Light-weight proactive approach for safe execution of untrusted code

A Thesis Presented
by
Anupama Chandwani

to
The Graduate School
in Partial Fulfillment of the Requirements
for the Degree of
Master of Science
in
Computer Science
Stony Brook University

May 2010

Stony Brook University

The Graduate School

Anupama Chandwani

We, the thesis committee for the above candidate for
the degree of Master of Science,
hereby recommend acceptance of this thesis.

Professor R. Sekar, (Advisor)
Computer Science Department, Stony Brook University

Professor Scott Stoller, (Chairman)
Computer Science Department, Stony Brook University

Professor Robert Johnson, (Committee Member)
Computer Science Department, Stony Brook University

This thesis is accepted by the Graduate School.

Lawrence Martin
Dean of the Graduate School

Abstract of the thesis

Light-weight proactive approach for safe execution of untrusted code

by

Anupama Chandwani

Master of Science

in

Computer Science

Stony Brook University

2010

Today's malware attacks are cleverly crafted and cause huge loss of resources. Existing proactive defense mechanisms against malware include isolation, sandboxing, information flow tracking, *etc.* These mechanisms completely block information flow on the system. But sometimes we do need the functionality provided by software from untrusted or unknown sources that are not malicious. The problem that we try to solve here is of executing this untrusted code on a real system so that it can coexist with other applications in the same environment, thus allowing safe information flow. At the same time we want to protect the system so that it does not get compromised due to untrusted information. Available approaches for information flow tracking are intrusive and require significant kernel changes, thus making them difficult to port and maintain across different operating systems or even newer version of the same OS. We propose a light-weight approach, based on *userid*, for proactive integrity protection and safe execution of untrusted code. We mediate all information flow in the system in order to provide protection from sophisticated malware and attacks.

To my Family,

Teachers

and Friends.

Contents

Acknowledgments

I take this opportunity to express my deepest gratitude to my advisor, Prof. R. Sekar, for his constant support, encouragement and guidance. I am greatly influenced by his dedication and high standards of research. I also wish to extend my sincere thanks to Prof. Scott Stoller and Prof. Rob Johnson for consenting to be on my defense committee and offering valuable suggestions.

I would like to thank Weiqing Sun, my mentor at Secure Systems Lab, for his patience and guidance during my initial phases of learning. I appreciate the continuous and ever increasing support by Arvind Ayyangar during my entire thesis. I thank Alok Tongaonkar, for his advice and encouragement.

I truly had a great time while closely working with Srivani Narra, Arvind Ayyangar, Bhuvan Mital, Prachi Deshmukh, Abhiraj Bhutala and Sumati Priya. I also thank Ashish Mishra, Alireza Saberi, Tung Tran, Riccardo Pellizi and Praveen Kumar for always keeping a lively work atmosphere in Seclab. In addition, I would like to extend my gratitude to Saatyaki Rajamani, Ravi Sarawadi and Anuradha Chandwani for their invaluable suggestions on the thesis report.

Finally, I would like to thank my parents and Darshan, without whose support this work would not have been possible. This research is supported by ONR grant N000140710928 and AFOSR FA9550-09-1-0539.

CHAPTER 1

Introduction

1.1 Problem

Malware is a serious problem on today's operating systems. Stealthy malware can cause huge loss of money and disrupt functionality. Malware can also misuse user confidential data. Today's malware largely targets end user desktops and is getting progressively sophisticated and adaptive to existing defense mechanisms.

Malware often takes the form of an executable. However, it may also enter a victim system in the form of malicious data. Such data can compromise benign applications on the system, and use their privileges to embed itself deeply in the victim. In this case, non-malicious applications are not intentionally helping malware, but they can have vulnerabilities in their code. Malicious input is designed to exploit these vulnerabilities.

1.2 Existing defense mechanisms

Defense mechanisms for protecting a system against malware can be broadly classified as reactive and proactive. Reactive approaches such as signature-based scanning and behavior monitoring are not sufficient for sophisticated malware. Moreover, they cannot handle Zero-day threats [1], which pose a significant threat to computer security today.

Proactive approaches for malware defense, on the other hand, rely on more systematic approach to solve this problem. Unlike reactive approaches, they do not reply on knowledge of existing malware code or their behaviors, and hence can protect against unknown malware. We will briefly discuss some of the existing proactive mechanisms with their problems and how our approach addresses them.

- **Policy based confinement of untrusted applications:** The aim here is to confine untrusted applications[1] to regulate access to system resources [3, 10, 8, 21]. The problems with this approach are,

    1. Confinement policies are complicated and difficult to define.
    2. Confining untrusted applications alone cannot be a complete defense mechanism. Since the effects of confined execution are visible on the system, a non-malicious applications can accidentally use it.

       For example, Acrobat reader used to view a malicious pdf file can compromise it.

[1]Untrusted applications are those that come from untrusted sources and can be potentially malicious

It is therefore necessary to regulate the execution of all applications on the system. Using mechanisms like sandboxing [3, 10, 8, 21] to do so will further complicate policy specification.

- **One-way isolation:** Isolating the execution of untrusted applications is another approach for confining their execution [25, 26, 14, 24, 12]. An isolated environment allows restricted access to an untrusted application by providing a copy of the system resources that it can/need to access. Interaction of other applications on the system with untrusted code/data is prevented by not allowing the effects of execution inside isolated environment to be visible on the rest of the system, where benign applications execute.

  The limitation of this approach is that the results produced inside the isolated environment cannot be used outside it. Based on the configuration of different untrusted applications, there might be a need to have multiple isolated environments. As each of these environments is isolated from one another, the whole system becomes difficult to use.

- **Information flow tracking:** Information flow tracking through label propagation can be an effective approach for malware defense [22, 4, 7]. Integrity of the system is ensured by preventing any influence from untrusted code/data to rest on the system. This approach has two parts,

  1. Each file carries its integrity label.
  2. Policy enforcement is based on subject and file label.

Unlike isolation, this approach allows the execution of untrusted code on the same environment, making the results usable. However, information flow based approaches are uncommon. The available implementations [22, 4] are very OS-specific. Moreover, they require significant changes to the OS kernel, thus making it difficult to port even among versions of the same OS.

1.3 Our approach

Our goal is to provide light-weight proactive integrity protection. We aim at using existing mechanisms for file labeling and policy enforcement. Our approach requires no kernel changes or extensive changes at the user level. This makes it easier to port the system across different flavors of UNIX, or at the least, avoid significant work each time the kernel is updated.

Our proposal is to use userids as a mechanism for confinement and information flow tracking. Userids serve two purposes:

1. They serve as "file labels[2]".

2. They serve as a basis for policy enforcement.

The concept of discretionary access control (DAC) with userid is commonly used in all modern operating systems. Hence, our approach is generic and relatively easy to port.

In addition to the use of userids, our approach adds additional functionality to standard system libraries, specifically *glibc* on GNU/Linux. We know that such libraries can be bypassed by malicious code. This

[2]In this context, we can map each integrity level into a (owner, group) pair, creating new groups if needed, to capture different integrity levels.

mechanism is therefore used only in contexts where there is no incentive for an application to attempt such bypass. In particular, we use library modifications to ensure that a high integrity subject downgrades itself before reading low integrity data.

We realize that strict information policy enforcement can impact usability of the system. We develop carefully controlled mechanism for permitting integrity aware applications to override strict information flow policies in certain context.

1.4 Summarizing our approach as an end-to-end mechanism

Trust level of new applications and its files is based on its source. This requires a mechanism to label all new files on the system during installation with appropriate trust level. Modern operating systems use an installer to download and install packages. For example, Ubuntu uses Aptitude, Red Hat Linux uses RPM, *etc.* Moreover, the installer executes application specific scripts while installing and uninstalling it. Therefore, it is required that the installation happens at the integrity level of the application. We develop a secure installer based on an approach by Weiqing Sun et al. [23]. This is used to achieve the objectives of labeling new files and executing installation scripts at application integrity level. Other mechanisms that files use to enter the system also need to propagate their trust level. For example, *ssh*, file sharing, *etc.*

Our approach confines all application on the system. We use DAC to enforce policies that prevent subjects at a certain level from corrupting files at a higher integrity levels. For all the files on the system, we encode the value of its integrity in the (owner, group) pair. Since it is possible to modify owner/group and permissions on files, it is necessary to make sure that our encoding exists across all such modifications.

We also need to prevent subjects at a certain level from randomly lowering their integrity level. This can happen when the subject reads from files which have an integrity levels lower than itself. We use system call interception, implemented in a modified system library, to achieve this objective.

The rest of the thesis is organized as follows. Chapter 2 presents the design of our approach. Chapter 3 presents the implementation and issues. Chapter 4 provides the mechanism we plan to use for evaluating our system. Related work on this problem is discussed in Chapter 5. Finally Chapter 6 concludes this report.

CHAPTER 2

Design

Our design for proactive integrity protection achieves the following goals:

- *Using portable mechanisms.* We use the existing mechanism of userid for tracking information flow. Additional policies are implemented using system call interception at the shared library[1] level. Both these mechanisms are generic and common in all flavors of UNIX, thus making our approach portable. We avoid any changes to the kernel and work with minimal changes at the user level. This largely reduces effort of maintaining code across newer versions.

- *End-to-end trust handling.* An application's trust can be determined by the trust of its code/executable provider. Installers like aptitude, rpm, use predefined repositories to obtain packages. We add functionality to an existing installer and assign trust to each such provider. Our approach associates package trust to an integrity level on the system. This gives a complete end-to-end trust handling mechanism.

[1]We use binary interception on libraries of *glibc* and *pthread*.

- *Address entire life cycle of an application.* An application entering the system is assigned a userid corresponding to its trust level. This userid is made the owner of all files in the application. We have mechanisms to ensure that all phases of the application's life cycle, installation, execution and uninstallation, happen in the context of this userid.

Having spoken about these goals, let us discuss in detail the design of our approach to achieve these goals.

2.1 Policy enforcement

Our approach involves confining all applications on the system. Untrusted applications are confined to preserve the integrity of trusted files. Whereas non-malicious applications are confined to protect themselves from being compromised when exposed to lower integrity input.

The policy enforcement has two parts:

1. No privilege escalation: Subjects at certain level should not be allowed to corrupt higher integrity objects or subjects.
2. Enforcing regulated privilege downgrade: Subjects should be prevented from performing operations that will lower its integrity.

2.1.1 No privilege escalation

Subjects at a certain level should not be able to write to objects at higher levels. This is accomplished by ensuring that a subject will always run at its own integrity level. Moreover, we set the file (owner, group) and permissions in such a way that it can be modified only by subjects running at its own or higher integrity levels.

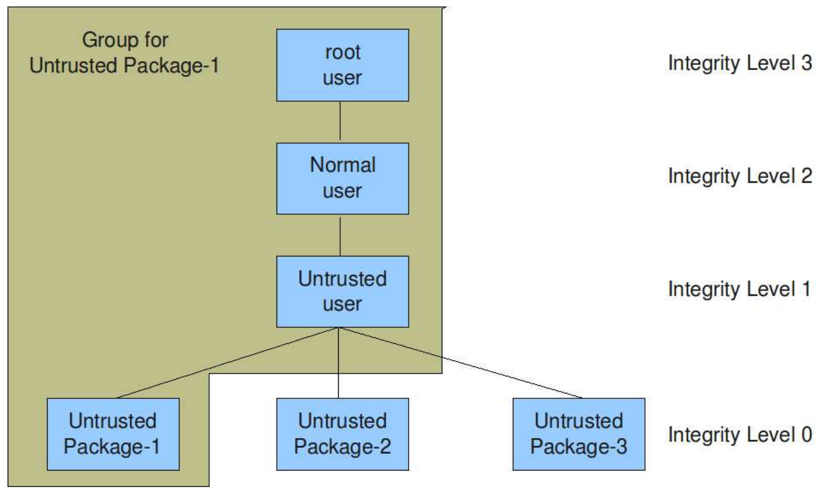

Figure 1: Integrity lattice

This is achieved by creating a trust lattice with userids as nodes. A userid in the lattice represents a unique trust level[2]. A group is used to define the set of users (levels) that can safely access the file at a certain level. Thus, a group g_l is created for each level l, which includes userids of all subjects that are intended to run at $level \geq l$. Group permissions are set to specify type of access by these subjects. No permissions are granted to users outside the group.

In figure 1, we have a lattice with six different userids across four integrity levels. Here, *Integrity level 3 > level 2 > level 1 > level 0.* Normal user is the user used to login. Untrusted user is created for normal user to downgrade while accessing untrusted data. Untrusted package-1 to Untrusted package-n is created for each untrusted package provider.

[2]There can be multiple userids in the same integrity level.

The user group for files with untrusted package-1 as owner is shown by the darkened portion. Subjects running with any of the userids in this group have access to objects owned by untrusted package-1. The type of access is specified by group permissions on those files.

2.1.1.1 Enforcing no write up

Files are assigned (owner, group) pairs based on its integrity. A group g_l is assigned, when its owner is at level l. With ownership of (l, g_l), the file is made accessible only to subjects with userids at a $level > l$. DAC policy enforcement is used to realize this policy.

Permissions on some existing files on the system need to be changed for enforcing this policy on them.

- **world writable files:** Typically, these files are readable/writable by the world. In other words, group information is unused. We can define a group g_l consisting of all users whose subjects can safely modify files of this level l. We then change world permissions to group permissions.

- **group writable files:** In this case, group name is already used. This is practically not an issue. A userid is put in a group when it needs specific access to that file. It is unlikely that the system would work if those permissions were taken away. In other words, we do not need to do anything since file access is already restricted.

- **owner writable files:** Again, the system has specified restricted write permissions. Users from arbitrary levels cannot modify this file. When we change its owner userid, the "no write up" policy is automatically enforced.

A file's integrity level is a function of its (owner, group) and write permissions. Its integrity is logically equal to the lowest integrity level of user that could modify it.

2.1.1.2 Enforcing no exec up

A subject belonging to an integrity level, should be allowed to execute only at its own level or at levels under it in the integrity lattice. This is achieved by setting the userid of subjects corresponding to their trust level. Linux does not support an *exec* with arbitrary userid.

- **Running setuid executables**
  A setuid executable can violate our policy of "no exec up". We need a mechanism to ensure that a subject at a certain level does not have execute permissions on setuid files that execute at a higher level. To accomplish this, we assign group[3] executable permissions to the setuid subject. However, this approach is not applicable for setuid files that are originally group-writable and world-executable. Since modifiable[4] setuid executables are themselves rare, such a scenario would be much rare.

2.1.2 Enforcing regulated privilege downgrade

Information flow from lower level files to higher level subjects is inevitable. For example, Acrobat reader, Music player, *etc.* have to accept files of all integrity levels as input. An approach that prevents non-malicious applications to read untrusted input largely obstruct usability. However, such applications can be exploited by untrusted input

[3]The group is a set of all userids that can execute this file safely.

[4]Such as scripts, that use same file for modifying and executing. Binary setuid executables would not have write permissions.

and get compromised. This in turn compromises the system by letting untrusted input use its privileges.

It is therefore required to execute the instance of a higher level subject at the level of untrusted input. This can be achieved by a mechanism to downgrade a process when exposed to lower integrity input. Due to our design of integrity lattice and the concept of group, g_l, file permissions cannot be used to prevent access to a lower level file by a subject from higher level. So we need another mechanism to enforce these policies. We enforce the policies of downgrading at system call level and achieve this by using library interception.

This mechanism is used by subjects to lower its privileges. An application does not get any extra abilities by bypassing this mechanism. This mechanism is used to protect itself from lower integrity input. If application intents to corrupt files at its own level, it can do so without depending on reading lower integrity files. It is thus safe to assume that applications will not bypass this mechanism.

2.1.2.1 Enforcing no read down

This policy allows a high integrity subject to open low integrity files, only after lowering its integrity level to that of the file. A subject can specify up to what level it can be downgraded. Any attempt to open files lower than that level should be denied[5].

2.1.2.2 Regulated exec down

When a subject at higher level exec's an application at lower level, our policy is to run the lower level application in its own trust level.

[5]Need special assistance with programs that handle files of different levels at the same time - For example, *tar*, *cp*.

This ensures that lower level executables cannot compromise parent application and misuse its privileges. An enhancement to *exec*[6] is made at library interception, which allows executing a process with arbitrary userid. Based on different combinations of parent's level, l_p, executable's integrity level, l_e and lowest level to which the parent process can downgrade level l_d, there are following choices for *exec*:

- When $l_p > l_e$, execute at level l_e. This can be achieved by using the enhanced *exec* call.
- When $l_p < l_e$ but $l_d > l_e$, execute at level l_p. All applications are executed at the level that can influence its execution.
- When $l_p < l_e$ and $l_d < l_e$, block execution. Return error.

To ensure that a lower level subject cannot use the enhanced *exec* wrapper to exec files at arbitrary levels, we need multiple wrappers for each integrity level with their execute permissions set appropriately. So a wrapper for a certain level will be allowed to execute at userids strictly less than its own integrity level.

2.1.3 Limited downgrade support due to userid

Our policy enforcement needs a mechanism to lower a subject's integrity level from its existing level. This is needed when there is information flow from a file with lower integrity level to a subject with higher integrity level. In general, information flow occurs when a subject executes another subject or opens a file for reading. Depending on these two possibilities, we need a mechanism to downgrade a subject when it starts execution (exec-time) and when it opens a file (run-time).

[6]A setuid executable is provided, which changes the userid to required level and then invokes the intended *exec* with original arguments.

There can be only one integrity level that is associated with userid 0. This is because operations that are currently done with root privilege will need to be done with root privileges even in our system, or else accesses (such as file writes) will fail. So we do not prevent a subject with userid 0 from accessing a file or other operations that should be permitted only for high integrity processes. This limits our integrity protection mechanism and we do not help improve security of root login sessions.

However, we do provide an invulnerable[7] execution mode. Subject executing in this mode can continue to execute at high level even after being exposed to low integrity data. This mode should be used only by integrity aware applications. More about it will be discussed in section 2.3.

2.1.3.1 Downgrading at exec-time

- Linux allows root owned processes to change its userid to any arbitrary value. Downgrading is implemented by using the *setuid*() call. The userid of the subject is changed to that of the executable file. This is enforced in the intercepted library.

- Ability to change the userid of non-root processes, to arbitrary values, is not supported in Linux. We achieve this by implementing a function $exec\_from\_level$ (*userid*, *executable*, $args$).
  This function is a setuid program and will always execute with root privileges. It is invoked from the intercepted library, when an *execve* system call is made. The *executable* and $args$ are parameters of original *exec* call. From the intercepted library, this

[7]We do not give unregulated access in this mode. Processes are still subject to downgrade if it receives untrusted input on channels that expect only high integrity data.

program is invoked by changing the parameters to original *execve* system call. This program eventually makes the same *execve* system call with original parameters, but with the userid assigned by our policy.

2.1.3.2 Downgrading at run-time

It is not right to change a process userid to an arbitrary value while it is executing. The process does not expect this change and hence might not be able to handle it gracefully. Our approach of using userid as file label has imposed this limitation. However, a file carries an effective userid, real userid and saved userid [9] during execution. These values are interchangeable during the execution of process. Therefore a process can handle userid change within the limited options.

- For root owned processes, the same mechanism for downgrading (using *setuid*()) can be used even at run-time.

- For non-root processes, *setresuid*[8] is used. This allows interchanging the values of the process userids. Effective userid is used for all permission checks, we use real userid and saved userid to encode userids of each level at which the process can downgrade. However, this approach has these following limitations:

    1. A process can execute at only 3 integrity levels during its execution. Also these levels (userids) have to be pre-encoded when the process executes.

    2. Supplementary groupids cannot be surrendered using this downgrading mechanism. We can overcome this by starting downgrade-able subjects with empty supplementary groupid

[8] *setresuid* (*ruid*, *euid*, *suid*) can be used to interchange the userids of a process.

list or with groups that the subject can legitimately have even after downgrade.

However, if a user before downgrading, wants to use the permissions provided by these supplementary groups, we can provide a command, *sgdo* just like *sudo*, that will add the required groups to the user's supplementary group list. It is a change in user behavior but we expect users will get used to it just like using *sudo*.

As mentioned, the *execve* system call can be intercepted to assign the downgrade-able userid of a subject. However, a more complete approach would be to set default values for each process which can be overwritten in *exec*. This is achieved by modifying the *login* utility to propagate default downgrade-able integrity levels.

As our run-time downgrade mechanism is based on *setresuid*, a subject can be downgraded to only one (or maybe two) other levels. This level will be used for accessing all low integrity data and hence needs to be shared across all untrusted applications.

2.2 End-to-end trust mechanism

The integrity of a file entering the system is determined by trust level of its source. We need a mechanism to correlate trust of file source and its integrity level in the lattice on our system.

2.2.1 Integrity of downloaded files

Files downloaded using a web-browser should inherit the trust of web site from where it was downloaded. This can be achieved by assigning trust to each web site and associating it to an integrity level in our lattice. By running the browser with the userid[9] corresponding to the trust level of web site, we ensure that files downloaded from this instance of browser will inherit web site's integrity level. We use our mechanism of "exec-down" to downgrade an instance of browser to the integrity level of the web site.

Currently, this mechanism of running browser with web site trust is not automated. User needs to open different instances for accessing web sites of different trust levels. By combining this mechanism with a method to infer web site trust, user intervention can be completely removed.

2.2.2 Integrity of installed packages

For packages installed using an installer, the trust of the package provider is propagated to the package. In our design, we define repository as a basic unit of trust. Each repository is associated with an integrity level in the lattice. Let us see how trust is associated from the repository to its packages.

When we trust a package from a repository, we trust the package provider and the repository for having rightly received the package from

[9]Firefox allows running multiple instances as independent processes by specifying $-no-remote$ option.

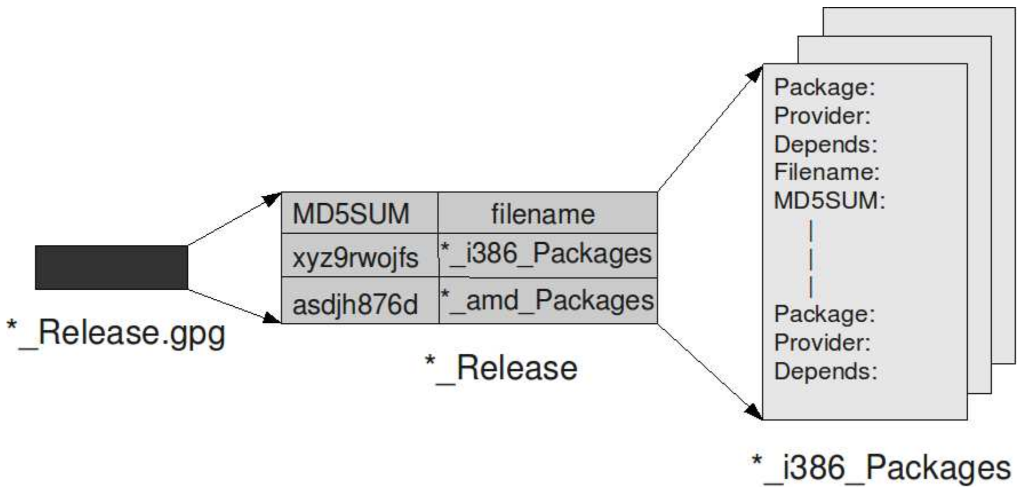

Figure 2: Repository files for Apt

its provider and checked it. If the same package[10] from another repository is not trusted, it means that we trust the package provider but not the channel of communication between the provider and repository, or the checking mechanism of the repository. Therefore, this repository cannot be trusted for any package it provides.

Now consider another package provided by the repository we trust. If we do not trust this package, it means we do not trust its provider. Irrespectively of the repository providing this package, we do not trust it.

All packages from a repository, inheriting the same trust level, will correspond to a single level in the integrity lattice. Moreover, packages from a repository will have the same (owner, group) pair. This gives a complete end-to-end mechanism, where a package trust is determined based on its provider. On entering the system, files and executables

[10]When we say the packages are same, it does not translate into packages of same functionality, rather means downloaded package files have same checksum.

from this package are confined to a trust level corresponding to its provider.

In figure 2, an overview of database files in apt is provided. Each repository in apt is signed for its contents, which in turn contains information about packages provided by it. So inherently, apt is based on trusting packages based on its repository. The $< repository_name >$ $._packages$ file provides information about each package and its checksum. This information is used by apt for determining the integrity of downloaded package. Depending on the installer, we can migrate our unit of trust from repository to per package.

2.3 Granular trust for applications

So far we divide the set of applications on the system as untrusted (which can be malicious) and non-malicious. The non-malicious applications are vulnerable to malicious input. Therefore, we have a policy that will lower a subject's level to the integrity level of the object it reads. This is coarse grain trust. Certain well written applications can handle low integrity data without a threat of compromising itself or the system. Examples of such applications are *tar*, *grep*, *ssh*, *etc.*

These applications need to handle data with multiple trust levels in a single instance. If they are downgraded, as our policy states, it will not be possible to execute them any further. So we need a mechanism to indicate that a given subject is *invulnerable* and can safely handle lower integrity data.

Therefore, invulnerable applications need a mechanism to override the downgrade policy. However, these applications cannot be expected to handle malicious input in any context. For example, *tar* or *grep* cannot protect itself from a malicious *glibc*. Similarly *ssh* reading from malicious configuration file will get compromised. Hence, the mechanism for overriding downgrading policy should not provide blanket trust to these applications.

A mechanism that allows specifying the integrity level at which the file needs to be accessed is required. We can provide this functionality at various levels. Lets us see each of them.

- **Specified by modifiable application**
  An integrity-aware application can explicitly provide the integrity level it needs to execute at while accessing a file. We provide special functions that allow an application to achieve this. This function takes filename and execution level as input.

- **Specified using modified library**
  The above approach requires changes in the application code. Applications can use variants of default system calls to indicate the same behavior. In general, *open* and *exec* system call need to be provided this information. Additional flags can be used to indicate the intent of user and hence downgrading can be overridden. Similarly, environment variables can be used to indicate filenames, which when appear as a parameter, different policy at library interception can be applied.

The environment variable approach[11] enables a knowledgeable user to set up the files that can be trusted (primarily because the user knows that a trusted process is going to use it in a safe way, but also may be used as a mechanism to "endorse" a file with a low label).

- **Inferred by user behavior**
  The mechanisms mentioned above require changes in the system, also user needs to explicitly convey this information in some way. For users who do not want the burden of having to set up the environment variables, we propose a mechanism of inferring this information. We track all possible methods of user specifying a file to an application, i.e., files explicitly mentioned by user. We infer that all explicit file accesses to invulnerable applications should happen without downgrading the application level.

  The logic for inferring this is that, since the user specified the file manually, the file access is intentional. We assume the user will not intentionally give a low integrity file to an application in a context that the application can be compromised. Our enforcement policies take care that any unintentional access in this context is blocked/downgraded. File names appearing in the following cases should be considered intentional,

  1. Filenames appearing in environment variables or command line arguments.

[11]This approach fails to distinguish between contexts, e.g. a file may be safe to open as input without downgrading a process but not if it is used as configuration file. However, this is a problem only if someone is trying to fool the application by supplying the same file as input and configuration. But this is not the case here: since we are dealing with invulnerable process its parameters (or environment) could not have been influenced by attackers.

2. Filenames obtained using standard file dialog boxes.
3. Filenames appearing in high integrity files, like configuration files.

However, this mechanism of inferring intent based on filenames might not be sufficient in certain special cases. For example,

- *Filenames specified using relative path.* We handle this case by resolving filenames to their absolute path from its current working directory. However, in certain cases the application might resolve the relative path of a file after changing its working directory. This behavior is rare for a well written application, since it expects the user to know about changes in environment before accessing the file it specified.
- *Application adding suffix to filenames.* An application usually appends a pre-encoded suffix to user specified file while creating new/temporary files. For example, creating new log files after a certain size is reached. Creating a temporary copy of existing file. Since these files are newly created, they do not pose the threat of low integrity information flow. Temporary copy of a file will propagate original file label.

  However, when a user ends a filename in "*", we consider all filenames starting with this string as explicit.

For the examples mentioned above, the default policy of downgrading the subject will hold when it implicitly reads low integrity files like malicious library or configuration file. But when files are specified by either of the above mentioned mechanism, subject continues to execute with its own privileges. This extra granularity in trust, for a class of invulnerable applications, helps achieve usability while realizing the original goal of preserving integrity.

CHAPTER 3

Implementation

We use Ubuntu 8.04, Linux kernel version 2.6.24-27, for our prototype. Glibc version 2.7 is intercepted using an in-house binary interceptor. Secure installation is implemented on apt 0.7.9ubuntu17.2 for i386 and dpkg 1.14.16.6ubuntu1 (i386). Note that there was no code change in any of the above mentioned packages. We add our functionality as a wrapper to the existing binaries. Assumptions are made on the interface provided by the package, which is likely to remain consistent. This makes our prototype easy to port and maintain. We successfully ported it to Ubuntu 8.10 without any changes in code. This section describes details of each of the module used in our prototype.

3.1 Creating integrity lattice

For encoding integrity information in the (user, group) pair for a file, new users and groups need to be added. Each new user defines a level in our lattice. The existing users on a system are associated with an integrity level.

The concept of group g_l, corresponding to each user at level l, is added to define a safe "world" for files. The world permissions of files are assigned to this new group, making the file inaccessible to lower integrity subjects. This enforces the "no write up" policy.

The group g_l for each user at level l contains userids of all users above its own integrity level and of its corresponding owner. Users at the same level are not related. This helps in ensuring that an application at a level is not allowed to corrupt files of other owners.

Some applications like apache, gnome-games, *etc.* create their own userid and expect to execute in its context. This is allowed by associating the new userid to an existing level in the integrity lattice. This ensures that all policies applicable to its integrity level are enforced on the package.

3.2 System call Interception

System call interception is achieved at the shared library level. Libraries like *glibc* and *pthread*, that are used by applications to make system calls, are intercepted. The binary interceptor used for this is developed as an in-house project in Secure Systems Lab, Stony Brook University. The interceptor provides the following functionalities:

- **Control on syscall enter**
  The intercepted code in the library invokes a handler just before a system call is made. System call number and all its parameters are made available in this handler. It is possible to read and change these parameters before returning to the system call site. This makes it possible to implement our enforcement policies at the

system call entry level. If a policy is being violated, the handler supports aborting the original system call.

- **Control on syscall exit**
  The interceptor also provides control on system call exit. After a system call returns to the library, another handler is invoked. This handler gives access to return values of the system call.

- **Making other system calls**
  It is possible to invoke system calls from within the handler invoked by the intercepted code. This allows us to implement policies in the system call handler. The handler avoids recursion by bypassing interception for the system calls made from the handler.

Interception should be applied to almost all executables. This is because any process that does not have intercepted library has the potential to compromise itself, thus threatening the integrity of the system. If the application bypasses the library interception, it is because its environment is influenced by information flow from low integrity. In this case, the application should already be running at low integrity. A non-malicious application cannot be influenced to bypass this mechanism while executing at high integrity.

The system call interceptor is used for implementing policies for preventing a subject from randomly downgrading. Let us see how the policy determines if a downgrade should be allowed.

3.2.1 Enforce no exec down

This is enforced at the entry of *execve* system call. The parameters are checked for determining the desired action. If a downgrade is needed, the parameters to *exec* are right shifted and *exec_from_level* is invoked instead. This function takes original parameters to exec and userid with which it should execute.

The wrapper, *exec_from_level* needs to be careful to ensure that userid changes are permanent, group changes are permanent, supplementary groups are reset properly and environment and command line arguments are checked. The execute permissions of this wrapper allow execution only by users with higher integrity level.

3.2.2 Enforce no read down

This is similar to enforcing no exec down. The policy is implemented at the entry of *open* system call. Here, we can downgrade the subject to only a set of pre-encoded integrity level, as described earlier. Note that we downgrade process to the level of the file being opened, but we do not ensure that the userid be file owner.

3.3 Granular trust for invulnerable applications

We provide a mechanism to specify granularity in application trust. A small subset of non-malicious applications is defined as invulnerable. Such integrity aware applications can use our mechanism to use lower integrity input without lowering its privileges. We implement support for this mechanism in the intercepted library, where downgrading policies are implemented. Hence, it needs to be communicated to the library that the system call is being made by an invulnerable application and in a context where it can handle lower integrity data.

Along with communicating invulnerability information to the library, we provide functions that can control the integrity levels at which an application executes.

3.3.1 Application functions

Integrity aware applications call these functions from its code to indicate the context of information flow.

1. *open_downgrade (..., level)* - Allow opening the file if the file level $l_f \geq level$. The process may downgrade as needed. This function indicates that the subject cannot handle input lower than specified, in this context.

2. *open_trust (..., level)* - Allow opening the file if the file level $l_f \geq level$. The process may **not** downgrade even if the file's integrity is lower than its own. This function indicates that the subject can handle input up to the specified level, in an invulnerable manner. However, any information flow from a level $l < level$ can compromise it.

3. *downgrade (level)* - Downgrade the userid of the calling subject to the specified level. This can be used by an application, when it realizes that it may be a potential threat to system security, at the level it is currently executing.

4. *exec_at_level(..., level)* - Execute the specified file at a given level. This in turn uses the *exec_from_level()* function. An application is allowed to execute the file at any level lower than its own.

An application is not forced to use these functions for using our policy enforcement. This is an added mechanism for an integrity aware application to have fine grain policy enforcement. This mechanism however requires modification to application code. Though it is acceptable to assume an application to make changes based on its intentions and security requirements. We provide added mechanisms to achieve this functionality for existing application.

3.3.2 Flags to system call

We provide variants of *open*, *creat* and *exec* system call to take added flags as parameters. These flags achieve the needed flexibility as follows:

1. *min_level* - Opening a file with level $l_f < min\_level$ should be denied.

2. *invul_level* - Opening a file with level $l_f \geq min\_level$ should not cause any downgrading in the subject's execution level.

3. *obj_min_level* -This is the default level for newly created objects. The functionality of this flag will be limited based on the permissions of subject using it. On Linux, non-root processes do not have permissions to create arbitrary userid files.

4. *exec_level* - This is the highest default level at which subjects will be exec-ed. Files with level $l_f < exec\_level$ will execute at l_f level.

This mechanism requires changes in shared library where system calls are made. For a completely transparent approach, the application can specify filenames and integrity levels by setting appropriate environment variables. We created an environment variable, TRUSTED_FILES, to specify filenames that can be opened by an invulnerable application without downgrading.

The functionality provided by these functions and flags, is used for finer granularity in policy enforcement. The functions are implemented in a way that it will not be able to specify random integrity levels of operation. These functions can be used only to specify variations binding to the existing policies.

3.3.3 Inferring from user behavior

For users who do not want the burden of setting up environment variables or using modified libraries, we infer these values in the following way:

1. *min_level* - This flag is calculated using the information of integrity levels of all the files opened by the process. When a file of level $l_f > min\_level$ is opened, min_level is made equal to l_f. On a file close, the *min_level* is recomputed based on the integrity of all open files at that moment. However, the value of *min_level* is always maintained greater than the process downgrade-able level, $l_d \geq min\_level$.

2. *invul_level* - As defined in the design section, we term all explicit file access as invulnerable. The mechanism developed for identifying explicit file access is as follows:

    - File names appearing in environment variables and command line arguments are considered as explicit access. This is achieved by getting the return value from *read* system call of *stdout*. Also environment variables are scanned to locate filenames. A list is maintained for all explicit file names.

    - File selected by GUI is considered explicit. The gnome library provides a finite set of widgets that can be used to specify file names. Application uses these inbuilt gnome widgets. We retrieve the file name from these widgets by modifying their code in *glib*.

- In case of a configuration file, specified by user, filenames appearing in it should also be considered as explicitly specified by user. This is a configurable option and can be disabled completely. It is achieved using an information flow tracking mechanism, based on content. The idea is to associate trust to data based on its origin. On a *read* system call, the contents read are saved in a hash table along with the integrity of its origin (file it was read from). On the *write* system call, the integrity of data is determined by finding its match in the hash table. This mechanism is also developed in Secure Systems Lab, Stony Brook. We use it to identify if a filename was read from a high integrity file. All such filenames are to be considered explicit and added to the list.

Once this list of filenames is populated, the *open* system call checks the file name against this list of explicit file accesses. If a match is found, we consider the file access as intentional and allow to open it without downgrading the subject. Note that the *invul_level* makes sense only for invulnerable applications. All file accesses, explicit or implicit for all other applications will abide to the downgrading policy.

3. *obj_min_level* - Unless explicitly specified, the default value of this flag is same as the process's downgrade-able level, l_d.

4. *exec_level* - This value is interpreted to be the minimum out of executable file's level, l_f and parent process's level, l_p.

3.4 Modification to existing utilities

- *chmod* - Changes in permission to file owner is unregulated. If the file group is as defined by the lattice, i.e., all users in the group have level $l \geq g_l$, any permission changes are allowed. For any group other than g_l, only *read* permission can be assigned. Similarly, we allow only *read* permissions to others. This is done for "no write up" policy to be enforced across *chmod*.

- *chown* - A change in file owner to another user from its own integrity level or userids under it in the lattice is allowed. Changing the file owner to a user with higher integrity is not allowed.

- *chgrp* - The group owner of a file can be changed to any other group. However, we remove all permissions on the file before allowing this change. Only *read* permissions, if initially did exist are retained.

- *login* - For propagating default downgrade-able levels for a subject, we change the *login* utility. When a new user is logged in, we assign its real userid to be its downgraded userid. In addition, saved userid can be used for flexibility in downgrade-able levels. This value is then inherited by all the child processes of *login*.

3.5 Secure Installer

A package installer is used for systematic package installation. Utilities like APT (Advanced Packaging Tool) and YUM (Yellowdog Updater, Modified) maintain a database to help underlying installers like dpkg and rpm in resolving package dependencies, downloading packages, *etc.* The installer installs a downloaded package and executes installation scripts specific to the package. For our discussion, we will consider these utilities as part of the installer.

Packages that modify system directories by placing or removing their files, expect the installer to have superuser privileges. Moreover, the installer needs to update its database files on each package installation or uninstallation. These files are root-owned, making it essential for installer to always execute with superuser privileges.

However, there is a huge security hole here, since the actions of an installer can be influenced by the package it is installing [23]. This is possible when package specific installation scripts are executed. These scripts also execute with superuser privileges, like the installer. Therefore, the installer needs to be confined. But behavior of the installer varies according to the package it is installing, thus making policy specification relatively difficult.

3.5.1 Policies for a secure installer

The secure installer we developed is implemented for *apt*. We aim to secure the installation and uninstallation phases of the package by confining the installer using following higher level policies:

1. *Installer will always execute at the integrity level of the package.* This policy ensures that application scripts do not exploit the installer privileges and that files created and copied on the system will inherit application integrity level.

2. *Regulated update to higher integrity files using state-based policies.* Since the installer needs to modify certain high integrity files, we cannot blindly enforce policy 1. State-based policies consider the intent of an operation by allowing changes to a copy of the original

file and then checking if updated file and system are in a security consistent state. Such state-based policies are more powerful than policy enforcement using runtime monitoring [19], where decisions regarding permissibility of an operation are made strictly based on each operation independently.

We use a daemon process to help implement these state-based policies. This daemon is a root owned process and executes all legitimate superuser operations on behalf of the installer.

3.5.2 Daemon for privileged installer operations

A daemon process which will make superuser operations on behalf of installer is created. This daemon manages redirection for files. The shared library used by installer to make system calls is intercepted for this purpose. Handlers in the intercepted library on system call entry and exit communicate with the daemon process to achieve its functionality. Let us discuss in detail how each system call is handled:

- **Intercept on syscall exit**
  When a system call fails with $EPERM$, $EACCES$ or $ENOENT$ error, the handler in the intercepted library communicates with the daemon and passes information about system call that failed along with it parameters. The daemon determines the reason for failure and handles it in one of the following ways:

  - *Failed due to permission denial* - The daemon checks if system call would succeed with root privileges. This is done by checking if all other parameters to it are valid. If this is true, depending on the semantics, the system call is either done by the daemon on behalf of the installer or it is redirected.

Redirection is done by creating a copy of the original resource, that needs root privileges. Permissions on the copy are set such that installer has required access to it. The parameter of system call that indicated the resource path is replaced with the redirected path and sent back to the intercepted library. The library then makes the system call with new parameters.

We describe in detail how each system call is handled.

  - *access, utime,* and *stat* - If any of these system calls or its variants fails due to permission denial, the daemon does the system call with original parameters. The return value is passed to the library and in turn used by the installer. Since these system calls query information about resources and do not make any changes, it is alright to execute them.
  - *chmod, unlink, mkdir, rmdir, chdir,* and *chown* - These system calls and their variants change system resources. Hence, the file/directory on which it needs to be performed is redirected. Any future reference to this file/directory will use the redirected copy.

- *Failed because original file was redirected* - If the system call did not fail due to permission denial and its parameters were not valid, the system call might have failed due to our redirection. This case is handled by checking the resource name in the table of all redirections, maintained by the daemon. If an entry was found, the parameter holding resource name is replaced with the redirected resource name and sent to the library. Intercepted code now makes the system call again with new parameters.

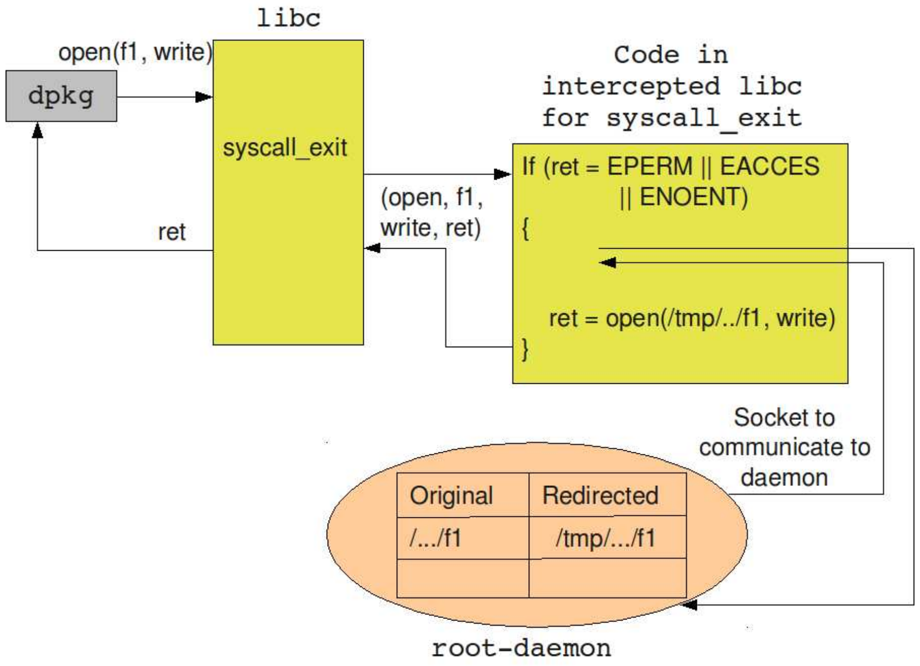
libc
open(f1, write)
dpkg
syscall_exit
ret
(open, f1, write, ret)
Code in intercepted libc for syscall_exit
If (ret = EPERM || EACCES || ENOENT)
{
ret = open(/tmp/../f1, write)
}
Socket to communicate to daemon
Original
Redirected
/.../f1
/tmp/.../f1
root-daemon

Figure 3: Interaction between installer and daemon process

If the failure is not due to any of the mentioned cases above, it is not handled and returned to the library.

In figure 3, we explain how *write* to a high integrity file is handled by the daemon.

- **Intercept on syscall entry**
  The system call interception is also used to propagate the integrity level of package to all its files. This is implemented in the entry hook of system call interception. We monitor the creation of new files on the system to override its (owner, group) information and file permissions. Following system calls are identified to create files on the system, *open*, *creat*, *rename* and *link*. The parameter to these system calls, containing filename is replaced with a redirected location. Redirection of all new files is done so that

if installation fails, all traces of the package can be completely removed.

3.5.3 Running installer with package userid

With a mechanism to handle privileged operations, we can now execute the installer with a userid corresponding to the integrity level of a package. The level of a package is determined by it's source repository, which in turn depends on the repository's source. Since there are limited trusted sources for a repository, the listing can be manually updated. For our prototype, we trust only default Ubuntu repositories.

There can be any number of repositories in a placed a group at each trust level. Repositories in the same group indicate same trust level and hence packages downloaded from them will be assigned to the same integrity level in our lattice.

3.5.3.1 Determining integrity level of a package

The *apt* utility takes package name to be installed as a parameter. It resolves package dependencies and adds all uninstalled packages, required by this package, to the installation list. After the dependency information is complete, *apt* downloads all packages. The order of searching repositories for a package is as follows:

1. It first checks for repositories with higher priority. This information can be provided by pinning repositories [2]. It is present in $/etc/apt/preferences$.

2. Among repositories with equal priorities, they are checked based on the order specified in */etc/apt/sources.list*.

We reorder repository listing in */etc/apt/sources.list* and sort them in order of their trust. This helps ensuring that if a package is provided by more than one repository, it is picked up from higher trusted source.

We need a mechanism to associate a Debian file with the repository it was downloaded from. This can be done by querying the repository information. Each repository contains a $< repository\_name >$ *_Packages* file, which contains information about all the packages it provides. Moreover, this file contains the name of Debian file that would be downloaded, along with its checksum information. We check repositories, in the same order as checked by *apt*, to determine source of given package. The package is then assigned an integrity level corresponding to the repository trust, and assigned a userid.

3.5.3.2 Installation phase

The secure installer has a wrapper for *dpkg*, where the integrity of each Debian file is calculated using the above method. If the files given to *dpkg* are of mixed integrity levels, the execution of *dpkg* is serialized. The dpkg-wrapper does the following for each package to be installed.

1. Change the userid and execute *dpkg* with default options, and single Debian file.

2. *dpkg* is made to use intercepted library for making system calls.

3. Daemon running with root privileges is used for making privileged operations and redirection for *dpkg*.

4. After *dpkg –unpack* and *dpkg –configure*, we validate changes to redirected installer database files. If the validation scripts pass, original database files are replaced with these redirection files.

5. All newly created files are copied on the expected location on the system.

6. However, if the validation of a high integrity file fails, the recovery mechanism, as described in section 3.5.5 is used to abort installation and get the system in a consistent state.

7. After completion of the installation, the next Debian file is taken and executed as elaborated above.

Installation of packages with userid 0 will bypass our mechanisms of library interception. This is because the intercepted library is used when system calls fail due to permission denial, which will not be the case when executing with userid 0. This is not a problem because we assume only high integrity packages to have userid 0.

3.5.3.3 Uninstallation phase

Similar to the installation phase, the uninstallation of a package also needs to execute at its own integrity level. The same dpkg-wrapper is used for this purpose. However, at this stage, *dpkg* take only package name as parameter. We need to determine the integrity of package being uninstalled from its name. This information is present in */var/lib/dpkg/status*, which is a database file for *dpkg*.

From the package name, we find files and libraries that belong to this package. The owner of these files on the system is used to determine package integrity[1]. The execution to *dpkg* is again serialized based on integrity of the package being uninstalled. Steps and mechanisms similar to installation phase are applied.

[1]Our execution time mechanisms ensure that userid of files is not changed to any arbitrary value. In addition to this, we have wrapper to *chown* to monitor changes.

3.5.4 Validating changes to high integrity files

We use state-based policies for regulating access to high integrity files. This is achieved by redirecting changes to these files and later validating if the changes left the file and system in a consistent state.

3.5.4.1 Installer database files

For validating changes to installer database files, the high level policy is to make sure that a package updates information belonging only to itself. We wrote scripts for validating changes to following high integrity files. Changes to sections belonging to other packages was considered as a violation. Each file needs a separate validation script, because the script will depend on the file format.

- */var/lib/dpkg/status*
- */var/lib/dpkg/diversions*
- */var/lib/dpkg/available*
- */var/lib/dpkg/statoverride*
- */var/lib/apt/extended_states*
- */usr/share/applications/mimeinfo.cache*

On an average, the script contains 50 lines. *perl* is used as a scripting language.

3.5.4.2 Other files

There are other files that are redirected by the daemon during installation. These are files that need to be copied in root owned directories, that modify existing files or are created during installation. The high level policy is to allow following actions,

1. To overwrite files with same or lower integrity userid.
2. To create new files in system directories.

Any file modification that does not fall in one of the categories stated above is rejected.

3.5.5 Recovery Mechanism

If at any stage of installation or uninstallation, a policy does not allow the installation to proceed, or validation to a high integrity file fails, a recovery mechanism is implemented to ensure that the system and the installer are always left in a consistent state.

The design of our secure installer is such that it does not interfere with the behavior of installer in an unpredictable way. Policies are enforced at well defined interfaces. This makes it possible to use existing mechanisms of *apt* of reverting installation. The existing mechanism of *apt* is to invoke a particular uninstallation script based on the stage at which installation fails. These scripts are provided by the package.

For a trusted package, these reverting scripts can be trusted, but for untrusted packages, early failure is preferred instead of leaving the system in an inconsistent state. Redirecting modifications to all newly created files on the system and high integrity files is one of the mechanisms to preserve system consistency.

CHAPTER 4

Evaluation

4.1 Evaluation of Secure Installer

We implemented the Secure Installer for APT and dpkg on Ubuntu 8.10 Linux. The code for policy enforcement using syscall interceptor was around 4.5K lines of C code. The validation scripts for installer database files were written in *perl*.

We tested the installer policies by installing three untrusted packages. These packages updated the installer database files, copied their files in system directories and executed their installation scripts. The performance numbers of the installation with and without our library interception is as follows:

Package Installed	Original Installer	Secure Installer	Overhead
webcam	6.568	8.296	26.29%
totem	6.552	8.704	32.82%
fruit	6.632	7.932	19.60%

Table 1: Performance overhead of Secure Installer. All numbers are in seconds.

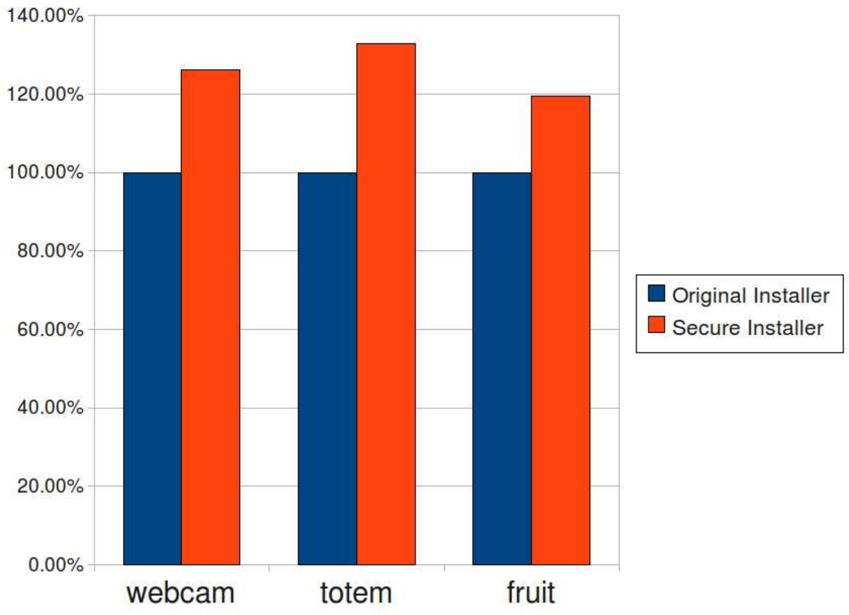

Figure 4: Evaluation of Secure Installer

The overhead seen is due to redirection of files. System call interception does not incur accountable overhead. Therefore, installation of packages from trusted repositories will not show this overhead. Since, redirection is done only on system call failure, and packages from trusted repositories will be installed with high integrity userid.

4.2 Evaluation of Information flow model

For evaluating the information flow model in the execution phase, we assume that the existing system is non-malicious, so all files on the system belong to the same integrity level. All files are assigned to a single userid. New packages entering the system will be assigned new userid based on its trust level.

Testing Policy enforcement

- *Untrusted subject trying to modify higher integrity files*
  We wrote sample test cases that check for "no write up" policy. However, it needs to be tested with real malware. We need to install malicious code that tries to modify system files.

- *Untrusted subject executing file at higher integrity level*
  The world executable permissions of all setuid programs on a system is changed to group permissions. No untrusted userid was added to this group, hence this policy was successfully enforced.

- *Non-malicious applications given malicious input*
  Packages like *acroread*[1] and *amarok*[2] were executed with untrusted input. The resultant instance of these applications ran with untrusted userid. However, we need to find malicious input that can successfully craft an attack.

- *Malicious subject executed by non-malicious applications*
  Sample test cases for checking this policy are created. We ran a subject at high integrity and executed a low integrity executable from it. The resultant execution was downgraded and "regulated exec down" policy was enforced. However, to check this on a real application, the application should execute a malicious file. This can be done by replacing a system utility by a low-integrity executable or changing configuration file of the non-malicious application to pick up malicious executables.

[1]Acrobat reader
[2]Music player - Amarok

- *Lower integrity input given to invulnerable applications*
  Applications like *ssh*, *cp* and *tar* are categorized as invulnerable. A set of files with mixed integrity levels was given as input to each of these. The output was generated and applications ran successfully. In the absence of finer granularity in trust, these applications would have failed due to downgrading.

We have not done a complete system-wide testing of the information flow model. There are few implementation issues that need to be resolved for it to be applied to the entire system.

CHAPTER 5

Related Work

The Biba model [4] introduces the concept of information flow by policies like no read down and no write up. However, these policies are very strict and break most of the applications. The low-water mark model [6] addresses this problem by relaxing these policies by allowing subjects to execute at the integrity level of input they are exposed to. LOMAC [6], a prototype implementation of low-water mark model on Linux, addresses the "self-revocation" problem for IPC but not files. SLIM (Simple Linux Integrity Model) [18] is developed as a part of the IBM research trusted client project, and is also based on the LOMAC model.

UMIP [17] and PPI [22] are information flow techniques for ensuring system integrity. The file labeling mechanism used by these approaches make their implementation very specific to a given version of the kernel. Also due to significant changes in the kernel code, porting these systems becomes very difficult. As opposed to this, we use existing mechanisms to label files and hence our approach is generic. Also we implement policy enforcement at user level, hence avoiding any changes to the existing kernel.

IX [15] uses dynamic labels on processes and files to ensure privacy and integrity via information flow. However, we do not address the problem of preserving confidentiality of data. But we improve usability by allowing applications to downgrade based on the context of low-integrity input.

Windows Vista enforces only the “no write up” policy of information flow. The “no read down” is not enforced as it breaks usability. However, this approach is not complete and attacks can be crafted based on this policy. On the other hand, Back to the Future system [5] enforces only the “no read down” policy. It thus identifies any attempt of malicious input being consumed by benign applications. However, the rollback mechanism is not straight forward and needs user intervention. Also the mechanism for recovering critical files, after they are overwritten by malware, is time-consuming.

While information flow techniques like SLIM [18], LOMAC [6] and PPI [22] protect host integrity during the execution phase of an application, they cannot be applied to the installation and uninstallation phases. SSI [23] tries to fill this missing gap by enforcing information flow policies on the installation phase and passing file integrity information to the execution phase of applications. We refer the higher level policies discussed in SSI as the basis of our secure installer design. However, due to different file labeling mechanisms, the implementation and issues of secure installer varies from SSI.

SoftwarePot [11] proposes a secure software circulation and deployment model. The notion of code provider is carried to the end user by encapsulating the execution of the application within a file system corresponding the code provider. This is similar to our concept of associating package provider identity during package execution. However, unlike our secure installer, it cannot be applied to existing installers or by using existing sandboxing techniques. More importantly, policy development of SoftwarePot is not automated and requires effort to support new applications. Secure Installer on the other hand uses state-based policies for preserving system integrity. A set of standard policies is applied to all untrusted applications, thus automating the policy development mechanism.

The concept of userid is used by Google to confine applications in Android [20, 13]. It ensures that each application is executed in its own userid environment. However, the concept of userid is used to create a sandbox for the application. With confinement based on userid, Android aims to ensure data confidentiality. The problem of preserving data integrity is completely avoided by isolating the executions within a given environment and not allowing to copy files outside its directory structure. Any request files and resources belonging to another application is made via an application manager, which checks if the request is legitimate before making it. This mechanism however, is possible on a mobile platform. But on a desktop we need to allow information flow and hence the problem to be solved is different.

iPhone security model [16], is also based on executing applications in a sandbox. However, Apple uses TrustedBSD MAC hooks similar to LSM to enforce sandboxing policies.

CHAPTER 6

Conclusion

We developed an approach to preserve system integrity while allowing safe execution of untrusted code. Our approach achieves following objectives of usability while preserving integrity.

- Untrusted applications need to be confined, i.e., they should not be able to overwrite data or code belonging to (or used by) benign applications that need to run with high integrity in order to carry out their functionality. This is achieved by using userid to encode file integrity information and enforcing "no write-up" and "no exec-up" policies using DAC mechanism.

- Most of the applications need the ability to be execute with lower privileges at some point during runtime, usually when they read untrusted input. Applications that can handle untrusted data may need downgrading as well, e.g., when they are exposed to untrusted input on channels where they expect high integrity input. We achieve this objective by enforcing a subject to downgrade its integrity level when it is exposed to lower integrity data.

- Some applications can be trusted to handle untrusted input, but only in certain context. For instance, even the best written application cannot be expected to protect itself from a malicious library or configuration file. We achieve this by assigning granular trust to invulnerable applications.

- Policy development should be largely automated. This is not an issue with our system, since our policies depend on file ownership. We take care that the integrity level of files entering the system is propagated from its source.

- User should not be prompted for input on security decisions. However, a sophisticated user should be able to override default policies in a carefully controlled way and applications should be able to go beyond default mechanisms if they are aware of the underlying mechanisms. This is achieved by the various methods provided for specifying and inferring user intention during file access.

- The mechanism to determine integrity of an application should be automated. This mechanism needs to translate the trust of an application from its source to its integrity on the system. A user cannot be expected to provide this information. We achieve this end-to-end trust handling by associating trust levels to repositories used for downloading applications. Also the repository trust level is translated to an integrity level in the system lattice.

- An application should not be able to violate the system security policies at any phase of its life cycle. This is achieved by implementing a secure installer, which ensures that installation and uninstallation happen at application integrity level.

Bibliography

[1] Hack of google, adobe conducted through zero-day ie flaw.

[2] Manual page for apt preferences. Technical report, 2003.

[3] A. Acharya and M. Raje. Mapbox: Using parameterized behavior classes to confine applications. In Proceedings of the 9th USENIX Security Symposium, 1999.

[4] K. J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, Mitre Corporation, ploits on commodity software. In Proceedings of 12th Annual Network and Distributed System Security Symposium, 2004.

[5] T. R. Francis Hsu and H. Chen. Back to the future: A framework for automatic malware removal and system repair. In Annual Computer Security Applications Conference (ACSAC), 2006.

[6] T. Fraser. Lomac: Low water-mark integrity protection for cots environments. 2001.

[7] S. D. G. Edward Suh, Jaewook Lee. Secure program execution via dynamic information flow tracking. ACM SIGARCH, 2004.

[8] F. M. T. B. Guido van t Noordende, Rutger Hofman and A. S. Tanenbaum. A secure jailing system for confining untrusted applications.

[9] D. W. Hao Chen and D. Dean. Setuid demystified. In *11th USENIX Security Symposium*, 2002.

[10] R. T. I. Goldberg, D. Wagner and E. A. Brewer. A secure environment for untrusted helper applications: confining the wily hacker. In *Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6*, 1996.

[11] K. Kato and Y. Oyama. Softwarepot: An encapsulated transferable file system for secure software circulation. In ISSS, pages 112132, 2002.

[12] B. L. Kevin Borders, Eric Vander Weele and A. Prakash. Protecting confidential data on personal computers with storage capsules. In *Proceedings of the 18th USENIX Security Symposium*, 2009.

[13] W. E. Machigar Ongtang, Stephen McLaughlin and P. McDaniel. Semantically rich application-centric security in android. Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC), Honololo, HI, 2009.

[14] S. E. Madnlck and J. d. Donovan. Application and analysis of the virtual machine approach to information system security and isolation. In *Proceedings of the 9th conference on USENIX Security Symposium - Volume 9*, 2000.

[15] M. D. McIlroy and J. A. Reeds. Multilevel security in the unix tradition. 1992.

[16] T. Moyer and D. Octeau. iphone security model. Technical report, 2009.

[17] Z. M. Ninghui Li and H. Chen. Usable mandatory integrity protection for operating systems. In IEEE Symposium on Security and Privacy, 2007.

[18] D. Safford and M. Zohar. Trusted linux client. ACSAC, Tucson, AZ, 2004.

[19] F. B. Schneider. Enforceable security policies. In *ACM Transactions on Information and System Security, 3(1):3050*, 2000.

[20] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer. Google android: A comprehensive security assessment. IEEE Security and Privacy, Vol. 8, No. 2. (2010), pp. 35-44., 2010.

[21] D. A. Wagner. Janus: an approach for confinement of untrusted applications. In *Technical Report: CSD-99-1056*, 1999.

[22] G. P. Weiqing Sun, R. Sekar and T. Karandikar. Practical proactive integrity preservation: A basis for malware defense. in IEEE Symposium on Security and Privacy, Oakland, CA, 2008.

[23] Z. L. Weiqing Sun, R. Sekar and V. Venkatakrishnan. Expanding malware defense by securing software installations. In *Detection of Intrusions, Malware and Vulnerability Analysis (DIMVA), Paris, France*, 2008.

[24] Y. Wen and H. Wang. A secure virtual execution environment for untrusted code. In *Information Security and Cryptology - ICISC 2007. Volume 4817/2007*, 2007.

[25] V. V. Zhenkai Liang and R. Sekar. Isolated program execution: An application transparent approach for executing untrusted programs. 19th Annual Computer Security Applications Conference (ACSAC), Las Vegas, NV, 2003.

[26] V. V. Zhenkai Liang, Weiqing Sun and R. Sekar. Alcatraz: An isolated environment for experimenting with untrusted software. ACM Transactions on Information and System Security (TISSEC), Volume 12, Issue 3, 2009.