CONTROLLED ENVIRONMENT TO RUN UNTRUSTED APPLICATIONS

A thesis submitted
To Kent State University in partial
Fulfillment of the requirements for the
Degree of Master of Science

by

TAHANI ALBALAWI

August, 2013
Thesis written by

TAHANI ALBALAWI

B.S., Taiba University, KSA 2007

M.S., Kent State University, USA, 2013

Approved by

Michael Rothstein,  Advisor

Javed Khan , Chair, Department of Computer Science

Raymond Craig,  Dean, College of Arts and Sciences
Abstract

Untrusted applications pose a security threat to systems they run on. At the same time sometimes a user wants to run such applications without risking unauthorized access to certain parts of the system. For this reason we propose a controlled environment to run host untrusted applications based on AppArmor tool. We intend to augment the usefulness of AppArmor by further restricting the permissions of the files accessible to the process using other system mechanisms, such as NFS and Chroot. Based on the principle of least authority, programs can run in a restricted secure environment limiting access to a subset of the files. This way untrusted applications are allowed to run with limits to protect the system.
# Table of Contents

LIST OF FIGURES ........................................................................................................ vii  
LIST OF TABLES ......................................................................................................... vii  
DEDICATION ................................................................................................................. viii  
ACKNOWLEDGMENTS .................................................................................................... ix  

CHAPTER 1 Introduction .............................................................................................. 1  

CHAPTER 2 Problem definition and history ............................................................... 4  
2.1 jailing process mechanisms ................................................................................... 5  
2.1.1 Change Root (Chroot) method .......................................................................... 6  
2.1.2 User-mode Linux (UML) method ..................................................................... 7  
2.1.3 Chroot vs UML ............................................................................................... 8  
2.2 Description of the problem .................................................................................... 10  
2.2.1 Chroot limitations .......................................................................................... 10  
2.2.2 Sandbox mechanism ...................................................................................... 11  
2.4 Access control models ......................................................................................... 12  
2.4.1 Mandatory access control (MAC) .................................................................. 14
2.4.1.1 Bell-LaPadulla Confidentiality Model ................................................. 14
2.4.1.2 Biba Integrity Model ........................................................................... 15
2.4.1.3 Advantages of Having A MAC Implemented Model ....................... 16
2.4.1.4 Issues in Having a MAC based Model ............................................... 17
2.4.2 Discretionary Access Control (DAC) ..................................................... 17
2.4.2.1 Access Control Lists (ACLs) ............................................................... 18
2.4.2.2 Advantages of Having a DAC Implemented Model ....................... 18
2.4.2.3 Issues in Having a DAC based Model ............................................... 19
2.4.3 Role-Based Access Control (RBAC) ..................................................... 19
2.4.3.1 Advantages of RBAC......................................................................... 19
2.4.3.2 Disadvantages of RBAC .................................................................... 20
2.4.4 Domain and Type Enforcement (DTE) ................................................ 20
2.5 Alternative security frameworks............................................................... 21
2.5.1 Selinux.................................................................................................. 21
2.5.1.1 SELinux Architecture................................................................. 21
2.5.1.2 Problems with SELinux................................................................. 26
2.5.2 Plash.................................................................................................... 26
2.5.2.1 Plash Architecture................................................................................................. 27
2.5.2.2 Problems with Plash.............................................................................................. 39
2.5.2.3 Plash vs Chroot...................................................................................................... 31

CHAPTER 3 current framework design........................................................................... 33
3.1 AppArmor .................................................................................................................. 33
3.2 AppArmor Architecture.............................................................................................. 35
3.3 AppArmor Advantages............................................................................................... 38
3.4 AppArmor disadvantages............................................................................................ 38
3.5 AppArmor VS SELinux.............................................................................................. 39

CHAPTER 4 Implementation........................................................................................... 41
4.1 First part: Confining application process using Parent-Child mechanism................ 41
4.2 Second part: Add AppArmor profile........................................................................... 45
4.3 Program test............................................................................................................... 46

CHAPTER 5 Conclusion and future work ...................................................................... 49

APPENDIX A .................................................................................................................. 52
APPENDIX B................................................................................................................... 60
REFERENCES.................................................................................................................. 67
List of Figures

Figure 2.1: Flask architecture as presented by RedHat.....................................................24

Figure 2.2: Plash Architecture .....................................................................................29

Figure 3.1: AppArmor architecture ...............................................................................36

Figure 4.1: flow chart for part one ................................................................................44

Figure 4.2: confine Firefox browser in (my_tmp).............................................................47

Figure 4.3: Using the process number to see where it is executing...................................47

Figure 4.4: Denying the access to Desktop files...............................................................48

List of Tables

Table 2.1: The list of glibc calls that modified or kept the ...........................................28
same as in the Plash documentation

Table 3.1: Example of some options for files or directories...........................................37
in the AppArmor profile
DEDICATION

With all love this thesis is dedicated to my family, A special feeling of gratitude to my loving parents, Mariam and Fahad, for their endless love, support and encouragement. My parents whose brought me up with their love and encouragement to pursue advanced degrees and never left my aside.

Also, I would like to give my heartfelt appreciation to my husband, Talal, who has accompanied me with his love, unlimited patience, understanding, helping and encouragement. Without his support, I would never be able to accomplish this work. So, I want to dedicate this thesis also to my husband.
ACKNOWLEDGMENTS

This thesis would not have been possible without the support of many people. I would like to thank my committee members who were more than generous with their expertise and precious time. Many thanks to my adviser, Dr. Michael Rothstein, for his countless hours of reflecting, reading, encouraging, and most of all patience throughout the entire process. Also a special thanks to Dr. Austin Melton who offered guidance and support. Thank you Dr. Arden Ruttan for agreeing to serve on my committee.

Tahani Albalawi

May 24, 2013, Kent, Ohio
Chapter 1: Introduction

The invention of computers has revolutionized the way we live our lives. It made many tasks easy and simple. For the last thirty years the world has relied heavily on using computers in education, business and in the military. The introduction of the World Wide Web made it easy for people to reach and communicate with others around the world with just a click of a button.

With this evolution of the Internet, many issues and concerns were raised when it comes to security. The users' intent also plays a role when it comes to computer security. There are crackers who look for vulnerabilities to attack systems for the sole purpose of harming others or entities. On the other hand you have a regular user who does not have the necessary security background when it comes to secure practices to using computers. For these users, downloading an attachment could infect an entire network although they did not intentionally do that. Some people would argue that educating this category of users is enough to prevent such actions, but education is not enough by itself. There must be a way to limit what such users can do in a certain environment by allowing them access only to what they need and not to all parts of the system.

The focus of security was to protect users from each other and from outsiders by implementing monitors that would check system accesses. However, since there is no
perfect program, these mechanisms were also subject to vulnerabilities, which meant continuous patching of holes in the Operating System to improve the security. The problem with this approach is two-fold: it is reactive, so that problems are fixed after they are discovered and there is a window of opportunity for vulnerabilities. In addition, the patching process itself often introduces new vulnerabilities. But if we set up an environment inside such Operating Systems where we can run untrusted applications but with restricted permissions in a way where we protect the Operating System from harmful actions triggered by the application, we will not be exposing the Operating System to the vulnerabilities. An example would be to limit the communication ports and java scripts in a web browser to prevent a hacker from using known vulnerabilities to attack the system.

In recent years a number of models were used to create a restrictive secure environment where an application can run. For example Plash, provides a secure mechanism to run untrusted applications by confining untrusted application in a sandbox and executing them and running them in an isolated environment. The sandbox takes control of accessing the file system. So the application running in it can have access only to the system files, which are needed for it to function properly; at the same time, it is isolated from other important parts of the system. Other packages, like SELinux and AppArmor placed restrictions on the ways applications could interact with the system.

In this thesis a sandbox is designed for the Linux operating system. A sandbox has multiple ways of implementation. Moreover we give a detailed analysis of these different options available in designing a sandbox. At the same time AppArmor will be used along with C++ code to create a further restricted environment to run untrusted applications.
In chapter 2, we will cover a detailed description of the limitations of Chroot and how this limitation introduced new ideas for sandbox mechanisms. We follow with a brief review of access control models that have been used in security frameworks. Also, we will discuss alternative existing tools to secure Linux such as Plash and SElinux.

In chapter 3 a detailed description of the AppArmor framework which is used in the implementation part of this thesis and the difficulties encountered will be discussed. In chapter 4 we will discuss an implementation of the sandbox to confining processes in a pre-specified directory and how to use the AppArmor tool to add further restrictions to the application. Finally we will present conclusions and suggestions for future work in chapter 5.
Chapter 2 : Problem definition and history

Untrusted applications pose a security threat to the systems they run on. Still, users often need to run such applications without risking unauthorized access to certain parts of the system; therefore, supporting the systems with a security environment to run untrusted applications will make it more convenient to deal with untrusted application threats.

There are a number of methods to provide systems with a secure environment to run these untrusted applications without risking unauthorized access to important parts of the system. For instance limiting the resources -jailing - is confining programs from accessing unnecessary files or file systems. The main technique using this approach is Chroot. Another approach, providing the security frameworks that are used to control the threats of untrusted applications by supporting access control security policies are more recent. This thesis will address in detail three types of security frameworks; they are Plash, SELinux and AppArmor.

Most of these methods depend on access control mechanisms to ensure that authorized access is allowed and unauthorized access is not allowed, so the system will be kept from improper modifications. The mechanisms of access control provide different levels of protection for different kinds of files, set the access permissions and make decisions as to who can access which files, and what the limitations are to this access. They determine the kind of access (read, write, etc.). This chapter will cover in detail different types of
access control models, their characteristics and some existing security mechanisms that try to implement these controls.

2-1 jailing process mechanism

The idea of jailing a process is “a process ideally places the desired process or processes in a self-sufficient environment that cannot reach the main system outside of the jail”[Lessard11]. A process inside this isolated environment cannot have any access to file system resources; thus the important system files e.g. password and the main system's log files will be kept out of the attacker's reach. Also, even if the attacker succeeds in damaging a process inside the isolated environment itself it will not affect the main system which adds another security layer to the system.

The jailing mechanism does not contradict with file system permissions settings. It would be another layer of protection for the system. The problem with file system permissions is that a user with limited file permissions would still be able to navigate to top-level directories e.g. the user can navigate and see other user's directories or even navigate system-critical directories. Although the user would not have permissions to edit these directories he or she would be able to see the files, target specific ones and try to exploit that information. This is considered one of the weakest points for the system security fence. On the other hand the jailing mechanism can avoid the previously weak point of the file system permissions setting.

Lessard in his paper [Lessard11] described two tools as examples for jailing processes.
These tools are Chroot- C(hange Root)- and User-mode Linux. Every one of them has its advantages and limitations.

2-1-1 Change Root (Chroot) method

In Chroot jailing, the root directory of a command is changed to another directory. Thus, we can use Chroot to create an identical directory to the main root directory and confine the processes inside the new root directory, which will be an isolated environment. So, in case there is an attack to any process inside the new root the basic root directories will be safe. In Chroot, system administrators have complete control over what services are to be provided inside the jail. This issue can be viewed from two points of view. First it could be considered as an advantage by allowing the system administrators to define exactly what services are needed and what services are not needed to be implemented in the jail. However, it could also be considered as a disadvantage because any necessary files such as shared libraries and configuration files, not present in the jail, will not be present at all, as far as the application is concerned, causing the application to fail. In fact, the process of determining the needed resources and adding them is considered to be very costly and complicated.

There are some security issues regarding the use of Chroot as a security mechanism. For instance, if the administrator adds a critical file which contains sensitive information or blindly adds unnecessary resources for the jail such as /etc/passwd which contains a list of all users on the system, it would make it helpful for attackers to use this information to exploit the system.

Another security issue is that having root access inside the jail is going to endanger the
system. With an attacker having root access they may cause a jailbreak by taking advantage of root administrator privileges, which gives the attacker the ability to change file permissions, changing root (again), etc.

**2-1-2 User-mode Linux (UML) method**

In this type of jailing the UML creates and sets up multiple virtual machines. It enables the user to boot a full Linux kernel inside of a user space which makes the application run within a normal Linux system (known as the host). So the user - guest- can run an application as guest and as a result the application can be run without special admin rights.

The multiple virtual machines are isolated from each other and from the hardware[Lesssard11] The kernel provides that by assigning isolated virtual resources, including a root file system and swap space, and can have a hardware configuration entirely separated from that of the host. This isolation gives the user the ability to run an application without taking the risk of breaking the host system in case of an attack happening like for example running a suspicious program.

UML security is a confusing subject to handle. The reason for that is the nature of the UML design where every virtual machine is considered as a full system. So UML security sometimes needs to get system administrator privileges to build up virtual machine security.

There are some security issues regarding UML as a secure method to run untrusted applications. For instance if a user runs the virtual machine as the root user, it makes the
system more vulnerable leading any attempt to hack into the system by intruders to have a high probability of being successful. They can then break into the main system, which is what we are trying to avoid.

2-1-3 Chroot vs UML

A choice between Chroot and UML is very highly dependent on the security target, what we are trying to protect, what we are trying to confine and what applications we want to have limited access to the system.

The Chroot jailing method is preferred with simple or compatible programs that do not need system overhead or system administration overhead. Thus, the system will achieve high performance due to the smaller overhead needed to jail applications. The main characteristics of Chroot is its age, “it is a tried, tested, and true way of securing an application, without a huge performance overhead” [Lessard11]. As a result of its age there are many studies published and many projects and tools created that depend on Chroot concepts. This enables the users or system administrators to obtain advantages of these previous studies and use the created tools to assist security for their application with full confidence.

The UML jailing method is preferred more in other situations. It is suitable for larger and complex applications because setting up a jail for every one of the applications on one host system will be very difficult to maintain, and that will influence the security of the applications negatively specially when maintaining some applications which need to cooperate. Thus, the UML can solve this problem by confining all these application and putting them in one jailed virtual machine. Furthermore, it supports the security situation
in the case of memory leaks in the system by limiting every process from consuming all of the system's resources

Moreover, the way the virtual system is designed in UML gives the user many security benefits. Since the virtual system can be entirely contained in one file or file system, the backup and restore of the virtual system in case of hacking will be easier. Also, incident containment and eradication time can be far reduced. Thus, this design of the single file gives the user the ability to experiment with a copy of the virtual system without risking the system to have any damage. In addition it is useful for applying and testing patches especially for complex patches. One last benefit with the single file, when the user builds up the wanted security rules on the virtual machine, he or she can deploy these policies to multiple physical systems.

Finally, the most distinguishing characteristic for the UML jailing method is that the user can run the virtual machine within a Chroot jail and as a result another security layer can be added to the applications.
2-2- Description of the problem

2-2-1- Chroot limitations

Just like any other sandbox mechanism Chroot has its limitations. One of the issues is that Chroot cannot counter or protect against internal attacks like an attack launched by a privileged user such as root. Another issue is Chrooted programs may perform another Chroot, which may cause them to break out. One approach to this problem is that Chrooted programs should relinquish root privileges as soon as they Chroot.

If there is a system that supports device nodes on ordinary file systems, a Chrooted root user can mount file systems on them. There is no way of blocking privileged users from accessing low level system devices.

Since Chroot can only be used by the root user, only some threats still prevail due to not being able to protect against attacks launched by privileged users. Thus, it can be seen that Chroot is about putting the user not its application or process into a jail and this is why it has issues like the ones mentioned above. Such limitations motivated developers to look for an alternative to confining applications, not users. This lead to the introduction of the concept of the sandbox which will be introduced in the following section.
2-2-3 Sandbox mechanism

As mentioned in the previous section in the Chroot limitations, Chroot cannot offer protection against privileged users' attacks. Confining a user is not enough; we also need to limit the running application or process. This forced developers to look into other ways for confining applications and not users to prevent any attack on the system in the form of unauthorized access to important parts of the system. The problem of having an untrusted application run and having access to all parts of the system is it can cause attacks to be triggered against the system. Such applications can exploit a system weakness allowing others to attack the system. This leads to the introduction of the concept of sandbox.

Just like the playground sandbox, where kids build their castles out of sand or play in that known environment, the software sandbox is a testing environment, which isolates untrusted code running, or applications from accessing important parts of the system. Such isolation will guarantee that whatever the intention of this untrusted application no changes can take effect outside this created environment. Some developers view virtual machines like for example virtual Box as a kind of sandbox. The user starts the virtual machine and inside it they might be running a different operating system than the one they are using which in turn is isolated from the other operating system which it runs on top of.

Sandboxes are not only environments to run untrusted applications. They can also be used to test new developed applications. It is now possible to write C++ code and compile it using a web browser only because some website provides C++ sandboxes for development purposes.
Though a sandbox can be viewed as a copy of the system, it is not the entire system but rather a small portion of the system. It is small but guarantees that the program or process confined inside the sandbox can perform its task without problems.

Many of the programming languages started to have new tags and keywords to sandbox the application developed for testing. This includes C# that introduced the domain concept and Java.

In this thesis a number of applications that use this concept will be presented. This includes Chroot, SElinux, Plash and AppArmor. These applications may differ in the way they perform the task but the goal is to confine the running application, process or users just like the description of the sandbox.

2-4- Access control models

The goal of any access control mechanism is to provide a way of guaranteeing the protection of information in any given system from unauthorized access or modification. It was thought originally that each user could dictate the policy they needed over the files they created. Thus, Discretionary Access Control was born. However, it was soon seen that no system could be secure with only discretionary access controls, because malicious users could alter any protections in place [Sandhu94]. Since the definition of ``secure" implies stating security goals, these goals are often listed in the security policy implementation. In addition what is to be protected has the most important aspect when it comes to choosing an access control policy. Usually in access control the goal is to control both or one of the following:
- **Confidentiality**: Prevent unauthorized disclosure of information.

- **Integrity**: Prevent unauthorized modification of information.

Both confidentiality and integrity are driven from two different environments. According to Bishop [Bishop04] military security policies' sole focus is the protection of information from disclosure to unauthorized personnel whereas integrity is driven from the commercial security policy that focuses on keeping information from being modified. Both of these require a Mandatory Control Access model.

In [Ausanka01] the author summarizes the methods for access control comparing them and what limitations they have. To the author such access control mechanisms have limitations if they fail in one of the following:

- **Inescapable**: The ability to break the security policy by altering the access control set by the model.

- **Invisible**: The smoothness of interaction of the user and administrator with the model.

- **Feasible**: Cost and practicality of implementing the model.

Some of the limitations that can break inescapability is the lack of support for the principle of least-privilege. Another limitation that affects invisibility is the difficulty in using the system. For feasibility too high a cost in implementing the model is viewed as a limitation. The author of the paper views all of these as things that would cause the model to break. There are other limitations but these are listed as examples to show what each one means.
In this chapter, two additional methods are listed also at the end. These are Role Based Access Control (RBAC) where the concept of role is used to add features for both MAC and DAC. The other method is Domain Type Enforcement (DTE), which is an extension of Type Enforcement (TE). It will be discussed in detail later in this chapter. In general, Access control models’ the primary object is to control the flow of information between subjects and objects, regardless of whether that flow is in terms of read or write. The goal is to have control of that flow by allowing such flow or blocking it from happening. In the following each model is discussed in further detail.

### 2-4-1 Mandatory Access Control (MAC)

The wide definition for this model is any security model that enforces the security policy regardless of what action or set of actions a subject initiates onto an object or objects. When MAC is mentioned it is usually associated with two known models. the Bell-LaPadula Confidentiality Model or the Biba Integrity Model. Both models have a focus on one of the two controls mentioned early in this chapter: Bell-LaPadulla for Confidentiality and Biba for Integrity.

#### 2-4-1-1 Bell-LaPadulla Confidentiality Model

The Bell-LaPadulla model was introduced in 1973 with the idea of multilevel security. In this model, security labels are assigned to each subject and object. Then it uses two security properties to force the model policy. These properties are the *simple property* and the *Star property* (the Star property). It tends to follow the military style of ensuring
security by restricting the flow of information from higher more secure level to a lower less secure level. The two properties:

- *simple property* Security: This handles the read case for a subject to read an object it has to have a security label placing it higher than or equal to the one of the object.
- *property* Security Prosperity: This handles writes for a subject to write to an object it has to have a security label less than or equal to the one of the object.

So basically in this model is read down and write up. It also features what is known as the tranquility feature. It has weak and strong tranquility. In strong tranquility the model restricts that no security label can change during system operations. The weak tranquility allows for labels to have the change they require but it must be done in a way to prevent any violation to the policy already defined and placed in action.

The weak tranquility allows for a starting subject at a security level lower than an object to upgrade to have the access it needs. This is also known as the high water mark principle.

**2-4-1-2 Biba Integrity Model**

As we saw in the previous section Bell-LaPadulla (BLP) has confidentiality as its primary focus. This is due to it following the military style of confidentiality. This was the reason behind having a separate model to cover integrity. So in general Biba is in the opposite direction of BLP. This is clear from the security properties it follows:
Table 1. *Simple Security Property:* This handles the read case. For a subject to read an object it has to have an integrity level lower than the one associated with the object.

Table 2. *_(Property) Security Property:* This deals with writes. For a subject to write to an object it has to have an integrity level higher than the one associated with the object.

So it is clear that it is the opposite of BLP. Reads are up and writes are down. Having such a model ensures the integrity of the system but there is an issue. Integrity was not the focus of the MAC model and this led to it having little or no impact on MAC based developed systems.

**2-4-1-3 Advantages of Having A MAC Implemented Model:**

The author in [Ausanka01] mentioned that having BLP implemented in a Multi-Layer Secure (MLS) systems proved to be beneficial and protective against attacks such as Trojan horse attacks. This is due to users not being able to declassify information making it the model of choice in military and intelligence agencies around the world. In these places the focus is on confidentiality with little consideration being given to integrity.
2-4-1-4 Issues in Having a MAC based Model

MAC models can place restrictions on the users or user actions. Even a user who is following the implemented security policy, might not be able to have access to other parts of the system. Such problems require modification of the operating system or even to the framework which might require placing parts of the system outside the MAC policy. Such actions might endanger system security. In addition MAC can over classify data through the *high-water-mark* principle, which may cause the limitation of productivity. MAC does not handle the concept of least privilege in a proper way. Finally MAC systems are expensive and difficult to implement. These are some of what is believed to be the problems with MAC.

2-4-2- Discretionary Access Control (DAC).

As mentioned in the MAC issues section, these issues and others not provided in this research caused MAC to be less widespread when compared to DAC. When MAC had the military as its basis of use DAC was designed in the commercial world and in academia. DAC was developed to implement the Access Control Matrices presented by the research conducted by Lampson [Lampson74]. Access Control Matrices are two-dimensional matrices having rows as subjects, columns as objects and the mapping of the set of subjects and objects pairs as the rights the subject have over an object.
In MAC we have predefined policies to what a subject can do to an object. In DAC the subject can decide what it can do to any objects they “own”. DAC access settings are usually stored as per-object file permission modes like in UNIX or as lists.

### 2-4-2-1 Access Control Lists (ACLs)

ACLs represent the object rights as a table of subjects mapped to the rights they have over an object. The problem with ACLs is it requires the operating system to look for the rights a subject has over an object each time an object is accessed. The operating system knows who is the owner of the process but does not know what access rights it has over an object. This makes ACLs not efficient when used in large systems with a large number of subjects and objects. There is another approach similar to the ACLs known as Capability Lists. These consist of lists of objects and rights a subject has; in this way, it makes it easier for the system to look information up..

### 2-4-2-2 Advantages of Having a DAC Implemented Model

DAC allows for the control of system objects better than MAC. It can also be used to implement the principle of least privilege with no issues. An object under DAC can limit the type of access a subject would have to that object with the minimum rights required. It is also cost effective compared to MAC and is easy to use and manage.
2-4-2-3 Issues in Having a DAC based Model

Having users control objects may open the system to attacks like Trojan horse attacks. In addition, system maintenance and security principles verification is extremely difficult under DAC. These are some of the limitations as mentioned by the author in [Ausanka01].

In the next section is the description of the other two access control methods that came after DAC and MAC.

2-4-3- Role-Based Access Control (RBAC)

RBAC is viewed as a more generalized model than both DAC and MAC. It can be used to implement or customize parts of both MAC and DAC or to use one of them based on the application at hand. When first introduced in 1992, the developers of the model wanted to have the focus on commercial and civilian uses as a primary focus. So integrity was first then confidentiality second. RBAC functions by assigning the rights to the role. So read and write rights will be based on the role of that person in the organization instead of subjects and objects like in MAC and DAC. In this way RBAC removes the ownership concept in DAC by preventing the transferring of rights and confining them to the role of that person.

2-4-3-1 Advantages of RBAC

Controlling what resources are accessed and how access happens ensures system integrity. Having a number of roles in given organizations and placing users in these roles
leads to easy management of the system. This leads to an easier way of verifying the security policies at hand. Finally it supports the principle of least privilege.

2-4-3-2 Disadvantages of RBAC

One of the problems of RBAC lies in the size of the organization. If it is too large it can cause a problem for maintaining the system by having a large number of roles. Also, a need for customized privileges can cause issues for the administrators of the system. Finally, for simple systems, it may be too complicated to be cost effective.

2-4-4- Domain and Type Enforcement(DTE)

DTE is an extension from Type Enforcement (TE). The idea is to assign objects to types so there is a column in the access matrix having types with listed objects. The DTE extension assigns subjects to domains then transforms the access matrix into a domain definition table (DDT) with rows of domains and columns of types. In this way, access to an object is based on what domain the subject is assigned to and if the object type is in that domain.

We presented in this section some of the known methods for access control. In the next chapters a set of applications are discussed that make use of these methods.
2-5- Alternative security frameworks.

In this section we will discuss some alternative security frameworks to confine applications within isolated environments that are expected to overcome the Chroot problems mentioned earlier. Each one of these frameworks has its own procedure and design that it follows to confine an application. Although some of them follow core principles like MAC, they differ in the way they implemented mechanisms to confine the applications.

2-5-1 SElinux

The National Security Agency (NSA) developed SELinux or Security Enhanced Linux back in the year 2000. It was integrated in the mainline Linux operating system kernel in 2003. It is an implementation of the Flask operating system Security architecture. It uses the MAC access model; as described previously, the Flask architecture handles the issues that MAC suffers from which is the restrictive access due to least privilege. This concept gives the requesting process the rights it only needs to do a certain task. Using the Linux security Modules (LSM) framework, the NSA evolved SELinux to be part of the kernel of Linux. Many of the current Linux distributions like Fedora, SUSE and RedHat have SELinux integrated into the operating system.

2-5-1-1 SELinux Architecture

As mentioned in the previous section, Flask was developed to overcome some of the problems in a MAC architecture. As mentioned previously, MAC is integrated with the MLS model (Multilevel Security). The access decision is based on what level of
clearance a subject has over an object. So no read-up and no-write down are allowed which basically follows the concept of confidentiality. In such a model integrity is not taken into account. In addition the least privilege concept is not handled. MAC has as discussed earlier no mechanism of addressing these security needs.

Flask solves these problems for MLS based on MAC by separating the policy enforcement from the policy logic, which is also known as the security server. The job of the security server is to interpret the security contexts or labels (set of attributes assigned to subjects and objects). These labels have the following format:

`<user>:` `<role>` `<type>`

An example is:

`System_u:object_r:httpd_exec_t`

The `system_u` is a standard identity used for daemons. The `object_r` is the role for system objects like files. The `httpd_exec_t` is the type applied to the executable, in this case `/usr/sbin/httpd`.

Prior to full integration with the Linux kernel, the security context or labels were maintained in a file as a set of security identifiers (SID). When it was integrated into the kernel the SID’s were placed in the kernel and could not be exported to the user space.
When a request for access happens, the security server needs only to look-up a pair of labels on a matrix of type labeled subject and object. It then puts the result in the access vector cache (AVC) to retrieve for any matching requests that come later.

Having a generalized form of Type Enforcement (TE) makes Flask flexible when it comes to labeling for transition and access decisions. Instead of having to make those decisions based on a predefined lattice, it is able to use the user identity to define other labels or even role attributes. Flask can use different methods to perform the access computations like lattice models, static matrix look-ups, historical decisions, and environmental decisions. The policy engine that produces the result and caches them deals with such computations. Another flexibility of Flask is the ability to change any of these subsystems without altering the other subsystems; this is due to the separation of the policy enforcement from the policy decision-making. The following diagram, Figure 2.1, shows the architecture of Flask as presented by RedHat.
Figure 2.1: Flask architecture as presented by RedHat
So looking at the previous figure we can see the steps of Flask taking place. It can be summarized to be the following:

- A subject wants to perform a task on a given object.
- The policy enforcement server gathers the security labels from the subject and object. It then forwards this information to the security server which has to make the decision of granting or denying access.
- The policy server checks the AVC for cached requests then returns the decision to the enforcement server. If no policy decision is in the AVC it asks the security server for assistance. The security server uses the binary policy loaded into the kernel. The AVC caches this decision and returns the result to the enforcement server.
- When the decision received is to grant access, the subject proceeds with the operation.
- If the decision is to not permit the subject from proceeding, then an access denied is given to the subject.

This way we can see that the security server handles the policy decision-making when the enforcement server takes care of the rest of the tasks.

SELinux is an implementation of the Flask architecture. The way it works is similar to what is outlined in the description of Flask. However SELinux is not an easy framework to use. It has many difficulties and issues surrounding it to the level where many users opt-out from using it to other more user-friendly frameworks.
2-5-1-2 Problems with SELinux:

SELinux has some issues related to its being user friendly. Throughout the process of preparing this thesis, we have experimented on using SELinux but ran into a lot of issues. The instructions on using SELinux seem to fail to produce the required results. It requires a deep understanding of the concept of MAC to build the policy accordingly. Another issue was that some programs when placed under a SELinux policy, do not run or even start at all. It was puzzling to why it is happening when some times all that is done is to prevent communication from taking place on a given port. But the program suffers from other problems created by the policy. Many users decided to use a more friendly, easy to manage applications like AppArmor. As we will see later on in a coming chapter, AppArmor policies are easy to create and manage. In fact when comparing the length of a SELinux policy to an AppArmor one, the convolutions and complications of SELinux become apparent.

2-5-2 Plash

The idea of the sandbox as mentioned earlier is to confine the application and prevent it from accessing parts of the system that it could damage. Plash is a sandbox in the Linux operating system. Plash was introduced to overcome the problems that jailing (Chroot) fails to handle. As we saw Chroot jails the user but nothing is done to handle the running application. Plash adopts the principle of Least Privilege by running the application with the minimum privileges needed to perform the task.
The way Plash works is by sandboxing the application process. Sandboxed processes are not allowed to use Linux kernel file name-based system calls like `open()`. It does the job by disabling the system calls. This is done by placing the sandboxed process into a very minimal jail (Chroot). The sandboxed process has access to the file system but only via a remoter procedure call (RPC) using a socket to a process on the server. For example in the `open()` case, when a process placed in the sandbox wants to open a file it sends the file name with the open request. The process running on the server responds by sending a file descriptor for a file across the socket.

GNU libc is where the system calls happen originally under Linux. What Plash does for the sandboxed process is dynamically link the sandboxed application to a modified version of glibc. In this modified version the restrictions are made to prevent direct system call and enforce an RPC. Some argue that having RPC to open a file like in the example mentioned above, makes the performance drop. However this is true if all communication is done via the RPC but once the sandboxed process obtains the file descriptor of a file it accesses it directly with no mediation required.

**2-5-2-1 Plash Architecture**

Plash limits the application process by having a Chroot environment under a dynamically allocated user ID. Figure 2.2 shows a simple diagram for the Plash architecture. The Chroot has only one executable file exec to start the program running in the process. When it requests an `open()`, it sends the file name across the socket to a server process that returns the file descriptor number which can be accessed. Some times
the running application needs to have another process running at the same time so it uses
the fork() to create a duplicate process and asks the server to create another socket for that
process to use its services. The following table (Table 2.1) from the original Plash
documentation shows what glibc library calls are alerted:

<table>
<thead>
<tr>
<th>Treatment</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intercepted and re-implemented</td>
<td>open, mkdir, symlink, unlink, rmdir, stat, lstat, readlink, rename, link, chmod, utimes, chdir, fchdir, getcwd, opendir/readdir/closedir, getuid/getgid</td>
</tr>
<tr>
<td>Intercepted but re-implemented</td>
<td>1. fork -- duplicates the connection to the server first</td>
</tr>
<tr>
<td>but re-implemented</td>
<td>2. execve -- invokes execve syscall on dynamic linker directly</td>
</tr>
<tr>
<td>using the original system call</td>
<td>3. connect, bind, getsockname -- changed for Unix domain sockets</td>
</tr>
<tr>
<td></td>
<td>4. fstat -- changed for directory FDs</td>
</tr>
<tr>
<td></td>
<td>5. close, dup2 -- changed to stop processes overwriting or closing the socket FD that is used to communicate with the server</td>
</tr>
<tr>
<td>Not Intercepted</td>
<td>read, write, sendmsg, recvmsg, select, dup, kill, wait, getpid (and others)</td>
</tr>
</tbody>
</table>

Table 2.1: The list of glibc calls that modified or kept the same as in the Plash documentation
2-5-2-2 Problems with Plash

Plash, just like any sandbox tool has it’s issues and problems. The following are some of these problems

Problem 1: Directory file descriptors vulnerability.

As we know, file descriptors may refer to a directory in Unix-like systems and this is considered as potentially high risk. When the server responds to a client's `open()` request it returns a real, kernel-level file descriptor for a directory through the socket. A client
might exploit these real, kernel-level file descriptor permissions to break out of its Chroot jail by using \texttt{fchdir()} system calls resulting in the client having access to the user's file system which it is not supposed to have.

In [Seaborn07] there is a suggested solution to the directory file descriptors problem as follows:

- Virtualize the full file descriptors, in this case all \texttt{glibc} system calls involving file descriptors are intercepted and replaced. This solution is somehow risky. First, it needs many modifications due to the fewer file descriptors related calls; thus the performance will be affected due to the confusion resulting from knowing what bits of code use real kernel file descriptors and which ones use virtualized file descriptors. In addition, if \texttt{glibc} fails to replace one of the file descriptors related calls, the result may end up using other file descriptors that might be already used in some operations and the reason for that is the file descriptor would be treated as a real kernel FD number.

- Another suggested solution to the same problem is to have the server return to the client as a response to the \texttt{open()} call a directory in two parts. The first one is to be the real kernel level file descriptor obtained by opening \texttt{/dev/null} and the second one is a reference to a \texttt{dir_stack} object to represent the directory. It returns the first part to the client but stores the second part object in a table maintained by \texttt{glibc}. So when \texttt{fchdir()} is used in Plash it checks this table and the only time it can work is if there is an entry for the given file descriptor in that table. This way the suggested solution makes sure that the file descriptor number
stays allocated from the kernel thus overcoming the file descriptors problem.

**Problem 2:** Plash doesn't have the ability to block network access. So to prevent a process from network vulnerabilities we should void `connect()` and `bind()` system calls.

**Problem 3:** The mechanisms of the Plash sandbox do not limit the system CPU and memory resource which might opens the door for denial of service attacks.

**Problem 4:** The construction of the modified `glipe` library is somewhat difficult.

### 2-5-2-3 Plash vs Chroot

Plash is similar in its functionality to Chroot. As mentioned previously, Chroot runs the program in a different file name-space. However Chroot has limitations. For instance creating a new name-space is limited with Chroot. All needed files have to be in the Chroot environment or hard linked to them. In addition Chroot does not allow for read or write access to individual directory entries. Another issue with Chroot is it has a static environment. Such an environment requires a new setup for each program running in it deleting the old files and directories to create the new one. One more problem with Chroot is that it has to be called by root and this is a problem. Many of the Linux distributions lately started to disable root access like in Ubuntu for better security. You may have the user as administrator but still they would need root access to do some tasks. In addition some programs are not allowed to work under root like Google chrome under
Ubuntu Linux. This goes to show how important it is not to have root access when it is not necessary.

Plash uses the Chroot environment to take control of the authority a process has but uses file descriptor passing to give limited authority back to the process. Plash moves the interpretation of file-names so it is done in the user space. Directories can be implemented in the user space. This way we have flexibility in creating file name-spaces.
Chapter 3- current framework design

3-1 AppArmor

AppArmor is one of the famous access control mechanisms used in Linux. It is integrated in the kernel as part of its design. However it has to be activated in order for it to function and enforce its policies. It is similar to previously discussed mechanisms in implementing the least privilege principle to confine an application. It was first introduced in [Cown00] to be an alternative to other complicated existing models to provide a security for Internet servers having problems related to the use of root on the server platform.

What makes AppArmor distinguishable from the other models is its flexibility in design and implementation. AppArmor defense depends on small implementations and simple operations. It is a path name-based system so in order to confine an application the full path of that application has to be specified.

Having this property makes it easy to deploy AppArmor without having to worry about the model failing because even though the profiles are the same, the path names differ depending on where they run. If it happens to be a profile for a
known application like for example Firefox then probably the path name will remain the same even if the same policy is used across multiple machines.

As seen previously in the case of Chroot, it confines the user, which has its problems. For example one of Chroot's problems is that you need a policy for each user. In AppArmor the confining is done to the application, which is better because eventually the security policy needs to be enforced to protect the system from a running application regardless of who is running it. It is better for distribution because if you have an application running on a server you just need to confine it, not the users who are using it. This leads to the question what is to be profiled under AppArmor? The answer to such a question is based on the type of application. AppArmor is originally intended for use on servers because it is simple to restrict the path an application can follow to access other parts of the system. So for this reason there is no way to profile users under AppArmor only applications because as we saw in the case of Chroot it has a lot of issues when confining users. Therefore no concept of role exists in AppArmor.

A user might not have direct access to certain resources in the system; but if the program they are using does, it should be confined using AppArmor to avoid issues where a malicious user directs a privileged program to alter a system file. Also any web application that could potentially invoke a program like java script that is known to have security threats should be thus confined. Another example: network applications that can open ports that invite attackers to use such
communication ports to attack the system. All of these types of applications can be confined using AppArmor. It succeeds by blocking unauthorized applications access to the appropriate paths.

3-2 AppArmor Architecture

AppArmor is implemented in the Linux Kernel so it is launched with the kernel at boot time. Its defense mechanism is based on Mandatory Access Control (MAC). Figure 3.1 shows the AppArmor architecture taken from [sergsoft08]. The AppArmor architecture places the restrictions on applications not users to make the profile simple, flexible and distributable. It confines an application to a specific domain and associates it with limited operations. The domain, Profile, specifies to the application what system files if any are allowed to be accessed. In addition it blocks or allows access to other files and directories in the system. That access can be in terms of read, write or read and write at the same time. Table 3.1 shows all access types that can be used in an AppArmor profile.
The profile language is somewhat flexible and easy to audit. It is loaded into the kernel when the system starts (boots) and in this state the profile has to be in one of two executing modes:

- **Enforcement mode**: Any profiles loaded in this mode will be enforced and any violation will be reported and blocked.
- **Complain mode**: Any profile loaded in this mode will not be enforced but rather will report any profile violation.
<table>
<thead>
<tr>
<th>File</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>read</td>
</tr>
<tr>
<td>w</td>
<td>write</td>
</tr>
<tr>
<td>l</td>
<td>link</td>
</tr>
<tr>
<td>k</td>
<td>File locking</td>
</tr>
<tr>
<td>a</td>
<td>File append</td>
</tr>
<tr>
<td>IX (Inherit )</td>
<td>The child profile is Staying in the same (parent's) profile.</td>
</tr>
<tr>
<td>UX (Unconstrained )</td>
<td>EXECutes the program without a profile.</td>
</tr>
<tr>
<td>Px (Profile )</td>
<td>Requires that a separate profile exist for the executed program.</td>
</tr>
</tbody>
</table>

Table 3.1: Example of some options for files or directories in the AppArmor profile

AppArmor can function in any of the following policy models:

**Targeted:** It is the standard way in which AppArmor is deployed. It is a compromise to the security of a system having a total lock down for ease of use and policy authoring. It does that by identifying a set of applications with high value and at the same time it is easy to confine these applications instead of having to do it for the entire system and all of this while providing the needed level of security. Most applications in this mode are network applications.
- **Total System**: This is the opposite of Targeted where in it all applications in the system are confined. This is somehow restrictive and a lot of issues can arise from having this model like lack of flexibility.

### 3-3 AppArmor Advantages

- Easy to use and flexible.

- Since AppArmor is a path-based system, diagnosing and fixing profile bugs is easy.

- AppArmor can disable certain profiles instead of having to disable the entire system.

- Profiles are simple text files and are easy to audit.

### 3-4 AppArmor disadvantages

- Since it is based on MAC it has the same problems.

- It fails to reduce the risk of attacking the kernel.

- Having any vulnerability exploited at the kernel level can lead to a break down of the AppArmor protection.

- The defined policy profile can weaken or restrict the policy. It is up to the administrator to create the policy so if the developer overseas any part it it might cause the protection to fail.
3-5 AppArmor VS SELinux

After looking at the AppArmor and SELinux models, it is essential to compare both models. The comparison is divided into:

- **Deployment:** Both use MAC. However in SELinux it uses the labeling security mechanism. SELinux places a tag on each file and application to determine what can access what based on the level of security. Such approach is difficult to implement and analyze so the entire system has to be relabeled and application settings have to be analyzed. On the other hand AppArmor is based on the concept that no relabeling has to happen which makes it easier to use. It uses the path of files and directories to confine the application to that path only.

- **Ease of Use:** SELinux requires the use of the command line. Although there is a GUI application, it is not easy to use. Since it requires the command line it requires special skills to use the tools correctly and build the policy accordingly. In contrast, AppArmor is easier to use. You can have a GUI, which can be used to analyze and create profiles. At the same time using the command line is not difficult because the syntax of the policy profile is easy to understand: it follows what users use under a Linux operating system. A problem with AppArmor is that profiling multiple applications is difficult. It is time consuming and difficult to the level compared to SELinux. The good thing is that if the user wants to have profiles for widely used applications they can use them but recreating profiles from scratch can still be time consuming.
- **Overhead:** AppArmor has lower overhead when compared to SELinux.

- **Policy Size:** The length of an AppArmor policy is short compared to the one in SELinux even if both do the same thing to the same application.

- **Web Servers:** Both work well in server environment but AppArmor is better than SELinux. It can place individual process in sub confinement by issuing individual polices for each process running on the server. AppArmor is easy to work with if the number of applications is small. SELinux has management problems making it difficult to work with. But if the goal is to protect an entire server SELinux is the option to follow but it requires highly skilled individuals to write the policy and enforce it.
Chapter 4: The Implementation

The implementation takes advantage of the AppArmor tool to increase the security of the applications. We intend to augment the usefulness of AppArmor by further restricting the permissions of the files accessible to the process using the Parent-Child mechanism. The implementation will be in two parts. First confining the application process in a specific directory (my_tmp) and the second part is to add further restricted permissions to the application itself using an AppArmor profile.

4-1 First part: Confining the application process using Parent-Child mechanism

In this part the application process is confined to run in a temporary directory (my_tmp) instead of its original directory. The Parent-Child mechanism is used to do this step where this mechanism is used to perform the system call when there is a relationship between two processes where the parent is the original process that spawned the child process. The child can also spawn its own children.

This method proved to be useful due to having simple control of memory management. It also reduces the risk of having memory leaks. When delete takes place only the objects
created get deleted and the child with no parent. At the same time if a child is deleted prior to its parent then the object will be removed automatically from the parent's list of children.

The program takes the entered application as its first input from the command line. Optionally, the user can also specify the directory in which the application needs to be run as a second argument in the command line. If this is not specified the application will be run from the "$HOME/Desktop/my_tmp" directory as a default directory to execute such applications. Then, the main program does validation for arguments and checks for the presence of the application and the temporary directory. If there are no errors then it spawns a child process via `fork()` system call. The child process will change the directory to the specified directory via the `chdir()` system call and calls the `execvp()` to execute the application. The application will run in the given directory and produces any write/read calls to/from this directory. The parent process just waits until the child process (application) completes running and exits once the child is finished. The parent process also prints the status to the screen.

The application (child process) is processed as follows:

- Gets the current directory before changing to the new directory.

- Builds the absolute path/relative path for the application depending on what application is entered in the command line.

- Changes directory to the given directory using the `chdir()` system call.

- Calls the `getcwd()` to get the current directory which outputs to the screen from
which directory the application will be launched.

- Calls the `execvp()` to execute the application

In the meantime, the parent process will wait until the child process finishes the job by calling the `waitpid()` system call, and once the child exits, it prints the child status on the screen and exits. Figure 4.1 shows the flow chart for this part.
Figure 4.1: flow chart for part one
4-2 Second part: Add AppArmor profile

Since the first part of the implementation is to force the process of an application to run in a pre-specified temporary directory, the second part is to add further restrictions to the application by creating a profile in AppArmor to confine and run the application inside of a directory in which it is isolated from other parts of the system.

The Firefox browser has been chosen as a target application to be confined within a previous pre-specified temporary directory in part one. To create a Firefox profile we have first to define what the privileges of Firefox would be as an application. For example which file system can or can not be accessible? Which directories can the user use for downloading or uploading? Are there any other applications that Firefox interacts with? And if yes how do we confine them?

Thus, the intention of the Firefox profile in these implementations is to restrict the browser as much as possible without breaking the browser functionality and confining all applications that Firefox may use as third party applications such as Java applications or document viewers by creating a sub-profile or separate profiles for them. In this case, we avoid reserved (unconfined execute) processes which might be used to penetrate the system. The danger of an unconfined execute is that the attacker can exploit the vulnerabilities in unconfined child processes to break through the isolated directory and then get access to system files or other confidential information.

The Firefox profile in this implementation has the following privileges:
- The user can download and upload from the same isolated directory (my_tmp) that had been defined in part one. So, most the file system is protected such as Pictures, Documents...etc.

- The browser has the ability to run Flash videos and there is a sub-profile to confine the Flash plug-in in the same Firefox profile.

- The browser can use the (qpdfview) plug-in to open PDF files directory and it is confined in the same Firefox profile.

- The browser can open and play video and audio using the (SMPlayer) multimedia player. It gives the user various options for running and controlling multimedia.

In the end by combining the two steps mentioned above the system would have two layers of protection by having a policy and further confining the running application to run inside of a directory in isolation from other parts of the system. The Firefox profile and the sub profile are listed in the appendix

**4-3 Program Test**

Figure 4.2 shows how to confine the Firefox browser in a certain directory. The Firefox (child process) is moved to a temporary directory (/root/Desktop/my_tmp) to run in isolation; if it fails to move the process then it will return 1 and an error message at the same time the program will terminate. No launch of Firefox will take place. You can see that it lists the default directory where the process is originally running and it did move it to the new directory.
Figure 4.2: confine Firefox browser in (my_tmp)

Figure 4.3: Using the process number to see where it is executing
Using the Linux command line we can see in Figure 4.3 that the process is indeed executing from the (/root/Desktop/my_tmp) directory which means that we have the process confined in the located directory so our sandbox works as required.

Figure 4.4: Denying the access to Desktop files

In the Apparmor part there is a custom profile for Firefox. This profile is restricting the access to most system parts such as desktop, downloads, pictures, etc. The user can download and upload from the same isolated directory (my_tmp) that we had in part one. Figure 4.4 shows the message of denying the access to Desktop files when the user tries to access the Desktop.
Chapter 5: Conclusions and future work

For the future, the work conducted in this thesis was done at the application level. A lot of the problems that these mechanisms suffer from are because of the underlying access models used. Future work would be to take one of these models and work on them at a theory level to further improve the model and come up with an enhancement to it. Looking at what causes some of it to fail in properly handling the principle of least privilege and propose a solution to this failure.

In this thesis we explore access control mechanisms in Linux OS; we started by examining Chroot. The problems that Chroot has like only confining the user not the applications have made system administrators not go with that option. Also Chroot usage in large systems would mean having a policy for each single user; this would cause management problems which would not be easy to handle. Then the underling access control methods were presented; they were mainly the DAC and MAC. There were other access control methods like the RBAC and DTE but for this thesis study the focus was in a detailed description of DAC and MAC.

The concept of the sand-box was introduced where we run an application but with the concept of least privilege in mind. So the application has access to what it needs to function properly while isolating it from important parts of the system.
The alternative models for access control were presented in the sense that they cover what Chroot lacked. When Plash was introduced, it has a modified *glibc* that controls certain system calls and programming in this level is not easy to work with for the non specialized user.

SElinux confines an application so it would restrict the application to work with only the associated labeled files that meet the security level. SElinux proved to be difficult to work with but it achieves the required level of security when used on servers running applications and large scale systems.

AppArmor was introduced as a path based system; it confines the applications to a certain number of files they have access to. AppArmor is easier to work with and understand but if we have a large number of applications it could be hard to profile each one of them.

Finally this thesis introduced a double layer of security; we created a sandbox to confine applications running processes to work from a certain directory. This way we sandboxed the application process so it cannot have access to other resources in the system outside this directory.

In addition, the second layer of security was added: an AppArmor profile to further confine the application so we will have a double layer of restrictions. This was tested and the results were presented in the previous chapter.

From what has been tested so far from all models each one of them has strong and weak points. Having a hybrid system that takes the advantages of two or more modules to overcome the issues on one of them is probably the best security solution. This is the aim
of having the sandbox and having policy profiles at the same time working together for a better level of security.
Appendix A

Firefox profile

#include <tunables/global>
/usr/lib/firefox/firefox {
    #include <abstractions/audio>
    #include <abstractions/base>
    #include <abstractions/browser_openjdk>
    #include <abstractions/dbus-session>
    #include <abstractions/fonts>
    #include <abstractions/nvidia>
    network inet dgram, 
    network inet stream, 
    /bin/dash rix, 
    /bin/grep rix, 
    /bin/ps Cx, 
    /etc/adobe/mms.cfg r, 
    /etc/firefox/syspref.js r, 
    /etc/gai.conf r, 
    /etc/gnome-vfs-2.0/modules/ r, 
    /etc/gnome-vfs-2.0/modules/* r, 
    /etc/gnome/defaults.list r, 
    /etc/host.conf r, 
    /etc/hosts r, 
    /etc/lsb-release r,
/etc/mailcap r,
/etc/mime.types r,
/etc/nsswitch.conf r,
/etc/passwd r,
/etc/resolv.conf r,
/etc/xul-ext/ubufox.js r,
/run/resolvconf/resolv.conf r,
/sys/devices/system/cpu/present r,
/tmp/ r,
/tmp/* mrwk,
/tmp/icedteapugin-*/ w,
/tmp/icedteapugin-*/[0-9]*-icedteanp** rw,
/tmp/orbit-*- w,
/tmp/plugtmp/ rw,
/tmp/plugtmp/* w,
#qpdfview & smplayer profiles are seperated.
/usr/bin/qpdfview Px,
/usr/bin/smplayer Px,
/usr/lib/firefox/plugin-container rix,
/usr/share/ r,
/usr/share/applications/*.desktop r,
/usr/share/applications/mimeinfo.cache r,
/usr/share/glib-2.0/schemas/gschemas.compiled r,
/usr/share/hunspell/ r,
/usr/share/icons/ r,
/usr/share/icons/** r,
/usr/share/libthai/* r,
/usr/share/mime/ r,
/usr/share/mime/** r,
/usr/share/mozilla/extensions/*/ r,
/usr/share/pixmaps/ r,
/usr/share/themes/** r,
/usr/share/xul-ext/ubufox/ r,
/usr/share/xul-ext/ubufox/** r,
/var/tmp/ r,
owner /{run,dev}/shm/pulse-shm* k,
/{run,dev}/shm/pulse-shm* rw,
@{HOME}/.ICEauthority r,
@{HOME}/.Xauthority r,

# Allow downloading files to /Desktop/my_tmp and uploading from /Public
@{HOME}/Desktop/my_tmp/ r,
@{HOME}/Desktop/my_tmp/** rw,
@{HOME}/Public/ r,
@{HOME}/Public/** r,

#to confine the smplayer multimedia
owner @ {HOME}/.cache/smplayer/plugin/smplayer* rw,
@{HOME}/.config/dconf/user r,
@{HOME}/.config/ibus/bus/ w,
@{HOME}/.fontconfig/* r,
@{HOME}/.icons/ r,
@{HOME}/.local/share/ r,
@{HOME}/.local/share/applications/mimeapps.list r,
@{HOME}/.local/share/applications/mimeinfo.cache r,
@{HOME}/.local/share/icons/ r,
@{HOME}/.local/share/icons/**/ r,
@{HOME}/.local/share/mime/ r,
@{HOME}/.local/share/mime/** r,
owner @{HOME}/.local/share/recently-used.xbel* rw,
@{HOME}/.macromedia/Flash_Player/#SharedObjects/ r,
owner @{HOME}/.macromedia/Flash_Player/#SharedObjects/** rw,
@{HOME}/.macromedia/Flash_Player/macromedia.com/support/flashplayer/sys/** r,
@{HOME}/.mozilla/** r,
owner @{HOME}/.mozilla/firefox/*.default/** rw,
@{HOME}/.nv/GLCache/ r,
@{HOME}/.nv/GLCache/** rwk,
@{HOME}/.pulse-cookie rwk,
owner @{HOME}/.{firefox,mozilla}/**/*.{db,parentlock,sqlite}* rwk,
owner @{HOME}/.adobe/Flash_Player/* w,
@{HOME}/.adobe/Flash_Player/AssetCache/ r,
@{HOME}/.adobe/Flash_Player/AssetCache/** rw,
@{HOME}/.cache/dconf/user rw,
@{PROC}/[0-9]*/cmdline r,
@{PROC}/[0-9]*/fd/ r,
@{PROC}/[0-9]*/mountinfo r,
@{PROC}/[0-9]*/mounts r,
@{PROC}/[0-9]*/net/dev r,
@{PROC}/[0-9]*/status r,

profile /bin/ps {
    deny capability sys_ptrace,
    /bin/ps r,
    /dev/tty r,
    /etc/ld.so.cache r,
    /lib/libproc-*.so mr,
    /lib/x86_64-linux-gnu/ld-*.so r,
    /lib/x86_64-linux-gnu/libc-*.so mr,
    /sys/devices/system/cpu/online r,
    /usr/lib/locale/** r,
    /usr/lib/x86_64-linux-gnu/gconv/gconv-modules.cache r,
    @PROC/ r,
    @PROC/[0-9]*/cmdline r,
    @PROC/[0-9]*/stat r,
    @PROC/[0-9]*/status r,
    @PROC/[0-9]/cmdline r,
    @PROC/[0-9]/stat r,
    @PROC/[0-9]/status r,
    @PROC/meminfo r,
    @PROC/stat r,
    @PROC/sys/kernel/pid_max r,
    @PROC/tty/drivers r,
    @PROC/uptime r,
@{PROC}/version r,

}

}

**smplayer sub-profile.**

# this profile to confine video file on the web using smplayer multimedia player.

#include <tunables/global>

/usr/bin/smplayer {
    #include <abstractions/audio>
    #include <abstractions/base>
    #include <abstractions/dbus-session>
    #include <abstractions/fonts>
    #include <abstractions/nvidia>

    network inet dgram,
    network inet stream,
    network inet6 stream,
    deny /etc/apparmor.d/** r,
    /etc/fstab r,
    /etc/gtk-3.0/settings.ini r,
    /etc/host.conf r,
    /etc/hosts r,
    /etc/smplayer/* r,
    /etc/nsswitch.conf r,
    /etc/passwd r,
    /etc/resolv.conf r,
/home/*/** r,
/home/*/.config/ibus/bus/ w,
/home/*/.cache/dconf/user rw,
/home/*/.local/share/recently-used.xbel rw,
/home/*/.local/share/recently-used.xbel.* rw,
/home/*/Desktop/my_tmp/ r,
/home/*/Desktop/my_tmp/** rw,
/media/** r,
/run/resolvconf/resolv.conf r,
/sys/devices/system/cpu/ r,
/tmp/smplayer* rw,
/usr/bin/smplayer mr,
/usr/bin/mencoder rix,
/usr/bin/smplayer rix,
/usr/lib/codecs/*.so mr,
/usr/lib/x86_32-linux-gnu/gtk-3.0/*/immodules/*.so mr,
/usr/lib/x86_32-linux-gnu/pango/*/modules/pango-*.so mr,
/usr/share/X11/XErrorDB r,
/usr/share/glib-2.0/schemas/gschemas.compiled r,
/usr/share/gvfs/remote-volume-monitors/ r,
/usr/share/gvfs/remote-volume-monitors/* r,
/usr/share/icons/ r,
/usr/share/icons/** r,
/usr/share/mime/mime.cache r,
/usr/share/pixmaps/ r,
/usr/share/terminfo/u/unknown r,
/usr/share/themes/** r,
owner /{run,dev}/shm/pulse-shm* rk,
/{run,dev}/shm/pulse-shm* w,
Appendix B

Setup AppArmor framework

AppArmor is implemented in the Linux Kernel so it is launched with the kernel at boot time. It confines an application to a specific profile and it associates it with limited operations. AppArmor is integrated in the default install in Ubuntu and openSUSE distributions. The profile is a text file that contains the sets of file permissions. It is stored in the directory called `/etc/apparmor.d/`. It is loaded into the kernel when boot starts and in this state the profile has to be in one of the two executing modes:

- Enforcement mode: Any profiles loaded in this mode will be enforced and any violation will be reported and blocked.

- Complain mode: Any profile loaded in this mode will not be enforced but rather report any profile violation.

To create a profile there are two main methods that can be used to create AppArmor profiles, and that is depending on the type of distribution used. The first method is the command line. This method requires knowledge of Linux commands and using terminal windows. The second method is the graphical YaST interface (YaST GUI); this type is provided in openSUSE distributions.
Creating a profile using the command line

1- open the terminal and type `aa-genprof` command to start generating the profile. You have to specify the full path of the application. In this tutorial the path of the application is `/usr/bin/gimp` where the gimp is the image edit application.

```
sudo aa-genprof /usr/bin/gimp
```

2- The application will be set into complain mode. Then start the application in another window and exercise the application functionality, once you finish close the application and type `S "Scan"` in the terminal in order to scan the system logs for AppArmor events.
3- In this step for each event you have to answer the series of generated questions regarding deny access to a specific resource or to define execute permissions for an entry and you have to select the answers from the provided options in the terminal. Table 1 shows these options.

4- Type S to save the rules which applied in previous steps.

5- Set the profile in enforce mode to apply the rules to the application by using the following command: sudo aa-enforce /usr/bin/gimp
Creating a profile using YaST GUI

1- From the main menu open the YaST2 control center then chose AppArmor Configuration under Security and users section.

2- Chose Generate Profile from Available AppArmor Modules.
3- Enter the full path of the Application that you want to confine or click on **Browse** to navigate the application path then click **Next**. In this tutorial the used application is **GIMP**, it is application for image editing.

![Application to Profile](image.png)

4- Open the application normally in another window and exercise its functionality as much as you can then select **Scan system log for AppArmor events**.
5- In this step the AppArmor profile wizard generates a series of questions regarding deny access to a specific resource or to define execute permissions for an entry and you have to select the answers from the provided options in the wizard. Table 1 shows these options.
6- Once finishing from the step 5 click **Finish** to save the profile.

### Table 1: Permissions options

<table>
<thead>
<tr>
<th>opposition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allow</td>
<td>Allowing the program to accessing the specified directory path entries.</td>
</tr>
<tr>
<td>Deny</td>
<td>Prevents the program from accessing the specified directory path entries.</td>
</tr>
<tr>
<td>Glob</td>
<td>Modify the directory path to include all files in the suggested directory.</td>
</tr>
<tr>
<td>Glob w/Ext</td>
<td>The program can access all files in the suggested directories that end with the specified extension.</td>
</tr>
<tr>
<td>Abort</td>
<td>Losing the entered rules for the profile.</td>
</tr>
<tr>
<td>Finish</td>
<td>Save the entered rules for the profile.</td>
</tr>
<tr>
<td>Inherit</td>
<td>The child inherits the parent's profile and get same access controls as the parent profile.</td>
</tr>
<tr>
<td>Profile</td>
<td>There is a separate profile for the child program which it is must be loaded into the kernel and if not the child going to get permission denied.</td>
</tr>
<tr>
<td>Unconstrained</td>
<td>Executes the program without a profile.</td>
</tr>
</tbody>
</table>
References


Detection Derived from MAC Policy, IEEE, 5-9 June 2011, Page(s): 1 – 6


[Li14] Z Li, H Cai, J Tian, W Chen, Application Sandbox Model Based on System Call Context, IEEE, April 2010, Page(s): 102 – 106


[Richard96] S Richard, Mandatory protection for Internet server software, IEEE, 1996,
[rookcifer12] rookcifer, Custom Firefox AppArmor Profile For Ubuntu 12.04


[Xu10] X Xu, C Xiao, C Gao, G Tian , A study on confidentiality and integrity protection of SELinux, IEEE, 11-12 June 2010 , Page(s): 269 – 273