An Improved Reference Flow Control Model for Policy-Based Intrusion Detection

Jacob Zimmermann, Ludovic Mé, Christophe Bidan
{firstname.lastname}@supelec.fr

Supélec, Campus de Rennes, équipe SSIR

Abstract. In this paper, we describe a novel approach to policy-based intrusion detection. The model we propose checks legality of information flows between objects in the system, according to an existing security policy specification. These flows are generated by executed system operations. Illegal flows, i.e., not authorized by the security policy, are signaled and considered as intrusion symptoms. This model is able to detect a large class of attacks, referred to as "attacks by delegation" in this paper. Since the approach focuses really on attack effects instead of attack scenarii, unknown attacks by delegation can be detected.

Keywords: Policy-based intrusion detection, information flow control, access control

1 Introduction

Traditional intrusion detection techniques build on two distinct approaches: scenario-based detection and anomaly-based detection. In the former case, an intrusion detector includes a database of existing attack scenarii and algorithms that correlate observed system events in order to detect known patterns. In the latter case, observed events are compared to a "normal behaviour" profile that usually results from a statistical learning process. While these intrusion detection systems (IDSes) have been effective in many cases, they raise a number of inherent problems [1]:

- **maintenance costs:** for a scenario-based IDS to remain useful, its database must be constantly upgraded to include new attack signatures as they are discovered. Anomaly-based detectors require a new profile building phase whenever new applications with legitimate but different behaviour are put in service;
- **detection reliability:** a scenario-based IDS is by definition limited to known attack scenarii, which are less likely to be used by a possible intruder. Novel attack detection remains limited or impossible. This may produce false negatives, i.e., successful attack not reported by the detector;
- **detection accuracy:** while successful attacks may remain undetected, an IDS also produces false-positives, i.e., alerts reporting intrusions that did *not* occur.

We believe that to a large extent, these problems arise from the fact that anomaly or scenario-based intrusion detection relies on a context-free definition of what a potential attack may be: while by definition any attack is a security policy violation, the security policy itself is not taken into account in these approaches.

There has been increasing interest in *policy-based* intrusion detector development in the recent years [2]. A policy-based IDS consists in a logical security policy specification and an execution trace validation algorithm. Such an approach has the potential to improve significantly over statistical anomaly detection and scenario detection in terms of reliability, accuracy and required maintenance. Ideally, maintenance is needed only to change the specified policy, there is no need for permanent upgrades or adaptation phases.

The present work pursues these goals by modeling a system in terms of objects, operations and information flows between objects. A criterion to distinguish between legal and illegal information flows according to a security policy is established. Information flows that appear as illegal are symptomatic of security policy violation attempts and should be reported as intrusion alerts.

Our purpose is not to enforce a security policy *a priori*. Instead, we use information flow control as a means to detect given access control criteria violations. We want the policy enforcement mechanism to remain transparent to users as much as possible. In this paper, we focus on discretionary access control policies, as these are used in the actual targeted operating systems. Nevertheless, the approach is general enough to be able to handle also for instance a Chinese Wall policy or a Bell&LaPadula policy.

In this paper, we present a formal intrusion detection model. Some implementation issues are discussed and we show the ability of the model to detect actual intrusions.

The rest of this paper is organized as follows. In section 2, we present the class of attacks we focus on and related work that provide partial solutions to prevent or detect such attacks. The proposed intrusion detection model is formally introduced in section 3. Section 4 describes our implementation on the Linux OS, as well as an actual attack detection example. Finally, section 5 presents the conclusion of our current work and some future directions.

2 Background

In this section, we present a class of host-based confidentiality or integrity violations, referred to as "attacks by delegation". These attacks can be described and modeled in terms of information flows. We briefly present and discuss several existing techniques and models that can be used to deal with these attacks.

2.1 Attacks by delegation

Integrity and confidentiality policies in operating systems are primarily enforced using access control mechanisms. If the access control implementation is correct, then it forbids any action that explicitly violates desired confidentiality and

integrity criteria. However, experience shows that security policy violations, i.e., intrusions, are still possible. Even if the attacker is prohibited from executing a specific action, an intrusion can be achieved by various means:

- The well-known access leakage problem, inherent to HRU-based access control models [3], can be exploited. In this case, the action eventually executed by the attacker is *explicitly authorized* by the access control, but is *semantically equivalent* to an action declared as illegal in terms of access to information;
- Race conditions can be exploited by interfering with the actions executed by some other process or subject. The attacker knows that a specific action will eventually be executed and schedules its own actions in order to take advantage of this. While all the actions performed by the attacker are legal by themselves, the security policy is violated by the *effect of their scheduling*;
- Abusing the privileges of other subjects is another form of interference, as in *confused deputy*-based attacks. In this case, the policy is violated by the fact that this action is executed as a consequence of an action of the attacker. Under some circumstances, *Trojan horses* and *back-doors* can be considered as confused deputy exploits, where the confused deputy itself is designed by the attacker to meet his needs. Once installed using some other attack, it can be activated by the attacker purposely to perform operations that are not allowed to him;
- We can consider some *buffer overflow*-based attacks as special cases of Trojan horses. Here, a misbehaving program executing actions programmed by an attacker using a buffer overflow exploit has the same effect as a Trojan horse installed by the attacker.

	lpr daemon	attacker
1		lpr /home/user/doc.txt
2	read /var/spool/lpd/job-****	
3		rm /home/user/doc.txt
4		ln -s /home/user/doc.txt /etc/shadow
5	read /home/user/doc.txt	
6	send document to printer	

Table 1. The *lpr* attack

These intrusion scenarii consist in series of actions, where each one is legal from the access control point of view when considered separately. However, when combined and executed in a particular context, they lead to some effect that is explicitly forbidden by access control.

We call such attacks *attacks by delegation*. Indeed, an attacker aiming at executing some action, but not allowed to do so, will "delegate" the action to

some other subject, or a special context he produces. Note that this concept is orthogonal to the more common classification of attacks (DoS, read access to data, execution of code), because in each of these categories, number of attacks can be modeled as attacks by delegation.

Attacks by delegation can be described in terms of information flows between objects in the system. As a simple example, let us consider the well-known attack against *lpr* (please refer to table 1). While each step is legal *per-se*, the final result is that the attacker prints /etc/shadow, which is prohibited by the DAC policy. This attack results in an information flow from a file (in this example, /etc/shadow) to the printer device, while the printer is not an allowed destination for this flow. The printer daemon is used as a confused deputy to produce such a flow (in step 6), by combining other legal information flows using a race condition (in steps 3 and 4).

2.2 Related work

Information flow control In the general case, access control mechanisms that consider each operation separately cannot protect a system against attacks by delegation. In most cases, rights to execute operations are granted on a strictly context-free subject/object identity basis, while interference and cooperation schemes take into account context and execution history when dealing with access to data and data modification. In other words - the security policy is implemented at the *object access control level*, while attackers operate at the *information flow level*.

In theory, this problem can be solved by using information flow control models; a large number of these has been proposed [4,5,6,7,8,9,10,11]. However, when focusing on implementing given security policies on existing general-purpose operating systems, their actual usability remains limited due to various practical reasons:

1. Models that enforce complete control over all possible information flow channels, including hidden ones, require knowledge of control flows and dependency graphs [4,5]. In our opinion, such a fine granularity is unrealistic on a large-scale OS running third-party software.
2. Coarse-grained, multi-level models that enforce control over well-defined channels have been proposed (such as the Bell&LaPadula [6], Biba [7] and Chinese Wall [8] models). A common basis to many of these models is that they enforce unidirectional information flow channels and ensure that the graph of all possible information flows is acyclic [9]. This is generally considered too strict for practical usage, so that exceptions have to be introduced, in the form of group intersections, declassification operations, method waivers [10], etc. Again, static program analysis may be required in such cases to avoid substantial overhead of run-time information flow checking [11];
3. Particular models designed for specific applications rely on assumptions met by a highly specialized operating environment, but remain unusable in the

general case. For instance, the model described in [12] exploits the Java visibility rules and security constraints to infer knowledge of possible information flow channels, which is a form of static program analysis.

Neither of these approaches is directly usable on a generic OS such as Linux, Unix or Windows, especially if the existing security policy remains unchanged and the system runs unmodified closed-source software.

Access control Confused deputy attacks can be avoided using a capability-based access control model such as [13,14]. By binding possible operations of a process directly to objects, these models prevents attacks such as those based on symbolic links or file renaming. Similarly, the problem of access leakage in access control models can be addressed in multiple ways. Typed access control models [15,16] provide means to define safety invariants and ensure that a subject will not be able to gain certain undesired privileges. Nevertheless these models remain unable to prevent malicious cooperation or interference. They also fall short in dealing with back-doors or operation side-effects.

To deal with privilege and information transfer from an access control point of view, the "Take-Grant" model and its derivatives were developed [17,18,19]. This formalism allows to distinguish between *de jure* transfers, where a subject gains privileges that allow him to access some informations, and *de facto* transfers, that consist in accessing some copy of that information [20]. Later, the problems of privilege theft, conspiracy [21,22] and information flow between subjects and objects [23] have been modeled using this framework. The proposed solutions provide means to validate a given access control policy, modeled using the "Take-Grant" formalism, against potential *de facto* transfers or conspiracy scenarii.

Some attacks by delegation actually create *de facto* transfers or privilege thefts, and could thus be prevented by modifying the policy using these approaches. However, others forms of attacks involving illegal information flows (such as Trojan horses or back-doors) cannot be modeled so easily this way.

Intrusion detection In the intrusion detection field, an interesting approach to race conditions-based attacks detection has been proposed [24]. The authors define a property of subjects non-interfering with data and propose an algorithm for system integrity enforcement. They show their model's ability to detect novel security policy violations, however security policies taken into account in this work focus only on subjects being forbidden to overwrite a piece of data. In Appendix A, we show that policies of this form can easily be handled also using our model.

An approach to prevent privilege abuse and confused deputy attacks is the "privilege graph" model [25], that takes into account implemented subject privileges, as well as vulnerabilities that allow some subject to gain additional privileges. This model can be used to evaluate a given security policy by estimating the global security level of that policy and the impact of policy changes (subject creation, subject privilege modification, etc.). >From an intrusion detection

point of view, this can be seen as an event correlation model. However it is not a runtime intrusion detection mechanism by itself.

2.3 Positioning of our work

Since our goal is to detect attacks by delegation that produce illegal information flows, we propose to enforce a simplified information flow control by restricting our focus only to flows between system objects. In practical terms, anything that is identified as an "object" by the underlying operating system fits in the model: in a Windows or Unix-like system, this includes files, pipes, FIFOs, sockets, message queues, shared memory buffers etc.

Our purpose is not to enforce a security policy *a priori*. Instead, we use information flow control as a means to detect given access control criteria violations by various forms of interference or confused-deputy exploits at runtime. In this paper, we focus on **discretionary access control policies**, as these are used in the targeted operating systems. Nevertheless, the approach is general enough to be able to handle also for instance a Chinese Wall policy or a Bell&LaPadula policy.

3 Intrusion detection model

We propose to simplify information flow control by considering only flows between system objects. Any access to an information is modeled by a method call on the object containing that information. By executing series of method calls, system operations produce information flows between objects. A security policy is enforced by defining domains that specify sets of method calls, each combination of these calls producing authorized information flows. By definition, to be legal, information flows must occur within an unique domain; any flow that violates this rule violates by definition a confidentiality or integrity requirement.

3.1 Objects and operations

The elementary unit of information in our model is an *atomic object state*. Atomic objects provide information containers. Their state is set and retrieved using method calls.

According to the usual definition, the state of an object is defined by the values of its attributes. Atomic objects [1] are objects whose attributes depend on each other and cannot be modified separately, thus the security policy can only deal with the object as a whole (for instance, a file contents, not a part of it). In practical terms, our model can enforce a policy if a file contents is accessed using the *read* and *write* methods as an atomic objects, but not if it is accessed

[1] In this paper, we further refer to *atomic objects* simply as to *objects*.

through a raw device or by mapping the file as part of the virtual memory of a process (which are not atomic)[2].

By allowing to access and modify an object state, its methods in effect produce information flows. Any operation executed on the system that accesses or modifies the state of an object thus has a meaning in terms of generated information flows between objects. As such, it is defined by a set of objects and the set of called methods. We use the symbol $\gg$ to denote such operations: an operation

$$\{o.m\} \gg \{o'.m'\}$$

creates an information flow from object o to object o' by accessing the state of o using its method m and, as a consequence, setting the state of o' using its method m'.

Since we focus only on flows produced by method calls, information flows using hidden channels are clearly outside of our scope.

Our operating system model is formally specified as follows:

Definition 31 *A system is a tuple* $(O, M, \Omega, S, exec)$ *where*

- O *is a set of object identifiers;*
- M *is a set of object methods;*
- $\Omega \subseteq \{(O \times M)^* \gg (O \times M)^*\}$ *is the set of operations that generate information flows between objects,* $*$ *being used as a notation for the powerset;*
- S *is a set of system states;*
- $exec : \Omega \rightarrow (S \rightarrow S)$ *are the information flow semantics of operation execution, i.e., the system state transition functions.*

This model provides an instance of an EM enforcement mechanism [27]. The set of system states S and state transition functions $exec$ will be formally defined in the following sections.

3.2 Domains and references

An attack by delegation symptom is an information flow between objects that violates the enforced security policy. To detect operations that create such flows, we need to define a "legality condition" for the generated flows.

As implemented by an access control mechanism, a security policy explicitly declares legal and illegal flows. Let us consider the sample DAC policy from table 2. When applied to a given system state (i.e., the states of all the system objects), this policy declares an operation such as $\{m.r\} \gg \{n.w\}$ as legal. On the opposite, we consider that any access that is not explicitly authorized is forbidden. Thus, an operation such as $\{m.r\} \gg \{p.w\}$ is illegal.

[2] In current operating systems, non-atomic object exist; however, the problem is known and is being addressed in current development [26]. Thus, our model only handles atomic objects, while the implementation will use the possibility of emulating atomic objects as soon as it will be released.

	Alice	Bob
m	$\{r, w\}$	$\emptyset$
n	$\{r, w\}$	$\{r, w\}$
o	$\{r, w\}$	$\{r, w\}$
p	$\emptyset$	$\{w\}$

Table 2. Sample access control policy

In the general case, a security policy implies legal combinations of object method calls to be used by operations. We call *domain* a set of method calls such that any combination of these is legal as regard to the security policy. Therefore, an operation that executes only method calls that belong to the same domain is legal by definition. Similarly, an operation that consists in method calls of several distinct domains is illegal, as the information flow it generates is not considered as authorized by the security policy.

Formally, the method calls that belong to a given domain are defined by a set of *references*. A reference $R_d o.m$ allows to execute the $o.m$ method call in the domain d. Given a system state (i.e., the states of all objects), each object is bound to a set of references.

Definition 32 *Let D be a set of domain identifiers. A system state is a objects-to-references binding:*

- $S : O \rightarrow \{R_d o.m | d \in D, o \in O, m \in M\}$

Given an object $o \in O$ and a state $s \in S$, $s(o) \subseteq \{R_d o.m | d \in D, m \in M\}$ is the set of all references that provide legal accesses to the o object in the s state.

Defining domains and related reference sets in an initial system state is the means to implement the existing security policy. The actual domain building algorithm depends on the security policy one wish to implement, the common DAC case is treated in section 4.2.

For a single information flow, the "domain uniqueness rule" is formally defined by the *islegal* predicate:

Definition 33 *We define the function $dom : (O \times M)^* \times S \rightarrow D^*$ as the set of all domains that allow a set of object method calls to be executed in a system state. Given an operation and a system state, we define the predicate $islegal : \Omega \times S \rightarrow \{true, false\}$ as true if all the executed method calls are allowed within some same domain, i.e. if there exists some domain where the appropriate method calls on both the source objects and the destination objects can be executed.*

- $dom[\{o_1.m_1, \ldots, o_p.m_p\}, s] = \{d \in D | \forall i \in [1 \ldots p] R_d o_i.m_i \in s(o_i)\}$
- $islegal(src \gg dst, s) = true$ *if* $dom(src, s) \cap dom(dst, s) \neq \emptyset$

This predicate checks that there exists at least one domain where accessing both the source objects (src) and the destination objects (dst) is allowed. Since any

flow within some domain is by definition legal, the "domain uniqueness rule" ensures that no illegal information flow occurs when executing the $src \gg dst$ operation.

3.3 Causal dependency and reference flow

Information flows between objects are also generated by sequences of causally dependent operations. According to Lamport's causal ordering [28], an operation $\omega_2 : src_2 \gg dst_2$ is a causal consequence of some operation $\omega_1 : src_1 \gg dst_1$ if it accesses an object state set by ω_1, i.e., if there exists an object $o \in O$ and two methods $(m, m') \in M \times M$ such that $o.m \in dst_1$ and $o.m' \in src_2$. In this case, the sequence $[src_1 \gg dst_1; src_2 \gg dst_2]$ actually creates an information flow $src_1 \gg dst_2$. Thus, after having executed $src_1 \gg dst_1$, $src_2 \gg dst_2$ is legal if, and only if, $src_1 \gg dst_2$ is legal.

Since information flow legality is enforced using references, that means that if $src_1 \gg dst_1$ occurs, the dst_1 objects must then have references similar to those of the src_1 objects. An information flow thus results in a flow of references. More precisely, whenever a legal information flow occurs, references from the source objects are propagated to the destination objects. Executing $src \gg dst$ in some system state s thus generates a new system state s' through the state transition function $exec : \Omega \rightarrow (S \rightarrow S)$.

Definition 34 *$exec(src \gg dst) = s \rightarrow s'$ such that $\forall o \in O$,*

- *if $\exists m \in M | o.m \in dst$*
  then $s'(o) = Prop(o, s, src \gg dst) \cup Create(o, src \gg dst)$ where
  - *$Prop(o, s, src \gg dst) = \{R_d o.m' | d \in dom(src, s)$*
    and $\exists o' \in O, m'' \in M | o'.m'' \in src \wedge R_d o'.m' \in s(o')\}$
  - *$Create(o, src \gg dst)$ is a set of new references.*
- *else $s'(o) = s(o)$*

When a flow from an object o' to an object o occurs in the state s, the operations possible on o in the new state s' must be at most the operations possible on o' in the previous state s. This is enforced by the reference flow, formalized by the definition 34: $Prop$ is the set of flowing references, whereas $Create$ is used in operations that create new objects. The first condition in $Prop$, i.e. $dom(src, s)$, selects all the domains in which a flow sourcing at the src objects in state s is legal. The second condition select all references that relate to source objects (i.e. objects that have methods called in src) in these domains. Notice that theoretically, some of these methods may not exist for the destination objects. However, a non-existing method will by definition not be called, so these references will never be involved in any further reference flow. Moreover, practically, all objects involved in a given operation have the same interfaces: thus this case does not even occur in the implementation.

Object creation operations are not formalized specifically, since they simply consist of information flows to objects. However, for the new objects being

accessible, appropriate created references to these objects must be bound to $dom(src, s) \cap dom(dst, s)$. Just like representing known operations using the $src \gg dst$ formalism, this depends on the actual system implementation.

As an example, let us consider again the DAC policy from table 2. In the scenario shown on table 3, operation at step 1 generates a flow[3] from m to n; a flow from m to p would have been illegal since the DAC policy allows either reading m or writing p, but never both simultaneously. Thus, the flow from n to p at step 2 is also illegal, because it would result in a flow from m to p through n.

s	operation	legal flows	islegal
0		$m \leftrightarrow n; m \leftrightarrow o; o \leftrightarrow n; o \rightarrow p; n \rightarrow p$	
1	$\{m.r\} \gg \{n.w\}$	$m \leftrightarrow n; m \leftrightarrow o; o \leftrightarrow n; o \rightarrow p$	true
2	$\{n.r\} \gg \{p.w\}$		false

Table 3. Illegal sequence of operations

3.4 Intrusion detection

Let us consider a $(O, M, \Omega, S, exec)$ system model with S and $exec$ as defined in the previous section. We suppose the system to be in some initial state s_0 that binds objects and their methods into domains as implied by the access control policy. Thus, any cross-domain operation is supposed to be an intrusion symptom. To detect such symptoms in an execution trace, i.e., a sequence of operations $[\omega_1, \omega_2 \ldots \omega_n]$, we have to check the "no cross-domain" property for each of the operations. That is:

1. For each operation ω_i, the *islegal* predicate must be *true*.
2. Each operation modifies the state of the system as defined by the *exec* function,

4 Implementation

We have developed a run-time intrusion detector using this model on Linux. This system matches well the object-based model with a discretionary access control policy. In addition, the system being open-source provides an ideal testbed environment.

Our implementation has shown promising abilities to detect different realistic attacks, including unknown attacks. In this section, we give some details about this implementation and show, on one interesting example, its ability to detect actual intrusions.

[3] In table 3, $n \rightarrow p$ denotes an authorized flow from n to p, i.e., $islegal(\{n.m_1\} \gg \{p.m_2\}, s) = true$, $m_1, m_2 \in M$

4.1 Modeling the Linux OS

In this implementation, we consider the following objects as information containers:

- Filesystem objects (i.e., regular files, directories and named FIFOs). Special "device files" are not considered;
- User and client related metadata: authentication accounts handled through the PAM system and IP ports
- Input/output descriptors for filesystem objects, pipes and sockets;
- POSIX inter-process messages;
- POSIX shared memory objects;
- process memory images.

The subset of system calls that we must handle to deal with these objects is quite small: while Linux kernel version 2.4 defines over 250 system calls, only 16 generate actual information flows between objects and are taken into account in our present implementation. These are calls that create or use I/O descriptors (file, pipe and socket operations), the `mmap` call, the `msgget` and `msgsnd` calls. Each has well-known semantics regarding information flows, from which we derive a precise specification in terms of reference requirements, such as those shown on table 4, and also reference creation where it applies. These rules are defined once and for all by the implementation.

As described in section 3.1, the implemented model assumes objects to be atomic. Nevertheless, some objects in Linux are not atomic: for instance, each line the */etc/shadow* file can theoretically be overwritten only by information flowing from a given user's console. This cannot be easily modeled, so some attacks (such as *Alice* changing *Bob*'s password instead of her own) will not be detected. However, a new filesystem is currently being developed that will eventually solve the problem by allowing to represent any information as a set of atomic files [26]. Once this feature will become available, we will take advantage of it.

System call	Reference requirements	Created flows
sendfile(out,in,offset,count)	$R_d out.w, R_d in.r$	$\{in.r\} \gg \{out.w\}$
read(fd,buf,count)	$R_d fd.r, R_d buf.r, R_d buf.w$	$\{fd.r, buf.r\} \gg \{buf.w\}$
mmap(s,l,p,f,fd,o)[4]	$R_d s.r, R_d s.w, fd.r$	$\{s.r, fd.r\} \gg \{s.w\}$

Table 4. Sample reference requirements

Algorithm 1 Building the initial state s_0:

1. Let U be the set of Linux subjects other than *root*.
2. Initially
   (a) $D = U$
   (b) $\forall o \in O, s_0(o) = \emptyset$
3. For each u in U :
   (a) for each method m in M, let $All_m(u)$ be the set of objects that u is allowed to execute m on.
   (b) If there exists an u' in U, $u' \neq u$, such that for any method m, $All_m(u) \subseteq All_m(u')$
      i. then $D \leftarrow D - \{u\}$
      ii. else for each o in $All_m(u)$, $s_0(o) \leftarrow R_u o.m$

4.2 Building the initial state

Since Linux enforces a discretionary access control policy, the initial object-to-references binding s_0 is easily deduced from the DAC matrix by forbidding *de facto* access rights transfers [20]. Since DAC is the primary means to implement security policies in Linux (and other current operating systems), we use it as a security policy specification. Thus, our goal is not to refine or correct DAC permissions, but to enforce the existing policy in cases where it could be violated by some attack by delegation.

Basically, by reading and writing objects he is allowed to access, any subject can generate information flows between these objects. Our interpretation of the policy is that such a flow is legal if some subject is allowed to read the source objects and write the destination objects. Thus, all flows a given subject can generate belong by definition to the same domain(s).

This interpretation of the Linux DAC is problematic in one case: suppose that *Bob*'s access rights are strictly a subset of *Alice*'s ones. That means that any flow *Bob* can possibly produce can also be generated by *Alice*; so all flows that either *Bob* or *Alice* can generate belong to the same domain(s). Therefore, if an attack by delegation attempts at creating a flow between *Alice*'s and *Bob*'s objects, it will remain undetected.

We argue, however, that there is no harm due to this. For the situation to arise, *Bob* would actually have to share *everything* with *Alice*, including her authentication account. If this is the case, then no action of *Bob* could be distinguished from an *Alice*'s action, which is undoubtedly a policy anomaly.

The set of domains D and the initial state s_0 are built by the algorithm 1. For any subject but *root*, it checks whether his rights are superseded by those of some other subject. If it is not the case, then all flows this subject can produce define one domain.

# of operations	12051434
default	9m 40s
with active checker	11m 23s

Table 5. Linux kernel compilation benchmark *(make bzImage)*

4.3 Running the checker

Our implementation consists in a user-space daemon running as a low-priority background process. A trivial kernel module hooks on the observed system calls and sends a message to the daemon whenever any of them is executed by a process. Thus, actual reference flow processing and checking the *islegal* condition is asynchronous. Using this design, the performance degradation on the target host is kept low (please see table 5 for an example).

Theoretically, the maximum number of domains is $number\,of\,objects * number\,of\,subjects$, assuming a DAC policy. However, most of the domains actually overlap; by considering all flows that may be generated by a given DAC subject as belonging to the same domain, the maximum number of automatically generated domains is the $number\,of\,subjects$. Let us notice that the initial domain construction phase takes approximatively 15 minutes on a standard PC running Linux with 13 users and over 120000 objects, or 8 minutes on our mail server (which has a much faster SCSI disk).

The *islegal* test complexity is linear in the number of checked references. There can be at most $number\,of\,methods * number\,of\,domains$ references per object; most objects have only three methods (*read*, *write* and *exec*). Because no new domains are created during execution[5], the maximum number of references per object remains constant.

The maximum number of propagated references per object is $number\,of\,methods * number\,of\,domains$, in the worst case (a flow being allowed in all the domains at the same time, such as a flow between two world-readable, writable and executable objects). In our test on the mail server, this worst number is 39. However, most of the time, a flow is allowed in only one domain; thus, the number if propagated references per object is $number\,of\,methods$, i.e. 2 (*read* and *write* methods, since executable objects are rarely involved in flows).

4.4 Actual Attack Detection Example

As an actual intrusion detection example, we present a Trojan horse-based attack against OpenSSH [29]. The purpose of this attack is to force the login process started by OpenSSH to execute a *setuid(0)* operation and thus start a *root* shell, regardless of the attacker's authentication. The exploit is based on the fact that the OpenSSH daemon can be configured to use an external login program, and

[5] Currently, new user creation or policy modification is not handled by our implementation.

to allow users set their own environment variables before the login procedure begins.

Please refer to table 6 for the attack scenario. It consists in creating a shared library (*libroot.so*) that overloads the *setuid* system call. When executed through this library, *setuid* effectively sets the process' *uid* to 0. The attacker then uses the *LD_PRELOAD* environment variable to force loading of this library when the */bin/login* executable is started. After successful authentication, the login program executes an *setuid* operation to set the authenticated user's identity. Because of *libroot.so* being loaded, this becomes in effect *setuid(0)*. As a result, the attacker gets a *root* shell.

	operation	required references
1	attacker creates *libroot.so*	$R_{d1}console.r, R_{d1}librootso.w$
2	attacker sets *LD_PRELOAD*	
3	attacker starts ssh session	
4	sshd executes */bin/login*	$R_{d2}login.r, R_{d2}login.x, R_{d2}img.w$
5	login loads *libroot.so*	$R_{d1}librootso.r, R_{d2}img.r, R_{d2}img.w$
6	authentication process	
7	login executes *setuid(0)*	
8	login executes */bin/sh*	
9	*root* shell is available	

Table 6. OpenSSH attack

At step 1, the *libroot.so* file is created as a destination object of an information flow whose source is primarily an object that represents the attacker's console. We consider that this flow occurs within some domain d_1, and that no flow is permitted between the console and the running *sshd* daemon. At step 4, the memory image of the *login* process (here noted as *img*) results from a child of the *ssh* daemon executing */bin/login*; that is, a flow from */bin/login* to *img*. It belongs thus to some domain d_2 where it is possible both to read the contents of */bin/login* (most probably, this is possible in any domain) and to overwrite the memory image of the calling process *img*.

These two flows converge at step 5, by the contents of *libroot.so* flowing into *img* when the *mmap* system call is executed. Since it is necessary to write *img* again, this could occur but in the domain d_2. However, the read reference available for *libroot.so* is $R_{d1}librootso.r$. Since this flow requires thus references from two domains, it is illegal and should be considered as an intrusion symptom.

The security policy goal enforced here is precisely that no information produced by the attacker should modify the *login* program. No hypothesis is made on how this modification may actually occur. It can be:

– by actually overwriting somehow the */bin/login* file;

- by using the described attack against OpenSSH;
- by using the same attack against another program[6];
- and so on.

All these cases lead to an identical effect in terms of information flows between objects: a cross-domain flow occurs between an object produced by the attacker and an object causally bound to the *login* program. As such, the attack will be detected in these terms in all cases, including possible novel ones.

Note: In this attack, the model in effect results in forbidding usage of user-defined shared libraries in programs that aren't executed under the user's own identity. Note that this very behaviour is implemented *ad-hoc* in the ELF dynamic linker, precisely to avoid such misuse. The OpenSSH attack is an attack by delegation that overcomes this prevention mechanism.

5 Conclusion

We have proposed a mechanism to detect host-based attacks by delegation by checking information flows between system objects through known and expected channels. This mechanism requires neither actual attack scenarii knowledge, nor empirical "normal behaviour" profile. As such, it is able to detect known as well as novel attacks and is suitable for host-based, policy-based intrusion detection.

The domains definition, resulting from a security policy specification, provides in effect legal execution scenarii constructs. The proposed approach is thus as special case of anomaly-based intrusion detection. Detected illegal operations provide information about attempted attack semantics (as an attacker trying to modify the *login* program execution), but no information about the precise attack scenario that led to this operation.

We have developed a runtime implementation on the Linux operating system. Our current test are encouraging, the system actually proves to be able to detect known as well as unknown host-based attacks, as well as some remote attacks that lead to a cross-domain information flow on the target system. The experiments show that the implementation of the model raises alarms only if an illegal flow actually occurs. False alarms exist, but are due to special cases where neither the policy specification, nor the model, are precise enough. These cases remain extremely rare and thus the number of false alarms is very low (4 in 26 million of processed events). These results are discussed in [30].

Acknowledgments

The work presented in this paper was partly funded by the French Ministry of research in the context of the DICO RNTL project. We would like to thank the

[6] Originally, this was a known problem of the telnet daemon, which was resolved. The same vulnerability being discovered later in OpenSSH shows the limits of this "patch and pray" approach.

Ministry and the RNTL for their support. We also would like to thank Heiko Mantel for in-depth comments and remarks about the detection model presented in this paper, as well as the anonymous reviewers for many valuable comments.

References

1. Herve Debar, Marc Dacier, and Andreas Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 31(8):805–822, 1999.
2. Chuck Boeckman. Getting closer to policy-based intrusion detection. *Information Security Bulletin*, pages 13–20, May 2000.
3. M.A. Harrison, W.L. Ruzzo, and J.D. Ullman. Protection in operating systems. *ACM*, 19(8):461–471, August 1976.
4. Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.
5. Richard A. Kemmerer. Shared resource matrix methodology: An approach to identifying storage and timing channels. *j-TOCS*, 1(3):256–277, August 1983.
6. D. Bell and L. LaPadula. Secure computer systems: Unified exposition and multics interpretation. Technical report, The Mitre Corp., 1976.
7. K. Biba. Integrity considerations for secure computer systems. *MTR-3153, Mitre Corporation*, 1975.
8. David F.C. Brewer and Michael J. Nash. The chinese wall security policy. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 206–214. IEEE Computer Society Press, May 1989.
9. K. G. Walter et al. Primitive models for computer security. Technical Report ESD-TR4 -117, Case Western Reserve University, 1974.
10. E. Ferrari, P. Samarati, E. Bertino, and S. Jajodia. Providing flexibility in information flow control for object-oriented systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 130–140, 1997.
11. Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Symposium on Operating Systems Principles*, pages 129–142, 1997.
12. Thomas P. Jensen, Daniel Le Metayer, and Tommy Thorn. Verification of control flow based security properties. In *IEEE Symposium on Security and Privacy*, pages 89–103, 1999.
13. Daniel Hagimont, Jacques Mossiere, Xavier Rousset de Pina, and F. Saunier. Hidden software capabilities. In *International Conference on Distributed Computing Systems*, pages 282–289, 1996.
14. Alan H. Karp, Rajiv Gupta, Guillermo Rozas, and Arindam Banerji. Split capabilities for access control. *HP Laboratories Palo Alto, HPL-2001-164*, 2001.
15. R. S. Sandhu. The typed access matrix model. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 122–136, 1992.
16. Jonathon Tidswell, Geoffrey H. Outhred, and John Potter. Dynamic rights: Safe extensible access control. In *ACM Workshop on Role-Based Access Control*, pages 113–120, 1999.
17. Richard J. Lipton and Lawrence Snyder. A linear time algorithm for deciding subject security. *JACM*, 24(3):455–464, 1977.
18. J. Biskup. Some variants of the take-grant protection model. *Information Processing Letters*, 19(3):151–156, 1984.
19. M. Dacier. A petri net representation of the take-grant model. In *6th IEEE Computer Security Foundations Workshop*, pages 99–108, Franconia, NH, 15-17 June 1993. IEEE Computer Society Press.

20. M. Bishop and L. Snyder. The transfer of information and authority in a protection system. In *Proceedings of the Seventh Symposium in Operating Systems Principles*, pages 45–54, December 1979.
21. L. Snyder. Theft and conspiracy in the take-grant protection model. *Journal of Computer and System Sciences*, 23:333–347, 1981.
22. Matt Bishop. Theft of Information in the Take-Grant Protection Model. Technical Report PCS-TR88-137, Dartmouth College, Computer Science, Hanover, NH, 1988.
23. M. Bishop. Conspiracy and information flow in the take-grant protection model. *Journal of Computer Security*, 4(4):331–359, 1996.
24. Calvin Ko and Timothy Redmond. Noninterference and intrusion detection. In *Proccedings of the IEEE Symposium on Security and Privacy*, 2002.
25. M. Dacier and Y. Deswarte. Privilege graph: an extension to the typed access matrix. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 1994.
26. The reiser4 filesystem - in progress draft document. http://www.namesys.com/v4/v4.html.
27. Fred B. Schneider. Enforceable security policies. *Information and System Security*, 3(1):30–50, 2000.
28. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
29. CMU CERT/CC. Vu#40327: Openssh uselogin option allows remote execution of commands as root. http://www.kb.cert.org/vuls/id/40327, November 2001.
30. J. Zimmermann, L. Mé, and C. Bidan. Experimenting a policy-based hids based on the reference flow model. Technical Report SSIR-2003-01, http://www.supelec-rennes.fr/rennes/si/equipe/lme/ZMB03.pdf, Supélec, March 2003.

Appendix A

The model published by Ko & Redmond in [24] focuses on noninterference between a group of users G and a piece of data o, noted $G\triangleright \parallel \{o\}$. Integrity policy goals consist in such noninterference requirements, that prevent specific users from modifying critical data. A system trace, as defined by the authors, is a sequence of system state changes where each information modification is performed on behalf of some subject.

We do not define "subjects" explicitly. However, any interaction with users is done through some terminal object, i.e. a *tty* device, a network socket, a removable device etc. Thus, an operation is executed by some user if it involves a flow from this user's terminal object(s).

It is up to the authentication procedure (in the case of Linux, this is done through the PAM authentication system) to create terminal objects and appropriate references. Let us consider a single terminal object per user. For each user u, $dlogin(u)$ is the set of domains the user's terminal object (denoted tu) is bound to. A user login operation is thus modeled as follows:

- $login(u) = \emptyset \gg \{tu.w\}$
- $Create(tu, \ldots) = \{R_d tu.read, R_d tu.write\}, d \in dlogin(u)$

Theorem 51 *Let G be a group of users, o an object, $M_w(o)$ the subset of its methods that change its state and $M_r(o)$ the subset of its methods that read its state. Given an initial system state s_0, the policy $G\triangleright \parallel \{o\}$ is enforced if $\forall u \in G, \forall m \in M_w(o), dlogin(u) \cap dom(o.m, s_0) = \emptyset$. $(*)$*

Proof:

Let us consider a system trace

$$[src_1 \gg dst_1, src_2 \gg dst_2, \ldots, src_n \gg dst_n]$$

1. If $o \notin dst_i$ for any $i \leq n$, then the policy is trivially enforced.
2. Let us suppose that $o \in dst_i$ for some operation i. If $tu \in src_i$, $u \in G$, then the policy is violated if $islegal(src_i \gg dst_i) = true$. Since $(*)$ implies $islegal(src_i \gg dst_i) = false$, the policy is enforced if $(*)$ holds.
3. If $o \in dst_i$ for some operation i and $tu \notin src_i$ for any $u \in G$, then the policy $G\triangleright \parallel \{o\}$ is enforced if $G\triangleright \parallel \{o'_k\}$ is enforced for any o'_k accessed in src_i or $islegal(src_i \gg dst_i) = false$.
   (a) If $G\triangleright \parallel \{o'_k\}$ is enforced for any o'_k, then $G\triangleright \parallel \{o\}$ is trivially enforced.
   (b) If $G\triangleright \parallel \{o'_k\}$ is not enforced for some o'_k, then, as a result from the *Prop* rule, $dom(src_i, s_i) \subseteq dlogin(u)$ for some $u \in G$. Thus, $(*)$ implies that $islegal(src_i \gg dst_i) = false$, and $G\triangleright \parallel \{o\}$ is thus enforced if $(*)$ holds.

Of course, practical effectiveness of this mechanism requires all considered system calls being accurately modeled in terms of information flows, including parameter passing (for instance, passing a file path parameter to the *open* system call means reading from memory).