Distributed Exchange of Alerts for the Detection of Coordinated Attacks

J. Garcia-Alfaro[†], M. A. Jaeger[‡], G. Mühl[‡], I. Barrera[⋆], and J. Borrell[⋆]

[†]Open University of Catalonia, Computer Science and Multimedia Studies
Rambla Poble Nou 156, 08018 Barcelona, Spain

[‡]Technical University of Berlin, Communication and Operating Systems
EN6, Einsteinufer 17, D-10587 Berlin, Germany

[⋆]Autonomous University of Barcelona, Dept. of Inf. and Comm. Engineering,
Edifici Q, 08193 Bellaterra, Spain

E-mail: {joaquin.garcia-alfaro,michael.jaeger,g_muehl}@acm.org
ibarrera@deic.uab.es, joan.borrell@uab.es

Abstract

Attacks and intrusions to information systems cause large revenue losses. The prevention of these attacks is not always possible by just considering information from isolated sources of the network. A global view of the whole system is necessary to react against the different actions of such an attack. The design and deployment of a decentralized system targeted at detecting as well as reacting to information system attacks might benefit from the use of the publish/subscribe model. In this paper, we discuss the advantages and convenience in using this communication paradigm for a general decentralized attack prevention framework and overview the design and implementation of our approach by using a combination of two different publish/subscribe middleware products. Furthermore, we present a quantitative evaluation of our approach.

Keywords: *Network Security, Attack Prevention System, Publish/Subscribe, Message Oriented Middleware, IDMEF*

1 Introduction

When attackers gain access to a corporate network by compromising authorized users, computers, or applications, the network and its resources can become an active part of a globally distributed or coordinated attack. Such an attack might be a coordinated port scan or distributed denial of service attack against third party networks—or even against computers on the same network. Both, distributed and coordinated attacks, rely on the combination of actions performed by a malicious adversary to violate the security policy of a target computer system. In order to prevent these attacks, a global view of the system as a whole is necessary. Hence, different events and specific information must be gathered and combined from all the sources. This affects, for example, information about suspicious connections, initiation of processes, and addition of new files.

In [1, 3], we presented an attack prevention framework that is targeted at detecting as well as reacting to distributed and coordinated attack scenarios. Our approach is based on gathering and correlating information held by multiple sources. We use a decentralized scheme based on message passing to share alerts in a secure communication infrastructure. This way, we can detect and prevent these kind of attacks performing detection and reaction processes based on the knowledge gained through alert correlation. In this paper, we extend the communication infrastructure of our proposal and evaluate a prototypical implementation of it. This infrastructure aims at fostering the collaboration between the different components of our framework in order to achieve a more complete view of the system as a whole. Once this is achieved, it is possible to detect and react on the different actions of a coordinated or distributed attack.

The structure of this paper is the following. We start in Section 2 with analyzing related work. We continue in Section 3 with an introduction of the publish/subscribe communication model and an overview of the communication mechanism proposed for the exchange of information among the components of our system. In Section 4, we

978-0-7695-3135-9/08 $25.00 © 2008 IEEE
DOI 10.1109/CNSR.2008.70

present the results of an evaluation of a first prototype implementation. We close in Section 5 with conclusions and give an outlook on future work.

2 Related Work

Traditional client/server solutions for the prevention of distributed and coordinated attacks can quickly become a bottleneck due to saturation problems associated with the service offered by centralized or master domain analyzers. A master domain analyzer is the entity on top of a hierarchy of IDSs consisting of multiple analyzers and different domains to analyze. Centralized systems, such as DIDS [11] and NADIR [5], use this approach to process their data in a central node although the collection of data is distributed. These schemes are straightforward as they simply push the data to a central node and perform the computation there. Hierarchical approaches such as GrIDS [12] and NetSTAT [15] have a layered structure where data is locally preprocessed and filtered. Although they mitigate some weaknesses present in centralized schemes, they still cannot avoid bottlenecks, scalability problems, and fault tolerance issues due to vulnerabilities at the root level.

In contrast to these traditional designs, alternative approaches try to eliminate the need for dedicated elements. The idea of distributing the detection process has some advantages regarding centralized and hierarchical approaches. Mainly, decentralized architectures have no single point of failure and bottlenecks can be avoided. Some message passing designs, such as CSM [16] and Quicksand [6], try to eliminate the need for dedicated elements by introducing a peer-to-peer architecture. Instead of having a central monitoring station to which all data has to be forwarded, there are independent uniform working entities at each host performing similar basic operations. To detect coordinated and distributed attacks, the different entities have to collaborate on the detection activities and cooperate to perform a decentralized correlation algorithm. These designs seem to be a promising technology to implement decentralized architectures for the detection of attacks.

However, these systems still suffer from some limitations. For instance, they might require complete knowledge of the system: All nodes have to be connected to each other which can make the matrix of the connections, which are used for exchanging alerts, grow explosively and become very costly to control and maintain. Another important disadvantage present in these designs is that the different entities always need to know where a received notification has to be forwarded (similar to a queue manager). This way, when the number of possible destinations grows, the network view can become extremely complex, which leads to a system that is not scalable. Other designs are based on flooding which makes the system easier to maintain on the cost of scalability, as the message complexity grows fast with the number of brokers and messages published.

Most of these limitations can be solved efficiently by using a publish/subscribe-based system. The advantage of this model for our problem domain over other communication paradigms is on the one hand that it keeps the producer of messages decoupled from the consumer and on the other hand that the communication is information-driven. This way, it can avoid problems regarding the scalability and the management inherent to other designs, by means of a network of publishers, brokers, and subscribers. A publisher in a publish/subscribe system does not need to have any knowledge about any of the entities that consume the published information. Likewise, the subscribers do not need to know anything about the publishers. New services can simply be added without any impact on or interruption of the service to other users.

3 Publish/Subscribe Model

The publish/subscribe communication model is intended for group communication, i.e., situations where a message (*notification*) sent by a single entity is required by and should be distributed to multiple entities. It is often used for efficient and comfortable information dissemination to group members which may have individual interests in arbitrary subsets of messages published. In contrast to multicast communication, clients have the possibility to describe to the events they are interested in more precisely (e.g., based on the contents of the notification). Clients can choose to either subscribe or unsubscribe to messages as time goes by, and all the subscribers are independent of each other.

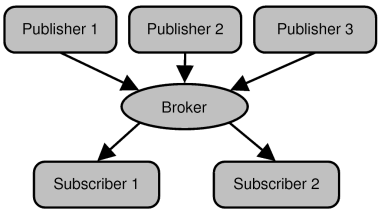

Figure 1. Simple publish/subscribe system.

3.1 Publish/Subscribe Systems

A publish/subscribe system consists of at least one broker forwarding notifications published by clients to other clients that are interested in them. For scalability reasons, it is common to implement a distributed broker network that forms a *notification service* through an overlay network consisting of brokers. This service provides a distributed in-

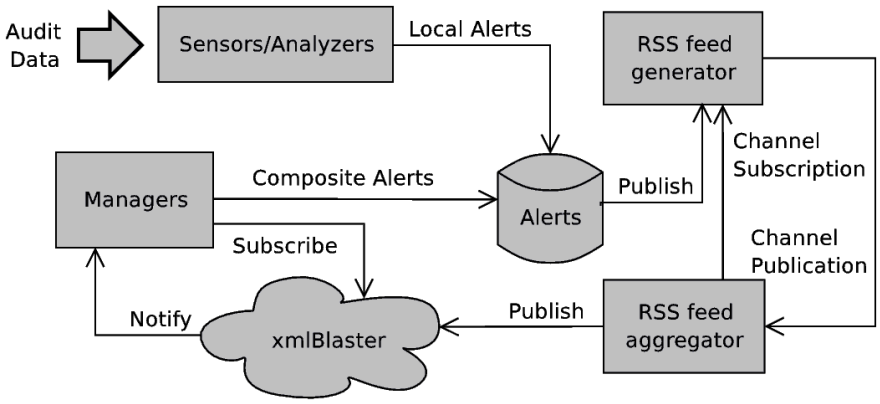

Figure 2. Proposed infrastructure for the exchange of alerts.

frastructure for notification routing which includes the management of subscriptions and the dissemination of notifications in a possibly asynchronous way. Clients can publish notifications and subscribe to filters that are matched against the notifications passing through the broker network. If a broker receives a new notification it checks if there is a local client that has subscribed to a filter that matches this notification. If so, the message is delivered to this client. Additionally, the broker forwards the message to neighbor brokers according to the applied routing algorithm. We refer to [8] for a survey on publish/subscribe systems.

An example of a simple centralized publish/subscribe system is shown in Figure 1. Here, five clients are connected to a single broker: three clients that are publishing notifications and two clients that are subscribed to a subset of the notifications published on the broker. Subscribers can choose to subscribe to the notifications available through the broker or cancel existing subscriptions as needed. The broker matches the notifications it receives from the publishers to the subscriptions, ensuring this way that every publication is delivered to all interested subscribers. This very basic publish/subscribe setup can be extended by connecting multiple brokers, enabling them to exchange messages. The extended design allows subscribers managed by one of the brokers to receive messages that have been published by clients of another broker, further freeing the subscriber from the constraints of connecting to the same broker the publisher is connected to. Available implementations usually make this transparent for the programmer by keeping the same interface operations as in the centralized design. This way, an application can easily be distributed. The subscribers are able to formulate their interests based, e.g., on the contents of the notifications and a special attribute they carry. This is known as content-based and topic-based subscription, respectively.

Topic-based subscriptions are easier to handle than content-based subscriptions. Subscribers specify their interest in a topic and receive all messages published on this topic. Two different matching mechanisms are commonly used here. One matches subscriptions successfully to notifications if the topic of the subscription exactly matches the topic under which the notification is published. Using this mechanism, topics become equivalent to "channels". The other mechanism arranges topics in a subject tree such that subscriptions not only match notifications if the topics are the same, but also if the topic of the subscription is an ancestor of the notification topic in the subject tree (in this case, a topic becomes equivalent to a "theme").

Content-based subscriptions allow for more sophisticated subscriptions on the cost of higher matching load and more complex routing decisions. Here, a subscription can be formulated extremely fine-grained based on the content of notifications using a query language. Moreover, there does not have to be a system wide agreement on the set of topics as it is in general needed for topic-based routing.

3.2 Proposed Architecture

In this section, we describe the main elements of our platform, as well as their interactions for the exchange of information. Let us start by having a look to the main components of our attack prevention system [1, 3]. A description of each component is presented in the following.

Analyzers – Analyzers are local elements which are responsible for processing local audit data. They process the information gathered by associated sensors to infer possible alerts. Their task is to identify occurrences which are relevant for the execution of the different steps of an attack and pass this information to the correlation manager via the publish/subscribe system. They are interested in local

events. Each event is detected in a sensor's input stream. The set of derived alerts is permanently stored in a local database and encoded in IDMEF (*Intrusion Detection Message Exchange Format*) [2]—which is an IETF proposal for the exchange of information between different security components, such as *Intrusion Detection Systems*, and *Firewalls*. Each alert expressed in IDMEF format has a unique classification and a list of attributes with their respective types to identify the analyzer that originated the alert (*AnalyzerID*), the time the alert was created (*CreateTime*), the time the event(s) leading up to the alert was detected in the sensor's input stream (*DetectTime*), the current time on the analyzer (*AnalyzerTime*), and the source(s) and target(s) of the event(s) (*Source* and *Target*). All possible classifications and their respective attributes must be known by all the components of the system (i.e., analyzers and managers) and all analyzers must be capable of deriving instances of local events of arbitrary types. This way, the correlation process presented in [1, 3] can be realized.

Managers – The use of multiple analyzers and sensors together with heterogeneous detection techniques increases the detection rate, but it also increases the number of information to process (both, local alerts generated within the same system and those received by external components). It is therefore necessary to distribute a set of managers which are going to be in charge of executing a complete process to fuse, merge, and correlate the received information [1, 3]. During a first stage, all the alerts associated to a single event, but reported by different analyzers, should be grouped and transformed into a single instance that uniquely identifies the occurrence of a single action in the system. Then, in a second stage, a correlation process allows the correlation of information between different alerts by looking to those logical links between single occurrences or actions in the system. The process presented in [1] shows how it is possible to implement this correlation process in order to find out which objectives are going to follow after these actions, and which possible countermeasures can be selected by the system in order to stop or neutralize malicious actions.

3.3 Publication and Subscription of Local Alerts based on RSS Channels

As we can see in Figure 2, the set of alerts stored within the database of each detection zone is disseminated to the rest of the zones through a syndication of alerts based on RSS (*Really Simple Syndication*) [4]. RSS is a publish/subscribe mechanism based on channels. RSS is currently very popular and extended on web services and weblogs for the dissemination of news and general information. The RSS architecture is quite simple: a set of clients manifest their interests by subscribing RSS channels (or *feeds*) upon which other clients are periodically publishing information. In this manner, and based on periodic polling, subscribers receive the information in which they declared interest. In our case, the RSS content of each channel is directly generated by exporting the set of local alerts stored in the database of each zone, and then encoded them into XML messages. Then, those stored alerts are integrated into a unique channel which can be accessed by authorized RSS clients.

Through a specific RSS aggregator integrated within the components of our platform, each zone processes the set of alerts of their local database and publishes them through a second publish/subscribe system in charge of distributing those messages with other detection zones. The alerts published with this second element, more complete and optimized for operations like searches and distributed subscriptions, can be finally accessed by the rest of managers of partners collaborating in the cooperative distribution process presented in [1, 3]. The RSS aggregator of each zone must be properly configured in order to process the set of subscriptions and publications. In oder to do so, a specific administration policy is set up by the administrator in charge of each zone.

3.4 Dissemination of Global Alerts through a Content-based Middleware

As our motivation for this work is not targeted on developing a new publish/subscribe middleware, we try to reuse as much available code and tools as possible. For our experiments (cf. Section 4) we used *xmlBlaster*, an open source publish/subscribe message oriented middleware [10]. It connects a set of nodes that build up the infrastructure for exchanging alerts using the interface operations offered by the underlying middleware. The notifications exchanged through xmlBlaster are expressed in XML-based IDMEF compliant messages. Publishers construct these messages from the set of local alerts received by the RSS-feed of each zone; subscribers express their interests by subscribing filters specified by means of *XPath* expressions. Each xmlBlaster message consists of a header, filtering that can be applied to, a body, and a system control section. The body of each xmlBlaster message is directly derived from the IDMEF messages of each detection zone; and filters are evaluated over the header of each xmlBlaster message to decide when a message has to be delivered to a subscriber.

The communication infrastructure offered via xmlBlaster can be viewed as a black box with an *interface* (cf. Figure 3) together with a set of *operations*, each of which may take a number of *parameters*. Clients can invoke *input operations* from the outside, and the system itself invokes *output operations* to deliver information to clients. To publish alerts, clients invoke the *pub*(a) operation, giving the alert a as parameter. The published alert can potentially be delivered to all clients connected to the system via an output operation

called *notify*(a). Clients register their interest in specific kinds of alerts by issuing subscriptions via the *sub*(F) operation, which takes a filter F as parameter. Each client can have multiple active subscriptions which must be revoked separately through the *unsub*(F) operation.

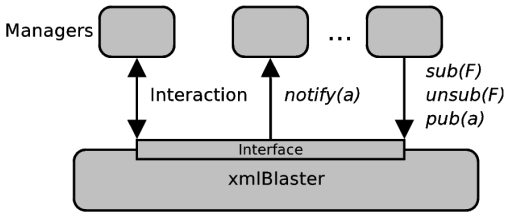

Figure 3. Operations offered by the xmlBlaster middleware.

All these operations are instantaneous and take parameters from the set of all clients $\mathcal{C}$, set of all alerts $\mathcal{A}$, and the set of all filters $\mathcal{F}$. Formally, a filter $F \in \mathcal{F}$ is a mapping defined by the following boolean expression:

$$F: \quad a \longrightarrow \{\text{true}, \text{false}\} \qquad \forall a \in \mathcal{A}$$

Please note that a *notification* n *matches filter* $F \in \mathcal{F}$ iff $F(a) = \text{true}$. We also assume that each alert can only be published once and that every filter is associated with a unique identifier in order to enable the alert communication infrastructure to identify a specific subscription.

For the experiments discussed in Section 4, the set of managers of each zone may register their interest in a subset $\mathcal{A}$ of alerts published in xmlBlaster by invoking the *sub*(A) operation, which takes the filter A as parameter, with

$$A(a) = \begin{cases} \text{true} & , \quad a \in \mathcal{A} \\ \text{false} & , \quad a \notin \mathcal{A} \end{cases}$$

Once subscribed to these filters, the communication infrastructure will notify the subscribed managers of all those alerts, previously grouped by the RSS aggregator of each zone, that apply to those filters.

4 Evaluation of CIDEX

In this section, we give an overview of CIDEX (which stands for *Communication Infrastructure for a Decentralized Exchange of Alerts*), a software prototype that implements the communication infrastructure proposed in this paper. The evaluation of CIDEX is based on a set of testbeds which we present in the following. The evaluation was carried out on a set of machines Intel-Pentium 4 at 1.5 GHz, with 256 MB of memory, and running Debian GNU/Linux 2.6 operating system configured with HTTP Apache 1.3 server, PHP 4.3 and Java HotSpot VM 1.5 (necessary for the execution of xmlBlaster brokers).

The generation of alerts was carried out by using a set of sensors of *prelude* [13] (among them, *snort* [9], which was compiled to be integrated as a prelude-compliant sensor within our platform). The complete set of alerts, managed and delivered by prelude, was permanently stored on a database managed by PostgreSQL and delivered consecutively to the components. We implemented an RSS generator in PHP with the objective of publishing the set of local alerts stored in the PostgreSQL database. The communication between the rest of components was based on the C socket library of xmlBlaster. Finally, the generation, parsing, and analysis of IDMEF messages was implemented by using a modified version of the libidmef library [7] based on libxml [14].

Performance of xmlBlaster brokers– In our first experiments, we evaluated the processing load and memory consumption of the xmlBlaster brokers for the delivery of an increasing set of alerts up to $10,000$ alerts. The average and confidence interval of the results of the first test set is shown in Figure 4. One can see in this figure that both, CPU load and memory consumption of xmlBlaster brokers, remained considerably stable during the whole experiment (about 16% and 4%, respectively). The significant big confidence intervals in memory consumption resulted from an interference with the Java garbage collector and is, thus, not relevant for our measurements.

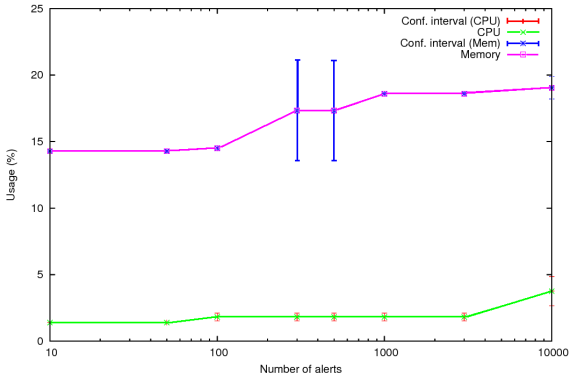

Figure 4. Process and memory consumption of xmlBlaster brokers.

Performance of xmlBlaster publishers– We also evaluated the CPU load and memory consumption of xmlBlaster publishers within our prototype. Figure 5 shows that both, CPU load and memory consumption, smoothly increased until an average of $3,000$ alerts. However, once raised this level of alerts, the consumption of resources started to increase heavily.

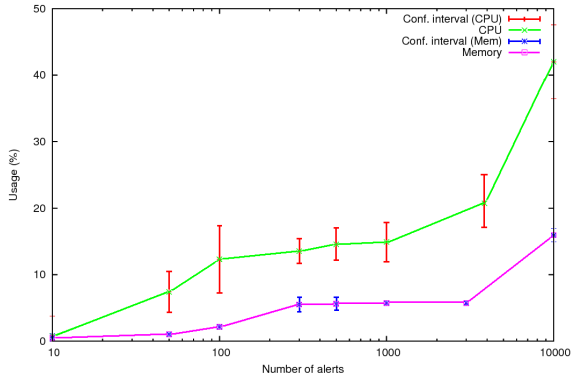

Figure 5. Process and memory consumption of xmlBlaster publishers.

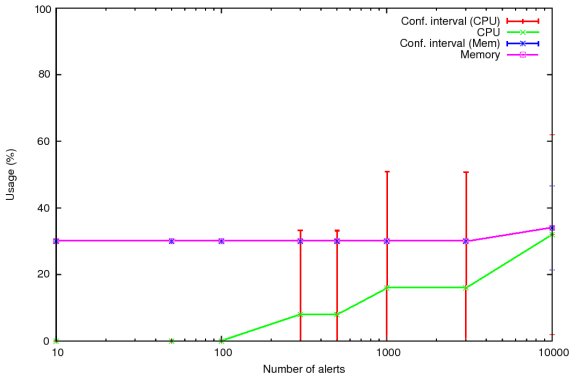

Figure 6. Process and memory consumption of xmlBlaster subscribers.

It is important to note that in the current version of our implementation, the set of publishers are fed by local RSS aggregators that send the set of local alerts as XML data. The complete set of alerts received through this process is parsed, analyzed, and republished by using libidmef which, in turn, is built over the libxml library. The libxml library provides two interfaces to parse XML data: a DOM style tree interface, and a SAX style event-based interface. Up to now, we are using the DOM interface for our implementation due to its ease of use. Its main drawback is, however, that its usage of resources is proportional to the size of the XML data. We thus think that the growing resource consumption by the set of publishers is due to the processing of XML data rather than the publication of the alerts. We are actually moving our current implementation to the SAX-based interface. We hope that this will help us to lower the current consumption of resources.

Performance of xmlBlasters subscribers– Figure 6 shows the consumption of resources by the xmlBlaster subscribers in our testbeds which asynchronously receive the set of alerts forwarded by the xmlBlaster brokers. The CPU load remains stable at about 30%, independently of the number of alerts. The consumption of memory, on the other hand, smoothly increases as alerts arrive, but never bypassed the 50% threshold in the experiments. These results are very positive and give us good hope that using this communication paradigm for the dissemination of alerts in our platform indeed increases the scalability and efficiency of our proposal. We envision that the consumption of resources by the set of subscribers during the reception of alerts on real case scenarios will be similarly. Moreover, we are confident that the new management of XML data pointed out above will improve the limitation imposed by our current XML handling.

Latency of the notification service– In the second stage of our experiments we analyzed the latency of the notification service. We evaluated the latency by publishing the alerts in three different ways: (1) all the sets of alerts were organized by xmlBlaster on a single principal node (*key*) from which all the alerts are associated; (2) the alerts were organized on three different categories taking into account the impact of the alert (high, medium, low) and leaving the organization of alerts to xmlBlaster on a tree with three principal nodes (one for each category of impact); and (3) each alert is classified by using its own alert identification– in this case, xmlBlaster generated a node for each alert.

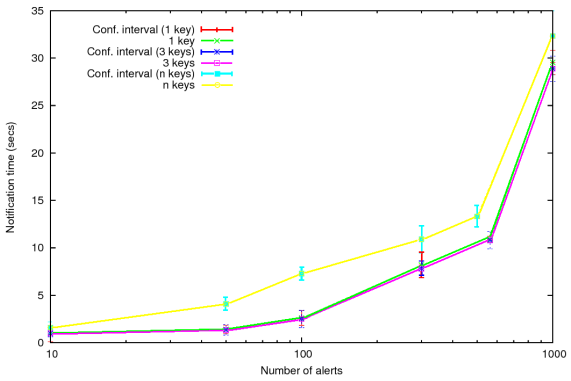

Figure 7. Latency of the xmlBlaster notification service.

When analyzing the delivery time for the two first scenarios (i.e., one and three keys), we see that for a number of alerts below 100 the difference in latency is minimal. However, as soon as the number of alerts passes this threshold, we the second scenario offers better results for the same number of exchanged alerts. We conclude that this difference is due to the routing algorithm of the notification service, which hap-

pens to be faster as the number of keys increases. However, the analysis of latency in the third scenario shows that if the number of keys increases proportionally to the number of alerts, the delivery of alerts takes much longer. Further experiments are going to be performed in future in order to find out which organization handles the best balance in order to guarantee the lowest delay. However, it is important to note that message delivery did not become a bottleneck in all three organization as all messages were processed in time and never reached the saturation point.

Latency of the delivery service of RSS messages– In a third stage we analyzed the latency of the aggregation and delivery of RSS messages. This process is responsible of parsing, analysis, and delivery of local alerts derived from the local events stored on the PostgreSQL database management system. The results plotted in Figure 8 show that this latency is proportional to the number of alerts to aggregate. We see that once more than $1,000$ alerts have been processed, the time needed for parsing the XML message generated by the RSS aggregator increases considerably. Please note that this parsing process is similar to the process performed by the publishers which has been discussed above. Hence, a similar reasoning applies in this case. We consider that once we will move our implementation to the SAX-based interface for the parsing of XML messages the consumption of resources will decrease considerably.

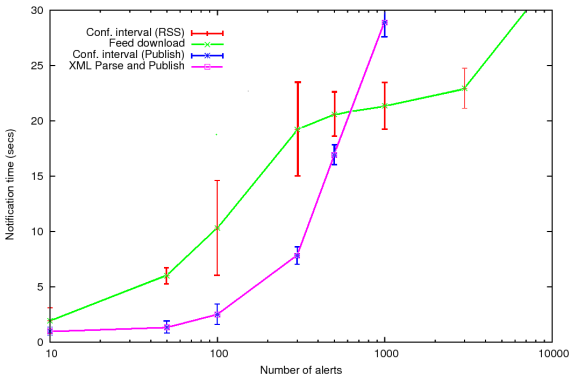

Figure 8. Latency of the delivery service of RSS messages.

Latency of the complete process– On a final stage, we evaluated the latency of the complete system, where we combined the complete set of elements of our testbeds with the set of alerts and traces generated by the sensors and analyzers managed by prelude and persistently stored within the database managed by PostgreSQL. The evaluation was performed by taking into account the three different organization of messages already discussed above (i.e., (1) set of alerts organized by xmlBlaster on a single principal node; (2) set of alerts organized on three different categories; and (3) set of alerts classified by using their alert identification). Figure 9 shows the latency of the alert delivery for each one of these three organizations. As expected, the results keep pointing to a moderate organization of messages as the best choice. Please note that during this final evaluation the whole system remained stable without reaching any saturation point. Only the RSS aggregator and the current management of XML data supposed a limitation to the proposed scheme. As we pointed out above, we are actually working on this point in order to improve and reevaluate the testbeds.

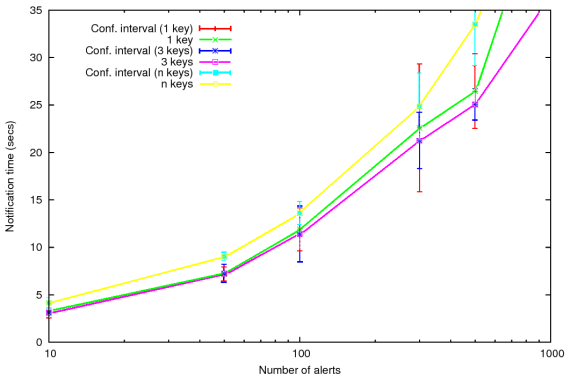

Figure 9. Latency of the complete process for the delivery of messages and alerts.

5 Conclusions and Outlook

In this paper, we presented an infrastructure to share alerts between the components of a security framework for the prevention of coordinated attacks. The framework itself is targeted at detecting as well as reacting to attack scenarios by using the publish/subscribe communication paradigm. In contrast to traditional client/server solutions, where centralized or hierarchical approaches quickly become a bottleneck due to saturation problems associated with the service offered by centralized or master domain analyzers, the information exchange between peers in our design achieves a more complete view of the system in whole. We believe that this is necessary to detect and react on the different actions of an attack. We have presented the use of two complementary publish/subscribe technologies for the deployment of our proposal. On the one hand, the use of RSS channels to spread local alerts which are persistently stored on each detection zone; on the other hand, the use of a publish/subscribe message-oriented middleware. The use of these two technologies have been evaluated. The set of experiments we conducted allows us to confirm that the

proposed architecture performs well enough for the application in real-world scenarios.

As future work we are considering to secure the communication partners by utilizing TLS/SSL, as well as to evaluate how this extension affect its use to the performance of the system. By adding TLS/SSL in our current infrastructure, each collaborating node must receive and handle a private and a public key. The public key of each node will be signed by a certification authority (CA), that is responsible for the protected network. Hence, the public key of the CA has to be distributed to every node as well. The secure TLS/SSL channel will allow the communicating peers to communicate privately and to authenticate each other, thus preventing malicious nodes from impersonating legal ones. The implications coming up with this new feature, such as compromised key management or certificate revocation, will be part of this work. We are also planning a more in-depth study about privacy mechanisms by exchanging alerts in a pseudonymous manner. By doing this, we hope that we can provide the destination and origin information of alerts (*Source* and *Target* field of IDMEF messages) without violating the privacy of publishers and subscribers located on different domains. Our study will cover the design of a pseudonymous identification scheme, trying to find a balance between identification and privacy.

Acknowledgments The collaboration between J. Garcia-Alfaro, F. Cuppens, F. Autrel, and T. Sans sharpened many of the arguments presented in this paper. The work has been supported by funding from the Spanish Ministry of Science and Education, under the project *CONSOLIDER CSD2007-00004 "ARES"*.

References

[1] F. Cuppens, F. Autrel, Y. Bouzida, J. Garcia-Alfaro, S. Gombault, and T. Sans. Anti-correlation as a criterion to select appropriate counter-measures in an intrusion detection framework. *Annals of Telecommunications*, 61(1-2):192–217, 2006.

[2] H. Debar, D. Curry, and B. Feinstein. Intrusion detection message exchange format data model and extensible markup language. *Request for Comments 4765*, March 2007.

[3] J. Garcia-Alfaro, F. Autrel, J. Borrell, S. Castillo, F. Cuppens, and G. Navarro. Decentralized publish/subscribe system to prevent coordinated attacks via alert correlation. In *6th Int'l Conf. on Information and Communications Security*, volume 3269 of *LNCS*, pages 223–235, October 2004. Springer-Verlag.

[4] B. Hammersley. *Content Syndication with RSS*. O'Reilly Ed., First Edition, March 2003, ISBN 0-596-00383-8, 202 pages.

[5] J. Hochberg, K. Jackson, C. Stallins, J. F. McClary, D. DuBois, and J. Ford. NADIR: An automated system for detecting network intrusion and misuse. In *Computer and Security*, volume 12(3), pages 235–248. May 1993.

[6] C. Kruegel and T. Toth. Distributed pattern detection for intrusion detection. In *Network and Distributed System Security Symposium Conference Proceedings: 2002*, 1775 Wiehle Ave., Suite 102, Reston, Virginia 20190, U.S.A., 2002. Internet Society.

[7] A. C. Migus. IDMEF XML library. `http://sourceforge.net/projects/lib-idmef/`, March 2004.

[8] G. Mühl, L. Fiege, and P. R. Pietzuch Distributed Event-Based Systems. *Springer-Verlag*, August 2006.

[9] M. Roesch. Snort: lightweight intrusion detection for networks. In *13th USENIX Systems Administration Conference*, Seattle, WA, 1999.

[10] M. Ruff. XmlBlaster: open source message oriented middleware. White paper [on-line]. `http://xmlblaster.org/`, 2000.

[11] S. R. Snapp, J. Brentano, G. V. Dias, T. L. Goan, L. T. Heberlein, C. Ho, K. N. Levitt, B. Mukherjee, S. E. Smaha, T. Grance, D. M. Teal, and D. Mansur. DIDS (distributed intrusion detection system) - motivation, architecture and an early prototype. In *14th National Security Conference*, pages 167–176, October, 1991.

[12] S. Staniford-Chen, S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland K. Levitt, C. Wee, R. Yip, and D. Zerkle. GrIDS – a graph-based intrusion detection system for large networks. In *19th National Information Systems Security Conference*, 1996.

[13] Y. Vandoorselaere and L. Oudot. Prelude-IDS, un Système de Détection d'Intrusion hybride open-source. MISC issue 3, July 2002.

[14] D. Veillard. The XML C library for Gnome (libxml). `http://www.xmlsoft.org`, 2006.

[15] G. Vigna and R. A. Kemmerer. NetSTAT: A network-based intrusion detection system. *Journal of Computer Security*, 7(1):37–71, 1999.

[16] G. B. White, E. A. Fisch, and U. W. Pooch. Cooperating security managers: A peer-based intrusion detection system. *IEEE Network*, 7:20–23, February 1999.