A Distributed Intrusion Detection Approach for Secure Software Architecture

Paola Inverardi and Leonardo Mostarda

Dip. di Informatica, Università di L'Aquila,
Coppito 67100, L'Aquila, Italy
{inverard, mostarda}@di.univaq.it

Abstract. This paper illustrates an approach to add security policies to a component-based system. We consider black-box-components-based applications, where each component can run concurrently in a different domain. The problem we want to face is to detect at run time that a component might start interacting with the other components in an anomalous way trying to subvert the application. This problem cannot be identified statically because we must take into account the fact that a component can be modified for malicious purposes at run time once deployed. We propose a specification-based approach to detect intrusions at architectural level. The approach is decentralized, that is given a global policy for the whole system, i.e. a set of admissible behaviors, we automatically generate a monitoring filter for each component that looks at local information of interest. Filters then suitably communicate in order to carry on cooperatively the validation of the global policy. With respect to centralized monitors, this approach increases performance, security and reliability and allows the supervision of complex applications where no centralized point of information flow exists or can be introduced.

1 Introduction

This paper describes a specification-based approach to detect intrusions at architectural level. We assume to have a black-box-components-based application where all components run concurrently and interact each other exchanging services. At architectural level, we speak about intrusions when legitimate components perform unauthorized actions; e.g., a rogue client can be built by an attacker that uses his authorizations to subvert the application.

Intrusions at architectural level may be detected by using static approaches, e.g. by using model checking techniques. However, components can be dynamically modified for malicious purposes and the statically validated properties can be violated. Run-time tools monitoring for evidence of intrusions can provide a solution to these problems. Nowadays, several run-time monitors are available: they are referred to as Intrusion Detection Systems (IDSs), and their main task is to analyze the observable behaviors of a system in order to recognize malicious behaviors. The effectiveness of an IDS is usually measured in terms of: **detec-**

tion efficiency: the amount of intrusions that are correctly recognized; **false alarms rate:** the amount of correct behaviors detected as intrusions.

There are three main types of IDSs detection techniques: *misuse*, *anomaly* and *specification-based. Misuse detection systems* [1] are explicitly programmed to recognize well-known attacks. These systems recognize intrusions by matching the pattern of observed data with the set of predefined (intrusion) signatures. They can perform focused analysis thus having a low false alarms rate. However, they cannot detect unknown types of attacks, since it is not possible to specify a signature for a still unknown vulnerability. Furthermore, IDS complexity grows with the number of well-known attacks. *Anomaly detection systems* assume that an attack will cause deviation from normal behaviors, thus detection can be done by comparing actual activities with known correct behaviors. Different approaches have been used to model normal behaviors: statistics-based [2], rule-based [3], immunology-based [4]. The advantage of this kind of systems is the ability of detecting novel attacks and the fact that it is not required specific knowledge about correct information flows. However, it is not easy to define what is a normal behavior, to set up anomaly thresholds, to have a good detection efficiency and moreover not all intrusions need to produce an anomalous behavior. *Specification-based systems* [5] use some kind of formal specification to describe the correct behaviors of the system. The detection of violations involves monitoring deviations from the formal specification, rather than matching specific well-know attacks. The advantage of this approach is the ability to detect previously unknown attacks at the expense of providing a formal specification of correct information flows.

Besides the problems mentioned above, all the described approaches when implemented suffer a number of further problems:

- the monitoring tools are subject to *tampering*, since they are software that can be itself target of attacks. [1].
- correct monitoring of points where there is a high level of information flow may be problematic (loss of data);
- in complex systems no *centralized point* of information flow can exist, so distributed solutions are needed;
- IDSs have to be *scalable* with respect to the number of components to be monitored, i.e. augmenting the components number must not result in an increased execution response time of the monitoring tool.

This paper presents a specification-based intrusion detection approach to face intrusions at architectural level. The application to be monitored is composed by black-box components that run concurrently in different domains. We assume to know the services required/offered by each component that is name / formal parameters / returned values, (i.e. the component interface), the topology of the application in terms of potential interactions among components (the

[1] Tamper is a term to indicate *"any act that results in the improper alteration of the application code." [6]*

connectors) and the specification of the acceptable behaviors the system has to comply with. The latter is given by means of a language that defines the correct behaviors of the system.

We propose a polynomial algorithm that combines the specification and the architectural information in order to distribute the specification checking on the component where the information to be supervised flows. More specifically, this algorithm builds a set of local wrappers (filters), one for each component. Wrappers locally monitor the component behavior and communicate with each other in a peer-to-peer fashion to discover attacks scattered over several components. Moreover, in order to address the attacks to the security measures filters are able to detect filter *tampering* [6]. In the remaining of the paper we will use the terms wrapper/filter interchangeably.

We choose a specification-based monitoring as opposed to anomaly base detection since it permits to detect unknown attacks and reduces the false alarm rate. Moreover, since our approach to intrusion detection is distributed it allows the monitoring of complex applications where there is no central point of information flow. However, the distributed approach also brings obvious overheads in terms of message exchange.

The contribution of this paper is in proposing a way to automatically generate a set of local filters (one for each component) that detect components dynamic misbehavior. Our approach also permits to build a tamper resistant IDS [6], i.e. an IDS which is resistant to modification and observation. The approach we propose is architectural since it relies on architectural information about components (interface) and connections (topology).

2 Related Work

Most of the advanced tools for intrusion detection send distributed data to a centralized unit that relates them in order to detect violations. This centralized design poses problems of: *scalability*, *fault tolerance* and *security.* Attacks to and faults of the central unit can deactivate the monitoring of the distributed system. An increasing number of sensors that forward data can cause loss of information or increase the monitoring system reaction time.

Systems like NetSTAT [7], and Emerald [8], and GrIDS [9] try to solve the above problems by means of a layered structure. Data are locally processed and events that are part of distributed attacks are forwarded to an higher entity. Although such systems try to address the problem of scalability, nodes close to the root of the hierarchy can still be overloaded and they represent a single point of failure or vulnerability.

CSM[10] faces these problems by means of a peer-to-peer design. It has no centralized unit thus data are exchanged among peers to correlate them.

All the above mentioned tools recognize well-known kinds of attacks by means of intrusion patterns (misuse based). Patterns are usually defined at networking level and they are monitored by distributed sensors that sniff traffic. Such tools have dedicated hosts and a management network separated from the one used by the application data. This choice is dictated by the fact that monitoring systems are themselves software that can be *tampered*.

Our approach aims at lifting monitoring technology from the operating system and network level to the application architecture level. We choose a peer-to-peer design to address *fault tolerance* and *scalability*. Our wrappers reside on the same host of the component code; thus we cannot rely on separated host/network to avoid their *tampering*. However, in our monitoring tool filters that interact with a tampered filter can detect its anomalous behavior.

The idea to monitor distributed systems at application level is not new. The ORA organization[4] monitors applications in anomaly based fashion. They characterize the normal behavior of the components interaction by using an immunology approach. While they are able to detect previously unknown attacks, not all intrusions deviate from normal behavior. We choose a specification-based monitoring to overcome such problem and to reduce the false alarm rate.

The DIANA tool[11] uses a specification-based approach to monitor distributed programs in a decentralized way. Safety properties are specified on each distributed process by means of a variant of a past time linear temporal logic. Formula related to a particular process can refer to remote states of other processes by using particular operators. Remote state information is delivered only when there is an explicit interaction with that process. Therefore a process locally computes a formula by using the information of remote states it is aware of. This logic seems not adequate to express security behaviors, since it may well happen to monitor applications (part of) whose components do not explicitly interact but their local states contribute to discover an attack.

Ponder [12] is an object oriented and declarative language mainly adopted for Object-Oriented distributed systems. A set of agents deployed at different hosts allow the monitoring. This language is specifically tailored to define roles, subjects domains and policies. However agents could overload the host to be monitored and they can be target of *tampering*. Breach on the security measures can become a means to attack the distributed application to be monitored.

Our monitoring language defines the security policy and it is tailored to express distributed correlation of information at architectural level. We deal only with observable messages exchanged at architectural level. Given the policy to monitor, filters are automatically generated and deployed.

3 Enforcement Mechanisms and IDSs Specification Based

Earlier IDSs were only involved in monitoring activities and analyzing log files. Today's IDSs embed reactive utilities that are undertaken when an attack is detected. For instance an IDS can react to an attack by terminating the session, blocking or shunning the traffic, creating session log files or restricting the accesses.

The time required to detect an attack and the time to react to such attack are relevant parameters that characterize the effectiveness of an IDS. Ideally an attack should be detected when it is in progress, this would allow either to avoid the attack or to have a faster recovery. In the worst case an attack that is terminated can be unrecoverable and important information can be lost.

Our specification based IDS captures every message exchanged among the components of the system to be monitored. It verifies that the messages comply with the formal specification, then it releases the messages. An attack is detected when the IDS finds a mismatch between the observed messages and the formal specification. In this case it reacts with the following default actions. The *log reaction* in which all activities related to the attack are logged. The *enforcing reaction* in which IDS releases every captured sequence verifying the formal specification. In other words, if our IDS captures a sequence of messages that verify the specification, then it deliveries the messages to the related components. Therefore our monitoring system can be also seen as an enforcement mechanism (EM).

As defined in [13] enforcement mechanisms compare a formal specification with the system steps. When there is a violation of the formal specification an EM can either terminate the system execution or replace an unacceptable execution step with an acceptable one. Any EM is assumed to be isolated from the system and any input to the system must be forwarded to it.

However in a system composed by black-box components running in different domains an EM might not have the right to terminate the system execution. Therefore our enforcing mechanism replaces an unacceptable behavior with an acceptable one.

We use the formal specification introduced in [13] to build our EM on the basis of an automaton that specifies the policies to be enforced. Our contribution is the algorithm to automatically distribute the EM on each component that composes the system. The distribution phase creates one filter for each component. Each filter embed only the part of control related to the local information of interest. The use of the automaton is twofold: on one side it permits to reduce the overhead of messages exchanged among the filters. On the other side it allows an acceptable tradeoff between detection time and expressiveness of the language used to describe the security policies. As described in [13], relevant security properties can be described by means of security automata. In the following section we categorize these security policies by means of definitions and examples.

4 Violations at Architectural Level

Systems must embed security features to resist to attacks. However, nothing is perfect. Even the best protected system must be monitored to detect security violations. In a component based system, we characterize attacks as: *interface attacks* and *trace attacks*. *Interface attacks* are carried out by requesting a service with bad formatted inputs: anomalous inputs can produce a buffer overflow or code injections, so that attackers can gain unauthorized accesses. *Traces attacks* aim at subvert the correct communication among components. In the following we list some subcases of *traces attacks*. *Sequence attacks* are related to the order in which messages are exchanged among components. For instance, a component may access a service before performing authentication, or a component may request exclusive access to a data base component without releasing it before exiting. *Synchronization attacks* are performed by synchronizing components in a suspicious way. This is the case in which two components require a write service offered by a data base component. This could lead the system in an erroneous state in which one of the component could not have access to the system any more. *Coordination attacks* concern an anomalous cooperation of a component to reach some global goal. For instance, in a collaborative writing system, a component cannot cooperate with another component in order to read or write a different piece of file. *Distributed attacks* are scattered over several sources. These attacks look innocents when local-component traffic is considered, but they result in a violation when data are related. An example of this type of attacks is given by a chain of requests among components.

We detect violations in a component based application by checking that the system behaviors match a well defined policy. Here, a *policy* is a set of rules that dynamically regulate the behavior of a system neither changing the components code nor requiring their cooperation. In particular, security policies define what actions are permitted or not permitted, for what or for whom, and under what conditions. Policies can define correct communications among components, access and protection to components, authentication, monitoring of the responses and correct use of services. To define policies, we provide an ad-hoc language based on state machines. We also provide an algorithm to automatically generate a set of wrappers, starting from the given policy and the system architecture. Our wrappers are distributed one for each component and embed the part of policies that define the component local interactions. Although wrapper can implement confidentiality, we mainly focus on policies related to communications, access, correct use of services, monitoring of the responses and protection of components.

5 The Model of the System

At the architectural level, a system is viewed as composed by a set of components communicating with each other. We consider distributed-black-box components running concurrently and communicating either asynchronously and/or synchronously. We know that messages exchanged among distributed components can always be totally ordered [14], thus, a global trace of the system can be obtained. In this Section, we will give the basic definitions which our frame relies on. In all these definitions, we will assume that a global system clock exists. However, this assumption is needed only for modelling purposes, and it will be relaxed in Section 6, where we describe how our filters distribution algorithm works.

We focus on architectural system traces, i.e. on strings containing all messages observed at architectural level. A message encodes information about the type of communication, i.e. a request or a reply, the kind of service and its parameters and the (returned) data. We also assume, without loss of generality, that all messages are uniquely identified. Two requests of the same service from two different components originate two distinct messages.

We introduce some definitions that will be used in the following.

Definition 1. *Let T be a string $m_1 m_2 m_3 \ldots m_i m_{i+1} \ldots$. T is an* architectural system trace *if the following properties hold:*

- *$\forall m_k \in T$ m_k codifies a service request to a component or a valid answer to some request.*
- *if $i < k$ then $t(m_i) \leq t(m_k)$, where $t(m)$ denotes the global system time at which the message occurred.*

Definition 2. *A sequence of messages $m_{l_1} m_{l_2} m_{l_3} \ldots m_{l_i} m_{l_{i+1}} \ldots$ is a* sub-trace *of some system trace $m_1 m_2 m_3 \ldots m_i m_{i+1} \ldots$ if $l_1 l_2 l_3 \ldots l_i l_{i+1} \ldots$ is a subsequence of* $1, 2, 3, \ldots, i, i+1, \ldots$

Definition 3. *Two subtraces $s_1^a s_2^a s_3^a \ldots s_i^a s_{i+1}^a \ldots$ and $s_1^b s_2^b s_3^b \ldots s_k^b s_{k+1}^b \ldots$ are said to be* distinct *if and only if $\forall i, j$ $s_i^a \neq s_j^b$.*

Definition 4. *Given two distinct subtraces $T1$: $s_1^a s_2^a s_3^a \ldots s_i^a s_{i+1}^a \ldots$ and $T2$: $s_1^b s_2^b$ $s_3^b \ldots s_k^b s_{k+1}^b \ldots$ of T,* a merge trace $T1 \oplus T2$ *is a subtrace of T defined by $s_1 s_2 s_3 \ldots s_j s_{j+1} \ldots$ where:*

- *$s_r \in T1 \oplus T2$ if and only if $s_r \in T1$ or $s_r \in T2$*
- *for each s_i and $s_j \in T1 \cup T2$ if $t(s_i) < t(s_j)$ then s_i appears before s_j in $T1 \oplus T2$*

The definition of subtrace permits to define for each component C a component local trace that is all messages locally sent/received by a component.

Definition 5. *Let C be a component and T an architectural system trace. $T_C = m_1^c\ m_2^c m_3^c \ldots m_k^c m_{k+1}^c \ldots$ is a* component local trace of C *if it is a subtrace of T and each m_i^c is a message that codifies either a request or a provided service of the component C.*

In our model, the architectural system trace is produced by messages exchanged among all the components. Each running component C_i in the system defines a, local to the component, subtrace T_{C_i}. These sets of local traces constitute a partition of the architectural-system trace. In other words, if T: $m_1 m_2 m_3 \ldots m_i m_{i+1} \ldots$ is an architectural-system trace, $\{T_{C_1}, T_{C_2}, T_{C_3}, \ldots, T_{C_n}\}$ the sets of local traces observed by the components of the system, then $\bigcap_{1 \leq i \leq n} T_{C_i} = \emptyset$ and the merge of T_{C_i} is equal to T.

Our purpose is to analyze the system architectural trace T produced at run-time to detect if T contains subtraces that violate the defined policies.

We provide an ad-hoc language based on state machines to specify policies (see [15]). It allows the definition of constrains on the input data of the services, on the ordering of the messages, on the synchronization among requests and on the relations among messages scattered over several components(see Section 4). The defined policy can establish when a component can access a service and it permits to monitor the response of a component. In this paper we do not show syntax and semantics of the language [15].

Our language permits to define the following automaton [13].

Definition 6. *A secure automaton is 4-tuple $A = (Q, q_0, I, \delta)$ where: Q is a finite set of automaton states, $q_0 \in Q$ the initial state, I is a finite set of input symbols and $\delta(Q \times I) \rightarrow Q$ is a transition function.*

Definition 7. *A secure automaton $A = (Q, q_0, I, \delta)$ parses a sequence $T = m_1 m_2 m_3 \ldots m_i m_{i+1} \ldots$ one symbol at a time from left to right. Let $q_{i-1} \in Q$ be the current state of A and let m_i be the next symbol to read. A* accepts *m_i if there exists a transition rule $\delta(q_{i-1}, m_i)$.*

Definition 8. *Let $A = (Q, q_0, I, \delta)$ be a secure automaton and $T = m_1 m_2 m_3 \ldots m_i m_{i+1} \ldots$ be a sequence of symbols in I. Let q_0 be the starting state of A and m_1 be the first symbol to read. A* accepts the sequence *T if for each current state q_{i-1} and next symbol m_i, A accepts m_i. $q_i = \delta(q_{i-1}, m_i)$ is the new state of A and m_{i+1} the next symbol to read.*

Definition 9. *The language $\ell(A)$ recognized by $A = (Q, q_0, I, \delta)$ is composed by all sequences of symbols in I accepted by A.*

This acceptance criterion permits to recognize finite and infinite sequences of symbols(see [13]).

In the context of component based systems, the 4-tuple of the secure automaton is constrained by the following rules. I is a finite set of symbols that represent messages at architectural level. Messages are of the form: $!s$ denoting outgoing message and $?s$ incoming message from/to a component, respectively. δ represents the policy that defines the correct messages exchange among components. We call such secure automaton: *Global Secure Automaton.* Global, since the alphabet I is a subset of all messages exchanged among components.

In Figure 1, we show a component-based system composed of three different types of components. A database component $C1$ can accept a login event which corresponds to an authentication service, encoded as $?login$. The message $!fail$ models a failure answer that $C1$ can send to a non-authorized client while the

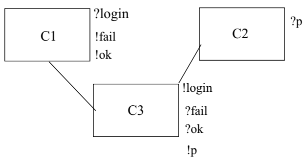

Fig. 1. Architectural view of the system

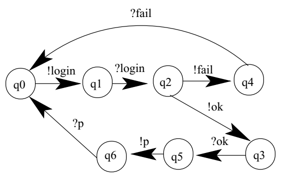

Fig. 2. Global secure automata

message $!ok$ is sent to an authorized client. A printer component $C2$ can accept incoming requests of print encoded as $?p$. A client component $C3$ requires login and print services encoded as messages $!login$ and $!p$, respectively and waits for incoming messages of successful/unsuccessful login encoded as $?ok/?fail$.

The global secure automaton (see figure 2) expresses a security policy in which a $!login$ request to the authentication component C1, once received, can be followed by either $!fail$ or $!ok$ messages. The service $!p$ can then be required only after a reception of an $?ok$ message.

The global secure automaton permits to monitor architectural system traces (see Definition 1). It performs a state transition for each observable message of the system and detects an attack when a message is not accepted.

Our main purpose is to automatically distribute the global secure automaton, so to monitor the distributed-component-based application in a peer-to-peer fashion. An algorithm produces a set of local secure automata that are assigned one for each component. Therefore, a local secure automaton can only observe the component local trace (see Definition 5) of the component it resides on. After the generation process each local secure automaton is implemented as a wrapper(filter) that envelops the component it supervises.

Notice that we consider deterministic automata. This permits to simplify the distribution algorithm and to reduce synchronization messages among filters. This choice is not a limitation, since, a non-deterministic automaton can be always translated to a deterministic automaton that accepts the same language. Moreover, we recall that a property is an high level description of the constraints imposed on the system components communications and not a complete description of the component-based-application behavior. In the remaining of the paper the notation $m^{C_i} \in C_i$ stands for messages locally sent/received by the component C_i.

6 Local Automata Generation

The monitor is conceived as a logically centralized process that makes a transition for each observable event of the system. A specification-based IDS can use the global secure automaton to realize the centralized monitoring by recognizing the languages defined by the security policies. Whenever these policies are violated an alarm is raised.

The algorithm to distribute the global secure automaton creates one filter for each component. The filter on a component C (in the following, we will denote it by $\Im_C$) implements a local secure automaton which, looking at the component local trace T_C (see Definition 5), detects a violation of the policies expressed by the global secure automaton. Obviously, by considering only the local-component trace of C $m_1^C m_2^C m_3^C \dots m_k^C m_{k+1}^C \dots$ is not sufficient to locally detect a violation of the policy. Therefore, $\Im_C$ has to parse an enriched trace that also contains context information provided by other filters. After the local parsing, $\Im_C$ can provide context information to other filters that need it. We call such information exchanged among filters *dependency information*.

Definition 10. *Let $\Im_C$ be the filter of the component C. Dependency information is of the form $!f(m, D)$ or $?f(m', S)$ where D and S range on the name of the application components. The message $!f(m, D)$,* outgoing dependency*, is sent by $\Im_C$ to filter $\Im_D$ in order to communicate that the C-component message m has been observed. The message $?f(m', S)$,*incoming dependency*, is an incoming information sent by filter $\Im_S$. With this information $\Im_S$ communicates to $\Im_C$ that it has observed a S-component message m'.*

Dependencies ensure that the merge (see Definition 4) of local-component traces result in a global trace of the system which satisfies the secure property expressed by the global-secure automaton. Hence, dependencies are a way to synchronize filters and to detect the violation of policies. Note that dependencies are sufficient to impose an order on messages; this allows us to relax the assumption that a global system clock exists (see Section 5).

A filter $\Im_C$ captures both local-component message and incoming dependencies ($?f(m, S)$) and outputs the outgoing dependencies ($!f(m, D)$) that will be used by other filters. In order to generate local automata we combine software architecture information with the global secure automaton. This combination is twofold: on one side permits to build local secure automata by projecting each transition of the global secure automaton (labelled with an architectural message, see Definition 6) on the component that accepts/sends the message. On the other side, it permits to enrich local secure automata with transitions that analyze and produce dependencies. Connections among components may be used to route context information messages through components filters [2]. Referring to the example in Figure 1 whenever $\Im_{C_1}$ needs to send a message to $\Im_{C_2}$ it has to route the message through the filter of C_3.
Informally, the algorithm for filters generation can be described as follows:

1. **Local automata generation**: For each component C, the set of automata $A_1 A_2 ... A_{n_C}$ is generated. These automata are the parts of the global secure automaton that processes events concerning interactions of the component C.
2. **Dependencies generation**:Let $A_1 A_2 ... A_{n_C}$ be the set of automata related to the component C. This step provides the needed context dependencies to ensure a complete and correct local message parsing. Furthermore, it connects the local automata $A_1 A_2 ... A_{n_C}$ of C in order to build a complete local secure automaton.

In the following we informally describe the algorithm and we illustrate its application by means of the example shown in Section 5. A more complete description is available in [16].

[2] The need of routing depends on the communication infrastructure on which the software architecture is built. For instance a component-based application may use communication layers that enable components to communicate with each other. In this case, a filter could send messages directly to another, which would avoid the overhead of routing messages via other filters.

6.1 Local Automata Generation

Given a global secure automaton $A = (Q, q_0, I, \delta)$, this step builds, for each component C of the system, a local secure automaton $\Im_C = (Q_C, q_{0C}, I_C, \delta_C)$. Informally $\Im_C$ is obtained by considering each rule $q_1 = \delta(q, m)$ defined in A. In the case that m is a C-component message such rule is reflected in a $\Im_C$-rule $q_1 = \delta_C(q, m)$, the states q,q_1 are added to Q_C and the message m is added to I_C. Therefore in the following we use two conventions: one is that we use exactly the same name q both for a state of the global automaton and the state of the filters where q has been projected. The other one is that, when it is clear from the context, we indifferently use either the rule $q' = \delta(q, m^C)$ or its $\Im_C$-projection $q' = \delta_C(q, m^C)$.

Looking at the global secure automaton A, the sequence of interactions that happen locally on a component C originates a local secure automaton $\Im_C$. In other words, $\Im_C$ does not include interactions among components that do not involve C. Therefore $\Im_C$ can result in a set of disconnected sub-automata $A_1 A_2 \ldots A_{n_C}$, each one modelling local interactions on C separated by interactions among different components.

The local automaton generation step is done locally to the component C. The time complexity is $O(|\delta|)$ where $|\delta|$ is the number of transitions of the global secure automaton. No new states are added, then the space complexity is linear.

Referring to the example in Section 5, after the local automata generation step the global secure automaton of Figure 2 is partitioned on each component of the system. Figure 3 shows such partition. For instance parts of the global automaton related to messages: $?login$, $!ok$ and $!fail$ constitute the local secure automaton on component $C1$, given that such messages are locally observed on that component. The same discussion can be done for the local secure automata of the components $C2$ and $C3$.

These local secure automata are not sufficient to validate the related component traces, therefore the next step shows how to add dependency transitions. Besides synchronization among local secure automata, dependencies are also used to link the disconnected automata $A_1 A_2 \ldots A_{n_C}$ of each $\Im_C$ (if any).

6.2 Dependencies Generation

The dependencies generation step takes as input the local secure automata and it adds dependencies information. Such information enforces the synchronization

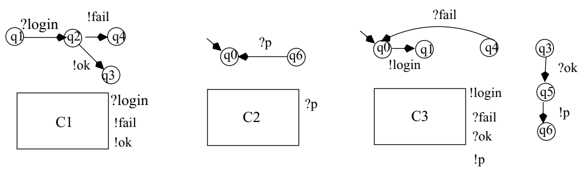

Fig. 3. Local automata generation

among the interested set of filters, so that merging the component local traces (see Definition 4) results in a trace accepted by the global secure automaton. For space reasons, we cannot provide a complete illustration of the dependencies generation step. Therefore we divide it into three phases and we sketch the basic idea of each phase. A complete and formal treatment of the whole step is described in [16].

Phase 1. In phase 1 the dependencies generation step provides the basic dependencies that are needed to synchronize a set of filters so that exactly one of them acquires the right to accept a local-component message.

A global secure automaton $A = (Q, q_0, I, \delta)$ can define a set of transitions exiting from a state q, with $q \in Q$. We consider the case in which from q two transitions exit: $q_1 = \delta(q, m^{C_1})$ and $q_2 = \delta(q, m^{C_2})$ with $m^{C_1} \in C_1$ and $m^{C_2} \in C_2$, and $q \neq q_1 \neq q_2$.

Local automata generation (see Section 6.1) ensures that the rules $q_1 = \delta(q, m^{C_1})$ and $q_2 = \delta(q, m^{C_2})$ are projected to the filters $\Im_{C_1}$ and $\Im_{C_2}$ respectively. Phase 1 adds the rules $q_1 = \delta_{C_1}(q, !f(m^{C_1}, C_2))$ and $q_2 = \delta_{C_1}(q, ?f(m^{C_2}, C_2)$ to $\Im_{C_1}$, and the rules $q_2 = \delta_{C_2}(q, !f(m^{C_2}, C_1))$ and $q_1 = \delta_{C_2}(q, ?f(m^{C_1}, C_1)$ to $\Im_{C_2}$.

From the point of view of A if it is in the state q then either the transition $q_1 = \delta(q, m^{C_1})$ or $q_2 = \delta(q, m^{C_2})$ can be applied. From the filters point of view such possibility is lost since these rules are independently applied by the two different filters residing on the two different components. The rules $q_1 = \delta_{C_1}(q, !f(m^{C_1}, C_2))$ and $q_2 = \delta_{C_2}(q, !f(m^{C_2}, C_1))$ are a means used by the filters to overcome this problem. Suppose that both filters $\Im_{C_1}$ and $\Im_{C_2}$ are in the state q. Each one of them can observe its local message, m^{C_1}, m^{C_2} respectively. In a single computation only one of them will participate in the (global) computation by parsing its message and leading to the state successor of q, that is either q_1 or q_2. However as far as the local automata $\Im_{C_1}$ and $\Im_{C_2}$ are concerned no matter who parses the message they must both move to the defined successor state, that is they will both reach either q_1 or q_2. For example, if $\Im_{C_1}$ observes the message m^{C_1}, it alerts $\Im_{C_2}$ of this observation by sending the dependency message $!f(m^{C_1}, C_2)$ and waits for an acknowledgment. If $\Im_{C_1}$ receives the $\Im_{C_2}$ acknowledgement, then this means that it has got the right from $\Im_{C_2}$ to move on and both filters move to state q_1. $\Im_{C_1}$ by consuming the message m^{C_1} by means of the rule $q_1 = \delta(q, m^{C_1})$. $\Im_{C_2}$ by consuming the dependency message $!f(m^{C_1}, C_2)$, by means of the rule $q_1 = \delta_{C_2}(q, ?f(m^{C_1}, C_1)$. In the case that both filters, at the same time, send the dependencies with each other then a synchronization protocol(see [16]) establishes that exactly one filter acquires the right to accept a component local message.

Phase 1 of the dependencies generation step considers a state q of a filter $\Im_C$ and its purpose is twofold: on one side it adds a set of transitions exiting from q labelled with outgoing dependencies. On the other side it adds a set of transition exiting from q labelled with incoming dependencies. The outgoing dependencies are needed to know the filters with whom $\Im_C$ has to synchronize, in order to acquire the right to parse a C-component message that labels a transition exiting from q. We call these outgoing dependencies *synchronization dependencies* and the protocol used by the filters to exchange these dependencies *synchronization protocol.* (see [16] for more details).

Phase 2. Phase 2 of the dependencies generation adds dependencies that are used by a filter to enable the parsing of local-component messages of other filters. A global secure automaton A can define a chain of rules $q_1 = \delta(q, m^{C_1})$ and $q_2 = \delta(q_1, m^{C_2})$, with $m^{C_1} \in C_1$, and $m^{C_2} \in C_2$, and $q \neq q_1$. The local automaton generation ensures that the rules $q_1 = \delta(q, m^{C_1})$ and $q_2 = \delta(q_1, m^{C_2})$ are projected on the filters $\Im_{C_1}$ and $\Im_{C_2}$, respectively. From the A point of view, this chain of rules defines a constraint among the messages m^{C_1} and m^{C_2}. That is, the message m^{C_1} must be accepted before the message m^{C_2}. However, from the local filters point of view, this constraint is lost, since the chain of rules is divided onto the filters $\Im_{C_1}$ and $\Im_{C_2}$. Therefore, the filter $\Im_{C_2}$ can autonomously accept the message m^{C_2} before the message m^{C_1} is accepted by the filter $\Im_{C_1}$. The problem is solved by adding dependencies. The dependencies generation adds to $\Im_{C_1}$ the rule $q_1 = \delta_{C_1}(q, !f(m^{C_1}, C_2))$ and to $\Im_{C_2}$ the rule $q_1 = \delta_{C_2}(q, ?f(m^{C_1}, C_1)$. Therefore $\Im_{C_2}$ can move to the state q_1, by means of the rule $q_1 = \delta_{C'}(q, ?f(m^{C_1}, C_1))$. However this rule can be applied only when the filter $\Im_{C_1}$ sends the outgoing dependency $!f(m^{C_1}, C_1)$. This is a means for filter $\Im_{C_1}$ to impose the right ordering among the messages m^{C_1} and m^{C_2}. We call such outgoing dependencies *enabling dependencies*, since they are used to enable the local-filter parsing when there is the right context condition.

Phase 3. Note however that, after the addition of such dependencies, some local automaton can still be disconnected. Phase 3 on one side links together the local disconnected automata $A_1 A_2 \ldots A_{n_C}$ through ε moves. On the other side it sets the initial state of all local automaton as the initial state of the global secure automaton.

The time-complexity to produce each local-automaton is $O(|\delta|^2)$ where $|\delta|$ is the number of transitions of the global secure automaton. The local dependencies generation does not add states with respect to the states of the global-secure automaton A. Therefore, the space-complexity is linear.

Figure 4 outlines the basic activities of a filter $\Im_C$ that is in a state q. In 4.1 a background thread buffers every C-component message in the message buffer and every incoming dependency in the dependencies buffer. In 4.2 $\Im_C$ picks up a C-component message m from the message buffer, if any. Steps 4.3-4.5 log and refuse m if it is recognized as an attack. On the contrary in 4.6 the filter $\Im_C$ tries to parse m by means of the rule $q1 = \delta_C(q, m)$. In 4.6b it starts the synchronization protocol in order to acquire the right to parse the message m. In the case that $\Im_C$ gains the right it applies the rule $q1 = \delta_C(q, m)$ and sends the enabling

1. a background thread buffers every C-component message in the message buffer and every incoming dependency in the dependencies buffer.
2. $\Im_C$-main process picks up a C-component message m from its message buffer, if any.
3. if m is not a C-local component message then it releases the message and logs a warning.
4. if m is a C-local component message that cannot be accepted in a successive state of $\Im_C$ then it trashes the message and raises an alarm.
5. if m is a message that cannot be accepted in the state q it logs a warning and it puts back the message on the buffer.
6. if m can be accepted by means of the rule $q1 = \delta_C(q, m)$ then
   (a) if $(q_1 = q)$ then it releases the message m and goes to step 7.
   (b) if $(q_1 \neq q)$ then it starts the synchronization protocol.
   (c) if it acquires the right to accept the message m then it sends the enabling dependencies, it applies the rule $q1 = \delta_C(q, m)$ and goes to step 7.
   (d) if $\Im_{C'}$, with $\Im_{C'} \neq \Im_C$ acquires the right to parse the message m' then
      – it puts back m on the local-component buffer.
      – it retrieves the rule $q' = \delta C(q, ?f(m', C'))$ and it moves without non-local message observation.
7. it picks up an incoming dependencies from its local-dependencies buffer that can be accepted, if any.

Fig. 4. $\Im_C$-filter behavior in a state q

dependencies. Otherwise it moves through incoming dependencies. Finally step 4.7 checks and applies the dependencies that are stored in the dependencies buffer.

In the remaining of the paper we make use of a set of assumptions that, although not mandatory, allow the simplification of the presentation. We assume first of all that messages among filters are not lost and that messages sent between local filters are received in the same order they are sent. When drawing the local secure automaton multiple transitions from the same source and target are indicated by using one arrow with multiple labels.

We use the example in Section 5 to illustrate the whole approach. Figure 5 shows the local automata related to components $C1, C2$ and C3 as produced by the dependencies generation. Initially all local filters have state q_0. When the component C3 sends a *!login* request to $C1$ the local secure automaton $\Im_{C3}$ captures the request. It observes that a *!login* message can be accepted then it sends the $!f(!login, C1)$ dependency and the *!login* message to $C1$. Finally, $\Im_{C3}$ moves to state q_1. The local filter on C1 receives the incoming message $!f(!login, C1)$ sent by $\Im_{C3}$ and it changes its state to q_1, since in q_0 it is waiting for an incoming message $?f(!login, C3)$. In state q_1 the filter on component $C1$ can accept the incoming *?login* message and it changes to the q_2 state. We can observe that component C2 can provide a print service *?p* only after some external events happened. These events are provided by $C3$ after a correct authentication is performed. At run time an attack is detected if a local automaton cannot accept a component local message, or if a local automaton is not able

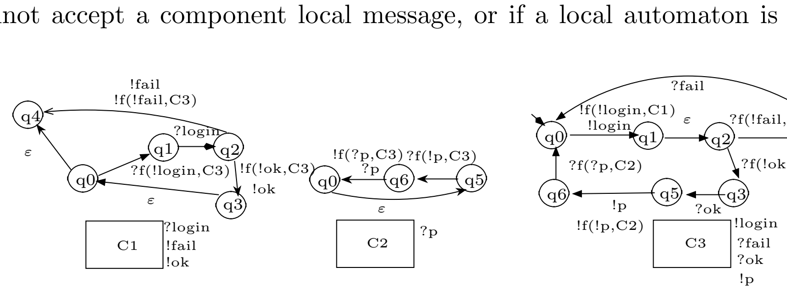

Fig. 5. Dependencies generation

to accept an external context information. In both cases, the information stored inside the filter gives details about the violation, providing a means to detect the source of attack. For example, in Figure 5 if component C3 requests the service $!p$ without previous $!login$, the local filter $\Im_{C3}$ captures the message and detects an error because it was waiting for a $!login$ request.

The overhead of messages generated by the filters is strictly related to the policies defined in the global secure automaton. A local automaton adds dependency messages when *non-interacting components* behavior has to be related. Let q be a state of the global secure automaton. Let $m_1 m_2 \dots m_n$ be n messages, related to n different components residing on n different hosts $H_1 H_2 \dots H_n$. Suppose that the messages $m_1 m_2 \dots m_n$ label a transition exiting from the state q. In the worst case when a local secure automaton on the host H_i moves from a state q to a state q',with $q \neq q'$, then at most n dependencies can flow on the distributed system. In practice dependency synchronization messages are relatively small in size and, depending on the system architecture, it is possible to bound the number of the messages exiting from a state q related to different components/hosts.

Correctness and completeness of our algorithm derive from the following theorem that is described in [16].

Theorem 1. Let $A = (Q, q_0, I, \delta)$ be a global secure automaton, $\{C_1, C_2, C_3 \dots C_i \dots\}$ a set of components. Let $\Im_{C_i}$ be the automaton related to component C_i as produced by the algorithm. A accepts an architectural trace $m_1 m_2 m_3 \dots m_i m_{i+1} \dots$ iff all component traces T_{C_i} are accepted by $\Im_{C_i}$.

As discussed in Section 1 the main problem of security tools is tampering. Intruders can blind the security measures, so to violate the policies or use security measure against the system itself. In our approach, a component changing behavior is detected but problems can still arise if an intruder decides to attack by modifying both the filter's and the component's behaviors. Referring to (Figure 5), the intruder can change C3 so that it requests a printer service $?p$ without no previously $!login$ and can change accordingly filter $\Im_{C3}$. $\Im_{C3}$ is therefore changed so that it does not have anymore a $!login$ transition and the related signaling message, leaving only the transition from q_5 to q_6. In other words, the component and its related filter are changed to provide a different behavior. When $C3$ sends a $!p$ request, the local filter $\Im_{C3}$ does not detect any violation. The filter on component C2, $\Im_{C2}$, receives the correct $!f(!p, C2)$ signal and provides the printer service. The solution to this problem is to add further dependencies on the local secure automaton. For instance, $\Im_{C2}$ can be enriched to rely on the context information that the $?login$ message took place on filter $\Im_{C1}$. A new step (tampering step) can use local automata as generated by the dependencies generation step to add further dependencies. In other words this means to add redundant context information so that Theorem 1 is still true and both a component and related filter tampering can be detected.

7 Conclusion and Further Work

We have presented a distributed specification-based approach to detect intrusions at architectural level. Its peer-to-peer design allows the supervision of complex applications where no centralized point of information flow exists or can be introduced. This distributed solution presents several advantages with respect to centralized monitors. It is scalable, since the monitoring of an increasing number of components does not rely on a single point of data correlation, so to avoid problems of detection reaction time, loss of data and scalability. It provides an approach to face the problem of security measures tampering. A filter can detect a component that violates the policy, and other filters control and analyze the filter behavior to discover its tampering. The disadvantage of our approach concerns the potential message traffic increase due to the dependency messages exchanged among filters. This is the inevitable cost to pay to achieve a filters distribution which is correct and complete with respect to a centralized approach. However this overhead depends on the software architecture of the system to monitor and on the adopted security policies. Thus the suitability of the approach has to be measured taking into account these two factors.

At present our research proceeds in three directions. A prototypal version of the tool to generate local filters starting from a global secure automaton specification has been developed [15]. The approach has been applied to an industrial component-based application [17]. We are refining and extending the approach considering cases in which more than one component and the related filters are changed at the same time. We are also considering the use of context free languages to specify policies.

References

1. T.Eckmann, S., Vigna, G., Kemmer, R.A.: Statl: An attack language for state-based intrusion detection. Journal of Computer Security **10** (2002) 71–104
2. Javitz, H.S., Valdes, A.: The nides statistical component description and justification. Technical report - Columbia University (1994)
3. Vaccaro, H., Liepins, G.: Detection of anomalous computer session activity. In proc. of the 1989 Synopsium on Security and privacy (1989) 280–289
4. Stillerman, M., Marceau, C., Stillman, M.: Intrusion detection for distributed applications. Communications of the ACM (1999)
5. Ko, C., Ruschitza, M., Levitt, K.: Execution monitoring of security-critical programs in distribute system: A specification-based approach. IEEE (1997)
6. Aucsmith, D.: Tamper resistant software: An implementation. LNCS (1997)
7. Vigna, G., A.Kemmer, R.: Netstat: A network-based intrusion detection system. In proc. of the 14th Annual Computer Security Applications Conf. (1998)
8. A.Porras, P., G.Neumann, P.: Event monitoring enabling responses to anomolous live disturbances. In Proc. of 20th NIS Security Conference (1997)
9. Snapp, S.R., Dias, J.B.G.V., Goan, T., Heberlein, L.T., Ho, C., Levitt, K.N., Mukherjee, B., Smaha, S.E., Grance, T., Teal, D.M., Mansur, D.: Dids (distributed intrusion detection system) - motivation architecture and early prototype. In proc. 14th National Security Conference **1** (1996) 361–370

10. White, G.B., Fisch, E.A., Pooch, U.W.: Cooperating security managers: A peer-based intrusion detectionn system. IEEE Network (1996) 20–30
11. Sen, K., Vardhan, A., Agha, G., Rosu, G.: Effecient decentralized monitoring of safety in distributed system. ICSE (2004)
12. Dulay, N., Lupu, E., Sloman, M., Damianou, N.: A policy deployment model for the ponder language. IM2001, Seattle,IEEE Press. (2001)
13. Schneider, F.B.: Enforceable security policies. ACM Trans. on Information and System Security **3** (2000) 30–50
14. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM **21** (1978) 558–565
15. Mostarda, L., Inverardi, P.: Distributed detection system for secure software architectures (desert) - a peer-to-peer tool for intrusion detection. http://www.di.univaq.it/mostarda/sito/default.php (2004)
16. Mostarda, L., Inverardi, P.: A distributed intrusion detection approach for secure software architecture - extended version. technical report http://www.di.univaq.it/mostarda/sito/default.php (2005)
17. Inverardi, P., Mostarda, L., Tivoli, M., Autili, M.: Automatic synthesis of distributed adaptors for component-based system. Submitted for publication (2005)