate. These operating system objects were found after reviewing the operating system directly and after research in [Fis01].

The operating system objects which the implementation shall regard are as follows:

Object, Target and Access Table		
Target Class	**Object Ref**	**Access Rights**
Clock	-	Write
Device	type:major:minor	Create, Remove, Read, Write
Directory	/a/directory	Read, Write, Execute
File	/a/file	Read, Write, Execute
FS Object Attribute	/an/fs/obj/path	Read, Write
FS Mount Point	/dev/dev:/mnt	Add, Remove
Kernel Module	/a/module	Add, Remove
Process	/a/program[:sig]	Signal, Trace
Socket IPC	protocol:address	Connect, Listen, Read, Write
System	-	Reboot, Halt, HWAccess
SysV IPC	key	Create, Remove, Read, Write
SysV IPC Attribute	key	Write
Swap	/dev/dev:prio	Add, Remove

5.4.1.1 Clock

The system clock is a resource which many processes in the system may depend on. Therefore modifications of the system clock must be restricted.

A program may be given the following access rights:

Write The program may modify the system clock

5.4.1.2 Device

A device is a communication channel with a peripheral unit or some kernel service. A device is named by two numbers; the major and minor number. Since devices may be communication channels with hard disks and other sensitive peripherals, modification of devices must be protected.

A program can be given the following device access rights:

Create The program may create a device of given type, major and minor number

Remove The program may remove a device of given type, major and minor number

Read The program may read from a device of given type, major and minor number

Write The program may write to a device of given type, major and minor number

5.4.1.3 Directory

A directory object is named using its absolute path which is unique for any file system object. A program can be given the following directory access rights:

Read The program may search the directory

Write The program may add or remove entries in the directory

Execute The program may enter the directory

The access right semantics mimic those which already exist for the user access control mechanism.

5.4.1.4 File

A file object is named using its absolute path which is unique for any file system object. A program can be given the following file access rights:

Read The program may read the contents of the file */a/file*

Write The program may modify the contents of the file */a/file*

Execute The program may execute the file */a/file* as a program file

The access right semantics mimic those which already exist for the use access control mechanism.

5.4.1.5 File System Object Attribute

File system object attributes are meta information. They describe any file system object. Such information may be ownership, time of creation, user access control information etc.

The reason for separating the access control on file system object contents and attributes is that there are scenarios where a program is allowed to modify the contents of a file, but may not change the user access control information. Such a program could be an user editor. It may need to update the database of user information, but it may not change attributes of the file such as file owner.

A program can be given the following file access rights:

Read The program may read the file system object attributes for object */an/fs/obj/path*

Write The program may modify the file system object attributes for object */an/fs/obj/path*

5.4.1.6 File System Mount Point

The UNIX file system allows "merging" of file systems into a single file system. When a file system is merged into the file system tree, it is *mounted* onto a mount point. A mount point in a file system is a node in the file system tree. When mounted, the tree is extended by the tree structure provided by the mounted file system. Figure 5.4 shows how file systems is merged when a new file system is mounted on a mount point.

A file system mount point is referred to as a file system device and a mount point. A file system device may be a UNIX block device or a file containing a file system image. The mount point is a path to where the file system tree is to be merged. A program may be given the following access rights:

Add The program may mount the file system tree contained in device/file */dev/dev* on the mount point */mnt*

Remove The program may unmount the file system mounted on the mount point */mnt* given that the mounted file system is contained in */dev/dev*

5.4.1.7 Kernel Module

A kernel module is a Linux specific feature. It is based on the "plugin" concept. Certain drivers such as the floppy driver need not be loaded all the time. The floppy is not used very often.

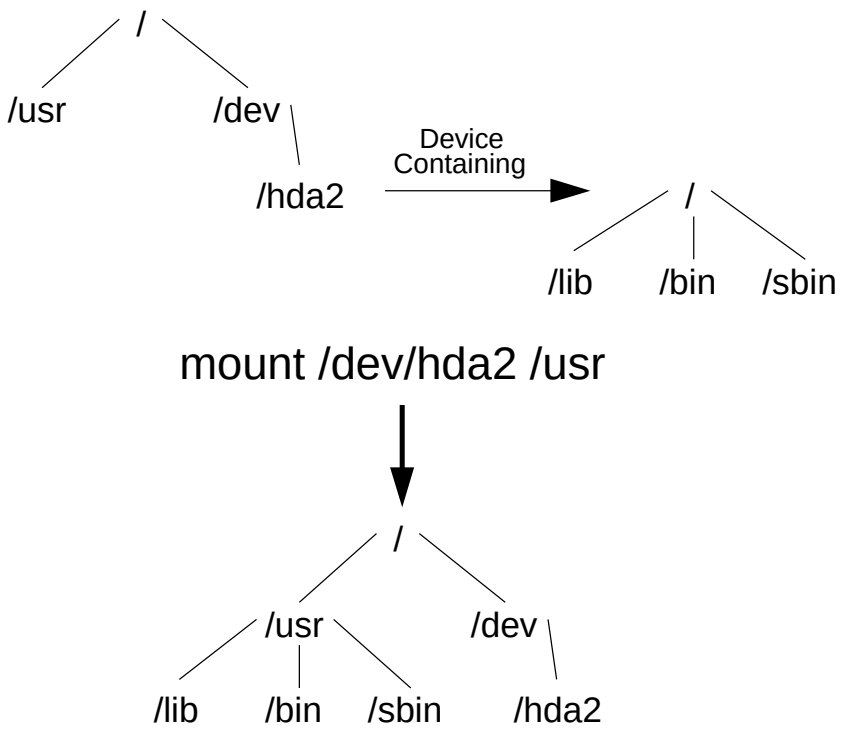


Figure 5.4: Mounting Device /dev/hda2 Onto Mount Point /usr

When not used, there is no reason to keep the floppy driver in memory all the time using valuable memory. The kernel module feature allows the system to be optimized at all times regarding memory. As a short summary, a kernel module can be thought of as a program running in kernel space and is invoked by the kernel or user space programs depending on its implementation.

The name of the file which contains the kernel module is the name of the kernel module.

A program may be given the following access rights:

Add The program may add the kernel module contained in */a/module*

Remove The program may remove the kernel which was loaded from the file */a/module*

5.4.1.8 Process

Process signals are notification signals sent from either the kernel or other processes. What is unique for a process signal is its destination and signal name. The destination is referred to as the program name which is a file path. The signal name tells the semantics of the signal. It may be a TERMINATE signal which causes the process to terminate, or it may be a CHILD signal which notifies the process that one of its child processes has terminated.

A process may connect to another process p, tracing the different system calls performed by p. This is useful in debugging purposes, but may provide a security problem. Therefore tracing processes needs to be regulated.

A program may be given the following access rights:

Signal The program is allowed to send a signal named *sig* to any process originating from program file */a/program*

Trace The program is allowed to trace system calls made by any process originating from program file */a/program*

5.4.1.9 Socket

Sockets are means of communication with other processes. These processes may be on the same host machine (UNIX sockets) or they be on a different host machine (TCP, UDP, Raw IP etc.). Sockets are named by a protocol, e.g. TCP, and an address, e.g. myhost.com:3454.

When establishing a socket connection, one of two access rights is needed. If the program tries to initiate a connection it needs the right to *connect*. If the program is passive and awaits an incoming connection call, it needs the right to *listen* for the incoming call.

Connectionless protocols such as UDP do not need an explicit connection, therefore reads and writes to a socket must also be monitored.

A program may be given the following access rights:

Connect The program is allowed to connection to a particular *address* using a particular *protocol*

Listen The program is allowed to listen for an incoming connection call from a particular *address* using a particular *protocol*

Read The program is allowed to receive data eminating from *address* using a particular *protocol*

Write The program is allowed to write data to *address* using a particular *protocol*

5.4.1.10 System

The system object is not really an object. It represents an interface to the system rather.

A program may be given the following access rights:

Reboot The program is allowed to reboot the system

Halt The program is allowed to halt the system

HWAccess The program may request access for hardware resources

5.4.1.11 SysV IPC

SysV IPC objects such as memory segments, messages and semaphores are initiated by some program. Such objects are created by specifying a *key* which uniquely identifies the object system wide. When creating such an object for the first time, the object is marked as owned by the process requesting the object. The object is also given a set of access rights. These rights are the same as for files, read, write or execute for user, group and others. Subsequent requests for creation of objects with the same key yields a handle for the same object, if the process requesting the object matches the access rights specified for the object.

A program may be given the following access rights:

Create The program may create the SysV IPC object identified by *key*

Remove The program may remove the SysV IPC object identified by *key*

Read The program may read from SysV IPC objects identified by *key*

Write The program may write to SysV IPC objects identified by *key*

5.4.1.12 SysV IPC Attribute

SysV IPC objects as mentioned in section 5.4.1.11, has attributes associated with itself.

Write The program may write attributes to SysV IPC objects identified by *key*

5.4.1.13 Swap

UNIX-like operating system uses a special file system for swapping and paging. These file systems are added by referring to the device which holds the file system. A priority is also given to each swap file system, so that the load can be balanced over file systems with different speed and size capabilities.

A program may be given the following access rights:

Add Adds a swap file system located in device */dev/dev* to the kernel. Associated priority is *prio*

Remove Removes a swap file system located in device */dev/dev*

5.4.2 Implementation Environments

5.4.2.1 The Choice of Operating System Environment

Linux The operating system used for the implementation is Linux. Linux was started by its creator Linus Torvalds in the early 1990's. Linux has since then grown into a fully functional POSIX compliant UNIX flavoured operating system.

What is unique with Linux as compared to other operating systems such as Solaris, Microsoft Windows etc., is that it is totally free. It is not only available free of charge, it is also free in a more strict definition as per the GNU Public License which Linux is licensed under. This license says that if anyone makes any modification of Linux, and redistributes Linux with these changes, then the source code must also be freely available. This ensures that any work done using the Linux operating system, will remain free and open for all.

There exists a security framework for Linux, also known as RSBAC Linux. This framework is discussed in section 5.4.2.5.

BSD Another viable operating system would be one of the free BSD variants. However, these are licensed under the Berkeley Software Distribution License which is not as strict as the GNU Public License. The BSD license allows anybody to make modifications to the operating system, and make it a non-open, non-free of charge version. There are also, to the authors knowledge, no known security framework available for any of the free BSD variants.

GNU/Hurd A third operating system is the GNU/Hurd operating system. It is a micro kernel based operating system which is in its infancy. Since it is a micro kernel based operating system, it is very modularized and thus very easy to work with. Modules, also known as *servers*, can be inserted and removed in the operating system at runtime. These servers execute in user space, thus a crash does not bring down the entire operating system, and debugging is a lot easier. This is ideal for a development environment. But as mentioned, GNU/Hurd is only in its infancy, and is therefore not as stable as Linux.

5.4.2.2 Standard Linux Kernel Space Implementation - General Considerations

Security Officer Standard Linux does not provide a security officer user, it only provides the super user (also known as root) and ordinary users. Therefore the security officer user must be implemented. Linux must be modified so that the data base of trusted signers may not be accessible by any other user for modification purposes.

The security officer user must also be a very restricted user. It shall not be possible to login as security officer in any other way than via the console - i.e., the security officer may only log in physically at the computer. This means that the user is physically secured. Also, it must not be possible to change the security officers password as a non-security officer such as root.

Kernel Execution Since the protection mechanisms reside in the kernel, the correct kernel must be executed at boot time. The installation of a new kernel which does not provide these protection mechanisms, must be prohibited. Therefore only the security officer may install a new kernel or modify a running kernel in any way.

5.4.2.3 Standard Linux Kernel Space Implementation for Stage 1

In order to prevent loading untrusted BEC's into memory, two system calls need to be addressed:

execve() Called for program BEC's

mmap() Called for arbitrary files which are to be mapped into memory

execve() The system call in the Linux operating system which loads and executes program BEC's is the *execve()* call. To test a programs authenticity, this system call is intercepted for testing the signature. If the program BEC is authentic and may be executed according to the database of trusted signers, then execve() may continue its execution. Otherwise the calling process is signalled "permission denied".

mmap() When function libraries and other non-program BEC files are loaded into memory, there is no single *load_bec()* system call which takes care of this. In fact, function libraries for instance, are not taken care of in kernel space. When the kernel executes a dynamically linked program BEC, it looks up a user space interpreter. For example programs using the ELF file format, a special ELF interpreter is invoked. The interpreters responsibility is to map the function libraries needed by the program into memory. The actual loading of the function library (and any other non-program BEC files) is done using the system call *mmap()*. mmap() maps a file into memory. This means that the file is accessed via memory pages. These pages are given access rights. The access rights are *read*, *write* and *execute*. In the BEC file case, the files are mapped as with the execute access right. If there are any attempts to map a file into memory with the access right execute, then it has to be tested for authenticity. In this case, mmap() is intercepted. If the BEC is properly signed, then mmap() is allowed to continue execution as normal. If not, the calling process is signalled "permission denied".

Data Base of Trusted Signers To implement the data base of trusted signers, the security officer user is needed.

5.4.2.4 Standard Linux Kernel Space Implementation for Stage 2

The operating system object as defined in section 5.4.1 can be manipulated via a limited subset of system calls. These system calls must be monitored and tested against the access control rules. These system calls mapping against the access requests are defined in table 5.2.

Object	Access Request	System Calls
Clock	Write	stime
Device	Create	mknod
Device	Remove	unlink
Device	Read	open, read
Device	Write	open, write
Directory	Read	open, read, getdents, opendir, readdir
Directory	Write	open, write, mkdir, creat, unlink, mknod, link
Directory	Execute	chdir, fchdir
File	Read	open, read
File	Write	open, write, truncate, ftruncate
File	Execute	execve, mmap
FS Object Attribute	Read	stat, fstat
FS Object Attribute	Write	chown, fchown, lchown, chmod, rename
FS Mount Point	Add	mount
FS Mount Point	Remove	umount
Kernel Module	Add	create_module
Kernel Module	Remove	delete_module
Process	Signal	kill
Process	Trace	ptrace
Socket IPC	Connect	socket, connect
Socket IPC	Listen	bind
Socket IPC	Read	read, recvfrom, recvmsg, recv
Socket IPC	Write	write, sendto, sendmsg, send
System	Reboot	reboot
System	HWAccess	iopl, ioperm
System	Halt	reboot
SysV IPC	Create	shmget, semget, msgget
SysV IPC	Remove	shmctl, semctl, msgctl
SysV IPC	Read	semop, msgrcv, shmat
SysV IPC	Write	semop, msgsnd, shmat
SysV IPC Attribute	Write	semctl, shmctl, msgctl
Swap	Add	swapon
Swap	Remove	swapoff

Table 5.2: Access Request - System Call Mapping

Access Control Rules Data Base This data base must also be protected so that only the security officer may modify it.

5.4.2.5 RSBAC Linux Implementation - General Considerations

RSBAC for Linux is *Rule Set Based Access Control for Linux* [Ott97], [OF00], [Ott]. It is a framework for implementing access control security models. It is based on the *Generalized Framework for Access Control* by Abrams and LaPadula et al [Abr90].

RSBAC Linux has several security models implemented such as Bell-La Padula MAC (Mandatory Access Control) [BL73], FC (Functional Control), RC (Role Compability), FF (File Flags), Simone Fischer-Hübner PM (Privacy Model), SIM (Security Information Modification) and more. The system can be configured to use one or more of these models. The models can coexist alongside each other. A great benefit by using this framework for the implementation of the proposed model is that it can coexist with these other models - i.e. any subset of the implemented models can function in parallel independently of each other. This allows for a very secure system. This framework is thus quite capable. RSBAC Linux also has the security officer user and the infrastructure to support it.

RSBAC is divided into three major components; *Access Enforcement Control Facility* (AEF), *Access Decision Control Facility* (ADF) and *Access Control Information* (ACI). All system call requests are dispatched through the AEF. The AEF will then query the ADF for a decision whether the system call request should be granted or not. The ADF will in turn test the request against all models. If all models grant the request (or the model does not care about the system call), then the system call request is granted, and the calling process may access the object which the system call refers to. If any model does not grant the system call request, then a "permission denied" signal is sent back to the calling process. Figure 5.5 gives a schematic overview of RSBAC.

AEF and possibly ADF makes use of ACI for storing persistant security attributes. The ACI is a set of files that resides on a file system. The ACI is protected by the kernel, so that unauthorized access is denied. It may only be manipulated by AEF, ADF and possibly system calls provided by the RSBAC framework.

Future work is to implement EAC as two ADF modules. One ADF module will handle access control rules, and the other other module will handle verification of signed BEC's.

The communication between the AEF and the ADF uses a well defined protocol. The protocol is based on a function call with a specified set of parameters. This function call is then dispatched to the various ADF policy modules for a decision. The function is named *rsbac-adf-request*. The parameters passed to it are as defined in table 5.3.

Parameter	Meaning
Request Type	Informs the ADF module what the request is. It may for instance be *execute* when the ADF modules are asked for a decision on execution of a program.
Process ID	All requests originates from a system call issued by a process. This parameter specifies the calling process.
Target Type	A system call always affect a target operating system object. This parameter specifies the type of the target, i.e. *file*.
Target ID	This parameter is a handle or reference to the target so that all properties of the target object may be examined by the deciding ADF module.

Attribute Type	Targets have a set of security attributes associated. These attributes may be modified by a system call. If such an attribute is about to change with as a result of the called system call, the type of the attribute is passed with this parameter.
Attribute Value	The new attribute value set by the calling system call is passed with this parameter.

Table 5.3: rsbac-adf-request parameters

RSBAC Linux defines a number of request and target types. These can be found in table 5.5. The table consists of two columns. The first column describes the request type. Also in column one, the target types which the target request applies to are listed. These target requests are described in table 5.4. The second column describes the meaning of the target request applied to the target types.

Target Type	**Description**
FILE	Files, including device files
DIR	Directories
DEV	Devices
IPC	Interprocess Communication; SysV IPC and sockets
SCD	System Control Data; Objects affecting the whole system. I.e. system clock
USER	Users
PROCESS	Processes
NONE	No target

Table 5.4: RSBAC Target Types

Request Type and Valid Targets	**Description**
ADD-TO-KERNEL (NONE)	Add a kernel module to the kernel
ALTER (IPC)	Modify control data of an ipc-type object
APPEND-OPEN (FILE, IPC)	Open a file or ipc-object to append data
CHANGE-GROUP (PROCESS, FILE, IPC, DIR)	Change the group of a file, directory, ipc-object or process
CHANGE-OWNER (PROCESS, FILE, IPC, DIR)	Change the owner of a file, directory, ipc-object or process
CHDIR (DIR)	Change current directory
CLONE (PROCESS)	Clone a process
CLOSE (FILE, DIR, IPC)	Close an open file, directory or ipc-object
CREATE (FILE, DIR, SCD, IPC)	Create a new object
DELETE (FILE, DIR, SCD, IPC)	Delete an object
EXECUTE (FILE)	Execute a file
GET-PERMISSION-DATA (FILE, DIR, SCD, IPC)	Read discretionary access permissions from the an object
GET-STATUS-DATA (FILE, DIR, SCD, IPC)	Read object status data
LINK-HARD (FILE)	Create a hard link (alias) for the file
MODIFY-ACCESS (FILE, DIR)	Modify access information for the object

MODIFY-ATTRIBUTE (USER, PROCESS, FILE, DIR, IPC)	Modify an attribute of the object
MODIFY-PERMISSION-DATA (SCD)	Change discrete access rights
MODIFY-SYSTEM-DATA (SCD)	Modify system data (e.g. time)
MOUNT (DEV, DIR)	Mount file system to specified mount point
READ (DIR)	Read data from directory
READ-ATTRIBUTE (USER, PROCESS, FILE, DIR, IPC)	Read attribute of the object
READ&WRITE-OPEN (FILE, DEV, IPC)	Open object for reading and writing
READ-OPEN (FILE, IPC, DIR, DEV)	Open object for reading
REMOVE-FROM-KERNEL(NONE)	Remove kernel module
RENAME (FILE, DIR)	Rename file or directory
SEARCH (DIR)	Read directory requested by AEF (kernel internal request)
SEND-SIGNAL (PROCESS)	Send signal to process
SHUTDOWN	Shuts down the system
SWITCH-LOG	Switch logging for ADF module (kernel internal request)
SWITCH-MODULE	Switch ADF module on or off (kernel internal request)
TERMINATE	Inform ADF that the system has terminated the process
TRACE (PROCESS)	Trace the process system calls
TRUNCATE (FILE)	Delete all data in file
UMOUNT (DIR, DEV)	Unmount mountpoint or device
WRITE (DIR)	Write data to directory
WRITE-OPEN (FILE, DEV)	Opens file or device for writing

Table 5.5: RSBAC Linux requests from AEF to ADF

By using the RSBAC ADF protocol, it is possible to provide access control for the defined operating system objects. It is also possible to provide verification of signed BEC files.

5.4.2.6 RSBAC Linux Implementation for Stage 1

Stage 1 is implemented in an ADF module of its own. The only request type which is of interest for this module is the EXECUTE request. When such a request is issued, the target file is then verified against its signature and against the trusted signers. If the verification passes, the request is accepted, otherwise rejected.

Trusted signer information is stored within the ACI. Adding and removing trusted signer information is handled by the ADF module. The ADF module will do this by registering a virtual device to which the security officer may send add- and remove commands. One can argue that this could be solved by adding extra system calls to the kernel. The decision was made mainly because of flexibility. A device can be operated using text-based commands which in theory allows configuration using only common UNIX tools such as the shell. A system call is only available for compiled programs, while a device is available both for compiled programs as well as simple shell commands and scripts.

5.4.2.7 RSBAC Linux Implementation for Stage 2

Stage 2 is also implemented in an ADF module of its own. Since RSBAC works on a lower level of abstraction, there is not always a one-to-one mapping between RSBAC targets and requests to this models operating system objects and access requests. Therefore the implementation has to map the RSBAC targets and requests to identified operating system objects and access requests. This mapping is shown in table 5.6. There are some mappings which are ambigous, and will be resolved by looking at the attributes passed to the ADF. Any other ADF request-target combination is ignored by the implementation, since such mappings have no meaning for this implementation.

RSBAC Request	**RSBAC Target**	**OS Object**	**Access Request**
GET-STATUS-DATA	SCD	Clock	Write
CREATE	DEV	Device	Create
DELETE	FILE	Device	Remove
READ-OPEN	DEV	Device	Read
READ&WRITE-OPEN	DEV	Device	Read
APPEND-OPEN	DEV	Device	Write
WRITE-OPEN	DEV	Device	Write
READ&WRITE-OPEN	DEV	Device	Write
CREATE	DIR	Directory	Create
DELETE	DIR	Directory	Remove
READ	DIR	Directory	Read
WRITE	DIR	Directory	Write
RENAME	FILE	Directory	Write
RENAME	DIR	Directory	Write
DELETE	FILE	Directory	Write
CHDIR	DIR	Directory	Execute
READ-OPEN	FILE	File	Read
READ&WRITE-OPEN	FILE	File	Read
WRITE-OPEN	FILE	File	Write
READ&WRITE-OPEN	FILE	File	Write
TRUNCATE	FILE	File	Write
GET-STATUS-DATA	FILE	FS Object Attribute	Read
GET-STATUS-DATA	DIR	FS Object Attribute	Read
GET-PERMISSIONS-DATA	FILE	FS Object Attribute	Read
GET-PERMISSIONS-DATA	DIR	FS Object Attribute	Read
CHANGE-GROUP	FILE	FS Object Attribute	Write
CHANGE-GROUP	DIR	FS Object Attribute	Write
CHANGE-OWNER	FILE	FS Object Attribute	Write
CHANGE-OWNER	DIR	FS Object Attribute	Write
MODIFY-ACCESS-DATA	FILE	FS Object Attribute	Write
MODIFY-ACCESS-DATA	DIR	FS Object Attribute	Write
MODIFY-PERMISSIONS-DATA	FILE	FS Object Attribute	Write

MODIFY-PERMISSIONS-DATA	DIR	FS Object Attribute	Write
MOUNT	DIR	FS Mount Point	Add
MOUNT	DEV	FS Mount Point	Add
UMOUNT	DIR	FS Mount Point	Remove
UMOUNT	DEV	FS Mount Point	Remove
ADD-TO-KERNEL	NONE	Kernel Module	Add
REMOVE-FROM-KERNEL	NONE	Kernel Module	Remove
SEND-SIGNAL	PROCESS	Process	Signal
TRACE	PROCESS	Process	Trace
CREATE	IPC	Socket IPC	Connect
READ&WRITE-OPEN	IPC	Socket IPC	Connect
READ&WRITE-OPEN	IPC	Socket IPC	Listen
READ	IPC	Socket IPC	Read
APPEND-OPEN	IPC	Socket IPC	Write
SHUTDOWN	NONE	System	Reboot
MODIFY-PERMISSIONS-DATA	SCD	System	HWAccess
SHUTDOWN	NONE	System	Halt
CREATE	IPC	SysV IPC	Create
DELETE	IPC	SysV IPC	Remove
READ-OPEN	IPC	SysV IPC	Read
READ-WRITE-OPEN	IPC	SysV IPC	Read
APPEND-OPEN	IPC	SysV IPC	Write
WRITE-OPEN	IPC	SysV IPC	Write
READ-WRITE-OPEN	IPC	SysV IPC	Write
CHANGE-GROUP	IPC	SysV IPC Attribute	Write
CHANGE-OWNER	IPC	SysV IPC Attribute	Write
ALTER	IPC	SysV IPC Attribute	Write
MODIFY-SYSTEM-DATA	SCD	Swap	Add
MODIFY-SYSTEM-DATA	SCD	Swap	Remove

Table 5.6: RSBAC Request & Target Mapping to OS Objects & Access Requests

After RSBAC targets and requests have been mapped into identified operating system objects and access requests, the objects and access requests are tested against the access control rules. If there is no access control rule which accepts the request or there any access control rule which denies the access request to the operating system object, the access request is denied. Otherwise the access request is accepted. The pseudo-code for this is relatively simple as can be seen in figure 5.6.

Access control rules are kept within the ACI. Modifications to the access control rules are done via a virtual device registered by the ADF module. The security officer may then add and remove access control rules by issuing commands to the device.

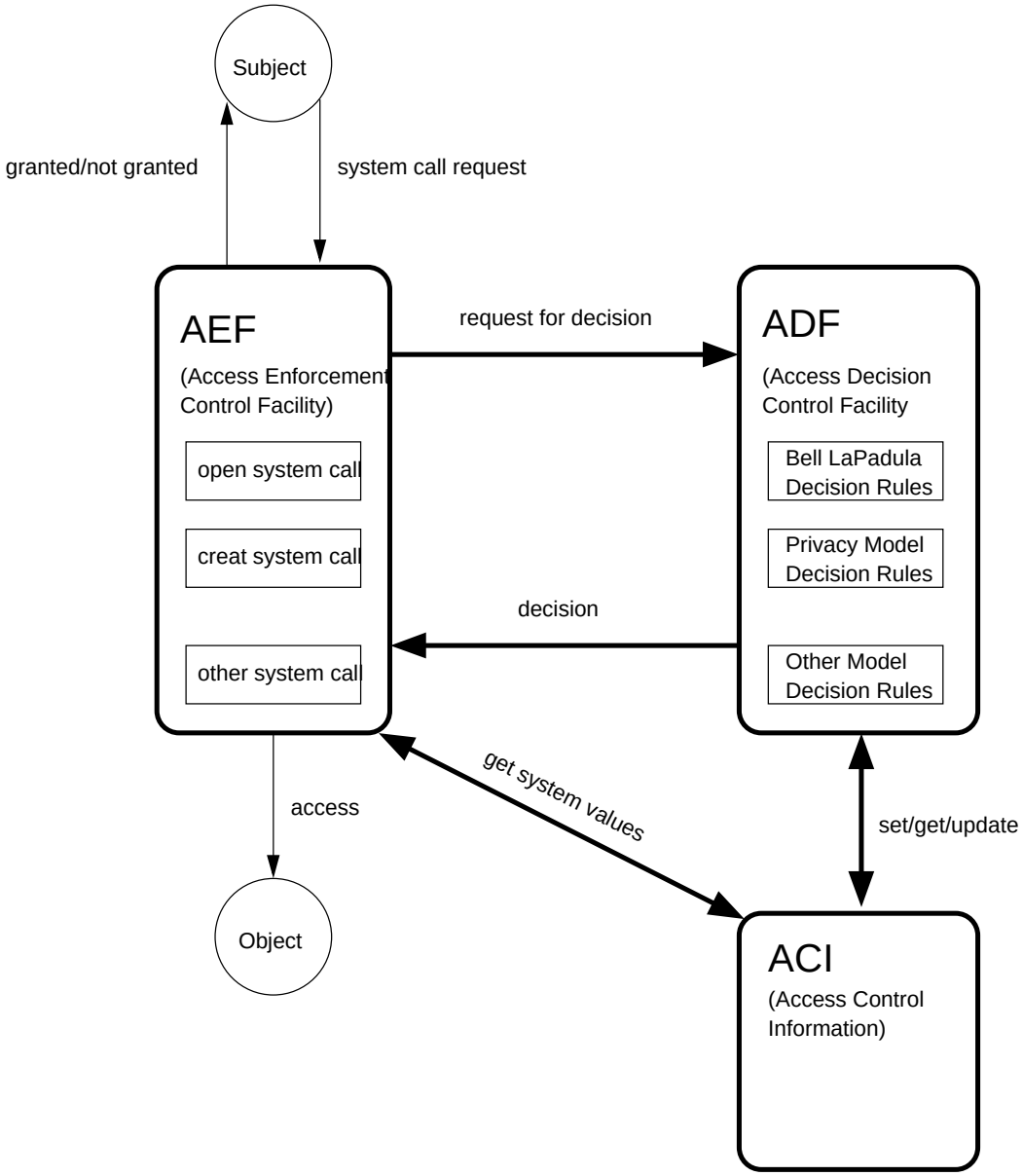


Figure 5.5: RSBAC Overview

```
proc process_request(rsbac_target, rsbac_request) ≡
  (os_object, access_request) = map_rsbac_request(rsbac_target, rsbac_request);
  if ¬exists_rule_for(os_object) ∨ is_denied(os_object, access_request)
    then
        result = DENIED;
    else
        result = ACCEPTED;
  fi
end
```

Figure 5.6: Pseudo-Code for Access Control Rules

Chapter 6

Conclusion

6.1 Future Work

6.1.1 Implementation

As this thesis only specifies a security model and suggests some guidelines for the implementation, an implementation still remains. An implementation is important to show that the model works in practice. It is also important to measure it for security, functionality, efficiency and usability.

6.1.1.1 Security Measure

The security can be roughly measured by trying to exploit well known vulnerabilities and see if the exploitation works or not. The measure is not very scientific, but it gives an indication if the security is good enough. Of course, the system must have been configured correctly to withstand these exploits, so the result depends greatly on the competence of the security officer. The configuration should preferably be done by me or someone else with an expertise in this implementation.

6.1.1.2 Functionaliy Measure

It is important that all software can function in a safe manner using this implementation without having to be rewritten. It would be ideal if this implementation could be "dropped into" a system and then work out of the box when configured properly.

6.1.1.3 Efficiency Measure

If the implementation degrades the efficiency of the system too much, the cause must be investigated. It may turn out that the implementation could have been better, or that the efficiency degradation is inevitable.

6.1.1.4 Usability Measure

Usability is important from a security perspective. It should be easily configured, and easy to comprehend. If not, the likelihood of making an erroneous configuration is bigger, and consequently the likelihood of a security breach is bigger. Usability could be measured by repeating

the security measuring techniques on several sites with several security officers who have never used the implementation before. If the results are identical to the measured results described in 6.1.1.1 are identical, then the usability of the implementation is high.

6.2 Problems

During the writing of this thesis, one problem arised above all other problems[1] - the lack of memory protection in contemporary architectures. The Intel [Int] platform has limited memory protection capabilities.

6.2.1 Definition of the Problem

The problem lies in how the access control bits work for memory access. There are three defined protection modes for memory access: *read*, *write* and *execute*. Logically, reading from memory involves fetching data from memory, writing involves updating data in memory, and execute involves fetching code from memory. Since the Intel architecture does not differentiate between code and data, the execute access right is equivalent to the read access right. From a CPU point of view, there is no difference between code and data until it reaches the CPU - there it is either put in a register or it is put in the instruction queue.

In the implementation of EAC, interception of code execution is done via two system calls: *exec()*, *mmap()*. By controlling these two "code entry points", all code execution can be controlled. If one tries to execute code in a data buffer, the architecture would deny it because it would not have sufficient access rights - *execute*. Only these two system calls can enable the execute access rights for any type of memory area.

However, since the Intel platform (and possibly other platforms) does not really differentiate between execute and read access rights - i.e. read implies execute, and execute implies read, these two system calls are not the only code entry points. In fact, the number of code entry points are close to infinite. It is for example possible to read data from a file into a data buffer, and then simply execute it by issuing a lowlevel CPU jump instruction - effectively bypassing the two entry points.

6.2.2 Workarounds

There are possibly three workarounds for this problem:

- Issue a trouble report to Intel to rectify the problem
- Accept the problem by not controlling execution of non-programs[2]
- Assume that the kernel always sets the memory access bits so that at least function libraries can be controlled

It is not likely that Intel will fix the problem in a near future. Such a fix would require a significant change of the architecture which would in turn cost Intel money for research and development. Also, many systems may rely on this "feature" so that a fix could potentially render many systems non-functional. Therefore Intel would be very reluctant.

[1] Problems such as defining, explaining, researching and any other normal research problems
[2] Function libraries etc.

Simply accepting the problem by not controlling execution of non-programs is not the solution that I am willing to do. Half of the work presented in this thesis is based on the fact that exection of code is and must be controlled.

I am however willing to rely on the Linux kernel code base. All programs must be loaded through the *exec()* system call, so for program BEC's, this problem is not an issue. Function libraries and other BEC's are different. They are normally loaded into memory by using the *mmap()* system call. On an architecture which honours the memory access rights, it would be possible to force all non-program BEC's to enter memory via this system call. However, a rogue program on an Intel architecture may bypass this by just reading the BEC into a data buffer and then execute it. Since the semantics of the system call *read()* it is not possible to determine whether a file is read into memory for data processesing or execution. Under normal operation this scheme would work well. Rogue programs are not "normal" in this sense.

The security is of course degraded by this architecture deficiency. Stage 1 of EAC can be compromised by this. Stage 2 however will still continue to function since access control rules apply to the program BEC, not the function libraries it may have loaded. Program BEC's must have been loaded into memory through the *exec()* system call.

On architectures which honour the memory access rights will of course not degrade the security of stage 1.

Appendix A

Glossary

A.1 Acronyms

Acronym	Explanation
BEC	Binary Executable Content
COM	Component Object Model
PKCS	Public-Key Cryptography Standards
PKI	Public-Key Infrastructure
TBD	To Be Determined
SysV	System V UNIX - a UNIX specification
X.509	CCITT Recommendation for Directory Authentication

Table A.1: Acronyms

A.2 Concepts

Binary Executable Content are files which contain code designed to run natively in the operating system. Examples of such files are program files, function libraries, class libraries, program modules etc.

Certificates are digital identification codes. These contain information about some entity, the certificate holder. They also contain public keys used for encrypted communication between the certificate holder and some other party. Certficates are also signed, or certified, by *certificate authorities* which are organizations which can vouch for the correctness of the certificates. For more information about digital certificates, see [Pfl97] pp 135-140, [Sta00] pp 73, [CCI89].

Component Object Model is a model invented by Microsoft. It is a model which specifies the binary layout of objects. Since the binary layout of the objects follows specification rules, any language can be extended to support COM objects. This means that any language, OO-enabled or not, can be used to implement or use COM objects.

Computer Viruses are programs whose main purpose is to replicate themselves. Viruses may be destructive in varying degrees such that they destroy data in the host machine. A more detailed description can be found in [Pfl97] pp 176-195.

Covert Channels are information channels which leak information. These channels may reveal secret information, or information which may lead to a possible system compromise. More information can be found in [Pfl97] pp 199-207.

Digest is the name of the result when applying a one way hash function over some data.

Digital Signature is meta information about some information, devised using cryptographic methods. The purpose of digital signatures is to ensure authenticity and integrity of the information being signed. For more information see [Sta00] pp 72, [Pfl97] pp 96-97 and [RSA78].

Exploit Script is a program designed to exploit weaknesses in a computer system to escalate privileges, bypass security mechanisms etc.

Hash Functions are functions which calculate a fixed sized code for any input data. A good hash function yields major difference in result for small differences in input data. See [Pfl97] pp 97-99 for more information.

Operating System Objects are entities such as files, directories, IPC facilities and other means for communication outside the process scope. They are also viewed upon as abstract data types.

Public Key Cryptography is an assymmetric cryptography scheme. The scheme makes use of mathematical properties to define a pair of keys for encryption and decryption. One key of the pair is considered private and may only be known by the key pair owner. The other key is considered public and may be distributed to anybody. It should not be possible to derive the private key by knowing the public key. The keys are used for two separate purposes; one key decrypts the data, the other encrypts the data. For more information on public key cryptography see [Pfl97] pp 82-96 and [Sta00] pp 62-72.

Reference Monitor is the part of an operating system which monitors and regulates access to operating system objects. Its two main properties are 1) it is *tamperproof* - no rogue process may circumvent it, and 2) *is always invoked* - it may never let anything pass by unhandled. For more information see [Pfl97] pp 293.

SysV IPC is the name of the IPC mechanisms as specified by SysV UNIX. These mechanisms include support for shared memory, semaphores and message queues.

Trojans are programs which contains hidden functionality. An example of such a program could be a mail reader application, which not only fetches email and displays them nicely, but also forwards them to a third party without the consent of the user of the application. A more detailed description can be found in [Sta00] pp 305-306.

References

[Abr90] M. Abrams. A Generalized Framework for Access Control: an Informal Description. Technical report, MITRE Corporation, August 1990.

[BL73] Bell and LaPadula. Secure Computer Systems: Mathematical Foundations and Model. MITRE Report, MTR 2547, November 1973.

[BS96] Bruce Schneier. *Applied Cryptography*. John Wiley and Sons, 2nd edition, 1996.

[CCI89] CCIIT/ITU. CCITT Recommendation X.509, The Directory-Authentication Framework. 1989.

[CER] CERT. http://www.cert.org/.

[CWH99] Campione, Walrath, and Huml. *The Java Tutorial Continued, The Rest of the JDK*. Sun Microsystems, 1999.

[ElG86] T. ElGamal. A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In *Advances in Cryptology: Proceedings of CRYPTO 84*, pages 10–18. Springer-Verlag, 1986.

[Fed94a] Russian Federation. GOST R 34.10-94 Digital Signature Standard. *Gosudarstvennyi Standard of Russian Federation*, 1994.

[Fed94b] Russian Federation. GOST R 34.11-94 Hash Function. *Gosudarstvennyi Standard of Russian Federation*, 1994.

[Fis01] Fischer-Hübner, Simone. *Privacy-Enhancing Design and Use of IT-Security Mechanisms*. Number LNCS 1958. Springer Scientific Publishers, February 2001.

[Gar00] Paul Garrett. *An Introduction to Cryptology*. Prentice Hall, 2000.

[Gas98] Morrie Gasser. *Building a Secure Computer System*. Van Nostrand Reinhold, 1998.

[GE94] Gene H. Kim and Eugene H. Spafford. The design and implementation of tripwire: A file system integrity checker. *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 18–29, 1994.

[GJSB00] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Sun Microsystems, 2nd edition, 2000.

[Hau98] M. L. Haungs. Extending Sim286 to the Intel386 Architecture with 32-bit processing and Elf Binary input. Technical Report, UC Davis, Computer Science Dept., Parallel and Distributed Computing Lab, September 1998.

[Int] Intel. http://www.intel.com/.

[KPS95] Kaufman, Charlie, Perlman, Radia, and Speciner, Mike. *Network Security - PRIVATE Communication in a PUBLIC World*. Prentice Hall, 1995.

[LLJ99] S. Lindskog, U. Lindqvist, and E. Johnsson. IT Security Research and Education in Synergy. In *Proceedings of the 1st World Conference on Information Security Education, WISE 1*, pages 145–162, Stockholm, June 1999. IFIP, FIP.

[Lu,95] Lu, H. ELF: From The Programmer's Perspective. Technical Report, NYNEX Science and Technology, Inc., May 1995.

[LY00] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Sun Microsystems, 2nd edition, 2000.

[McE78] R.J. McEliece. A Public-Key Cryptosystem Based on Algebraic Coding Theory. Deep Space Network Progress Report, Jet Propulsion Laboratory, California Institute of Technology, 1978.

[Moo85] J.H. Moore. Some Public-Key Functions as Intractable as Factorization. In *Advances in Cryptology: Proceedings of CRYPTO 84*, pages 66–70. Springer-Verlag, 1985.

[Moo86] J.H. Moore. An M^3 Public-Key Encryption Scheme. In *Advances in Cryptology: Proceedings of CRYPTO 85*, pages 358–368. Springer-Verlag, 1986.

[Moo92] J.H. Moore. Protocol Failures in Cryptosystems. In *Contemporary Cryptology: The Science of Information Integrity*, pages 541–558. IEEE, IEEE Press, 1992.

[Mor97] M. Morrison. *Java*. Sams.net Publishing, 2nd edition, 1997.

[MS] N. Mavroyanopoulos and S. Schumann. *Mhash Implementation*. http://mhash.sourceforge.net/.

[MS96] Microsoft authenticode technology, ensuring accountability and authenticity for software components on the internet. October 1996.

[NIS91] NIST. Proposed Federal Information Processing Standard for Digital Signature Standard (DSS). *Federal Register*, 56(169):42980–42982, August 1991.

[NIS92] NIST. Proposed Federal Information Processing Standard for Secure Hash Standard. *Federal Register*, 57(21):3747–3749, January 1992.

[OF00] Ott, Amon and Fischer-Hübner, Simone. Rule Set Based Access Control in Linux. Technical report, Compuniverse, Brauch und Ott GbR and Karlstad University, Dept. of Computer Science, 2000.

[Ott] Amon Ott. http://www.rsbac.org/.

[Ott97] Amon Ott. Regelsatz-basierte Zugriffskontrolle nach dem "Generalized Framework for Access Control"-Ansatz am Beispiel Linux. Master's thesis, University of Hamburg, Fachbereich Informatik, November 1997.

[Pfl97] Pfleeger, C. P. *Security in Computing*. Prentice Hall, 2nd International edition, 1997.

[PGP] PGP. http://www.pgp.com/.

[Rab79] M.O. Rabin. Digital Signatures and Public-Key Functions as Intractable as Factorization. Technical Report, MIT/LCS/TR-212, MIT Laboratory for Computer Science, January 1979.

[RAC92] RACE. RIPE Integrity Primitives: Final Report of RACE Integrity Primitives Evaluation (R1040). Research and Development in Advanced Communication Technologies in Europe, RACE, June 1992.

[RSA] RSA. http://www.rsasecurity.com/.

[RSA78] Rivest, R.L., Shamir, A., and Adleman, L.M. On digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[SG94] A. Silberschatz and P.B. Galvin. *Operating System Concepts*. Addison Wesley Longman Inc., 4th edition, 1994.

[Sta00] Stallings, William. *Network Security Essentials, Applications and Standards*. Prentice Hall, 2000.

[TIS93] TIS Committee. *Tool Interface Standard (TIS) Portable Formats Specification Version 1.1*. Tool Interface Committee, Intel Corporation, Literature Center, Order No: 241597, October 1993.

[Tri] http://tripwire.com/.

[Wil80] H.C. Williams. A Modification of the RSA Public-Key Encryption Procedure. *IEEE Transactions on Information Theory*, IT-26(6):726–729, November 1980.

[ZPS93] Y. Zheng, J. Pieprzyk, and J. Seberry. HAVAL - A One-Way Hasing Algorithm with Variable Length of Output. In *Advances in Cryptology: Proceedings of CRYPTO 92*, pages 83–104. Springer-Verlag, 1993.