Trojan Horse Resistant Discretionary Access Control

by Ziqing Mao, Ninghui Li, Hong Chen, Xuxian Jiang
Center for Education and Research
Information Assurance and Security
Purdue University, West Lafayette, IN 47907-2086

Trojan Horse Resistant Discretionary Access Control

Ziqing Mao, Ninghui Li, Hong Chen
Purdue University
{zmao, ninghui, chen131}@cs.purdue.edu

Xuxian Jiang
George Mason University
xjiang@gmu.edu

Abstract

Modern operating systems primarily use Discretionary Access Control (DAC) to protect files and other operating system resources. DAC mechanisms are more user-friendly than Mandatory Access Control (MAC) systems, but are vulnerable to trojan horse attacks and attacks exploiting buggy software. We show that it is possible to have the best of both worlds: DAC's easy-to-use discretionary policy specification and MAC's defense against trojan horses and buggy programs. This is made possible by a key new insight that DAC has this weakness not because it uses the discretionary principle, but because existing DAC enforcement mechanisms assume that a single principal is responsible for any request, whereas in reality a request may be influenced by multiple principals; thus these mechanisms cannot correctly identify the true origin(s) of a request and fall prey to trojan horses. We propose to solve this problem by combining DAC's policy specification with new enforcement techniques that use ideas from MAC's information flow tracking. Our model, called Information Flow Enhanced Discretionary Access Control (IFEDAC), is the first DAC model that can defend against trojan horses and attacks exploiting buggy software. IFEDAC significantly strengthens end host security, while preserving to a large degree DAC's ease of use. In this paper, we present the IFEDAC model, analyze its security properties, and discuss our design and implementation for Linux.

1 Introduction

Modern commercial off the shelf operating systems use Discretionary Access Control (DAC) to protect files and other operating system resources. According to the Trusted Computer System Evaluation Criteria (TCSEC) (often referred to as the Orange Book) [10], Discretionary Access Control is "a means of restricting access to objects based on the identity of subjects and/or groups to which they belong. The controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject (unless restrained by mandatory access control)." It has been known since early 1970's that DAC is vulnerable to trojan horses. A Trojan horse, or simply trojan, is a piece of malicious software which in addition to performing some apparently benign and useful actions, also performs hidden, malicious actions. Such trojans may be email attachments, programs downloaded from the Internet, or obtained from removable media such as USB thumb drives. By planting a trojan, an attacker can get access to resources the attacker is not authorized under the DAC policy, and is often able to abuse such privileges to take over the host or to obtain private information. DAC is also vulnerable when one runs buggy programs that receive malicious inputs. For example, a network-facing server daemon may receive packets with corrupted data, a web browser might visit malicious web pages, and a media player can read malformed data stored on a shared drive. An attacker can form the input in a way to exploit the bugs in these programs and take over the processes running them, e.g., by injecting malicious code. In essence, a buggy program that takes malicious input can become a trojan horse.

In other words, for existing DAC mechanisms to be effective in achieving the specified protection policies, one has to assume that *all* programs are benign (be functional as intended) and correct (won't be exploited by malicious inputs). This assumption does not hold in today's computing environments. This weakness of DAC is a key reason that today's computer hosts are easily compromised. Host compromise, in turn, leads to a number of other serious computer security problems today. Computer worms propagate by first compromising vulnerable hosts and then propagating to other hosts. Compromised hosts may be organized under a common command and control infrastructure, forming botnets. Botnets can then be used for carrying out attacks such as phishing, spamming, and distributed denial of service. To effectively address the problem, one needs to address the root cause of these threats, namely, the vulnerability of end hosts.

There are a lot of research efforts on making computer hosts more secure by adding mandatory access control (MAC) to operating systems. Perhaps the most widely-known among these is Security Enhanced Linux (SELinux) [28], which came out of the National Security Agency. Systems such as SELinux take the approach of specifying an additional layer of access control, with new policies and mechanisms. Such systems are flexible and powerful. Through proper configuration, they could result in highly-secure systems. However, they are also complex and intimidating to configure. The SELinux policy interface is daunting even for security experts. While SELinux makes sense in a setting where the systems run similar applications, and sophisticated security expertise is available, its applicability to a more general setting is unclear.

Even though DAC's weaknesses is widely known since early 1970's, DAC is today's dominant access control approach in operating systems. We believe that this is because DAC has some fundamental advantages when compared with MAC. DAC is easy and intuitive (compared with MAC) for users to configure, many computer users are familiar with it, and the discretionary feature enables desirable sharing. In this paper we show that it is possible to have the best of both worlds: DAC's easy-to-use discretionary policy specification and MAC's defense against trojan horses and buggy programs. This may sound impossible based on conventional wisdom. In fact, it has been asserted that "This basic principle of discretionary access control contains a fundamental flaw that makes it vulnerable to Trojan horses." [26]. We now show why this assertion is inaccurate.

We dissect a DAC system into two components: the discretionary policy component and the enforcement component. Take the access control system in UNIX-based systems as an example. The policy component consists of the following features: each file has an owner and a number of permission bits controlling which users can read/write/execute the file. The owner of a file can update these permission bits, which is the discretionary feature of DAC. The policy component specifies only which users are authorized, whereas the actual request are generated by processes (subjects) and not users. The enforcement component fills in this gap. In enforcement, each process has an associated user id (the effective user id) that is used to determine this process's privileges, and there are a number of rules that determine how the effective user id is set. We point out that these rules are mandatory in the sense that they are specified by the system and is not controlled by users. Such rules include the behavior of various setuid-related system calls.[1] In short, the policy part specifies which users can access what resources and the enforcement part tries to determine on which users' behalf a process is executing.

The key new insight that enables us to have the best of both worlds is *DAC's advantage over MAC lies in the policy component, whereas DAC's weakness comes from the enforcement component*. Thus we can keep DAC's policy component while revamping the enforcement component. We now explain this insight in more detail. DAC's discretionary policy protects both system core resources and user-owned files. System core resources are authorized only to system administrators. In UNIX, those resources are owned by root and readable or writable only by root. For user-owned files, the discretionary policy asks owners to decide which other users can access the file. We claim that the DAC policy used in today's DAC systems is sufficient to preserve the host integrity provided that they can be properly enforced to be resilient to trojan horse and buggy software.[2]

They key weakness in existing DAC enforcement mechanisms is that they assume that a *single* principal is responsible for any request, whereas in reality a request may be influenced by multiple principals; thus these

[1] The actual DAC enforcement in UNIX-based systems is much more complicated than described here. See [6] for an excellent discussion of the complexities.

[2] It is certainly true that in any UNIX-based distribution, there may exist errors in the DAC policy specification. However, such policy mis-configuration errors can occur in any access control system. Given that DAC is simpler and more intuitive than other MAC systems and that the DAC information for key resources have already been used by millions of users and examined for decades, we believe that a DAC policy configuration is likely to have fewer and less serious errors than a fresh MAC policy configuration.

DAC mechanisms cannot correctly identify the true origin(s) of a request and fall prey to trojan horses. This is the *fundamental limitation* of existing DAC enforcement mechanisms. When one user's process executes a program controlled by another user, both the invoker and the controllers of the program content affect any requests made by the new process. That is, the master of this new process is a set containing both the invoker and the controllers. The new process is guaranteed to act on behalf of the invoker only when the program is *benign*. After reading data, the process continues acting on behalf of the old master only when the program is *correct*; otherwise the (potentially maliciously formed) data could be used to exploit the program. If the program is not assumed to be correct, the controllers of the data must be added to the set of masters.

Utilizing this insight to solve DAC's weakness, we keep the discretionary policy component, but change the enforcement component. In the enforcement component, one should maintain a set of principals, rather than a single one, for each process. When a request occurs, it is authorized only when every principal in the set is authorized according to the DAC policy, since any of those principals may be responsible for the request. We develop this idea into the Information Flow Enhanced Discretionary Access Control (IFEDAC) model that enhances DAC to defend against trojan horses and buggy software. IFEDAC uses the notion of a contamination source to trace the identity of requesters. For each subject and object, we use an integrity level to track potential contamination sources that may have affected the subject or object. The system grants access to an object if the contamination set of the subject is a subset of the set of users allowed to access the object by DAC. We believe that IFEDAC can still be called DAC, even though the contamination tracking techniques are from mandatory information flow control. The fundamental difference of IFEDAC from MAC is that protection labels are not centrally specified, but rather discretionally specified. In IFEDAC, who can access an object is based on whether the requesters' identities satisfy the DAC policy, rather than the integrity level of the object. IFEDAC follows the discretionary control principle, and allows owners to decide which other users can access the file. Second, while IFEDAC uses mandatory rules to track the set of contamination resources and uses this in making access control decisions, all DAC mechanisms must use some kind of mandatory rules to track who is the requester. In short, IFEDAC is DAC with enhanced enforcement techniques borrowed from MAC.

IFEDAC is the first DAC model that can defend against trojan horses and attacks exploiting buggy software. This is achieved by precisely isolating and fixing what makes DAC vulnerable to trojan horses and buggy programs. IFEDAC can significantly strengthen end host security, while preserving to a large extent DAC's ease of use. We have implemented IFEDAC for Linux as a kernel module, using the Linux Security Modules (LSM) framework [32]. While the description of the IFEDAC model in this paper is based on our design for Linux, we believe that the model can be applied to other UNIX variants with minor changes, and the general approach would be applicable also to non-Unix operating systems such as the Microsoft Windows family.

The rest of this paper is organized as follows. We first give an overview of IFEDAC in Section 2, then present a formal model of IFEDAC in Section 3, and analyze the security properties in Section 4. We discuss our implementation of IFEDAC for Linux and its evaluation in Section 5 and Section 6. We discuss related work in Section 7 and conclude in Section 8.

2 An Overview of the IFEDAC Model

One key concept in IFEDAC is the contamination source. Each contamination source represents a channel potentially controlled by a different entity who may compromise the system integrity. Each DAC user account that has a login shell and is not root is viewed as a separate contamination source. Remote network communication is another contamination source (denoted as net), which represents the remote attacker who do not have a local account. In the following description, we use subject and process interchangeably, and object and file interchangeably.

IFEDAC maintains an integrity level for each subject and object. The value of an integrity level is a set of contamination sources that indicate who may have gained control over the subject or who may have changed the content stored in the object. The integrity level is tracked using information flow technique, which is roughly described as follows. The integrity level of the first process started in the system is initialized as an empty set, which means the process has not been contaminated by any entity. A newly created process inherits the parent's integrity level. When a process receives remote network communication, the contamination source net is added into the process's integrity level, which indicates the remote attacker may have exploited and controlled the process. When a process reads data from a file, all contamination sources in the file's integrity level will be added to the process's integrity level, because after the reading, all entities that have changed the file's content may gain control over the process by feeding malicious input data or exploiting vulnerabilities. Also, when a process writes to a file, all contamination sources in the process's integrity level will be added to the file's integrity level. Similarly to DAC, when a process logs in a user who is not a system administrator, the contamination source corresponding to that user account is added to the process's integrity level. A formal definition of the integrity level tracking is given in Table 1 and a more detailed discussion is given in Section 3.

In IFEDAC, the access control policy is specified by associating each object with a read protection class (rpc) and a write protection class (wpc); either value is a set of contamination sources that indicates which entities are authorized to read from and write to the object. When a subject requests access to an object, the access is allowed if all of the contamination sources in the subject's integrity level are allowed to access the object, i.e., the subject's integrity level is a subset of the object's protection class.

Generally IFEDAC enforces the same policy specified in the DAC system, in which cases the rpc and wpc contain the entities who are authorized to read from and write to the object determined by the DAC policy. In a few situations, the IFEDAC policy is different from the DAC policy (e.g., there is no entity corresponding to net in the DAC system [3]), in which case the rpc and wpc can be explicitly set. As that in the DAC policy, only root or the owner can set an object's rpc and wpc, and only root can set an object's owner.

Most real-world attacks are prevented by the default policy model of IFEDAC we have introduced so far. For example, if a remote attacker breaks in by exploiting a vulnerability in a network server, the server process controlled by the attacker will have net in the integrity level, and hence cannot access the system core files and the files that are authorized only to local users. Similarly, if a careless administrator executes a trojan horse downloaded from a malicious site or open an email attachment that is a mal-formed file exploiting a vulnerable application, these files will have net in their integrity levels, as will the processes running and reading these files. Last, if a malicious local user u exploits a vulnerability in a setuid-root program to gain root privilege, the exploited process will have u in its integrity level, and hence it cannot access the system core files and other files that are not authorized to u.

While the default policy model of IFEDAC provides strong security guarantees to protect both core system files and user-owned files against trojan horses and vulnerability exploiting attacks, the model will disallow some legitimate operations, i.e., some processes need to access files that they are not authorized to access according to the policy. The same problem also exists in the DAC system and is handled by the setuid feature, which poses many security risks when the programs are buggy. IFEDAC handle this problem by specifying exceptions for programs. Exceptions imply trusts over programs and such trusts are strictly limited and can be clearly specified. We give a definition of exceptions in Section 3, analyze the underlying security assumptions for exceptions in Section 4, and discuss the exception policy configuration in practice Section 6.

It is known that DAC protects both core system components and user-owned files only when all programs are benign and correct. IFEDAC can enforce the same policy as that in DAC without relying on this unrealistic assumption by tracking the potential contamination sources for subjects and objects using the information flow technique. We analyze the security properties IFEDAC can provide in Section 4. In short, IFEDAC weakens the assumption to be (1) the programs that are explicitly identified as benign are benign (2) the programs that have exceptions are correct in processing input data.

[3] In Section 6, we will discuss policy configuration in IFEDAC and when the IFEDAC policy is different from that in DAC.

3 The IFEDAC Model

We now give a formal definition of the IFEDAC model for Linux.

3.1 Elements in the IFEDAC Model

The IFEDAC model has the following elements:

- S denotes the set of all subjects (i.e., processes).
- O denotes the set of all objects (i.e., files).
- U denotes the set of users. The set $U \cup \{\mathsf{net}\}$ is the set of all contamination sources.

  The users are partitioned into two subsets: $U = A \cup N$, where A denotes system administrators and N denotes non-administrators. The users in A are trusted to perform system administration through certain limited channels, whereas the users in N are not. In IFEDAC, compromising the passwords of users in A can lead to remote system compromise, if remote system administration is allowed. Compromising the passwords of users in N will affect only the compromised user.
- $\mathcal{L} = 2^{U \cup \{\mathsf{net}\}}$ is the set of all security labels that are used for labeling integrity levels and protection classes. These labels form a lattice under a partial order $\geq$ such that $\ell_1 \geq \ell_2$ if and only if $\ell_1 \subseteq \ell_2$. The meet (i.e., greatest lower bound) of ℓ_1, ℓ_2 in the poset $\langle \mathcal{L}, \geq \rangle$ is therefore $\ell_1 \cup \ell_2$. The greatest element in $\langle \mathcal{L}, \geq \rangle$ is $\emptyset$, which we use $\top$ to denote; and the least element is $U \cup \{\mathsf{net}\}$, which we use $\bot$ to denote.
- A function $\mathit{int} : S \cup O \rightarrow \mathcal{L}$ assigns an integrity level to each subject and each object.

  Directories and a few special files, such as /dev/null, do not have integrity levels, because contamination through them is considered unlikely. The file /dev/null is a pseudo device that discards all data written to it and provides no data to any subject that reads from it. For directories, this is because actual reading and writing are performed by the kernel code. A malicious subject writes to a directory by creating files. We assume that a subject that reads the directory (but not the files) cannot be exploited from this reading. Alternatively, one can think of these special objects as always having integrity level $\top$.
- Each object o has three protection classes.
  - A function $\mathit{rpc} : O \rightarrow \mathcal{L}$ assigns a *read protection class* to each object.

    The value $\mathit{rpc}(o)$ can be explicitly set. If it is not explicitly set, then $\mathit{rpc}(o)$ is inferred from DAC as follows: If o is world-readable, then $\mathit{rpc}(o) = \bot$. Otherwise, $\mathit{rpc}(o) = U_r(o)$, where $U_r(o)$ is the set of users in U who are authorized to read o. If o is group-readable, then $U_r(o)$ may change when group membership changes. IFEDAC uses the group membership information at the time of access to determine $\mathit{rpc}(o)$.
  - A function $\mathit{wpc} : O \rightarrow \mathcal{L}$ assigns a *write protection class* to each object.

    Unless explicitly set, the value $\mathit{wpc}(o)$ is inferred from DAC similar to $\mathit{rpc}(o)$. We expect that for the vast majority of objects, $\mathit{rpc}(o)$ and $\mathit{wpc}(o)$ are inferred from DAC.
  - A function $\mathit{apc} : O \rightarrow \mathcal{L}$ assigns an *admin protection class* to each object.

    This function determines which subject can change the $\mathit{rpc}(o)$ and $\mathit{wpc}(o)$ either directly or indirectly, through changing its DAC permission bits. As the Linux DAC mechanism allows only root or the owner of an object to change the permission bits, in IFEDAC we choose $\mathit{apc}(o) = \{\mathsf{owner}(o)\}$.
- A function $\mathit{spc} : S \rightarrow \mathcal{L}$ assigns a *subject protection class* to each subject. The value $\mathit{spc}(s)$ determines which subjects can send signals or use ptrace to interrupt or control the subject s.

	condition or effect		exceptions
Subject Integrity Tracking[a]			
After creating the first subject s_0	$int(s_0) \leftarrow \top$	(m1)	no
After s creates s'	$int(s') \leftarrow int(s)$	(m2)	no
After s executes o	$int(s) \leftarrow int(s) \cup int(o)$	(m3)	no
After s reads from the network	$int(s) \leftarrow int(s) \cup \{\mathsf{net}\}$	(m4)	yes
After s reads from o	$int(s) \leftarrow int(s) \cup int(o)$	(m5)	yes
After s logs in a non-administrator u	$int(s) \leftarrow int(s) \cup \{u\}$	(m6)	no
After s_1 receives IPC data from s_2	$int(s_1) \leftarrow int(s_1) \cup int(s_2)$	(m7)	yes
Object Integrity Tracking			
When o is created by s	$int(o) \leftarrow int(s)$	(o1)	no
When $int(o)$ is not previously assigned	$int(o) \leftarrow wpc(o)$	(o2)	no
After o is written by s	$int(o) \leftarrow int(s) \cup int(o)$	(o3)	yes
Object Protection			
For s to read o	require $int(s) \geq rpc(o)$	(a1)	yes
For s to write o	require $int(s) \geq wpc(o)$	(a2)	yes
For s to change $rpc(o)$ or $wpc(o)$	require $int(s) \geq apc(o)$	(a3)	yes
For s to change $apc(o)$	require $int(s) \geq \top$	(a4)	no
For s to change $int(o)$ to ℓ'	require $int(s) \geq apc(o) \wedge int(s) \geq \ell'$	(a5)	no
IPC Protection			
For s_1 to interrupt s_2	require $int(s_1) \geq spc(s_2)$	(i1)	no
For s_1 to ptrace s_2	require $int(s_1) \geq spc(s_2) \wedge int(s_1) \geq int(s_2)$[b]	(i2)	no

[a] In our subject integrity tracking rules, the integrity levels of processes can only go down and can never go up. One may worry that the whole system converges to $\bot$ after a time. This is not the case. It is true that the integrity level of any individual process can never go up after it goes down. However, as some processes, e.g., the root process of the system (i.e., init), are high, there are always new processes coming up that are also high. One can use a tree as an analogy. Once a leaf is dead, it does not become alive again. However, because the root is alive, new leaves keep coming up.

[b] If $int(s_1) \neq \top$, s_2 cannot have any exception privileges. All these conditions should be satisfied during the whole tracing period. A violation will stop the tracing.

Table 1: The 17 access control and label maintenance rules in IFEDAC. The last column indicate whether the rule can have an exception.

3.2 Access control and label maintenance rules in IFEDAC

IFEDAC has 17 rules for access control and label maintenance. They are separated into four parts: subject integrity tracking, object integrity tracking, file system protection, and inter-process communications (IPC) protection. These rules are summarized in Table 1. Some of these rules can have exceptions. We describe them in Section 3.3. We point out that end users do not need to know these rules to use a Linux system with IFEDAC, just as they do not need to know the intricacies of setuid-related system calls to use current Linux.

Subject Integrity Tracking The subject's integrity level is determined as follows. The first process, init, has integrity level $\top$. When a new process is created, it inherits the parent process's integrity level. When a process receives network traffic, its integrity level is updated to include net as an additional contamination source. This represents that the attacker who controls the network may have gained control over the subject by exploiting vulnerabilities in the subject. When a process s executes or reads an object o, its integrity level is contaminated by o, that is, $int(s) \leftarrow int(s) \cup int(o)$. This represents whichever source that may have contaminated the object o now may have gained control over s.

When a process logs in a user u, the process is contaminated according to the type of the user. If u is an administrator, then the process's integrity level remains unchanged. Otherwise, if u is a non-administrator, the process's integrity level is updated to include u as an additional contamination source, which represents that the user u has gained control over the process. We use the fact that a login in Linux triggers an event wherein all three uids (real uid, effective uid, saved uid) of a process are changed to a new user. This event occurs whether the login is through a terminal and the X desktop (the "login" process), via an ssh or ftp server, or by the execution of the "su" command. We rely on the DAC mechanism to trigger the login event because only the applications know which user is logged in and the applications are only aware of the DAC system. Note that missing this event is only significant when the process is currently at $\top$, in which case the process behaves in a benign way. If the attacker controls a process, IFEDAC ensures that the process does not have $\top$. Trying to evade this event does not increase the attacker's power.

Object Integrity Tracking For subject integrity tracking to be effective, we also need object integrity tracking. When a new object is created by a process, the object's integrity level is initialized to be the process's integrity level. For an object that is created before IFEDAC is deployed, its integrity level is initialized as its write protection class, because the write protection class is derived from the DAC permissions which can indicate how the object is protected before deploying IFEDAC. When an object o is modified by a process s, the object's integrity level is contaminated by s, that is $\mathit{int}(o) \leftarrow \mathit{int}(o) \cup \mathit{int}(s)$.

File System Protection The access control rules for file system protection are as follows. For a subject s to read o, we require that $\mathit{int}(s) \geq \mathit{rpc}(o)$. Similarly, we require that $\mathit{int}(s) \geq \mathit{wpc}(o)$ for s to write to o. For s to change the rpc (o) and wpc (o), we require $\mathit{int}(s) \geq \mathit{apc}(o)$. Linux DAC allows only root to change the owner of a file; thus IFEDAC adopts the policy that for s to change $\mathit{apc}(o)$, we require $\mathit{int}(s) = \top$.

An object's integrity level can be updated explicitly. This is necessary, for example, to allow system updates. The integrity level of downloaded updates will include net, and needs to be upgraded to $\top$ before the updates can be installed. However, for s to update o's integrity level to ℓ, we require both $\mathit{int}(s) \geq \mathit{apc}(o)$ and $\mathit{int}(s) \geq \ell$. The former requires that s represents the owner of o, and the latter prevents malicious upgrading beyond one's own integrity level.

Inter-process Communications Modern Linux supports various mechanisms for inter-process communication (IPC). IFEDAC handles IPC by categorizing the IPC mechanisms into three types.

We call the first type *Data Sending*. The IPC mechanisms that belong to this type include pipes, FIFO, message queues, shared memory, local sockets and loopback network communication. They can be used to send free-formed data, and such data can be crafted to exploit bugs in the receiving process. Therefore after $s1$ receives IPC traffic from $s2$, $\mathit{int}(s1) \leftarrow \mathit{int}(s1) \cup \mathit{int}(s2)$. IFEDAC does not apply additional control to these IPCs, because they require active participation of both the sender and the receiver. Without the receiver's active participation, the sender cannot force the receiver to receive data.

We call the second type *Interrupting*. The IPC mechanisms that belong to this type include sending signals and changing scheduling parameters of another process (through the sys_set_priority() and sys_set_scheduler() system calls). For most signals, the default behavior of the receiving process is to terminate, core-dump, or stop, unless the process registers its own signal handlers to overwrite the default actions. We do not want the attacker to be able to terminate a critical system service or change the execution state of a process that belongs to another user. In other words, a user can only interrupt his own processes and only system administrators can do so to the system processes. IFEDAC achieves that by defining the subject protection class to indicate the "owner" of a process. The spc value is determined as follows. Initially when a new process is created, it inherits the parent process's protection class. When a process logs in a user u, its protection class is updated to $\{u\}$. Then, for s_1 to deliver an interrupting IPC to s_2, we require $\mathit{int}(s_1) \geq \mathit{spc}(s_2)$. If succeed, unlike the data sending IPCs, the receiver's integrity level does not change because it is difficult to use signaling to exploit a vulnerable process.

We call the third type *Controlling*. The only IPC mechanism that belongs to this type is ptrace. It enables the tracing process to observe and control the traced process and is used primarily for debugging. The tracing process can arbitrarily manipulate the memory and registers of the traced process, and even inject code into the traced process. As with interrupting IPCs, IFEDAC requires $int(s_1) \geq spc(s_2)$ for s_1 to ptrace s_2. In addition, because the tracing process can easily abuse the privileges of the traced process, IFEDAC requires s_2 does not have any privileges that are not available to s_1. That is, $int(s_1) \geq int(s_2)$, and s_2 does not have any exceptions if $int(s_1) \neq \top$. These conditions should be satisfied during the whole tracing period. Any violation will stop the tracing immediately.

3.3 Exceptions to the Rules

As the saying goes, rules are made to be broken. IFEDAC allows a number of exceptions to the rules. These exceptions are associated with program binaries, and imply that these programs are trusted in certain ways. Activation of these exceptions has minimal integrity restrictions. When a program binary that has exceptions is loaded (through the execve system call), if the current process's integrity level satisfies the minimal integrity restriction and the program binary has the integrity level $\top$, the exceptions are enabled. Once a new binary is loaded, the old exceptions are gone.

Exceptions to the subject integrity tracking Exception to the network contamination rule ((m4) in Table 1) is by the notion of a *remote administration point (RAP)*. A process running a RAP program maintains its integrity level when receiving network traffic. If one wants to allow remote system administration through, for example, the secure shell daemon, then one can identify the SSH daemon as a RAP. The trust assumption underlying a RAP declaration is that when the program is started in a benign environment it will process the network input correctly and the attacker cannot gain control of it by sending malformed network packets. We stress that whether to declare a program as RAP is a decision made by the local system administrator.

Similarly, exceptions to the file reading and IPC contamination rules ((m5) and (m7) in Table 1) are done under the notion of a *local service point (LSP)*. The process running a LSP program maintains its integrity level when reading from files or receiving IPC data from other processes. We do not distinguish between the exception for file reading contamination and that for IPC contamination because many programs also use shared files to achieve IPC (e.g., lock files). The trust assumption underlying the LSP declaration is that the program will process file and IPC input correctly.

The concepts of RAP and LSP are similar to the ring policy in the Biba model, in which a subject can read objects of an arbitrary integrity level without dropping its own integrity level. This is also similar to the notion of a transformation procedure in the Clark-Wilson model, which can read low integrity unconstrained data items and write to high integrity constrained data items.

Exceptions to object protection and integrity tracking For some programs, the integrity level at which it is normally running does not dominate the protection class of some objects it needs to access. For example, the ftp daemon will be running at the integrity level $\{\mathsf{net}\}$, but it needs to read from the /etc/shadow file to authenticate users. However, the shadow file has the read protection class $\top$, and thus the default policy will stop the access. We deal with this by allowing exceptions to object protection rules ((a1) and (a2) in Table 1). One can specify a set of file access exceptions for a program. Each exception enables a process running the program to read from or write to a file while violating the object protection rules. For the example of ftp server, one can specify the ftp daemon program to have a file access exception to read from the file /etc/shadow.

A file write exception contains an additional field to enable an exception to the object integrity tracking rule ((o3) in Table 1). When that field is set, after the program writes to the file, the file's integrity level remains unchanged. For example, the program /etc/passwd needs an exception to write to the file /etc/shadow when it is executed by a non-administrator u at the integrity level $\{u\}$. Moreover, the shadow file's integrity level should remain as $\top$ after being modified by passwd.

Note that since the old exception privileges are gone after a new binary is loaded, even if a vulnerable program is granted some exception privileges and the process running that program is exploited by the attacker, a shell (or other programs) spawned from the exploited process won't have any exception privileges.

4 Security Properties of IFEDAC

Recall that for DAC to be effective, all programs need to be assumed to be benign and correct. By introducing information flow techniques, IFEDAC aims at weakening this unrealistic assumption. We now analyze what the security properties IFEDAC can provide and what are the necessary assumptions to achieve them. The high-level security goal of IFEDAC is that confidentiality and integrity properties of a system are preserved under attacks.

4.1 Defining Integrity

Defining integrity in the context of operating systems is a difficult task. One can start by defining integrity as the property that key components do not change. This definition is too strong, as key files (e.g., /etc/shadow) and the kernel data structures need to change. As key components must change, one may modify the property to state that the resulting state after a change must satisfy certain constraints that can be precisely specified and checked. However, it is infeasible (and often impossible) to characterize these constraints. Next, one could refine the definition of integrity as the property that key components are changed only through certain programs. This property, though, is insufficient. Text editors must be allowed to modify key system script files. Yet, one cannot say these files have integrity solely because all updates are performed only through these editors. Finally, one can define integrity by declaring that key components are changed only by certain users. We believe this last choice most accurately reflects the intuition. If the change is intended by authorized users, then integrity is preserved; otherwise, it is violated.

We thus define integrity informally as

Integrity means all updates reflect authorized users' intentions.

To formalize this, we must identify two things: (1) who is authorized to perform an update, and (2) whose intention a subject (process) reflects. In IFEDAC, the former is specified by the write and administration protection classes. Any user in $wpc(o)$ (as well as root) is authorized to update o. For the latter, we observe that an integrity label has a natural interpretation as a representation of intentions. A label of $\top$ means the intention of the root user. A label of $\{u_1, u_2\}$ means the intention of u_1, u_2, or root. If we have $int(s) = \{u_1, u_2\}$ and $wpc(o) = \{u_1, u_3\}$, then s cannot update o, because the update may reflect the intention of u_2, who is not authorized to do so.

Therefore, a key property we need to show is that IFEDAC maintains the integrity levels for subjects correctly. That is, if a subject has integrity level ℓ according to IFEDAC, then the subject is *benign for integrity level* ℓ in the sense that any operation performed by the subject reflects the intention of only those users in ℓ.

To achieve this goal, we start by noting that integrity protection requires some degree of trust that programs do not introduce bad data. We can contrast this with confidentiality protection, for which if an untrusted subject never reads any secret information, it can not later write or leak secret information. For integrity, it is not enough to control what the subject reads, as it can create bad data without reading bad data. This observation suggests that integrity is not simply an information flow property. The strict integrity policy in the Biba model allows a subject at integrity level ℓ to read objects at ℓ or higher and write objects at level ℓ or lower. This implicitly requires that one trusts a subject at integrity level ℓ to be able to generate data at integrity level ℓ when reading data only at level ℓ or higher. Therefore, the code executed in the subject must be both functional and not malicious for integrity level ℓ. We say such a program is assumed to be *benign* for integrity level ℓ. Intuitively, the behavior of a benign program reflects the users' intention. For example, the basic utilities on a system such as editors and file manipulation tools are considered benign, not because they cannot be used to do bad things, but because they reflect the users' intentions.

We still need to translate the benign property of a static program file to the benign property of a running process. To do this, we assume the following axiom.

Axiom 1 *If a program is benign for an integrity level ℓ, then when it is executed by a subject that is benign at integrity level ℓ or higher, and the subject reads only input at integrity level ℓ or higher, the subject is benign for integrity level ℓ.*

We note that assuming that a program is benign is a weaker assumption than that the program is both benign and correct. A benign program is not trusted to handle malicious input. In short, a benign program mostly works as expected. But when it is exposed to malicious input, it may not do so anymore.

4.2 Integrity Protection Properties of IFEDAC

We now show that IFEDAC achieves the integrity goal that all updates reflect authorized users' intentions, under a number of assumptions. It suffices to show that IFEDAC maintains the following three invariants: (1) Every subject with integrity level ℓ is benign for that integrity level. (2) The content of every file with integrity level ℓ is only controlled by the users in ℓ. (3) For every file o, $wpc(o)$ correctly identifies the authorized users.

These are maintained by IFEDAC under the following assumptions. (1) When IFEDAC is enabled, the integrity levels of files are correct. For example, a program labeled with integrity level ℓ is benign for that level. (2) When IFEDAC is enabled, files are labeled with the correct write and administration protection classes. (3) The hardware has not been compromised. (4) The kernel and the programs that have exceptions are trusted either to process input correctly or not to fail in a way that the attacker can directly exploit the exceptions. (5) When a legitimate user *intends to* upgrade a file's integrity level, the decision is correct. When a legitimate user *intends to change* the write or admin protection class of an object, the decision is correct.

Assumptions (1) and (2) say that the initial labels are correct. IFEDAC cannot defend against physical attacks such as changing the BIOS settings to boot from the attacker's media; hence assumption (3). Assumption (5) means that the system must trust the legitimate user's intentions. Rather than assuming all programs are benign, assumptions (1) and (5) indicate that IFEDAC requires only the programs that are explicitly identified as benign (by setting the program's integrity level) to be benign.

Assumption (4) requires more examination. First, as IFEDAC works within the kernel, we must assume the kernel has no vulnerabilities the attacker can exploit. This assumption is also needed for similar protection systems, such as Security Enhanced Linux (SELinux) or AppArmor. IFEDAC extends this assumption so that a process running a program specified as RAP cannot be compromised by receiving network traffic, as the program is assumed to process network data correctly. Similarly, any program specified as LSP is assumed to process IPC inputs correctly. Read exceptions do not affect integrity, as it does not involve an update. If a program has a write exception with integrity preservation, it is assumed that (1) the program correctly handles bad input (similar to the previous discussion of RAP and LSP), or (2) if the program is exploited, the attacker is unable to inject malicious code directly into the address space to take advantage of the exception. In a typical exploit, the attacker injects the shell code into the vulnerable process, then runs malicious code in the spawned shell. Under IFEDAC, the spawned shell loses the write exceptions. The other possibility is for the attacker to inject all of the malicious code directly into the address space, but this task is more difficult than getting a shell, and is more easily defended against (e.g., with a non-executable stack). For write exceptions without integrity preservation, the assumption can be further weakened by observing that files written by the remote attacker will have an integrity level that includes $\{\mathsf{net}\}$. This level restricts the actions of any subject that then reads this file. For example, httpd has an exception to write to /var/log/httpd. While this file needs to be protected, it is not critical, and any direct or indirect attack that uses this exception cannot affect other parts of the system.

Almost all exceptions we have are also allowed in the SELinux Targeted policy. We point out that each of our exception specifications makes the underlying security assumption explicit. This is not the case in, for example, SELinux. Because our default access rules allow many accesses, the number of exceptions is small, and each exception reflects a new assumption about maintaining the integrity of the system. By examining them, one has a clear picture of the assumptions one is making for maintaining security. In SELinux, the default rule is no one has any access; therefore all legitimate accesses must be specified, making it very difficult to extract the assumptions one is making in the policy.

4.3 Confidentiality Protection in IFEDAC

As in integrity protection, DAC assumes all programs to be benign for confidentiality. For example, when one uses /bin/cat to view a file's contents, one implicitly trusts that it will not secretly send the file through an email, or create a world-readable copy of the file. Some programs will also write to, for example, files readable by others while reading files readable only by the user. Those programs are trusted to correctly declassify information. In IFEDAC, if we assume that a subject that is benign at integrity level ℓ can correctly declassify information at level ℓ, then confidentiality is also preserved by IFEDAC, under similar assumptions for integrity protection. Of course, we need to assume that the initial read protection classes of objects are set correctly. Also, when a program has a read exception, we assume that the program either (1) can handle malicious input, or (2) cannot be exploited in a way that the attacker injects malicious code into the address space and takes advantage of the exceptions.

4.4 The Attacker's View

We assume physical attacks are not possible, either because the attacker does not have physical access or there is another physical security mechanism. The attacker controls the incoming network traffic (the source net), and a set P of accounts for non-administrators (either the user are malicious or the attacker knows their passwords). That is, we assume the attacker does not know the passwords for administrator accounts in A. (If a system has no remote administration points, i.e., administration can be performed only through terminals, and an attacker has only network access, then they can be defended against even if they know administrator passwords.) Under such an attacker model, IFEDAC ensures that the attacker cannot control a subject with integrity level l, where $l \subseteq U \setminus P$, because the subjects with integrity level $l \subseteq U \setminus P$ reflect only the intention of the users in $U \setminus P$. In other words, an attacker-controlled subject's integrity level must contain at lease one source in $P' = P \cup \{\mathsf{net}\}$. As a result, the attacker cannot read objects for which the principals in P' are not authorized to read, or write to objects for which the principals in P' are not authorized to write. Therefore, the objects authorized only to administrators and un-malicious non-administrators are protected.

5 Implementation

We have implemented the IFEDAC model for Linux using Linux Security Module (LSM) framework. The implementation does not require any changes to the kernel source. We use extended attributes to store the additional fields IFEDAC introduced for each file; they are the integrity level, optional read protection class and optional write protection class. The module also maintains a label for each process, which includes the integrity level and the exceptions (if any) for the process. The module implements a number of hook functions to handle events that will trigger the IFEDAC rules and perform access control and label maintenance. The rest of this section discusses some implementation issues in more detail.

5.1 Capabilities Protection

The protection provided by the IFEDAC model described in Section 3 focuses on the file system. In Linux, there are also non-file critical resources that require protection – capabilities. Modern Linux uses capabilities to break the privileges normally reserved for root down to smaller pieces. As of Linux Kernel 2.6.22, there are 31 different capabilities; they control critical operations such as loading a kernel module, administering an IP firewall, and mounting a file system. IFEDAC protects the capabilities by categorizing them into three types (see Table 6 in Appendix). (1) Allowed. IFEDAC does not restrict processes using the allowed capabilities. The capabilities controlling file accesses are allowed because IFEDAC relies on its own access control for files, which is sufficient to protect the file system integrity. (2) Restricted. In the default policy, the restricted capabilities can only be used by processes running at the integrity level $\top$. Other processes can gain restricted capabilities by exceptions. (3) Reserved. The reserved capabilities (e.g., CAP_SYS_MODULE, CAP_RAW_IO) can only be used by processes running at the integrity level $\top$. Other processes cannot gain reserved capabilities by exceptions. The reserved capabilities correspond to extremely critical operations, which should be performed only by system administrators. Many attacks rely on those reserved capabilities. For example, kernel rootkits are installed either by loading a kernel module (controlled by CAP_SYS_MODULE) or by modifying the kernel memory image (controlled by CAP_RAW_IO). By reserving them for processes running at $\top$ we can effectively disable many attacks.

5.2 Resident vs. Removable Drives

Files coming from removable drives can contain malicious contents. IFEDAC handles this by explicitly distinguishing between resident drives and removable drives. Local drives are called resident. IFEDAC performs access control and integrity tracking for the files stored in resident drives. All resident drives must be identified in the policy. Removable drives are often plugged into other machines with untrustworthy operating systems. By default, IFEDAC will not perform access control for files stored in removable drives, and will not track those files' integrity levels. All those files are assigned the integrity level $\bot$.

Setting all files on removable filesystems to the integrity level $\bot$ is not desirable when users want to install software or do a system update from a removable filesystem. IFEDAC handles this situation by providing a command to upgrade the integrity level of either all or a portion of a removable filesystem. Such upgrade has two requirements: (1) the upgrade should satisfy the condition given by rule (a5) in Table 1, i.e., only the owner can upgrade a removable filesystem to its own integrity level (2) the upgrade is revoked once the removable filesystem is unmounted.

6 Evaluation

We evaluated the implementation on the Fedora Core 5 distribution of Linux with kernel version 2.6.15, along the following dimensions: usability, security and performance.

6.1 Usability

We established a server and a personal workstation with the IFEDAC module loaded during system boot. On the server machine, we installed some commonly used server applications (e.g., httpd, ftpd, samba, svn) and provided services to our research group. Multiple user accounts exist on the server, some of which are allowed to perform system administration (specified as a sudoer and a member of A, the set of administrators). On the personal workstation, we perform everyday jobs on the Gnome desktop. The jobs we tested include web browsing, emailing, file downloading, instant-messaging and normal system administration. We report some interesting experiences of deploying, configuring and using the IFEDAC module.

A Usage Case We use the email client ThunderBird as a usage case to describe how to configure and use IFEDAC in practise. When a local user u launches the application of ThunderBird, the process inherits the parent's integrity level and runs at $\{u\}$. After the process receives network traffic from remote servers, its integrity level is updated to $\{u, \mathsf{net}\}$. The process needs to read from and write to the configuration files and the files storing the downloaded messages, which are located in the directory $HOME/.thunderbird by default ($HOME refers to u's home directory). In DAC, those files are writable only by u; hence in IFEDAC they have the write protection class $\{u\}$, which is higher than the process's integrity level. To enable the access, we grant the binary executable of ThunderBird an exception privilege to read from and write to the directory $HOME/.thunderbird/ recursively. In this way, the email client can function normally.

If the user wants to save a file from an email attachment to the file system, this is achieved by the *Internet Directory*. The user u can create an Internet directory and set its write protection class to be $\{u, \mathsf{net}\}$. When he wants to save an email attachment, he first saves the file to the Internet directory. The saved file's integrity is initialized as the process's integrity level, $\{u, \mathsf{net}\}$, which can be manually upgraded later if the user has confidence in the file and wants to use it with a higher integrity level. The Internet Directory is not only used by the email client; in fact the user may create multiple Internet directories and can store all downloaded files (e.g., through a web browser, ftp client, instant messenger) to those directories and later upgrade their integrity levels if he wants to. The Internet directory is an example where the write protection class is lower than that inferred from DAC permission. In DAC, that directory is writable only by the owner.

Possible attack channels exposed by the email client include executing a mal-ware in an email attachment, opening an attachment that is a mal-formed file exploiting a vulnerable application and a vulnerability in the email client being exploited by a remote attacker. In all these attacks, the process controlled by the attacker will have the integrity level $\{u, \mathsf{net}\}$ and can only access the files writable by the world and the user's Internet directories. See the security evaluation for details about testing against attacks.

System Administration and Automatic Update Many modern Linux systems allow normal user accounts to perform system administration through the sudo tool. One benefit is better accountability. With IFEDAC we can still use this common usage practise with better security property. These accounts should be in A, the set of administrators. Even though users in A are trusted, each of them still corresponds to a contamination source. This separation helps to enforce the DAC policy. Additionally, most tasks these users perform are user-level jobs that do not need full privileges. Viewing these users as separate contamination sources limits any errors made for a user-level job to that particular user.

For a user $u \in A$, most of his files have the write protection class and integrity level at $\{u\}$ or lower, except for some startup files (e.g., the startup script of the shell) that are used during login. When making u an administrative user, one upgrades the write protection class and integrity level of the user's startup files to $\top$. For example, the startup scripts for Bash Shell include: /.bash_rc, /.bash_profile and /.bash_logout. When he logs in, he gets a shell at $\top$, where he can perform system administration tasks. However, any descendant process that reads his normal files will drop to the integrity level $\{u\}$. He can also downgrade the shell's integrity level to $\{u\}$ by executing a utility program provided by IFEDAC, when he starts performing user-level jobs. To perform system administration later, he needs to obtain a fresh channel with at $\top$ by logging in again.

The startup files owned by users in A provide an example where the write protection class is higher than that inferred from DAC permissions. In DAC, those files are owned by normal users, rather than root. However, if they do not have write protection class $\top$, and the user u accidentally updates it while at level $\{u\}$, then the user cannot later login and perform system administration. Assigning those files with the write protection class $\top$ also helps protecting system integrity, because those files are critical and should only be modified at level $\top$.

Remote administration through a secure shell daemon is expected in some situations. As mentioned in Section 3, one can allow that by specifying the program /usr/sbin/sshd to be a remote administration point (RAP). Also, automatic updates are commonly used in today's commercial operating systems. These programs download updated packages and automatically install them. To enable automatic updates in IFEDAC, the administrator can specify the update program as a RAP, trusting that it is not vulnerable. For example, the automatic update programs in Fedora Core include /usr/bin/yum and /usr/share/rhn/rhn_applet/applet.py.

Exception Policy Configuration Most programs can work with IFEDAC without any modification and policy configuration. Two kinds of programs need exceptions in IFEDAC: network programs and setuid root programs.

Network Programs Like the email client described before, network programs run at the integrity level $\{\mathsf{net}\}$ or $\{u, \mathsf{net}\}$, but need to access configuration and log files that have higher protection class. See Table 2 for a sample policy for some commonly used server and client programs. For each program, only a small number of exception privileges are needed. The policy can be easily understood.

Setuid Root Programs The setuid-root programs run at integrity levels $\{u\}$ when they are executed by a non-administrator u. The default policy will forbid them from performing system critical operations that require the integrity level $\top$. However, most of these programs need to perform such high-integrity tasks. A sample exception policy for setuid-root programs in Fedora Core 5 is shown in Table 3. Those exceptions will be activated only from an integrity level $\{u\}$. That is, if a process has integrity level $\{u_1, u_2\}$ or $\{u_1, \mathsf{net}\}$, it does not get any exceptions when loading the setuid root programs.

IFEDAC provides better protection for setuid-root programs than DAC in three aspects. First, in IFEDAC the privileges gained by those programs are restricted based on the least privilege principle. For example, the program "ping" needs to be setuid-root only because it performs raw socket operations (controlled by the capability CAP_NET_RAW). IFEDAC grants only that exception privilege to "ping", whereas DAC allows "ping" to perform any critical operation. IFEDAC significantly reduces the damage caused by an exploit in "ping". Second, the shell spawned from an exploit loses the exception privileges. In order to abuse the exception privileges, the attacker must inject all malicious code into the address space of the vulnerable process, which is more difficult. Third, with IFEDAC, only malicious local users are able to take advantage of buggy setuid-root programs. Remote attackers breaking in through network programs cannot use setuid-root program to elevate their privileges, because they cannot use the exception privileges if the integrity level contains net.

X Window In X Window environment, X applications access the X server and other desktop daemons such as desktop managers through local domain sockets. We specify the X server and desktop daemons as Local Service Points (LSP). In this way, those processes will maintain the integrity level when serving X applications with lower integrity levels through local domain sockets and other IPC channels. As a result, X applications running at different integrity levels can access the same X server and users can perform system administration and networking activities (such as web browsing) in the same desktop environment. For the Gnome desktop in Fedora Core 5, the desktop daemons include Xorg, gconfd, gdm-binary, gnome-panel, gnome-terminal, gnome-session, gam-server and gnome-setttings-daemon.

What end-users need to know about IFEDAC? In practice, the exception policies should be specified and distributed by the software and OS vendor (e.g., included in the installation packages). System administrators only need to make high-level decisions such as whether to allow remote administration or not. Similarly, administrator only need to specify which users are allowed to perform system administration; the configurations are done automatically by the system. Normal users should understand the basic meaning of read protection class and write protection class for objects (which are similar to ACL). In most situations, the protection classes are derived from the DAC policy and configuring them are achieved by changing the permission bits. In our experiments, the only case that a normal user need to explicitly manage the protection class is to setup the Internet directory, which can be done automatically by the system when a new user is created. Normal users should also understand the integrity level for objects and, in a few situations, users need to manually upgrade an object's integrity level. For example, when a user wants to use a downloaded program to manage his own files, he need to upgrade the program's integrity level from $\{u, \mathsf{net}\}$ to $\{u\}$.

6.2 Security

IFEDAC can defend against most attacks caused by trojan horse and buggy software. To evaluate the effectiveness of IFEDAC, we test IFEDAC against two sets of attack scenarios.

Vulnerability exploitation attacks We used the NetCat tool to provide an interactive root shell to remote attackers. We ran NetCat as root on the victim machine, listening on a port. When the attacker connects to the port, NetCat spawns a shell process, which takes input from the attacker and also directs output to him. In this way, NetCat can be viewed as a vulnerable network server exploited by the attacker. In the interactive root shell, we performed the following three attacks. (1) Installing a rootkit: we attempted to install the kernel mode rootkit "Adore-ng" by loading a kernel module and the user mode rootkit "Linux Rootkit Family (LRK)" by replacing existing system programs. (2) Stealing /etc/shadow: we attempted to send out the /etc/shadow file as an email attachment. (3) Altering another user's web page: we attempted to modify another user's web page files. All attacks succeed on the system with DAC because the spawned shell controlled by the attacker has root privilege as the NetCat process. In contrast, all the attacks fail when IFEDAC is enabled because the NetCat process is running at the integrity level $\{\mathsf{net}\}$ and so is the spawned shell. All those attacks require either executing critical capabilities reserved for the integrity level $\top$ or accessing files with the protection class $\top$ or $\{u\}$, which are not authorized to the processes running at $\{\mathsf{net}\}$.

Trojan horse attacks We assume the victim root user accidentally execute a bot program called Agobot from

Programs	File Exceptions	Capability Exceptions
Servers		

FTP Server
/usr/sbin/vsftpd
	read /etc/shadow;
	write to /etc/vsftpd, /var/log/xferlog;
		CAP_SYS_CHROOT
		CAP_NET_BIND_SERVICE

Web Server /usr/sbin/httpd	read /etc/pki/tls, /var/www; write to /var/log/httpd, /var/run/httpd.pid	
Samba Server /usr/sbin/smbd	write to /var/cache/samba, /etc/samba, /var/log/samba, /var/run/smbd.pid	CAP_SYS_RESOURCE CAP_NET_BIND_SERVICE
NetBIOS Name Server /usr/sbin/nmbd	write to /var/cache/samba, /var/log/samba	
SMTP Server /usr/sbin/sendmail	read /etc/aliases.db; write to /var/spool/mqueue, /var/spool/mail, /var/spool/clientmqueue, /etc/mail, /var/log/mail	CAP_NET_BIND_SERVICE
Clients		
Browser /usr/lib/.../firefox-bin	write to /tmp, $HOME/.mozilla/firefox	
Email Client /usr/lib/.../thunderbird-bin	write to /tmp, $HOME/.thunderbird	

Table 2: Exception privileges for network programs

Usage Types	Setuid Root Programs	Exception Privileges [a] [b],

User information updates	passwd, chage: change user password and expiry information	create files in /etc;
		write to /etc/passwd, /etc/shadow;

	chsh, chfn: change user login shell and finger information	create files in /etc;
		write to /etc/passwd;

PAM (Pluggable Authentication Module) utilities	unix_chkpwd: check user password	read /etc/shadow

	userhelper: update user information	create files in /etc;
		write to /etc/passwd, /etc/shadow;
		read from /var/run/sudo;

Group configuration	gpasswd	create files in /etc;
		write to /etc/gshadow, /etc/group, /var/run/utmp;

User identity switches	newgrp: login to a new group	read /etc/group, /etc/passwd, /etc/shadow, /etc/gshadow; write to /var/run/utmp
	su, sudo, sudoedit: run a shell or other commands as another user	read /etc/shadow, /etc/sudoer
Network utilities	ping, ping6: ping network hosts	use CAP_NET_RAW

Mounting utilities	mount, umount	create files in /etc; read /etc/fstab;
		write to /etc/mtab, /etc/filesystems;

r-commands	rlogin, rcp, rsh: remote login, copy and shell	write to /etc/krb5.conf, /etc/krb.conf; use CAP_NET_BIND_SERVICE
Job scheduling	at: schedule a command	write to /var/spool/at, /var/run/utmp; read /etc/at.allow, /etc/at.deny

	crontab: edit the regular job schedule	write to /var/spool/cron;
		read /etc/cron.allow, /etc/cron.deny;

[a] The write privilege over a file infers the read privilege over the same file.

[b] All write exceptions keep the integrity level of the written files.

Table 3: Exception privileges for setuid-root program in Fedora Core 5

Benchmark	excel throughput	file copy	pipe throughput	process creation	shell scripts
Base	666.3	696.8	452.8	697.37	939.5
IFEDAC	607	661.6	366.7	675.3	856.2
Overhead(%)	9	5	19	3	9
SELinux(%)	5	5	16	2	4

Table 4: The performance results of Unixbench4.1 measurements.

Benchmark	stat	open/close	pipe latency	AF_UNIX sock	fork+/bin/sh-c	RCP/udp latency	TCP/IP cost
Base	3.36	4.70	9.56	16.35	1617	31.31	79.20
IFEDAC	3.69	5.94	11.51	19.82	1842	39.13	82
Overhead(%)	9	26	20	21	14	25	4
SELinux(%)	28	27	12	19	10	18	9

Table 5: The performance results of lmbench 3 measurements.

an email attachment. The program connects to an IRC server, receives commands from a remote attacker, and executes the commands locally. The bot can execute local Linux commands, download files, send spam, and launch DDoS attacks, among others. With DAC, the bot runs as root and can arbitrarily corrupt the system resources. With IFEDAC, the downloaded bot program has the integrity level {net} and so does the process executing the bot. As a result, IFEDAC successfully prevents the bot from corrupting critical system resources and user-owned protected files, but offers no protection against launching the DDoS, spamming, and port scanning attacks. To maximize the effect of the bot, the attacker will try to add the bot to the boot script. The attempt is prevented by IFEDAC and the attacker loses access to the victim once the process is terminated.

6.3 Performance

We compared our performance result with SELinux [22]. Our module has comparable performance overhead. Our performance evaluation uses the Lmbench 3 benchmark and the Unixbench 4.1 benchmark suites. We established a PC configured with RedHat Linux Fedora Core 5, running on an Intel Pentium IV processor 3GHZ with 1GB memory. The test results are given in Table 4 and Table 5.

7 Related Work

The limitations of DAC have been discussed in many sources, e.g., [11, 26]. While such analysis is invaluable, it did not accurately pinpoint the exact problem that makes DAC vulnerable to trojan horses. We show that the key problem lies in trying to associate a *single user* with a request. Traditionally, people deal with the weaknesses of DAC by replacing or enhancing it with Mandatory Access Control (MAC). There are three classes of approaches to add MAC to operating systems: confidentiality-based, confinement-based, and integrity-based.

Perhaps the best known example of confidentiality-based MAC is the Bell-LaPadula (BLP) model [4]. Systems that implement protection models similar to BLP include Trusted Solaris and IX [23]. The BLP model assumes that programs are either trusted or untrusted. This results in a strict security policy rule (the *-property) that will break today's COTS operating systems, unless almost all components are declared to be trusted. IFEDAC, however, divides programs into trusted, benign, and untrusted. This enables it to work on COTS operating system such as Linux while offering meaningful protection.

Confinement-based MAC systems include SELinux [28], AppArmor [2, 8], systrace [29], LIDS [21], PACL [31]. These approaches develop an access control system completely separate from DAC to offer additional protection. Our approach is different in that it preserves the policy goals of DAC. This leads to a smaller policy size and easier policy configuration. Also, it is difficult to configure these other systems to implement information flow policies beyond the creation of a dichotomy of high and low integrity. Jaeger et al. [15] analyzed the SELinux example policy to separate the domains and types into those in a Trusted Computing Base (TCB), i.e., high integrity, and those are not, i.e., low integrity. They found many information flow channels from low to high, due to the nature of Linux. The approaches in AppArmor, systrace, and PACL are to identify a number of programs that, when compromised, could be dangerous, and confine them by a policy. These techniques require a larger policy than that used in IFEDAC, because they do not have default policy rules to allow some accesses and must explicitly specify every access. Furthermore, these approaches remain vulnerable to trojan horse attacks. As most programs, such as shells, obtained through normal usage channels are unconfined, the execution of a trojan horse program will not be subject to the control of the system.

The Biba model [5] is perhaps the earliest mandatory integrity protection model. It provides five integrity policies, which offer important insights into integrity protection and contamination tracking. LOMAC [13] is based on the subject low-water mark policy in Biba. IFEDAC can be viewed as an approach that integrates Biba's integrity tracking and DAC's policy specification and enforcement. We use integrity audits (subject and object low-water mark at the same time) to track the contamination sources of subjects and objects, and then we use these sources as identities in DAC. Microsoft Vista introduced a security feature called Mandatory Integrity Control (MIC) [1]. The approach partitions files and programs into four different integrity levels: low, medium, high, and system. A program running at one level cannot update objects that are at a higher level. There is no information flow tracking; a subject's integrity level is fixed. This can be viewed as a simplified version of SELinux, where there are only four types.

In addition to enforcing DAC policies and allow discretionary control, IFEDAC has also the following two features distinguishing it from the existing Biba-based integrity models. (1) Existing models use a single integrity level for each entity (subject or object) to determine both how the entity contaminates other entities and how the entity is to be protected. This mixing of contamination tracking with the protection classification is undesirable. For example, system logs are inevitably affected by channels that the attacker may control (implying low integrity), but need protection from modification by the attacker (require high integrity). In IFEDAC, protection levels are separate from integrity levels and are derived from DAC policies. (2) Existing models use a linear integrity level that forces users to operate at the same level. The result is that compromising one account with a weak password can compromise all other users' accounts (by trojan horses and exploitations). IFEDAC uses a lattice for integrity labels and provide strong separation between users.

The work that is most closely related to ours is the UMIP model, recently introduced by Li et al. [19]. UMIP uses only high and low integrity levels, and can be viewed as an approximation of IFEDAC, where all labels not containing $\{\mathsf{net}\}$ collapse into high, and all others collapse into low. Consequently, UMIP is unable to provide strong user separation, and UMIP does not separate subject protection classes from integrity levels. Our approach of integrating information flow and DAC is inspired by UMIP's strive for usability and the adoption of DAC information in MAC. Compared to UMIP, IFEDAC is able to enforce all DAC policies correctly and protect against malicious users and weak passwords. We also provide a formal model and security analysis for IFEDAC, which the UMIP paper [19] does not have. As UMIP can be viewed as an approximation of IFEDAC, our analysis can be easily adapted for UMIP, providing a more sound foundation for its security. Another well-known integrity model is the Clark-Wilson model [7], with follow-up work by Karger [16] and Lee [18], among others. These integrity-protection approaches have not been applied to operating systems and do not support user-specific integrity, e.g., separating one user from another.

Language-based information flow security has been studied extensively in the programming language context [9, 24, 25]. This line of work is related to ours because the underlying principles for information flow tracking are often the same. For example, Li et al. [20] discussed four models for integrity labels. In the Writer Model, labels are sets of principals who may have modified the data. Our contamination tracking uses this model. This line of work is orthogonal to ours in that it focuses on analyzing and controlling information flow within programs, and our work uses information flow at the process level and applies that to operating system access control. Our work can benefit from language-based information flow security. In IFEDAC, programs that have exceptions are trusted to process inputs correctly. This trust has to be based on the correctness of these programs, which can be addressed using techniques in language-based information flow security. Some recent papers are starting to bridge this gap. The CW-Lite work [30] addresses this issue of trust by explicitly analyzing source code of programs. Hicks et al. [14] proposed an architecture for an operating system service that integrates a security-typed language with MAC in operating systems, and built SIESTA, an implementation of the service that handles applications developed in Jif running on SELinux.

Dynamic information flow tracking within programs has been used to detect attacks and generate signatures of attacks. Examples include TaintCheck [27] and taint-enhanced policy enforcement [33]. These techniques provide protections orthogonal to ours. They defend particular programs against being exploited by attackers. We focus on general operating system access control techniques that apply to all processes. The idea that a request may represent the intention of one of many principals appeared also in the work of Adabi et al. on access control in distributed systems [3]. There are several recent works on developing new operating system access control models that use information flow. Asbestos [12], HiStar [34], and Fluke [17] use decentralized information flow control, which allows application writers to control how data flows between pieces of an application and the outside world. We have a different goal of fixing DAC without affecting application programmers.

8 Conclusions

The DAC mechanism in operating systems suffers from trojan horses and buggy software. We point out that this is because existing DAC mechanism tries to associate a single principal with each request and cannot do so correctly with the presence of trojan horses and buggy software. We have proposed the IFEDAC model, which uses information flow techniques to track which principals are responsible for a request, thereby achieving DAC policy without assuming that softwares are bug free and benign. While using techniques from mandatory information flow, IFEDAC follows the discretionary control principle and allows owners to decide which other users can access the file and uses the identities of the requester to decide access. In this sense, IFEDAC is the first DAC model that can defend against trojan horses. We have presented the formal model and security analysis of IFEDAC, clearly identifying the assumptions under which it achieves its goal. We have also reported the experiences and evaluation results of our implementation of IFEDAC under Linux.

Acknowledgements

This work is supported by NSF CNS-0448204 (CAREER: Access Control Policy Verification Through Security Analysis And Insider Threat Assessment), and by sponsors of CERIAS. We thank Michael Kirkpatrick for proofreading an earlier version of this paper.

References

[1] The advantages of running applications on Windows Vista. *http://msdn2.microsoft.com/en-us/library/bb188739.aspx*.

[2] Apparmor application security for linux. *http://www.novell.com/linux/security/apparmor/*.

[3] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM TOPLAS*, 15(4):706–734, Oct. 1993.

[4] D. E. Bell and L. J. LaPadula. Secure computer systems: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, Mitre Corporation, Mar. 1976.

[5] K. J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, MITRE, April 1977.

[6] H. Chen, D. Dean, and D. Wagner. Setuid demystified. In USENIX Security 2002.

[7] D. D. Clark and D. R. Wilson. A comparision of commercial and military computer security policies. In IEEE SSP 1987.

[8] C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. D. Gligor. Subdomain: Parsimonious server security. In LISA 2000.

[9] D. Denning. A lattice model of secure information flow. *CACM*, 19(5):236–242, 1976.

[10] DOD. *Trusted Computer System Evaluation Criteria*. Department of Defense 5200.28-STD, Dec. 1985.

[11] D. D. Downs, J. R. Rub, K. C. Kung, and C. S. Jordan. Issues in discretionary access control. In IEEE SSP 1985.

[12] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazieres, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In SOSP 2005.

[13] T. Fraser. LOMAC: Low water-mark integrity protection for COTS environments. In IEEE SSP 2000.

[14] B. Hicks, S. Rueda, T. Jaeger, and P. McDaniel. From trusted to secure: Building and executing applications that enforce system security. In *Proceedings of the USENIX Annual Technical Conference*, June 2007.

[15] T. Jaeger, R. Sailer, and X. Zhang. Analyzing integrity protection in the SELinux example policy. In USENIX Security 2003.

[16] P. A. Karger. Implementing commercial data integrity with secure capabilities. In IEEE SSP, 1988.

[17] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard os abstractions. In SOSP, Oct. 2007.

[18] T. M. P. Lee. Using mandatory integrity to enforce “commercial” security. In IEEE SSP 1988.

[19] N. Li, Z. Mao, and H. Chen. Usable mandatory integrity protection for operating systems. In IEEE SSP 2007.

[20] P. Li, Y. Mao, and S. Zdancewic. Information integrity policies. In *Proceedings of the Workshop on Formal Aspects in Security and Trust (FAST)*, Sept. 2003.

[21] LIDS: Linux intrusion detection system. *http://www.lids.org/*.

[22] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the FREENIX track: USENIX Annual Technical Conference*, pages 29–42, June 2001.

[23] M. D. Mcllroy and J. A. Reeds. Multilevel security in the unix tradition. *Software—Practice and Experience*, 22(8):673–694, Aug. 1992.

[24] A. C. Myers. Jflow: Practical mostly-static information-flow control. In POPL 1999.

[25] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, October 2000.

[26] NCSC. National computer security center: A guide to understanding discretionary access control in trusted systems, Sept. 1987. NCSC-TG-003.

[27] J. Newsome and D. Song. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In NDSS 2005.

[28] NSA. Security enhanced linux. http://www.nsa.gov/selinux/.

[29] N. Provos. Improving host security with system call policies. In USENIX Security 2003.

[30] U. Shankar, T. Jaeger, and R. Sailer. Toward automated information-flow integrity verification for security-critical applications. In NDSS 2006.

[31] D. R. Wichers, D. M. Cook, R. A. Olsson, J. Crossley, P. Kerchen, K. N. Levitt, and R. Lo. Pacl’s: An access control list approach to anti-viral security. In NCSC 1990.

[32] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman. Linux security modules: General security support for the linux kernel. In USENIX Security 2002.

[33] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In USENIX Security 2006.

[34] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazires. Making information flow explicit in histar. In OSDI 2006.

No	Name	Description	Type
0	CAP_CHOWN	Allow changing file ownership and group ownership	Allowed
1	CAP_DAC_OVERRIDE	Bypass DAC checks for file read, write and execute permissions	Allowed
2	CAP_DAC_READ_SEARCH	Bypass DAC checks for file read permission and directory read and execute permissions	Allowed
3	CAP_FOWNER	Bypass checks on operations that normally require filesystem UID of the process match the owner UID of the file	Allowed
4	CAP_FSETID	Allow setting the setuid and setgid bits of files	Allowed
5	CAP_KILL	Allow sending a signal to any process	Restricted
6	CAP_SETGID	Allow manipulating process GIDs	Allowed
7	CAP_SETUID	Allow manipulating process UIDs	Allowed
8	CAP_SETPCAP	Allow transferring or removing capabilities in caller's permitted set to or from any other process	Reserved [a]
9	CAP_LINUX_IMMUTABLE	Allow modification of the S_IMMUTABLE and S_APPEND file attributes	Reserved
10	CAP_NET_BIND_SERVICE	Allow binding to a port below 1024	Restricted
11	CAP_NET_BROADCAST	Allow broadcasting and listening to multicast	Restricted
12	CAP_NET_ADMIN	Allow network administration, e.g., interface configuration, firewall configuration	Restricted
13	CAP_NET_RAW	Allow using RAW sockets and PACKET sockets	Restricted
14	CAP_IPC_LOCK	Allow locking of shared memory segments	Restricted
15	CAP_IPC_OWNER	Bypass permission checks for operations on System V IPC objects	Restricted
16	CAP_SYS_MODULE	Allow inserting and removing kernel modules	Reserved
17	CAP_SYS_RAWIO	Allow I/O port operations and accessing /proc/kcore	Reserved
18	CAP_SYS_CHROOT	Allow using chroot()	Restricted
19	CAP_SYS_PTRACE	Allow ptracing any process	Reserved
20	CAP_SYS_PACCT	Allow configuration of process accounting	Reserved
21	CAP_SYS_ADMIN	Allow system administration, e.g., configuring kernel's syslog, using mount() and umount()	Restricted
22	CAP_SYS_BOOT	Allow executing reboot()	Reserved
23	CAP_SYS_NICE	Allow setting scheduling parameters of any process	Reserved
24	CAP_SYS_RESOURCE	Allow manipulating resource limits, e.g., overwriting disk quota limits	Restricted
25	CAP_SYS_TIME	Allow setting system clock	Restricted
26	CAP_SYS_TTY_CONFIG	Allow configuring tty devices	Restricted
27	CAP_MKNOD	Allow creating special files using mknod()	Restricted
28	CAP_LEASE	Allow establishing file leases on arbitrary files	Restricted
29	CAP_AUDIT_WRITE	Allow writing records to kernel auditing log	Reserved
30	CAP_AUDIT_CONTROL	Allow performing auditing control, e.g., enabling and disabling kernel auditing	Reserved

[a] CAP_SETPCAP is disabled for all processes in current Linux kernel implementation (v.2.6.22)

Table 6: Categorizing 31 capabilities in Linux kernel 2.6 into three types