An Approach to Ensuring System Integrity in the Presence of Untrusted Applications

A Dissertation Presented
by

Gaurav Poothia

to

The Graduate School
in partial fulfillment of the
Requirements
for the degree of

Master of Science
in
Computer Science

Stony Brook University
August 2006

Stony Brook University

The Graduate School

Gaurav Poothia

We, the thesis committee for the above candidate for the

Master of Science degree,

hereby recommend acceptance of this thesis.

Dr. R. C. Sekar, Thesis Advisor,
Computer Science Department

Dr. Scott Stoller, Chairman of Thesis Committee,
Computer Science Department

Dr. Erez Zadok, Comittee Member,
Computer Science Department

This thesis is accepted by the Graduate School.

Dean of the Graduate School

Abstract of the Thesis

An Approach to Ensuring System Integrity in the Presence of Untrusted Applications

by

Gaurav Poothia

Master of Science

in

Computer Science

Stony Brook University

2006

The threat to the integrity of a host from untrusted applications is a serious one. This has become more relevant in the personal computing age where untrusted and trusted applications end up running on the same system.

This thesis has a dual objective. The first objective is to survey existing approaches for policy-based confinement. We examine Systrace, SELinux, Model Carrying Code (MCC) and Multi-Level Security (MLS) as part of this survey. Our second objective is to take the best elements from these approaches and design an approach that has the potential to preserve system integrity without unduly impacting usability of (trusted and untrusted) applications on the protected host. The key technical contribution of the approach is the development of a methodology that uses past file accesses made by these applications to automate most aspects of generating policies that would ensure overall system integrity, while permitting almost all these accesses. In our implementation, we investigated the feasibility of this approach in the context of a recent Linux distribution.

In addition to preserving system integrity we also try and solve some practical issues and enhance the usability of the system. We invent a methodology for automating the assignment of trust labels to all the applications on a system. By doing so we in fact solve a problem shared by other approaches. We also address the issue of maintaining integrity of file names which is distinct from the integrity of the contents a file. Finally we present a way of protecting the integrity of untrusted applications when attacked by other untrusted applications.

Table of Contents

List of Figures

ACKNOWLEDGEMENTS

I would like to thank Dr. R. Sekar for expecting me to analyze ideas more carefully than what I was used to. This research has been possible and enjoyable as a result of that expectation.

I would also like to thank Dr. Scott Stoller and Dr. Erez Zadok for taking the time and effort to be on the committee.

Thanks to all my labmates for the many lighter moments we've shared. Sandhya for simply being there. Ma and Papa for making it all possible.

Chapter 1

Introduction

Users today find themselves running untrusted applications on the same system where they run security-critical applications. The goal of this thesis is to develop practical ways to protect security-critical apps from untrusted apps and do so without compromising system usability. We do not consider it feasible to run untrusted apps on a separate virtual machine, or other solutions that require users to "switch contexts" in order to use different applications.

Our approach to solve this problem falls under the broad category of Policy-based confinement. In such an approach the policy protects the system's resources from any damage the untrusted application might cause.

Below we define the broad goals of the thesis:

- Our first goal is to compare existing approaches for policy based confinement against the above threat model. We provide insights into the strengths and weaknesses of the following infrastructures
  - Systrace
  - SELinux
  - Model Carrying Code (MCC)
  - Multi-Level Security (MLS)
- Our second goal is to propose a new approach that attempts to combine the useful elements from the existing approaches and overcome some of their important limitations. The focus is on minimizing usability problems associated with confinement policies as well as provide extensibility to confine a new untrusted application without having to first figure out its behavioural patterns. Automation of policy generation is the final goal. We have not achieved complete automation but reduced the need for human judgement in policy creation. We try and find a good trade off between reducing human intervention

and providing strong security assurances against the threat from untrusted applications. It is our hope that our approach can ultimately lead to automated policy generation.

Chapter 2

A Critique of Related Security Policy Infrastructures

In this section we trace our investigation of relevant security infrastructures which do policy based sandboxing. In the first section we discuss them as a survey of the infrastructures that address the same threat model we are trying to protect against. In the second section of this chapter we try and share the insights we gained about each. We present our learnings in a discussion of which elements, from each of these approaches, is best suited for solving the problem. We end the chapter by motivating a new approach that combines the useful elements identified in our critique.

2.1 Overview of Policy Infrastructures

2.1.1 Systrace

Systrace [11] is a system designed to confine applications based on system call monitoring. The policy is essentially an enumeration of system call arguments that are deemed safe and is completely independent of operating system permissions. The protection it offers is based on the observation that system calls are the means to make privileged kernel operations and by monitoring and restricting system calls malicious intent can be thwarted. This observation has been used in security research earlier [5] [7]. If the systrace policy is to deny a system call it can choose the error code returned by the call. This gives the programs a chance to handle the problem gracefully.

There are two possible approaches to developing systrace policies. One possibility is to record the system calls made by the application during the **training run**. As the application proceeds the system calls are translated into policy statements for that application.This approach is only applicable if both the application can be trusted in the first place to allow a training run and the training run captures most if not all the execution paths. When these preconditions are not satisfied systrace depends on **interactive policy generation**. Here the application is run against a generic policy template and any system call that violates the template is presented to the user to accept and add to the policy or deny. This approach to policy definition is iterative and involves the user actively because it relies on her to relax the deviations from policy as the application proceeds.

Systrace finds another utility for the system call monitoring approach. This is the idea of *privilege elevation* which eliminates the need for programs to obtain root privilege, by means of the setuid option, to execute a few privileged instructions. So systrace simply elevates the privilege of the process when required by allowing the system call in question. If compromised, the application can only be used to do a handful of privileged operations versus the earlier scenario where it had *all* the privileges of root.

2.1.2 Security Enhanced Linux(SELinux)

The Security Enhanced Linux(SELinux) project implements MAC for Linux. This approach tries to secure the operating system [9] itself rather than provide application security in user space. The scope of the solution goes beyond protecting the system from untrusted applications, to protect the system from compromised root processes[15], which are likely to be trusted applications. The way it achieves these two objectives [8] is by giving the policy author abstractions using which she can restrict privileges available to executing processes or in other words by enforcing the principle of least privilege. By providing strong separation of applications it can achieve both safe execution of untrusted applications as well as limit the potential for damage when a system service or application is compromised.

It focuses on providing an administratively-defined security policy that can control all subjects and objects, basing decisions on its security model. The basic security model used is Type Enforcement (TE)[1]. TE is aimed at providing fine-grained control over what resources are accessed by processes by giving type labels to both subjects and objects.

We attempt to explain the SELinux framework by explaining the concepts behind Flask, the MAC architecture, of which SELinux is an implementation. We then explain the Type Enforcement (TE) model which is the primary security model for SELinux. Finally we look at the SELinux policy language and the policy goals.

Flask Concepts

Flask [14] is the operating security architecture that SELinux implements. So it is important we look at the concepts behind this architecture. The basic idea is that every subject and object (e.g. file, socket, device, IPC object etc) is given a security context. The security context is the security relevant information about the subject or object in question. What the security context consists of is left unspecified. It depends on the security model used by the particular implementation. In the case of SELinux the primary security model is Type Enforcement. In any case the security context is only understood by the security server.

To provide efficiency, the policy enforcement modules in SELinux deal in security identifiers(SIDs) because the security context can be of arbitrary size. The security server has a mapping of SIDs to security contexts available to it on the fly.

Each access needs security clearance for which the policy enforcement part sends the SID of the subject and the object to the security server for a decision. It also sends along the object security class which basically tells the server the kind of object being accessed. The server takes these three inputs and makes a decision. There are a couple of different security decisions referred to the server: **labeling decisions and access decisions**.

The **labeling decisions** come into play when the security context of a new subject or object needs to be decided. The security context for a new object is based on the security context of the subject creating it and the security context of a related object, like the directory in which a file is created. The security context of a new subject is based on the security context of the parent process and the program which is executed.

In SELinux the files security contexts are specified at init time by grouping them into types (types are explained in the section on TE) based on regular expressions over path names. The labels thus assigned are stored as extended attributes in the filesystem.

The **access decisions** are used to allow or deny based on the two SIDs and the object class. The object class has a fixed set of operations allowed on it. The result of the access decision is an access vector that represents which of the operations are allowed and which are denied.

Type Enforcement

The SELinux security model is a combination of Type Enforcement, Role-Based Access Control(RBAC) and Multi-Level Security. The TE model is the dominant security model responsible for providing fine-grained control over both processes and resources.

TE traditionally associates a type with each object and a domain (a specialized term for types associated with processes) with each subject. The TE model treats all processes in the same domain as equivalent and all the objects with the same type as equivalent. A TE access matrix decides which domains can access which object types. Additionally it also specifies what types can be executed by each domain and conversely which programs can be used as entry points into a domain. SELinux differs from traditional TE in that here security class information is also used. For instance the policy does distinguish between a regular file from a device file while making a decision.

We have already explained the input to the security server when it needs to decide the type for a new subject or object. However if there is no special rule for the given set of inputs then the following defaults that come into play. For new subjects they simply take the type of the parent. For new objects the type of the related object, like parent directory, is taken. The related object varies based on the security class of the new object.

Policy Language

We do not provide a formal or comprehensive description of the policy language here. Instead we introduce the various possible TE statements and provide an informal treatment of what they mean. The various TE statements are:

Attribute Declarations: An attribute of a type is used to identify a set of types with a similar property. The set of attributes and set of types share a many-to-many mapping. Examples of attributes include `daemon` which is self-explanatory, `privlog` for domains that can communicate to `syslogd`, `logfile` for files or directories that exist only for logging purposes etc.

Type Declarations: These are simply the declaration statements for each type in the system. It can be used to specify, in addition to the type declaration, things like aliases and attributes of the type.

Transition Rules: TE transition rules are used to decide the type for a new object or subject created in the system. The new type in both cases depends on a source type, a target type and the security class. For a process the source type is the current domain, the target type is the type of the executable. For an object the source type is the type of the parent process and the target type is the type of the related object. In case of a file the related object is the parent directory and its type is the target type. For example consider the transition rule:

```
type_transition initrc_t sshd_exec_t process sshd_t
```

Here a process running in the `initrc_t` domain, the `init` program for instance, will transition to `sshd_t` on executing a file with the type `sshd_exec_t`

TE Access Vector Rules: Such a rule is at the heart of the TE model. It specifies a set of permissions based on a type pair and the object security class. It is these rules that constitute the TE access matrix. The flexibility provided is that an Access Vector rule can use a set for any of its parameters i.e. for source and target types as well as for security objects and permissions. For example consider the rule:

```
allow sshd_t sshd_exec_t:file {read execute entrypoint }
```

Here the `sshd_t` domain has the above three permissions on any file with the type `sshd_exec_t`

Policy Goals

Given the number of different types and security classes in SELinux there is a lot of flexibility to express which subjects can access which objects. In order to make things easier for policy authors the infrastructure provides macros for various policy idioms. For example the macro `uses_shlib` gives the domain access to execute the types for the dynamic linker and system shared libraries.

But at this point we answer the question: what are the SELinux policies trying to achieve. The answer is that they are trying to enforce least privilege on each application on the system by leveraging the fine granularity the infrastructure affords the policy author. This achieves a separation of applications that combats the threat from untrusted applications as well as compromised root services.

It is important to note that SELinux policies do not *define* a secure system although it can definitely form a basis for a secure system. However here is no way inside the SELinux infrastructure to state a set of security goals and verify that they have been achieved. For instance there is no way to prove that the integrity of the trusted computing base(TCB) is preserved. There has been research which has tried to answer such questions [6] but those efforts lie outside the infrastructure itself.

2.1.3 Model Carrying Code (MCC)

The Model Carrying Code (MCC) [12] approach requires the untrusted applications to be accompanied by a *model* that captures the security-relevant behavior of the code. This allows the consumer to statically verify the behaviour of the application against consumer-specified security policies. The process is called *verification* and is used to determine *policy satisfaction*. Further there is a provision for runtime enforcement to ensure that the model does indeed capture a safe approximation of the program behavior. The process is known as *enforcement* it is used to determine *model safety*.

Policy Language

The MCC policy language is based on *extended finite state automata* (EFSA) that extend finite state automata by incorporating state variables to remember argument values. For example we can find the file involved in a `read` operation by comparing the file descriptor with the return value of an earlier `open` system call. The file name passed to that `open` call gives us the file being read. MCC uses the BMSL language to express a dual representation of EFSA known as *regular expressions over events* (REE)[13]

Model Generation

In MCC the code producer is responsible for generating the model, which should be usable against all policies the consumer may have. The MCC framework develops a technique to generate EFSA (or EPDA) models by execution monitoring. The technique is fairly sophisticated and here it suffices to state that it manages to learn temporal relationships between arguments of *different* system calls.

Verification

Verification is the process of determining whether the model M of the untrusted application satisfies the security policy P of the consumer. Here M is the EFSA model for program behavior and $\bar{P}$ is the policy automaton. The verification consists of building the product automaton $M \times \bar{P}$, which accept the intersection of the behaviors accepted by each of automatons individually. If there are paths in this product automaton that lead to violating (final) states of $\bar{P}$, then the policy is violated by the model.

Policies

It is important to make a distinction between the MCC infrastructure and policies that will be specified inside this infrastructure. Until now we discussed the infrastructure itself and now we look as the policies proposed for use by the infrastructure.

MCC adds the abstraction of *owned* files for applications. These are file that the application depends on directly to get its job done such as executables, libraries, configurations, cache etc. As a result the application can modify these without possibly compromising any other application. If only we can automate the way the set of *owned* files is derived we have solved a problem we encountered with SELinux, which is the hassle of creating policies for each new untrusted application.

We propose to use the following techniques to find these files.

- **Static Information:** We lookup the Package Management database to create a dependency graph for *all* packages installed in a system. We then find the package which owns the application and find all the packages this package directly or indirectly depends on. We create an *Owned database* for the application which has all the files belonging to any package that it depends on.

- **Dynamic Information:** We determine the per-user configuration files for the untrusted app based on heuristics that guess the config filename. For example user configuration file for `emacs` is `/home/user/.emacs`. We add such files also to the *owned database*

Having created this context for each untrusted application automatically we can now use it to specify broadly two types of policies:

- **Conservative Policy:** This allows the untrusted application to access *only* those files that it *owns*. Good candidates for this policy with respect to the *write* access are untrusted document viewers and media players.

- **Permissive Policy:** This allows the untrusted application to access *any* file in the system as long as it is not *owned* by another application. For the *write* access this policy is suitable for untrusted editors and other applications which need to access different files based on usage patterns.

MCC creates policy templates for groups of applications with similar functionalities. These groups are known as *Application Classes.* Policy for each application class combines conservative and permissive policies for different accesses like read, write, creation and deletion. Some examples of applications classes are

- Multimedia and Document Viewers
- Archivers and Format Converters
- Games and Peer to Peer Applications
- Vulnerability Scanners

Existence of application classes makes it easier for the administrator or the framework to choose the right policy for a new untrusted application.

2.1.4 Multi-Level Security (MLS)

Multilevel Security (MLS) [2][3] has been primarily focused on confidentiality issues faced in military settings based on the Bell-LaPadula model. This approach has met with moderate success in tackling confidentiality issues.

For integrity concerns there are two approaches:

- **Biba's Model (Strict Integrity Model):** What Biba does is assign integrity levels to all subjects and objects in the system. A high integrity level for an object suggests trustworthiness of data and for a subject it suggests correctness of execution. To prevent information information transfer paths from objects of low integrity to objects of high integrity Biba enforces the "no write up" and "no read down" rules.

- **Low-Water-Mark Policy:** Here[4] is an alternative approach to the Strict integrity Model while providing the same integrity guarantees. It retains the "no write up" rule but modifies the "no read down" rule. Instead it allows any subject s with integrity level $i(s)$ to read any object o with integrity level $i(o)$ with the subject assuming the integrity level $min(i(s), i(o))$ after the read.

Policy Goals

In order to understand the assurances provided by MLS we need to define an information transfer path. An [3] **information transfer path** is a sequence of objects o_1 ,..., o_{n+1} and a corresponding sequence of subjects s_1 ,..., s_n such that subject s_i can read o_i and subject s_i can write object o_{i+1}, for all i, $1 \leq i \leq n$.

Both the Strict and the Low-Water Mark policies constrain information transfer path from any object o_1 to any object o_{n+1} such that the integrity label on o_{n+1} is higher than on equal to the label on o_1.

Thus the MLS policies prevent the flow of low integrity data into high integrity objects.

2.2 Key learnings and Motivating a New Approach

We present the key insights we gained about the infrastructures described above. An important analysis in each case is the tradeoff it makes between reducing the need for human intervention and judgement while providing strong integrity assurances against the threat from untrusted applications.

2.2.1 Systrace

Systrace, as we saw, can provide extremely fine grained control and thus gives strong assurances. Unfortunately the level of user effort and judgement involved is very high.

Systrace *cannot* be used in training mode with untrusted applications simply because the application cannot be trusted to run even once. As stated earlier, we don't consider the use of a virtual environment for this purpose as desirable either. So that leaves us with the only other option of interactive policy generation. The disadvantage of this approach is that it rests on the user's judgement almost completely and involves her extensively. The approach here is simple: monitor and judge *all* accesses by untrusted applications. Since it does not introduce any new abstractions to help with the judgement, it places a large burden on the end user. The use of template policies to alleviate the problem has the risk of either starting with a template that is too permissive or too conservative. The former compromises security and the latter inconveniences the user, by needing her to iterate many times while trying to relax the policy sufficiently.

Another disadvantage of systrace is its inability to sandbox applications whose behaviour is parametrized by the caller or is based on user interactions. For example an editor like `vi` would most likely access a different file in each run, which will trigger a violation of its earlier, interactively generated systrace policy.

To borrow from the MCC terminology, systrace policies are essentially about generating models rather than policies. The philosophy that an accurate model can be used to confine an application has the disadvantages we mentioned above.

All the other infrastructures we looked at, invent additional abstractions to help the policy author specify the confinement policy.

2.2.2 SELinux

Now lets turn our attention to the SELinux idea and how well suited it is to defend against untrusted applications. It shares with systrace the basic philosophy of using least privilege for confinement. So in that respect it brings the same fundamental advantages and disadvantages as we will see. The advantages of the TE paradigm relevant to our discussion are:

- Provides enough flexibility for enforcing least privilege or separation of duty. Because the policy author has a type system, he can express what privileges a subject can have at that abstraction. This flexibility is in contrast to the rigidity of an approach like Multi-Level Security.

- A well designed type system is a powerful tool to create policy idioms, such as the set of accesses which can be granted to all daemon processes. SELinux does a good job of providing intuitive types that lend themselves to expressing security policies and then building such idioms for policy authors to reuse.

- Since it takes a least privilege approach to security policies it address problems beyond the threat model we are trying to address. It is important to note that SELinux not only protects the system from untrusted applications but also protects the system from a root process (likely to be a trusted application) that may be compromised over the network.

However the TE flavor of policy based confinement poses practical difficulties for addressing the threat model of malicious untrusted applications:

- SELinux by construction does not state any meaningful security goals besides enforcing principle of least privilege. As mentioned earlier it can act as the basis for a secure system but by itself cannot provide guarantees of preserving system integrity.

- It requires the policy author to have an a priori understanding of what resources the untrusted application needs in order to function. This information is almost always unavailable. So it follows from this observation that adding any new trusted or untrusted application to this infrastructure, is a time consuming task. Ideally we would want to run a new untrusted binary without even worrying about how it promises to behave and at the same time have the assurance that it will not compromise the system.
- As in the case of systrace, some applications whose behaviour is parametrized by the caller or is based on user interaction, will be difficult to confine based on least privilege.
- The additional problem with TE based confinement is that the predefined types for files tend to group files based on various regular expressions. This is a bottom-up way of grouping filesystem resources which does not represent any logical relationship between the files sharing the same type label.
- A related problem with file types based on regular expressions is that in the future there can arise a need to divide a type into further sub types or even recombine existing types to form new types. In such a scenario, all existing policies, which drew on the reasoning of the earlier classification, will have to be revisited. This problem can also be traced back to the lack of logical grouping in the first place.

In summary SELinux has an elaborate type system which provides the flexibility to express policies for least privilege. However least privilege, as we have seen, is perhaps not the best approach when defending against untrusted applications. This approach does not provide clear and verifiable assurances for system integrity. Besides it hurts the usability of a system to have the author think of least privilege policy for every new untrusted application. We did however note that SELinux is well suited to combat a different threat model. where the root processes (or trusted applications) are compromised over the network.

2.2.3 MCC

At this point we realize that to protect trusted applications from untrusted applications we need a paradigm that has the ability to somehow deduce automatically what files are safe for an untrusted application to modify. The reasoning is that then we can add a new untrusted application to the system and confine it safely without needing policy to have a detailed understanding of the behaviour of the new application. The *owned files* abstraction in MCC uses exactly that idea. In contrast to SELinux and systrace, MCC solves the usability problem to a large extent by automatically deriving the *owned* context for each new application.

Additionally MCC provides the idea of Application Classes to help group similar applications. Policy for each Application Class combines conservative and permissive policies for file reads, writes, creation and deletion. It must be noted that even the most permissive policy possible is still preserving the integrity of other applications. It is only relaxing the access to the relatively less important *orphan* files in the system which are not critical to the operation of any application. These may include data files etc. The availability of policy templates for Application Classes makes it easier for the security administrator to choose the most appropriate policy for a new untrusted application. As mentioned earlier, if it is unclear as to which classification is most suitable, the infrastructure automates the choice by finding the best match based on verification results.

We seemed to have solved a problem we had earlier with systrace and SELinux and can now induct new untrusted applications into this infrastructure painlessly. A subtle problem however exists with this approach. Notice that we are only monitoring write operations by untrusted applications but not tracking the reads of trusted applications nor keeping track of the files written by the untrusted application. So it is possible for an untrusted application to feed malicious data into a trusted application via any file that it wrote and the trusted app read. This is actually a well known problem regarding information transfer paths from objects of low integrity to objects of high integrity.

2.2.4 MLS

Multi-Level Security concerns itself with solving exactly the problem we had with MCC: that of preventing information flow from untrusted objects to trusted applications.

Since it does not use least privilege, but instead tracks data flows, MLS is the most intuitive solution to the problem of confining untrusted applications. It can deliver guarantees for system integrity which can be verified. But it has well known usability issues when it tries to preventing bad information flows in the system

Strict Integrity Model

Although the Biba model delivers strong guarantees of preventing bad data flows, the model cannot be employed without severe usability issues. Under the Biba model untrusted data will not be readable by any subject with a greater trust level due to the “no read down” rule. Unfortunately that would render all untrusted files inaccessible except by subjects that hold low trust levels themselves, which presumably will not include many installed system utilities.

Low-Water-Mark Model

The Low-Water-Mark model has not been popular because of its need to demote subject integrity levels, which can cause failures in application functionality. This is because of the *self-revocation* problem: Consider a process has opened a trusted file for writing and then reads an untrusted file. If the process is downgraded at this point it will have to revoke its permissions to the trusted file. Since many applications don't expect to have their access revoked they may not handle these errors gracefully or may simply be unable to provide a certain functionality without the revoked accesses

Besides the usability issues MLS has some further problems:

- To get started the administrator needs to perform the nontrivial task of partitioning *all* the files and applications on the system into multiple integrity levels.
- It does not give us the option of protecting untrusted applications from each other. Since all untrusted applications and files have the same label, these are indistinguishable in the model.

In summary the Biba model is practically unusable on a regular desktop because of its strong constraints while the Low-Water Mark policy causes self-revocation. MLS also has the usability issues of classifying the applications into trust levels on a regular desktop. But as far as integrity assurances in concerned, MLS provides strong and verifiable assurances.

Our observation at this point is that the usability problem can be alleviated to some extent by using a hybrid of the above models. The hybrid decides to demote the subject or object versus deny the access, based on the context of the subject and object involved. The important thing is that if the policy were to be context aware and act accordingly it would still deliver the same integrity assurances as the original MLS models. Another insight is that in the real world there are instances where we can break the formal semantics of MLS by trusting the subject to protect itself against untrusted data and thus the policy takes need not take any action when such an access takes place. Finally we also note that there is a way to leverage the *container* concept in MLS to protect untrusted applications from each other.

2.2.5 Goals of Proposed Infrastructure

After looking at the pros and cons of the related infrastructures we are ready to defined the goals for our infrastructure as follows:

- Like MLS, we should provide integrity assurances which account for information flows from untrusted sources to trusted applications rather than take the least privilege approach which doesn't work well for untrusted applications.

- We should automate the process of inducting a new untrusted application into the framework.
- We want to address the problem of classifying the applications into trust levels on a regular desktop.

We also want to use the following insights in our approach:

- We should leverage the insight that a hybrid of the two MLS models (Strict integrity and Low Water-Mark) will work better in terms of usability while providing the same integrity assurances.
- We want to use MCC's idea of *owned* files and use that as the context for building *compartments* in MLS. This gives us stronger assurances than traditional MLS by protecting untrusted applications from each other.

Chapter 3

Methodology

In this chapter present our approach to preserving the integrity of the system against the threat from untrusted applications. We present the fundamental idea in the first section. At the end of the section we list the practical challenges in implementing a framework that leverages the idea. The subsequent sections present our solutions to these problems.

3.1 Distinguishing Integrity Labels and Policy

Our goal, as stated, is to to preserve system integrity against the threat from untrusted applications. For this purpose it is intuitive to think in terms of a system which associates trust levels with all the subjects and objects in the system. A higher integrity level for an object indicates more trustworthy content and a higher integrity level for a subject indicates higher confidence that it will execute correctly. One important objective is to prevent the flow of untrusted data into trusted applications. We share this objective with traditional MLS infrastructures we discussed earlier, such as Strict Integrity Model and the Low-Water Mark policy. The way MLS models deal with this problem is by enforcing a policy strictly based on labels. Since in MLS the policy is driven only by labels, it inevitably ends up causing usability issues as discussed earlier.

In order to overcome such problems we make a distinction between *integrity labels* and *policy*. Integrity labels, just like in MLS, indicated whether the content of a file or the nature of a process are trustworthy. However in our solution the labels do not alone determine whether a process can read from or write to a file. Our policy decides based on labels and other factors whether read or write accesses are allowed and if there should be any side effects like demotion of subject or object.

The two scenarios where our policy has to make a choice is either when a high integrity subject attempts to read a low integrity object or a low integrity subject attempts to write a high integrity object . Unlike traditional MLS, where there is exactly one course of action, we have multiple choices.

When a **high integrity subject attempts to read a low integrity object** we have the following choices:

- Deny the access
- Downgrade the process to low-integrity
- Allow the access without downgrading the process, our expectation here is that the process will protect itself

  A look at some examples will clarify the need for this flexibility. If a `ssh` server is provided a untrusted configuration file `/etc/ssh/sshd_config`, the best course of action is to deny access (alternative 1). The second alternative of downgrading the server will prevent it from overwriting its own state stored in other trusted files like `/etc/ssh/ssh_host_key`. It is of course absurd to trust the server to protect itself against a bad configuration so the third option is also eliminated. Lets take another example where a utility like `cp` may access high integrity objects in some runs(e.g. create a copy of `/etc/shadow`) and copy low integrity files at other times (e.g. user files). In such a case when `cp` reads an untrusted file it is best to downgrade it according to the Low-Water Mark policy(alternative 2). This way a copy of an untrusted file will also be untrusted. However there are other scenarios where downgrading sometimes leads to the *self-revocation* problem: Consider a process has opened a trusted file for writing and then reads an untrusted file. If the process is downgraded at this point it will have to revoke its permissions to the trusted file. Since many applications don't expect to have their access revoked they may not handle these errors gracefully or may simply be unable to provide a certain functionality without the revoked accesses. Our previous example where a `ssh` server read an untrusted configuration file demonstrates why the second alternative of downgrade is not a panacea. Finally lets consider a trusted web browser that reads untrusted input from the Internet. The only real choice we have is to trust the browser to protect itself (alternative 3) since the other options will prevent the browser from functioning.

When a **low integrity subject attempts to write a high integrity object** we have the following choices

- Deny the access
- Downgrade the object, and allow the write.

Here the rule of thumb is to deny access to files which are definitely going to be read by trusted applications at a later point of time. All other accesses are allowed and downgrade the object, indicating they are no longer trusted and have become untrusted. Here we deviate from the traditional MLS, where object integrity levels never decrease.

The key idea in the above discussion is that we deviate from standard MLS in proposing **multiple ways of resolving a conflict**. This affords us more flexibility. Now that we have stated the central idea, we look at the practical issues with building a system that uses the concept. The solutions are presented in the subsequent sections.

- The first problem is the issue of how to easily classify applications on a typical desktop into trusted and untrusted. This is non trivial because we are dealing with hundreds of active applications on a regular desktop. We share this problem with the traditional MLS approach.

- The second problem has to do with the high number of conflicts we encounter. A *conflict* is defined as either a high integrity subject attempting to read a low integrity object or a low integrity subject attempting to write a high integrity object. As we stated earlier there are multiple resolutions possible in such conflicts. The problem is that given the large number of subject-object interactions in a typical system there will be a large number of conflicts and we need a methodology to help automate choosing the appropriate resolution in each case.

- The third problem has to do with attacks where untrusted applications can rename or unlink files which are critical to the functioning of a trusted application. The approach we described above solved the integrity problem by preventing bad data flow, so the integrity labels were related to the *contents* of a file not its name. So we need a way to preserve integrity of file names as well.

- The fourth problem is that of protecting untrusted applications from each other. The trust based classification we talked about, does not give us a way to prevent arbitrary subject-object interactions *within* a label.

Before we go about explaining the solutions to these issues, it is useful to make a mention of the inputs used to analyze and solve the problems. We rely on the **analysis of system wide logs of filesystem accesses** as well as the information from the **package manager database**. The logs are obtained over reasonably large periods of time, ranging from a week to a few months. The events logged include reads, writes, creation of new links, deletion of links and renaming links.

3.2 Assigning Trust Levels to Applications

We first need to partition the set of applications running on the system into two levels: trusted and untrusted. This is a step which is common for all MLS based approaches but is a nontrivial task simply because there are hundreds of applications active on a typical desktop. One basis for partitioning could be that only processes that run as root are trusted. But that criterion turns out to be weak because very often utilities critical to maintaining system integrity run in underprivileged mode such as utilities like `cp`, `cat`, `vi` and many others.

We have two complimentary approaches to help automate the classification process. The first is labelling applications based on their interactions with other applications in the system, got by log analysis. The second draws on the package dependencies, got by querying the package management database.

3.2.1 Labelling Applications Based on Log Analysis

Our approach is based on a two step process:

Infer set of trusted applications based on exec relationships between applications

We mine the logs for exec events and create a graph representing exec based relationships between applications. Each node represents an app and an edge from Vi to Vj means that Vi execed Vj. The idea is to start from `init` and create the complete process tree. Now we should simply ignore all nodes which are reachable from nodes where user interaction starts such as `login`, `sshd` and `gnome-panel`. We cannot say much about the trust level of the apps that are reachable from these nodes but we can label the rest of the apps as trusted.

However there is a slight problem with this algorithm: Apps like `bash` are invoked both in trusted and untrusted contexts such as startup scripts as well as user login. In the graph we have constructed *none* of the descendents of `bash` can be labeled trusted despite the fact that some apps, like daemons etc created at startup time, are obviously trusted.

So we modify this algorithm to create separate nodes: app_t and app_u depending on whether the parent node is trusted or untrusted. We have identified `gnome-panel` and `sshd` as 2 nodes that provide an untrusted context because of user driven behaviour. So all nodes reachable from such nodes are labeled untrusted and all the other descendents of init are labeled trusted.

So at the end of this step we have identified a set of Trusted apps. As for the nodes with the untrusted label we simply cannot assign a trust label at this point. This is done in the following step.

Infer the trust level of the remaining applications based on data flow interaction between the applications

This step uses a two prong strategy of

- Deriving trust label for applications based on their interactions with other well known trusted and untrusted applications.
- Identifying groups of applications that typically do not have data flows with applications outside of that group

Well known trusted applications besides those identified in the previous step, include the system software such as the operating system, editors, compilers and linkers . It also includes system services and core system utilities. Well known untrusted applications can be site specific but can include viewers, editors, games etc.

The steps for assigning trust levels to the applications are

- **Construct Graph of Application Interactions:** Create a graph G where vertices represent applications which appear in the log i.e. all the applications that are active in the system. We add a directed edge (V_i, V_j) if application V_i writes into an inode which is *almost always* read by application V_j. We can set a threshold to decide what fraction of application V_j's runs must read that inode in order to qualify. We discount, from the analysis, inodes that belong to device files which cannot act as a conduit for data, such as `/dev/null` and `/dev/tty`. The graph thus formed shows *typical* data flows between applications and tries to be independent of user behaviour. For example `passwd` will almost always read the `/etc/shadow` file written by `adduser` and thus typically these two applications interact and there is a directed edge from `adduser` to `passwd`. On the other hand say `vi` reads a file downloaded by `wget`. This is not likely to happen for most instances of `vi` and there will be no edge between the applications as result of this. Thus arbitrary user behaviour does not create an edge between applications.

- **Infer Labels of Applications from well known Applications:** Label all vertices corresponding to well known trusted application as T and all well know untrusted applications as U

  For any edge (V_i, V_j) in graph G, if the vertex V_j has a T label then we mark the vertex V_i as T. The reasoning is that if the application V_i writes a file typically read by the trusted application V_j then we can assert that for security, V_i needs to be a trusted application.

For any edge (V_i, V_j) in graph G, if the vertex V_i has a U label then we mark the vertex V_j as U. The reasoning is that if the untrusted application V_i writes a file typically read by the application V_j then we can assert that for security, V_j needs to be an untrusted application.

- **Remove Uninteresting Edges:** We remove any edge (V_i, V_j) in the graph originating from a trusted vertex V_i. The reasoning is that information flow from a trusted application is of no interest from a security point of view. We also remove any edge (V_i, V_j) in the graph which ends on an untrusted vertex V_j. The reasoning is that information flow to an untrusted application is of no interest from a security point of view

- **Remove Labelled Nodes:** We now remove all the labelled nodes, which as a result of the previos steps will be isolated nodes at this point. So we dont lose any edges from the graph in this step.

- **Find Groups of Interacting Unlabelled Applications:** In the resulting graph we look for connected components. These indicate groups of applications which do not have data flows with any application outside the group. We are free to assign either label to the group. But the same label should be assigned to all the applications in the group.

3.2.2 Labelling Applications Based on Package Dependencies

The strategy here is to derive trust labels for applications based on the dependencies between their owning packages.

The steps for assigning trust levels to the applications are

- **Construct Graph of Package Dependencies:** We construct a graph G where vertices represent installed packages. We add a directed edge (V_i, V_j) if package V_i has a dependency on the package V_j. This information is got by querying the package management database.

- **Infer Labels of Applications:** We label any package that is known to be trusted as T i.e. it houses a well known trusted application. Example: `coreutils` houses trusted applications like `cat` and `cp`. Similarly we label packages that house well known untrusted applications as U.

  For any edge (V_i, V_j) in graph G, if the vertex V_i has a T label then we mark the vertex V_j as T. The reasoning is that if the package V_i depends on package V_j then we can assert that for security, V_j needs to be a trusted package.

For any edge (V_i, V_j) in graph G, if the vertex V_j has a U label then we mark the vertex V_i as U. The reasoning is that if the package V_i depends on the untrusted package V_j then we can assert that for security, V_i needs to be an untrusted application.

Now many of the vertices in the graph G have a label. If we want to derive the trust label for an unknown application we can simply look up the label of the owning package.

3.3 Log Analysis for creating a Data Flow Policy

We analyze the filesystem access logs, mentioned earlier, in order to develop a policy that prevents flow of untrusted data into trusted applications. We note that there is a different class of attacks by malicious applications involving renaming critical links or surreptitiously creating a link usually associated with a trusted file. We handle these in the next section. For now we concentrate on more simple attacks involving reads and writes to inode only. In this section we use the terms *file* and *inode* interchangeably. The reason we can do that is because the only way to read from or write to a file is by accessing the inode. We have a two step procedure to achieve safe data flows.

- **Policy to Prevent Overwriting Files Regularly Read by Trusted Applications:** We find, from the logs, all inodes that meet the following criteria:

  - Inodes that are *always* read by trusted applications.
  - Inodes that are *ever* written by untrusted applications.

  At this point we get only trivial conflicts like `/dev/null` and `/dev/tty` which are not going to act as conduits for data flow. However if we do encounter any nontrivial conflicts then we can conclude that the untrusted application which wrote into that inode cannot be safely run on the system. The reasoning is simply that the inode seems to be of critical importance to the trusted application. The file is probably a binary, a library, a configuration or a cache file used by the trusted application.

- **Policy to Resolve all other Data Flow Conflicts:** We find from the logs, all inodes that meet the following criteria:

  - Inodes that are *sometimes* read by trusted applications
  - Inodes that are *ever* written by untrusted applications

  These conflicts are typically the result of some arbitrary user behavior such as using a trusted editor to edit a downloaded file. These conflicts can be resolved by one of the 3 possible resolutions for when a high integrity subject reads a low integrity object mentioned earlier.

Our goal in these resolutions will be to deny such a read to a trusted application if it always needs to write to trusted files thereafter. On the other hand if the trusted application doesn't need to write to trusted files we can downgrade the application on such a read. Some rules of thumb here are:

- Trusted utilities will typically be downgraded for the sessions in which they read an untrusted file. e.g. copying, editing or compiling an untrusted file
- If a trusted application breaks due to the self-revocation problem we deny the read.
- If an application can be assumed to be resilient against malicious inputs we can trust the subject and simply allow the access

The other way untrusted data can flow into trusted applications, from within the system, is **Inter-Process Communication (IPC)**. The policy for these data flows is the same as those via files.The question is: Do we capture these flows?

- Named pipes or FIFOs: Data flow via these can be captured because the reads and writes are directed at a common disk inode.
- Anonymous Pipes: Data flow via these can be captured because the reads and writes are directed to the same inode, this time on the pipefs.
- Shared Memory: We can discount shared memory because it tends to be used by cooperating processes and is thus an unlikely path for bad data flows.
- Unix Domain Sockets: Data flow via these is also captured because read and write accesses look up the same inode.
- Loopback sockets: This data flow we do not capture since the data doesn't flow through an inode on the filesystem. However we can build a simple extension that declares processes connecting to and serving at the same ports as having established a data flow.

3.4 Policies for Preventing Attacks Based on File Names

We noted earlier that there is a class of attacks where an untrusted application can manipulate links to critical inodes or in other words manipulate file names to compromise trusted applications. Such attacks could include:

- An untrusted application renames the configuration file of a trusted application and breaks the trusted application as a result. The reason the data flow policy will not catch this is because the inode of the file is intact

- An untrusted application unlinks a trusted file with multiple links i.e without destroying the inode. Now the link used by a trusted application is gone and breaks the application. Again since the inode is unaffected the data flow policy will not catch it.

- An untrusted application can surreptitiously create a link usually associated with a trusted file. This of course is possible if that link didn't exist in the first place. The attack is feasible with applications for which configuration files are optional and when they are absent some defaults are chosen.

Our solution for solving this problem is again based on integrity labels, this time associated with links. In this case however the policy is based strictly on labels. The reason we cannot separate labels from the policy in this case is because of deletion of links. Once a link is deleted we lose the label associated withon that link. Despite deletion, this information is still relevant because the application which accesses this link in the future needs to be aware of this deletion. We can of course maintain the label information outside the links themselves but that probably adds more latency to each link access. To avoid this situation we implement a policy based strictly on labels which will simply prevent such a deletion.

When protecting integrity of links we choose to interpret the special link related operations as reads or writes depending on the case. This helps us fit this new problem into the old access control framework we already have from before. The operations are interpreted as:

- `unlink` operation is treated as a write access to the link being deleted.

- `rename` operation is interpreted as a 2 step procedure. Deletion of the old link, interpreted as a write access , followed by creation of a new link

- `read` operation in a file needs a lookup on the filename and is thus a read access on the link.

- `write` operation on a file similarly needs to lookup the filename and is also a read access for the link.

The policy enforced is the **Strict Integrity Model of MLS** which is described in section 2.1.4. We label all trusted and untrusted applications as before. However this time we label the links that are almost always accessed by trusted applications as trusted and the rest an untrusted. The untrusted application is unable to unlink or rename a link with a high integrity label because that constitutes a low integrity subject writing a high integrity object. Since the Strict Integrity Model prohibits this, it takes care of guarding against the first and second attacks mentioned above. It is still possible to create surreptitious links to untrusted files as suggested in the third attack. However when the trusted application reads or writes this bogus link, it will translate into a read access for the link in question. Now the application is a high integrity subject and the link is a low integrity object, because it was created by an untrusted process. As we know read access of a low integrity object by a high integrity subject will fail under the Strict Integrity Model. This we foil the third attack as well.

3.5 Protecting Untrusted Applications

To protect the integrity of untrusted applications from each other we need a different approach from that used to protect trusted applications. Our aim now is to ensure that the files which are critical to the integrity of an untrusted application are not interfered with by any other untrusted application. These files include executables, configuration files, libraries etc. We use the concept of *compartments* in MLS and can thus be easily added into the existing policies we have discussed thus far. We create a compartment for each untrusted application. The compartment holds the files that are critical to the integrity of the untrusted application.

We have two complimentary approaches to enumerate the members that constitute the compartment for each untrusted application.

3.5.1 Building an Owned Database

We create an *owned database* of each untrusted application just as was done in MCC. We propose to use the same techniques to find these files.

- **Static Information:** We lookup the Package Management database to create a dependency graph for *all* packages installed in a system. We then find the package which owns the application and find all the packages this package directly or indirectly depends on. We create an *owned database* for the application which has all the files belonging to any package that it depends on.

- **Dynamic Information:** We determine the per-user configuration files for the untrusted app based on heuristics that guess the config file name. For example user configuration file for `emacs` is `/home/user/.emacs`. We add such files also to the *owned database*

We add the files from the owned database into the compartment associated with that untrusted application.

3.5.2 Log Analysis Based Approach

For each untrusted application we find the inodes that are *always* read by the application. These files are presumably critical to the operation of the application and may include many trusted files like `libc.so` and `/etc/ld.so.cache`. We add these files to the compartment associated with that untrusted application.

The reason we can't always use the log based approach is because for an new untrusted application we may not have logs which help us learn which files should end up in that application's compartment.

Chapter 4

Implementation

In the first section of the chapter we discuss how the filesystem access logs were generated. The analysis of the logs is discussed in the next chapter. In the second chapter we talk about a prototype that was built to test the impact a conflict resolution choice has on the usability of an application.

4.1 Setup to Obtain the Logs

4.1.1 Events and Data Captured

In order to test the methodology we need to log all the filesystem accesses over a reasonably long period of time. Since the number of `read` and `write` system calls is very large, we treat the `open` system call in read-only mode as a *Read* event and in write or read-write mode as a *Write* event. In addition we track the access to files from inside the kernel by probing all calls to the kernel function `filp_open` which is often used for this purpose. Again we translate that call into the appropriate event by using the mode.

In addition we also needed to track system calls which have some effect on links such as `link`, `unlink` and `rename`. Since the integrity of directories is independent of data flows we didn't need to track system calls like `mkdir` and `rmdir`

For all the calls we track above we log the triplet of ($Filesystem$, $Inode\ Number$, $Inode\ Generation\ Number$) that is unique for each file as well as the absolute path for each file.

4.1.2 Logging Tool

In order to log these accesses on a system-wide basis we used SystemTap tool[10] on Linux. This allows the user to write simple scripts which trigger on any probe points of the user's choice, as long they are supported by the `kprobe` infrastructure inside the Linux kernel. At the probe points we obtained and logged the information we needed from the kernel data structures. It must also be noted that the logging started during system bootup, as soon as the root filesystem is mounted in read-write mode. Thus we capture the accesses made by all the init scripts as well.

4.1.3 System Setup

The hosts on which we were logging these accesses were setup like regular desktop machines, with network services like `ftp`, `http`, `https` etc turned off. We also had installed on these hosts untrusted applications like

- Multimedia and Document Viewers
  - xview
  - acrobat reader
  - mplayer
  - xpdf
  - gv
- Archivers/Format converters
  - ascii2pdf
  - html2latex
  - atool (archive manager)
- Games/P2P Applications/Web Agents
  - bittorrent
  - yahoo messenger
  - gaim (instant messaging client)
- Vulnerability Scanners
  - crack (password cracker utility)

4.1.4 Usage patterns

We deployed the hosts as regular desktops for users in our lab. They were free to install and use any application, in any way they wished. We needed to capture multiple instances of each application. Some of them, like daemons, would only run once so we restarted the system at reasonable intervals of a few days. We ensured that the usage patterns were *typical* of a regular desktop by not interfering in any way. The logging tool ran transparent to the users inside the kernel.

4.2 Prototype to Test Usability Impact of Conflict Resolution

We built a prototype where we were able to test the usability issues associated with making a policy choice based on the log analysis, such as deny, downgrade or trust on read or write access. The idea is to have a monitor process that uses the `ptrace` mechanism to monitor and control every system call made by the application in question. The integrity level of the process was maintained in the state of the monitor and the labels of the files was maintained in a `BDB` database. So we were able to change integrity labels of subjects and objects as required by simply updating these external states. The monitor is able to execute policy logic both at the entry as well as the exit of a system call. Because we could execute policy logic at both of these points we had the options of

- On system call entry, check if access causes a policy conflict. Choose a resolution such as deny or trust. In case of denial we return an appropriate error code.

- On system call exit, check whether the call was successful and if necessary provide the third conflict resolution, that of demoting the subject or object by updating the integrity levels.

Chapter 5

Analysis and Results

5.1 Assigning Trust Levels To Application

On processing the logs we found that 326 applications had made filesytem accesses. The total number of distinct inodes opened in read-only mode by the applications was 28235. Total number of distinct inodes opened in read-write mode by the applications was 11989.

The exec realationship graph yielded 217 trusted applications. The list of applications which we could label as trusted from this step are listed in the appendix

The graph of interacting applications was constructed as described in section 3.2.1 . It turned out to be a connected graph which meant that all applications in the system were typically interacting with at least one other application.

We could identify 31 well known trusted applications and 47 well known untrusted applications. Using the procedure for inferring labels by observing interactions with well known applications we could infer 2 applications as untrusted and 4 applications as trusted. For example a data flow from `sessreg` to the well known trusted application `sshd` via the `/var/log/lastlog` file helped us infer that the former is to be labelled trusted.

Now we proceeded to remove the uninteresting edges which represent writes by trusted applications and reads by untrusted applications. After doing so only 49 applications remain connected while all the other vertices become disconnected. Our algorithm has yielded many different connected graphs, a majority of which consisted of only a single node. We were now free to classify the applications in these disconnected graphs either way using the procedure described earlier.

We noticed that a majority of the edges in the original interaction graph originated at trusted nodes. This is because some trusted applications write into trusted files that are in turn read by many other applications. For example `ldconfig` writes into `/etc/ld.so.cache` which is read by practically all applications on the system.

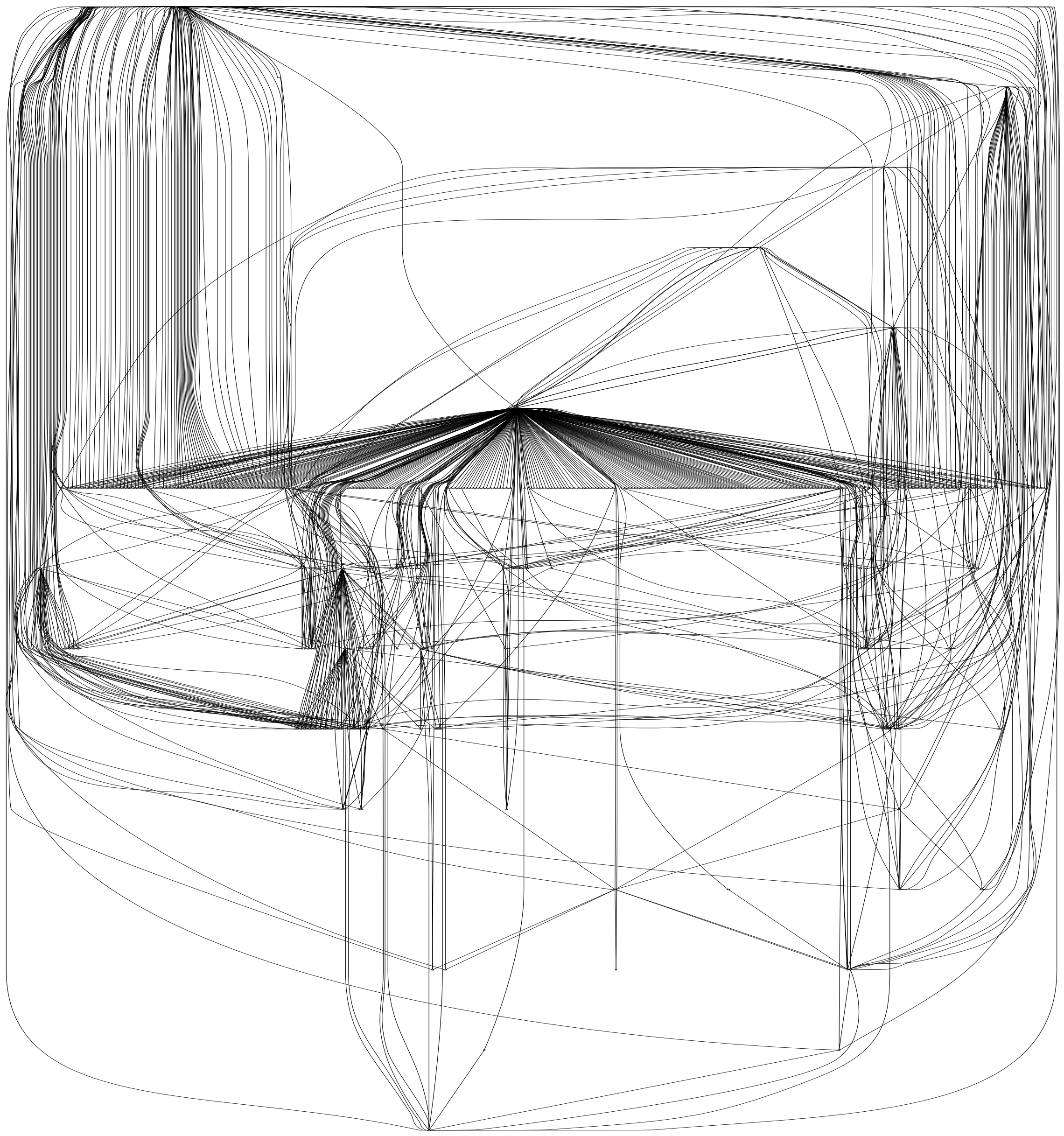

Figure 5.1: Graph of interacting applications

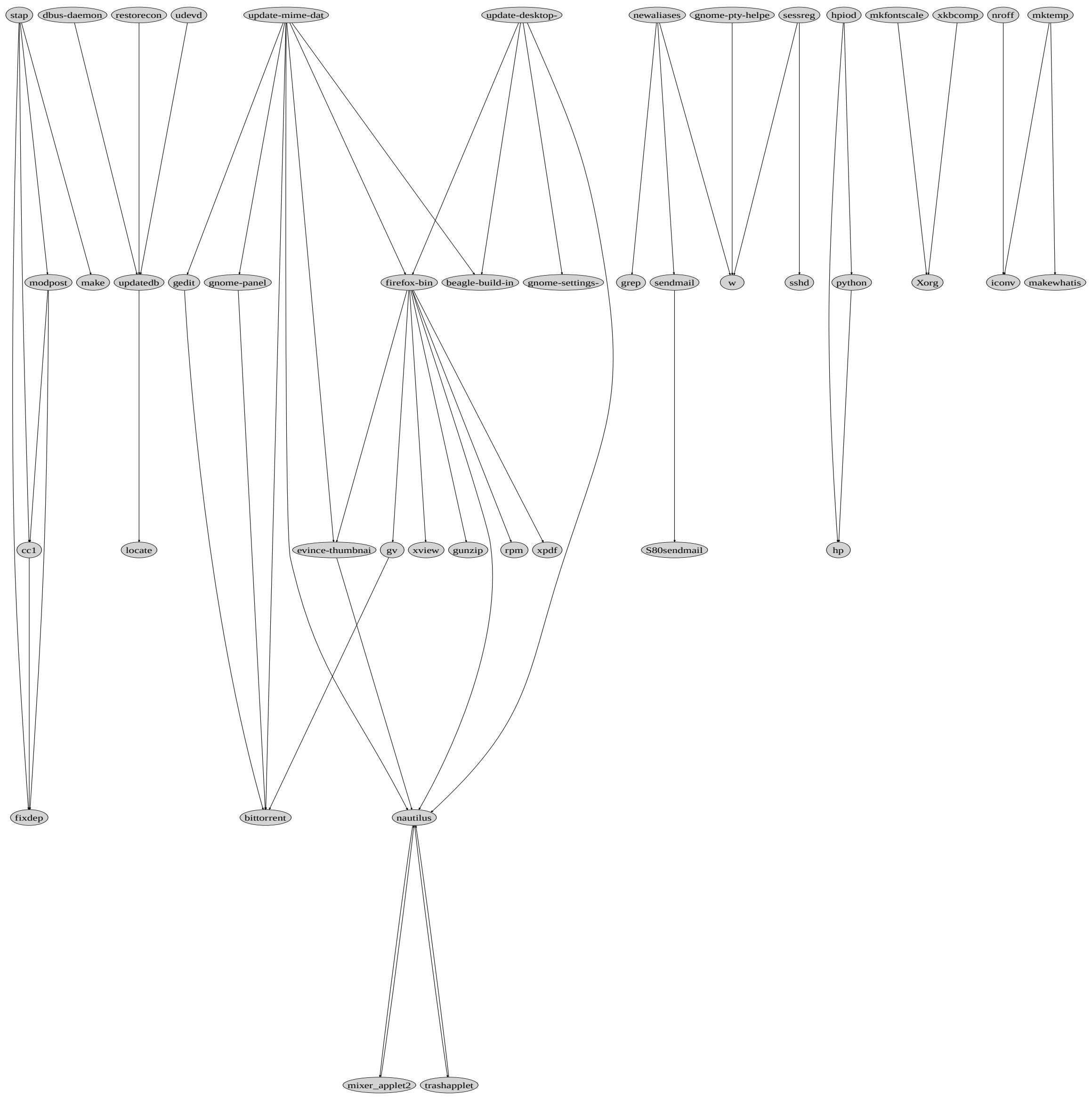

Figure 5.2: Graph with only unlabeled connected nodes

Another example is the `passwd` application writes into `/etc/passwd` which is accesses by scores of applications to resolve UIDs to names. As a result applications like `ldconfig` and `passwd` have many edges originating from them. As explained earlier, since these applications are trusted, we can eliminate these edges without losing any data flow that affects security.

Trying to assign trust levels to applications based on human judgement was difficult because many application were either obscure or did not seem obviously trusted or untrusted. Our procedure for labelling applications makes the task easier and more reliable.

5.2 Log Analysis for Creating Data Flow Policies

Here we present the results of the procedure for analyzing logs to create a Data Flow Policy, as described in section 3.3. In the first step of the analysis we are trying to prevent untrusted applications from overwriting files regularly read by trusted application. As expected there were only trivial conflicts due to inodes of device files which do not act as conduits of untrusted data and can be safely ignored.

In the second step we find a number of inodes that cause conflicts. On analyzing the conflicts we can reduce them to 11 conflicting scenarios. The reduction involved eliminating multiple inodes that cause conflicts between the same pair of applications.

Here we present a couple of typical conflicts and the proposed resolutions:

- System Utilities Reading Untrusted Files: Trusted utilities like `cp` and `cat` regularly read files written by untrusted applications such as web browsers. These were in fact files downloaded from the Internet. The resolution is to *downgrade* the process. So any copy created of an untrusted file will also be untrusted. A not so desirable result is achieved in the scenario when a batch copy is made of a mixed bag of trusted and untrusted files. All copies created *after* reading the untrusted file will be untrusted.

- Trusted Editor Reading Untrusted Files: Trusted editor `vi` read files written or created by less trusted editors like `gedit`. The proposed resolution is *downgrade*. We took this scenario back to our prototype for testing usability problems. We notice that `vi` gracefully handles self-revocation. When demoted, it is denied access on trying to overwrite some of its own configurations, but manages to successfully handle the edit operations.

- Secure FTP of an Untrusted file: The `sftp` utility was used to transfer out untrusted files. On reading untrusted files the resolution we proposed was *downgrade*. Again when testing this approach on our prototype we saw that since `sftp` was not writing any trusted files the demotion posed no usability issues.

We did encounter a couple of applications for which our proposed approach needed modifications. Here we present special security policies for these apps:

- Archivers like `tar` are trusted to protect themselves. So while archiving files, even if some or all of the input is untrusted, the resulting archive is trusted. While unzipping, if the archive is trusted, we faithfully let each created file retain its archived trust label. This is an exception to the usual approach of creating new objects with the same label as the subject creating them. On the other hand, while unzipping an untrusted archive, we label all the resulting files as untrusted. This is because we cannot rely on the labels suggested by an untrusted archive.

- Package managers like `rpm` are trusted to read any package and protect themselves. However on similar lines as above, while installing from an untrusted package, all the newly installed files are labelled as untrusted.

It was interesting to contrast the conflicts we see here with the conflicts we found when we did not employ our methodology to classify the applications into trusted and untrusted. When not using the methodology explained in section 3.2.1 we used our judgement to partition the 326 applications into trusted and untrusted. When we did data flow analysis on the same logs with this partitioning, it threw up many more pairs of conflicting applications. For example `newaliases` was labelled as untrusted and `sendmail` was labelled as trusted. The folly of not accounting for their interaction via the file `/etc/aliases.db` caused a conflict which shouldn't arise in the first place because `sendmail` should have the same trust label as `newaliases`. Of course we can make that assertion now based on our methodology for inferring labels. Such examples suggest the superiority of our methodology over relying completely on human judgement for the process of deciding what is to be trusted and what is not to be trusted.

Chapter 6

Conclusion

We have presented a survey of multiple infrastructures that do policy-based confinement. In analyzing their strengths and weaknesses we have gained insights which were put to use in our approach to protect system integrity from untrusted applications.

We first discarded the principle of least privilege as an effective approach since it does not translate into integrity assurances in a verifiable manner. We decided to create variations on the theme of MLS, since it has strong assurances. By separating the policy from the labels we have created more choice for conflict resolution. This has helped overcome, to a large extent, the usability issues with traditional MLS since now we can chooses a resolution which is better suited based on the context. Further we provide a log analysis based approach to making the conflict resolution easier.

Going beyond protecting the data integrity of trusted applications, we attempt to solve a couple of practical issues in actually deploying such a system. The first is to provide a methodology which can be used to give integrity labels to applications installed on a regular desktop. The second is to look at integrity of file names in addition to contents of files. We bring this problem back into the MLS framework to keep things uniform. Thirdly we provide a methodology to protect untrusted applications from each other, again by creating appropriate MLS compartments.

We hope that our approach will lead to a methodology where policies will be generated automatically to protect against untrusted applications.

Bibliography

[1] L. Badger, D. F. Sterne, D. L. Sherman, and K. M.Walker. A domain and type enforcement unix prototype. In *USENIX Computing Systems*, 1996.

[2] K. J. Biba. Integrity considerations for secure computer systems. In *Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Hanscom Air Force Base, Bedford, Massachusetts*, 1977.

[3] Matt Bishop. Computer security art and science. pages 151–155. Addison-Wesley, 2002.

[4] Timothy Fraser. Lomac: Low water-mark integrity protection for COTS environments. In *IEEE Symposium on Security and Privacy*, 2000.

[5] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the 6th Usenix Security Symposium*, San Jose, CA, USA, 1996.

[6] T. Jaeger, R. Sailer, and X. Zhang. Analyzing integrity protection in the selinux example policy. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

[7] K. Jain and R. Sekar. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. pages 19–34.

[8] Peter A. Loscocco and Stephen D. Smalley. Meeting critical security objectives with security-enhanced linux. In *Proceedings of the 2001 Ottawa Linux Symposium*, 2001.

[9] Peter A. Loscocco, Stephen D. Smalley, Patrick A. Muckelbauer, Ruth C. Taylor, S. Jeff Turner, and John F. Farrell. The inevitability of failure: The flawed assumption of security in modern computing environments. In *Proceedings of the 21st National Information System Security Conference*, 1998.

[10] Vara Prasad, William Cohen, Frank Ch. Eigler, Martin Hunt, Jim Keniston, and Brad Chen. Locating system problems using dynamic instrumentation. In *Proceedings of the 2005 Ottawa Linux Symposium*, 2005.

[11] N. Provos. Improving host security with system call policies, 2002.

[12] R. Sekar, C. Ramkrishnan, I. Ramamkrishnan, and S. Smolka. Model-carrying code : A new paradigm for mobile-code security, 2001.

[13] R. Sekar and P. Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. pages 63–78.

[14] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, David Andersen, and Jay Lepreau. The Flask security architecture: System support for diverse security policies. pages 123–139.

[15] Kenneth M. Walker, Daniel F. Sterne, M. Lee Badger, Michael J. Petkac, David L. Shermann, and Karen A. Oostendorp. Confining root programs with domain and type enforcement (dte). In *Proceedings of the 6th USENIX Security Symposium*, 1996.

Appendix A

Applications inferred as Trusted from exec relationships

- acpid
- anacron
- arping
- atd
- automount
- avahi-daemon
- awk
- basename
- bash
- beh
- bluez-pin
- bonobo-activation-server
- cat
- cc1
- change_console

- chkfontpath
- chmod
- chown
- clock-applet
- consoletype
- cpp
- cp
- crond
- cups-config-daemon
- cupsd
- cut
- dbus-daemon
- dbus-launch
- Default
- dhclient-script
- dhclient
- dircolors
- dmesg
- dmidecode
- eggcups
- egrep
- ethtool
- fgrep
- find
- free

- gconfd-2
- gconf-sanity-check-2
- gdm-binary
- gdmflexiserver
- gdmgreeter
- gdm
- gnome-keyring-daemon
- gnome-panel
- gnome-power-manager
- gnome-screensaver
- gnome-session
- gnome-settings-daemon
- gnome-vfs-daemon
- gnome-volume-manager
- gnome-wm
- gpm
- grep
- hald-addon-acpi
- hald-addon-keyboard
- hald-addon-storage
- hald-probe-input
- hald-probe-pc-floppy
- hald-probe-serial
- hald-probe-smbios
- hald-probe-smbi

- hald-probe-storage
- hald-probe-volume
- hald-runner
- hald
- hal-system-storage-cleanup-mountpoints
- hal
- hcid
- head
- hid2hci
- hidd
- hostname
- hpiod
- hpssd.py
- hp
- http
- id
- ifconfig
- ifup-aliases
- ifup-eth
- ifup-post
- ifup-routes
- ifup
- init
- ipcalc
- ipp

- ip
- iptables-restore
- iptables-restor
- irqbalance
- iwconfig
- khelper
- klogd
- krb5-auth-dialog
- kudzu
- logger
- lpd
- lsmod
- ls
- make
- mapping-daemon
- metacity
- mii-tool
- mingetty
- mixer_applet2
- mkdir
- mkfontdir
- mkfontscale
- mktemp
- modprobe
- mount

- nautilus
- ncp
- newaliases
- nice
- nm-applet
- notification-area-applet
- pam_console_apply
- pam-panel-icon
- pamimestamp_check
- parallel
- pidof
- portmap
- prefdm
- printconf-backend
- python
- rc.sysinit
- rc
- restorecon
- rfcomm
- rhgb-client
- rhgb
- rm
- rpc.idmapd
- rpc.statd
- runlevel

- S04readahead_early
- S04readahead_ea
- S05kudzu
- S06cpuspeed
- S08iptables
- S09isdn
- S10network
- S12syslog
- S13irqbalance
- S13portmap
- S14nfslock
- S15mdmonitor
- S18rpcidmapd
- S19rpcgssd
- S22messagebus
- S25bluetooth
- S25netfs
- S26apmd
- S26hidd
- S28autofs
- S40smartd
- S44acpid
- S50hplip
- S55cups
- S55sshd

- S80sendmail
- S85gpm
- S90crond
- S90xfs
- S95anacron
- S95atd
- S95firstboot
- S96readahead
- S98avahi-daemon
- S98cups-config-daemon
- S98haldaemon
- S99local
- scsi
- sdpd
- sed
- selinuxenabled
- sendmail
- serial
- sessreg
- setxkbmap
- sh
- sleep
- smartd
- smb
- socket

- sort
- ssh-agent
- sshd
- sysctl
- syslogd
- tee
- touch
- trashapplet
- true
- udevd
- udevinfo
- udev_run_devd
- udev_run_hotplugd
- umount
- uname
- uniq
- usb
- which
- wnck-applet
- Xclients
- xfs
- xkbcomp
- xmodmap
- Xorg
- xrdb
- Xsession
- xsetroot