Information Flow Control for Intrusion Detection derived from MAC Policy

Stéphane Geller, Christophe Hauser, Frédéric Tronel, Valérie Viet Triem Tong

Information Flow Control for Intrusion Detection derived from MAC Policy

Stéphane Geller, Christophe Hauser, Frédéric Tronel, Valérie Viet Triem Tong
{firstname.lastname@supelec.fr}
SUPELEC, SSIR group (EA 4039), Rennes, FRANCE

***Abstract*—Most of today's MAC implementations can be turned into *permissive* mode, where no enforcement is performed but alerts are raised instead. This behavior is very close to an anomaly IDS except that the system is configured through a MAC policy. MAC implementations such as SELinux and AppArmor come with a default policy including real life and practical rules ready to be used as is or as a basis for a custom policy. In this paper, we first propose an extension of an IDS based on information flow control. We address issues concerning programs execution and improve its expressiveness in terms of security policy. This extended model can be configured to reach a wide variety of different security goals. Particularly, it allows for information flow checking based on users and/or programs dependent policy rules. Furthermore, suspicious modification of binary programs can be detected to avoid malware execution. We also propose an algorithm for deriving an AppArmor MAC policy into an information flow policy, and thus get the advantage of having a ready to use policy offering good security. We finally show a practical example of deriving such a policy in order to configure our IDS.**

***Index Terms*—Intrusion detection, Information flow control, Mandatory Access Control**

I. Introduction

Over the past years, access control mechanisms in most operating systems have been improved. While traditional *discretionary access control* (DAC) remains widely used, previous research works on *mandatory access control* (MAC) have led to implementations in common operating systems, such as Linux, FreeBSD, MacOSX and Windows. Examples include SELinux [11], AppArmor [1], Smack [10], Tomoyo [5]. By using those mechanisms, one can finely control the operations each subject is allowed to perform on the objects of the system. A significant amount of work has been spent on defining default security policies for SELinux and Apparmor, offering rules for a lot of applications. This makes those tools valuable for system administrators, reducing the work needed to set up complex security policies in real life systems.

In the same spirit, several models of information flow control have been proposed to address one persistent weakness of access control models, namely the possibility for users or programs to indirectly and illegally access to pieces of information by collaborating with users who have legal access to it. In this article, we focus on intrusion detection and propose a model of information flow control based on mandatory access control. We perform information flow tracking and consider that the detection of an illegal information flow is an intrusion symptom. As we do not enforce the security policy, an alert is raised in case of intrusion and illegal flows are not forbidden. This behavior is known as *permissive mode* and is available with most MAC implementations. Our model is inspired from a previous model presented in [12], with three new major contributions. First, our extended model supervises programs execution. Then, policy expressiveness has been improved by using a generic tag system. It is now possible to specify an information flow policy based either on user rights, programs rights, or both at the same time. Finally, in order to determine a practical security policy for this model, we define an algorithm to derive an information flow policy from an AppArmor MAC policy. The paper is organized as follows. First, we briefly present previous works in the literature, related to MAC and information flow tracking. A formal model for representing information flows is then introduced, and our extended model is presented. Finally, we show how it is possible to derive *information flow control* from a MAC policy.

II. Background and Related Work

In the following, files, sockets and other resources are referred to as *objects* while processes are referred to as *subjects*. Any subject or object containing information is a *container of information*. *Discretionary access control (DAC)* is the most commonly used access control mechanism and is the default on UNIX based systems. Access is restricted given the *identity* and the *group* of the subject who tries to access to an object. It is said to be discretionary because subjects are allowed to transfer certain permissions to each other at their own discretion. Each subject and object has a set of *security attributes*, and any operation requires to test that it is conform to the policy. *Mandatory access control*, at the opposite of discretionary access control, is based on authorization rules (policy) enforced by the operating system. The policy is centrally controlled by a security policy administrator, and users cannot modify it. Regular users cannot declassify information, and it is then possible to verify the policy consistency against a given set of security goals [3], [7]. Therefore, a number of security mechanisms are based on MAC, and MAC is central to our approach. This aspect will be further detailed later in section V-B.

Advances in common operating systems such as Linux and FreeBSD include the introduction of generic access control frameworks, including LSM [14] (Linux Security Modules) and TrustedBSD [13]. LSM has led to several implementations, among which SELinux [11], Tomoyo [5], Smack [10] and AppArmor [1] are the most commonly used. When used in *enforcement mode*, they block illegal accesses to resources before those can be conducted. When used in *permissive mode*, their behavior is comparable to a model-based IDS.

SELinux [11] is the first security module available in Linux, and it has been designed to implement a flexible MAC mechanism called *type enforcement* (TE). With type enforcement, all subjects and objects have a *type identifier*. When accessing an object, a subject must have an authorized type of operation (read, write, etc…) with respect to the object's type, and regardless of its user identity. AppArmor [1] is a simple MAC implementations available in the Linux kernel as an alternative to SELinux. AppArmor aims at being easier to use and configure than SELinux. It is used by default by Novell in their products and comes with a predefined policy, and a set of generic definitions to ease the creation of new policies. AppArmor will be further detailed later in this paper in section V-B.

Contrary to DAC or MAC systems which ensure security in controlling access to containers of information, information flow control ensures security in preventing illegal information flows. Models of information flow control have been introduced in the eighties by Denning, Biba, Bell and Lapadula [4], [2], [9]. These models are the origin of the Multilevel Security (MLS) model. In this model, subjects and objects are labeled with a security level, which represent their sensitivity or clearance. Any information flow from lower-level containers to higher-level containers is illegal. Implementations of MLS models try to precisely observe data manipulations in order to prevent illegal information flows. Flume [8], and Histar [15] are modern implementations of information flow control. Flume is an implementation of distributed information flow control (DIFC) for Linux, acting at the OS level, and using standard OS abstractions (processes, pipes, …). In Flume, processes are confined according to a flow control policy. Histar is an operating system aiming at minimizing the amount of code that must be trusted. It provides a secure operating system using mostly untrusted user-level libraries (the only fully trusted code being the kernel). It uses Asbestos labels on six OS level object types (threads, address spaces, segments, gates, containers and devices).

Blare [17], [12] is a policy-based intrusion detection system aiming at providing fine grained information flow tracking. Content and containers are distinguished, and information flows are observed using tainting techniques. Contents are the data and containers are physical or logical data storage. Tags are associated to containers, independently describing the content as well as the policy for each container in the system. The information flow policy can either be automatically constructed from a DAC policy or adjusted by an administrator. This model has been implemented in Blare[1] and has proved to be helpful to detect attacks (see [6]). Nonetheless the model proposed in [12] is not aware of execution of programs and processes behavior, and does not take different users into account, which is necessary to ensure a fine observation of information flows. Consequently, in this model and its implementation, the authors were not able to derive a Blare policy from a MAC policy, and illegal flows between processes were ignored. In this article we propose to address this problem and we present in section III an extension of the model introduced in [12]. We also explain how we can now derive an information flow policy from a MAC policy. As an example, we give a general algorithm to derive a Blare policy from an Apparmor policy.

III. Extended Blare Model

In our extended model, we introduce three kinds of containers : "on-disk" containers such as files are called *persistent* containers (*i.e.* long-term storage), "in memory" containers are called *volatile* containers (memory pages, shared memory, IPC…). *Processes* are considered as a third kind of containers (even though these are volatile) as they correspond to active subjects as opposed to passive memory. We note $\mathcal{C}$ the set of all containers, $\mathcal{PC}$ the set of all persistent containers, $\mathcal{VC}$ the set of all volatile containers and $\mathcal{P}$ the set of all processes. Hence, $\mathcal{C} = \mathcal{PC} \cup \mathcal{VC} \cup \mathcal{P}$. As multiple processes can in fact execute the same program (*e.g.* multiple forks of the apache daemon), we also define Π, the set of all classes of processes. Two processes running the same program are in the same class. In the same manner, we distinguish code of running programs from other passive data. We attach meta-information to each element of information in the system that we want to supervise. We note $\mathcal{I}$ the set of all meta-information attached to data (*e.g.* all the personal data of a user, or the code of the *apache* daemon stored "on disk" in */usr/bin/apache*), and $\mathcal{X}$ the set of all meta-information attached to running code (as the "in memory" code of the *apache* process). In practice, elements of $\mathcal{I}$ and elements of $\mathcal{X}$ are integers. Finally we also model users and we note $\mathcal{U}$ the set of all users.

The information flow policy is divided into three independent parts. The first part ($\mathbb{P}_{\mathcal{PC}}$) defines the authorized combinations of atomic information for the set of persistent containers that we want to supervise, the second part ($\mathbb{P}_{\mathcal{U}}$) defines the allowed combinations of atomic information for the users of the system, and the third part ($\mathbb{P}_{\Pi}$) defines the authorized combinations of atomic information for the set of all the classes of processes (each class being attached to the code of a program).

Definition 1 (Information flow policy): An information flow policy is a triplet $\mathbb{P} = (\mathbb{P}_{\mathcal{PC}}, \mathbb{P}_{\mathcal{U}}, \mathbb{P}_{\Pi})$ where[2] $\mathbb{P}_{\mathcal{PC}} \subseteq \mathcal{PC} \times \wp(\mathcal{I} \cup \mathcal{X})$, $\mathbb{P}_{\mathcal{U}} \subseteq \wp(\mathcal{I} \cup \mathcal{X}) \times \mathcal{U}$ and $\mathbb{P}_{\Pi} \subseteq \Pi \times \wp(\mathcal{I} \cup \mathcal{X})$.

- A pair $(c, a) \in \mathbb{P}_{\mathcal{C}}$ expresses that the container c is allowed to contain any subset of a
- A pair $(u, a) \in \mathbb{P}_{\mathcal{U}}$ expresses that any subset of a can be read or executed by the user u.
- A pair $(\pi, a) \in \mathbb{P}_{\Pi}$ expresses that any subset of a can be read or executed by a class of processes π running the

[1] Blare is freely available at `http://www.rennes.supelec.fr/blare/`

[2] $\wp$ (A) denotes all the subset of a set A

same code.

Then, we introduce the three following notations $\mathbb{P}_{\mathcal{PC}}(c)$, $\mathbb{P}_{\mathcal{U}}(u)$ and $\mathbb{P}_{\Pi}(\pi)$ whose respective values are $\{a \in \wp(\mathcal{I} \cup \mathcal{X})|(c,a) \in \mathbb{P}_{\mathcal{PC}}\}, \{a \in \wp(\mathcal{I} \cup \mathcal{X})|(u,a) \in \mathbb{P}_{\mathcal{U}}\}, \{a \in \wp(\mathcal{I} \cup \mathcal{X})|(\pi,a) \in \mathbb{P}_{\Pi}\}$.

Thus, the definition of the information flow policy is defined for *persistent containers*, for *users*, and for *classes of processes* through sets of rules accurately stating which combinations of atomic information those can receive, and which information are authorized to mix together.

These rules are stored in a distributed fashion : we attach three tags to each container : an *information tag*, a *policy tag* and an *execute policy tag*. Dynamic detection of illegal information flows can then be performed at the container level.

The *information tag* of a container $c \in \mathcal{C}$ is a set of elements of $\mathcal{I} \cup \mathcal{X}$ describing the content of c (*i.e.* which elements of information it contains). It is updated everytime an information flow modifies the content of c. Note that because we cannot monitor all the information flows at the OS level without hardware memory tagging [16], the information tag contains an over-approximation of the actual content.

The *policy tag* of a container $c \in \mathcal{C}$ is a set of subsets of $\mathcal{I} \cup \mathcal{X}$. It defines all the possible combinations of information that are allowed to flow towards the container. This tag is updated when a modification of the policy affects the container c (in practice, we do not change the policy at runtime).

The *execute policy tag* of a container $c \in \mathcal{C}$ is a set of subsets of $\mathcal{I} \cup \mathcal{X}$. It defines all the possible combinations of information that processes running the content of c as code are allowed to read or execute. Thus, it is attached to the code of programs, and as the information tag, it is updated everytime an information flow modifies the content of c (*i.e.* changing the code of a program changes the policy for running it). Note that it is independent of users rights, those are taken into account at execution time (see IV-2).

IV. Information flow tracking in a Live system

We denote $0, 1, 2, ..., n, ...$ the states of the system and we note t_i the transition between the states i and $i+1$. These transitions correspond to operations requiring an update of the tags content (fork, execution, creation of objects and information flows).

In the following, we will use the functions $itag$, $ptag$ and $xptag$ that respectively associate an information tag, a policy tag and an execute policy tag to a container.

- $itag : \mathcal{C} \mapsto \wp(\mathcal{I} \cup \mathcal{X})$, $itag_i(c)$ is the *information tag* attached to the container c at state i
- $ptag : \mathcal{C} \mapsto \wp(\wp(\mathcal{I} \cup \mathcal{X}))$, $ptag_i(c)$ is the *policy tag* attached to the container c at state i
- $xptag : \mathcal{PC} \mapsto \wp(\wp(\mathcal{I} \cup \mathcal{X}))$, $xptag_i(c)$ is the *execute policy tag* attached to the container c at state i

The relation $\sqcap$ is defined on the sets of sets as follows : $A \sqcap B = \{a \cap b | a \in A, b \in B\}$. We also introduce the notations $\top$ and $\bot$ which are respectively the set of all sets of tags and the set containing the empty set.

An information flow towards a container c is legal if and only if the new content of c (characterized by $itags(c)$) is authorized into c, *i.e.* it appears in $ptags(c)$.

Definition 2 (Legality of an information flow): An information flow towards a container c happening during the transition t_i is legal iff $\exists p \in ptag(c)_{i+1}$ such that $itag(c)_{i+1} \subseteq p$. We note $itag(c)_{i+1} \preccurlyeq ptag(c)_{i+1}$ when this condition is verified.

In the following subsections, we are going to detail accurately the propagation of the tags for each operation.

1) $fork$: When a process p forks, a clone q is created. We also clone all of its tags.

2) execution of a program: recall that we consider *passive* data and running code differently. A data is considered as running code once it is executed by a process through an $exec()$ system call. We note $Run : \mathcal{I} \to \mathcal{X}$, where $Run(d)$ characterizes the running code out of the execution of a data d at transition t_i. After an $exec()$ call, the three tags of the calling process p running the object o on behalf of u are initialized as follows. Its *information tag* becomes :

$$itag(p)_{i+1} := \bigcup_{k \in itag(o)_i \setminus \mathcal{X}} \{Run(k)\}$$

Note that elements of $\mathcal{X}$ from o are discarded, because these meta-information are only used to compute the legality of write and append operations (see IV-6).

Its *execute policy tag* is initialized to the execute policy tag of o.

$$xptag(p)_{i+1} := xptag_i(o)$$

Its policy tag is computed from its execute policy tag and from the legal combinations of information for the user running it :

$$ptag(p)_{i+1} = xptag_i(o) \sqcap \mathbb{P}_{\mathcal{U}}(u)\}$$

3) Persistent object creation: When a process p creates a new persistent object o on behalf of u, the new object receives an empty information tag.

$$itag(o)_{i+1} = \emptyset$$

We associate a policy tag to the new object as follows. The authorized flows towards the created object are the flows composed of atomic informations that are legal for u (*i.e.* the policy states that u is allowed to access this information).

$$ptag(o)_{i+1} = \mathbb{P}_{\mathcal{U}}(u)\}$$

An execute policy tag is also attached to the new object, and it is set to $\top$ by default : there is no restrictions on code execution as it contains no (executable) information.

$$xptag(o)_{i+1} = \top$$

4) Volatile object creation: When a new volatile object is created, it is assigned an empty information tag as it contains no information yet. It is then updated appropriately when further information flows occur (as it will be detailed in the following).

$$itag(o)_{i+1} = \emptyset$$

In the same manner, the *execute policy tag* is initialized to $\top$.

$$xptag(o)_{i+1} = \top$$

In this model, we consider that anything is allowed to flow into volatile containers. The legality of information flows involving volatile containers depend on the processes operating on it, *i.e.* processes can only access to information that matches their *policy tag*. Thus, the policy tag of volatile containers is initialized to $\top$ so that anything can flow into it.

$$ptag(o)_{i+1} = \top$$

Using these tags, we are able to perform dynamic detection of illegal information flows by checking if $itag(c) \preccurlyeq ptag(c)$ stands everytime a flow occurs.

We classify information flows as *read_like*, *write_like* and *append_like*. When a process p reads from an object o, there is a *read_like* information flow between the containers o and p. In the same way, when a process p replaces the content of an object o, there is a *write_like* information flow. Finally, if the process writes without erasing the existing content of the object, it is an *append_like* information flow. When an information flow occurs, we make an over-approximation of the actual flow, considering all the data that might have flown (*i.e.* all the information tag from the source). It behaves differently whether a $read\_like$, $write\_like$ or $append\_like$ information flow happends, and tags are updated accordingly for the process and the object.

5) $read\_like$ operations: When a $read\_like$ flow occurs on an object o by a process p, we update the *information tag* and the *execute policy tag* of p. Note that, as in the case of the execution of a program, elements of $\mathcal{X}$ from the *information tag* of o are discarded (process execution history is not saved in our model).

$$\begin{array}{lcl} itag(p)_{i+1} & := & itag(p)_i \cup (itag(o)_i \setminus \mathcal{X}) \\ xptag(p)_{i+1} & := & xptag(p)_i \sqcap (xptag(o)_i) \end{array}$$

6) $write\_like$ operations: If the process p overwrites the content of the object o, we simply replace the information tag and the execute policy tag of the object by those of the process. Note that the elements of $\mathcal{X}$ in the information tag are used to check the legality of this *write_like* operation performed by this particular code being executed by p.

$$\begin{array}{lcl} itag(o)_{i+1} & := & itag(p)_i \\ xptag(o)_{i+1} & := & xptag(p)_i \end{array}$$

7) $append\_like$ operations: A process p can also append pieces of information to an object o. In this case, the new information tag of o is the concatenation of both information tags. As for *write like* operations, the elements of $\mathcal{X}$ are used to check the legality of the operation. The *execute policy tag* is updated so as to match both the *execute policy tags* of the process and of the object.

$$\begin{array}{lcl} itag(o)_{i+1} & := & itag(o)_i \cup itag(p)_i \\ xptag(o)_{i+1} & := & xptag(p)_i \sqcap xptag(o)_i \end{array}$$

The figures 1 and 2 summarize the propagations of the tags in the different cases mentionned above.

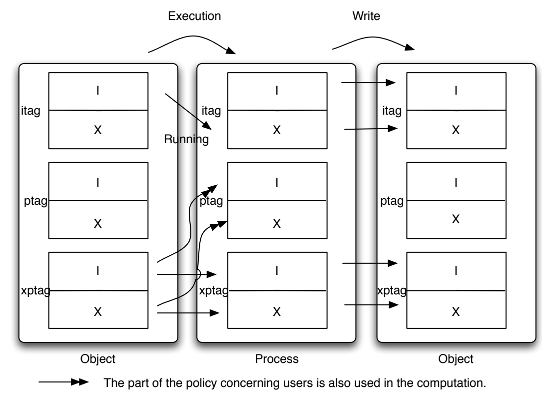

Fig. 1. Diagram with an execution and a write-like operation

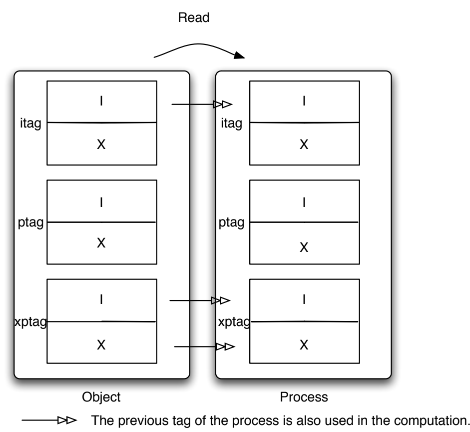

Fig. 2. Diagram with a read-like operation

V. Setting up the initial tags

In previous works, the authors have presented how to automatically derive an information flow policy starting from a DAC policy [6] or how administrators can compute an ad-hoc policy[12]. In this section, we will present an algorithm to derive a Blare policy from a MAC policy.

A. Initialization of tags starting from an existing policy $\mathbb{P} = (\mathbb{P}_{\mathcal{PC}}, \mathbb{P}_{\mathcal{U}}, \mathbb{P}_{\Pi})$

At initialization time, *i.e.* the initial state of the system, before we start to track information flows, persistent containers are attached an information tag, a policy tag, and an execute policy tag matching the policy.

1) Initial information tag : a unique meta-information describing the initial content of the container is stored into its information tags. This initial information is considered as being atomic (atomic information are the smallest pieces of information that we are able to distinguish in the system).

2) Initial policy tag: for any persistent container c, the associated *policy tag* is the set of elements in the policy regarding this container.

$$\forall c \in \mathcal{PC}, ptag_0(c) := \mathbb{P}_{\mathcal{PC}}(c)$$

3) Initial execute policy tag: for any persistent container c, the associated *execute policy tag* is the set of elements in the policy regarding the execution of the content of c. We note $Pclass : \mathcal{PC} \rightarrow \Pi$ the relation that associates a class of processes to any persistent container c. Any process executing the content of c belongs to this class.

$$xptag_0(c) := \mathbb{P}_{\Pi}(Pclass(c))$$

If the object does not contain executable code, the corresponding class will not appear in $\mathcal{P}_{\Pi}$ and the $xptag$ will be empty.

B. Deriving an information flow policy from an AppArmor Policy

In the following, an information flow policy (centered on programs) is derived from an AppArmor MAC policy. Such a policy does not specify rules based on users, and thus ($\mathbb{P}_{\mathcal{U}}$) is empty. As Blare monitors information flows, we do not take into consideration access control rules that would not be related to any flow transition. To derive a Blare policy from a set of AppArmor profiles, the following is performed : for each statement in the AppArmor policy, we check whether such a statement is related to a potential information flow, and transform it into a Blare statement if it does. The ability to derive such a Blare policy will be useful for future works in comparing different models in terms of intrusion detection, as each model would be configured with a common security policy.

1) AppArmor: In an AppArmor profile, the permission granted to a program π over a resource o can be one of the following : (r,w,l,m,ix,px,Px,ux,Ux).

r	**read** (executing also needs this permission)
w	**write**
a	**append**
l	**link mode** (mediates access to symlinks and hardlinks)
m	**allow** executables mapping (mmap)
ix	**inherit** execute mode (the resource inherits the current profile, even if a profile already exists for this resource)
px	**discrete profile** execute mode (if no profile is defined for the resource, execution is denied)
Px	**scrub** the environment (same as px but will use kernel's unsafe exec routines : tells glibc to clean the environment before executing the resource. It helps protect against e.g. LD_PRELOAD abuse)
ux	**unconstrained** execute mode (no profile is needed)
Ux	**unconstrained/scrub** the environment

Fig. 3. AppArmor access modes

AppArmor profiles also constrain access to network resources and POSIX capabilities. However, these are access control rules and aren't taken into account in this paper. Instead, possible information flows related to those accesses are captured at another level (*i.e.* actual illegal flow occurs). Such rules would add false positives and are discarded in our derivation.

Definition 3: An AppArmor policy $\mathbb{P}$ is a set of profiles. A profile $p \in \mathbb{P}$ is a set of rules of the form (o, α) where o is an object and α is a permission. All these rules confine a given program $\pi \in \Pi$. Such a profile is defined as : $(\pi, \{(o_1, \alpha_1), \ldots, (o_n, \alpha_n)\})$

For each AppArmor policy statement, if it allows a potential flow between a subject and an object, such as defined in section, we update the Blare tag system accordingly.

2) Algorithm: The following algorithm transforms an AppArmor policy (a set of profiles) into an expression of a Blare policy (set of policy labels on containers). Let P be the set of all the AppArmor profiles in the policy. For any profile $p \in P$, $p.container$ is the container associated to the binary program constrained by p, $p.canread()$ is the list of files on which a $read\_like$ access is authorized, $p.canexec()$ is the list of executable allowed to be executed, and $p.canwrite()$ is the list of paths where it is allowed to write. TOP represents the set of all atomic information tags in the system (it corresponds to $\top$), $inherit(p) : bool$ returns $true$ if the profile p inherits from its parent's profile and $false$ otherwise. $unconstrained(p) : bool$ returns $true$ if the associated program (subject) is unconstrained and $false$ if not. $Run(I)$ is defined in section III.

```
function tag(P)
for each p in P ; do
    class = Run(itag(p.container))
    if unconstrained(p)
        data = TOP
        code = TOP
    else
        for r in p.canread() ; do
            data += itag(r)
        end
        for x in p.canexec() ; do
            code += Run(x)
        end
    end
    xptag(p.container) = data + code
    for w in p.canwrite() ; do
        w.ptag += data + class
    end
end
end
```

VI. Example

The following is an example of intrusion detected by this model, when configured with an information flow policy derived from an AppArmor policy. Here, the security is centered on programs, with no user dependent policy rules. Consider the following AppArmor policy example, where two programs are confined : $apache$ and $ftpd$. Both own files that the other is not allowed to read. We consider AppArmor being setup in *permissive mode*, and we compare its behavior to our IDS in terms of detection potential.

```
{/usr/bin/apache,
    {(/etc/apache2.conf, w),
    (/etc/apache2.conf, r),
    (/www/index.php,r),  (/usr/bin/ftpd, px)}
}
{/usr/bin/ftpd,
    {(/etc/ftpd.conf,w),  (/etc/ftpd.conf,r),
    (/home/ftpd/data,w)}
}
```

Using the previously introduced algorithm, we can derive a Blare policy and compute its expression on the tag system (*the function* $Run()$ *is written* $R()$ *in the following table*):

path	itag	ptag	xptag
/usr/bin/apache	$\{i_1\}$	$\{i_1\}$	$\{R(i_1), R(i_2), i_3, i_6\}$
/usr/bin/ftpd	$\{i_2\}$	$\{i_2\}$	$\{R(i_2), i_4\}$
/etc/apache2.conf	$\{i_3\}$	$\{R(i_1), i_3, i_6\}$	$\top$
/etc/ftpd.conf	$\{i_4\}$	$\{R(i_2), i_4\}$	$\top$
/home/ftpd/data	$\{i_5\}$	$\{R(i_2), i_4, i_5\}$	$\top$
/www/index.php	$\{i_6\}$	$\{R(i_1), i_3, i_6\}$	$\top$

Now, the following execution sequence takes place (see figure 4). The apache process first reads its configuration file $/etc/apache2.conf$. Then it reads and interprets $/www/index.php$, containing a security flaw. Arbitrary code is injected and executed through apache. It introduces a malware in the binary code of $/usr/bin/ftpd$.

In this first part of the execution, the process running apache is not expected to write into $/usr/bin/ftpd$: the policy tag of this container is not allowed to receive information by a process running apache. Furthermore, the information apache previously read (and figuring in its information tag) does not belong to the policy tag of $/usr/bin/ftpd$. This would trigger an alert with both AppArmor (configured in *permissive* mode) and Blare.

Then, $apache$ runs the modified $ftpd$. The process running $apache$ is allowed to execute $ftpd$ in the security policy, hence AppArmor would allow this execution. But here, the information tag of $ftpd$ has been modified when the arbitrary code was written into it, and meta-information have been added to it. Those new meta-information do not figure in the policy tag of the process running $apache$, thus it is not authorized to run $ftpd$ anymore, and this would trigger a second alert for illegal code execution with Blare. This quite simple example reveals one of the major goals of Blare : the security administrator can specify a fine-grained information flow policy including processes behavior. Many real life viruses would trigger alerts in this model as soon as the code of a process is changed or confidential information is moved.

VII. Conclusion

In this paper we have presented a model of intrusion detection based on an information flow policy, dynamically checking that it is respected. The policy specifies which information may be combined together and which information the containers are allowed to contain. This model offers high expressiveness since we are able to assign meta-information to any data in the system and to constrain the behavior of programs when those data are involved. The policy expresses restrictions on access to information regardless of where it is located in the system by using a tag system associating meta-information to information containers. We explain how we maintain tags when information flows occur and how we can check if the policy is respected. A central concept of this model is the execution of programs. This model performs dynamic checking at execution time, and is able to detect executions of illegal code or illegal flows of information.

Today's MAC implementations in the Linux kernel come with extensive default security policies. It is possible to set up a policy for the model we propose out of an existing MAC policy. We show how to derive a Blare information flow policy from an AppArmor MAC policy, and give an example of practical use.

In our future works we will focus on an implementation of this model in the Linux kernel as a LSM module. We also aim to further enrich this model on two main aspects. First, a user owning information will be able to declassify it. Second, we will provide a high-level language to specify an information flow policy for Blare.

References

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

[2] K. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR 76-372, MITRE Co., April 1977.

[3] Yi-Ming Chen and Yung-Wei Kao. Information flow query and verification for security policy of security-enhanced linux. In *Proceedings of IWSEC*, pages 389–404, 2006.

[4] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[5] Toshiharu Harada, Takashi Horie, and Kazuo Tanaka. Access policy generation system based on process execution history. *Network Security Forum*, 2003.

[6] Guillaume Hiet, Valerie Viet Triem Tong, Ludovic Me, and Benjamin Morin. Policy-based intrusion detection in web applications by monitoring java information flows. *Int. J. Inf. Comput. Secur.*, 3(3/4):265–279, 2009.

[7] Amy L. Herzog Joshua D. Guttman and John D. Ramsdell. Information flow in operating systems : Eager formal methods. *Workshop on Issues on the Theory of Security (WITS)*, 2003.

[8] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard os abstractions. In *Proceedings of the 21st Symposium on Operating Systems Principles*, Stevenson, WA, October 2007.

[9] Leonard J. LaPadula and D. Elliott Bell. Secure computer systems: A mathematical model. MTR-2547 (ESD-TR-73-278-II) Vol. 2, MITRE Corp., Bedford, may 1973.

[10] Casey Schaufler. "the simplified mandatory access control kernel". *"White paper"*.

[11] Chris Vance Stephen Smalley. Implementing selinux as a linux security module. Technical report, NAI Labs, 2002.

[12] Valérie Viet Triem Tong, Andrew Clark, and Ludovic Mé. Specifying and enforcing a fine-grained information flow policy: Model and experiments. In *Journal of Wireless Mobile Networks, Ubiquitous Computing and Dependable Applications*, 2010.

[13] Robert Watson, Brian Feldman, Adam Migus, and Chris Vance. The trustedbsd mac framework. In *Proceedings of DISCEX (2)*, pages 13–, 2003.

state	action	itag(π_1)	itag(π_2)	itag(/usr/bin/ftpd)	AppArmor Alerts	Blare Alerts
0	$\pi_1 = exec(/usr/bin/apache)$	$R(i_1)$		i_2		
1	(apache,/etc/apache2.conf,r)	$R(i_1), i_3$	$\emptyset$	i_2		
2	(apache,/www/index.php,r)	$R(i_1), i_3, i_6$	$\emptyset$	i_2	X	X
3	(apache,/usr/bin/ftpd,w)	$R(i_1), i_3, i_6$	$\emptyset$	$i_2, i_3, i_6, R(i_1)$		
4	(apache,/usr/bin/ftpd,x)	$R(i_1), i_3, i_6$	$R(i_2), R(i_3), R(i_6)$	$i_2, i_3, i_6, R(i_1)$		X
	$\wedge\pi_2 = exec(ftpd)$			$i_2, i_3, i_6, R(i_1)$		
5	(ftpd,/home/ftpd/data,w)	$\emptyset$	$R(i_2), R(i_3), R(i_6)$	$i_2, i_3, i_6, R(i_1)$		

Fig. 4. Execution sequence

[14] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *USENIX Security Symposium*, pages 17–31, 2002.

[15] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 263–278, Berkeley, CA, USA, 2006. USENIX Association.

[16] Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. Hardware enforcement of application security policies using tagged memory.

[17] Jacob Zimmermann, Ludovic Mé, and Christophe Bidan. Introducing reference flow control for detecting intrusion symptoms at the os level. In Andreas Wespi, Giovanni Vigna, and Luca Deri, editors, *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID'2002)*, volume 2516 of *Lecture Notes in Computer Science*, pages 292–306. Springer, 2002.