AN OPEN AND SECURE INFRASTRUCTURE FOR DISTRIBUTED INTRUSION DETECTION SENSORS*

Marko Jahnke
Research Establishment for Applied Science (FGAN)
Research Institute for Communication, Information Processing and Ergonomics (FKIE)
Neuenahrer Str. 20, D-53343 Wachtberg, Germany
E-mail: jahnke@fgan.de

ABSTRACT

Although a few distributed Intrusion Detection Systems (IDS) allow integration of 3rd party sensors and analyzing modules, it is rarely possible to deploy arbitrary programs or processes within a distributed IDS architecture. Even if 3rd party components may be integrated, they are not protected against attacks (beside their own capabilities), although these issues are nowadays very important within heterogeneous high-assurance networks.

This paper describes a universal, secure and extensible infrastructure for distributed sensor applications, primarily focussed on, but not limited to IDS. A full modular design allows easy adaptation and integration of sensor data preprocessing, analyzing and storage components as well as response mechanisms and communication protocols. Additionally, a generic sensor adapter design allows virtually any sort of external program to be deployed as a sensor, protected by the infrastructure against process termination and unauthorized configuration tampering.

The described prototypical instantiation implements a hierarchical IDS architecture and currently supports the IETF conforming IDMEF data format (Intrusion Detection Message Exchange Format), transmitted over IAP (Intrusion Alert Protocol). Adapters for the "snort" network sensor and the "logsurfer" logfile analyzer as well as for other command-line tools have been implemented.

1. INTRODUCTION

For several tasks, it is necessary to have distributed sensors at some places in a network. *Sensors* gather information which is needed to describe state parameters of a system. An integrated analyzer stage examines the sensor data in order to determine if a previously defined condition is fulfilled, e.g. if a certain threshold for the sensor data is exceeded. In this case, the analyzer generates a so-called *event*, or, in terms of intrusion detection, an *alert*. Thus, from a more general point of view, events are equivalent to sensor data on a higher abstraction level.

The majority of distributed IDS has chosen a centralized architecture with different types of sensors, where local entities (frequently called *agents*) collect the sensor events within one host system, preprocess it, and send it over the network to a central command-and-control entity (*console* or *manager*) which then analyzes, stores and displays the received data. In other systems, like in AAFID [17], so-called *autonomous agents* can additionally include analyzer capabilities. If they primarily communicate with each other, rather than with a central controlling entity, they are called *cooperating agents*.

A few systems, like EMERALD [13], use an open modular design, with an API for integrating sensors and analyzing components. The STAT project provides an infrastructure (MetaSTAT [18]) with on-line configurable sensors. Generally, deploying arbitrary programs or processes for sensor data acquisition and protection services for 3rd party sensors are not foreseen.

This paper describes our approach for an open infrastructure for arbitrary distributed sensors. Due to its universal character, it may be used as a technical basis (or "toolbox") for different IDS architectures. Section 2 of this paper lists the design requirements for a distributed IDS.

* Published in: *Proceedings of the NATO RCMCIS'02*.

In section 3, the necessary abstract components and their instantiation in a hierarchical IDS architecture are described. In section 4, the different requirements for the deployed communication protocols and possible solutions are outlined. Section 5 describes a generic sensor adapter design which can easily be adjusted for virtually any sort of program or process to act as a sensor. In section 6, there are some basic techniques shown to protect the architecture components against possible attacks. Section 7 describes a prototypical implementation of a hierarchical IDS architecture with both host based and network based sensors. Finally, there are some initial results and possible future work presented.

2. DESIGN REQUIREMENTS

Before designing and implementing an IDS infrastructure, it is necessary to collect all relevant requirements. As described in [5], and furthermore in [17], the following general requirements for (distributed) IDS exist:

(R1) *Continuous Running.* The IDS must run continuously without human supervision.

(R2) *Fault Tolerance.* The IDS must be able to recover from hardware and software system crashes.

(R3) *Resist subversion.* The IDS must be able to detect attacks against itself.

(R4) *Minimal Overhead.* The IDS must not have an effect on the performance of the system it protects.

(R5) *Configurability.* The IDS must be able to enforce a changed security policy.

(R6) *Adaptability.* The IDS must be able to be adapted for changes in system or user behaviour over time.

(R7) *Scalability.* The IDS must be able to protect even a large number of systems.

(R8) *Graceful Degradation of Service.* The overall IDS must not be affected if some components stop working for any reason.

(R9) *Dynamic Reconfiguration.* The IDS must be able to be reconfigured without restarting the whole system.

Beside these basic requirements, we have additional ones concerning the implementation:

(R10) *Deploy Standard Protocols and Data Formats.* To be consistent with other IDS components, it is necessary to use standardized or commonly agreed communication protocols and data formats.

(R11) *Short Development Cycle.* Due to limited development resources, a fast prototype development process was required. We do not have the time for developing (theoretical) frameworks over years.

(R12) *Deploy COTS products where possible.* In several cases, it is not necessary to reinvent the wheel, so the possibility to integrate 3rd party components was essential.

(R13) *Platform Independency.* Due to applicability to protect existing desktop and server systems, dependencies of the operating system or dedicated hardware have to be avoided.

(R14) *Modularization and Component Reusability.* The IDS components have to be generic enough to be adapted for different applications, architectures and other environmental conditions.

3. INFRASTRUCTURE COMPONENTS

This section describes a framework of abstract components which are needed to build a sensor infrastructure. After that, it is shown how a concrete architecture can be instantiated. The example is focussed on a hierarchical architecture, but the components could also form different variations, like cooperating agents.

3.1 Abstract components

A sensor infrastructure consists of the following components:

- *Sensor*
  A sensor is a program or a process that collects or generates measurement data. This paper is only focussed on event generating sensors which feed their output into the distributed sensor system. In

reality, a more or less complex classification process has to be performed inside the sensor component.

- *Sensor Adapter*
  For every sensor, a dedicated adapter is needed. It has two main objectives: The first is to control the execution of the sensor program or process, and the second is to translate the sensor output into the internally used data model for sensor events. The translated events are then submitted to an event message dispatcher.
- *Message Dispatcher*
  Dispatchers receive event messages from inbound channels and distribute them to event processing units or to outbound channels for further transportation. Additionally they are able to send and to receive control messages, to interprete them and react accordingly.
- *Communication Channels*
  There are basically two types of communication channels, one for event message submission, and one for control messages. Their nature is discussed in section 4.
- *Event Message Processor*
  To process event messages, additional components are required. Like outbound communication channels, they are event message "consumers". There are several possibilities for processing events: The most important processors are *event analyzers* which examine incoming event messages and classify them with respect to a potentially critical system state. Other types of event processing units are *databases*; they collect and store incoming events in a suitable format for later inspection. *Event filters* collect messages and produce some sort of output, like events with modified priorities, summarizing events or other forms of "meta events". This output is then again fed into a message dispatcher.
- *Response Units*
  The responder is able to perform specified internal or external actions if instructed by a message dispatcher or an analyzer unit. These actions could not only be some sort of administrator notification but even active system or network reconfiguration.

3.2 A Hierarchical IDS Architecture

One commonly used architecture for distributed IDS is hierarchical. As a proof-of-concept, we decided to implement this architecture, based on the abstract components described in the last section.

- *Agent*
  An Agent includes all processes which are needed to establish an event message flow from the sensors on a single host to the central command-and-control entity. Beside a number of sensors and sensor adapters, a certain kind of message dispatcher is needed. This *communication client* is used for establishing and maintaining the network connection to the controlling entity. Additionally, it controls the different sensor adapters which are attached to it. If a new sensor event is received by one of the adapters, it is transmitted over the network to the console. Even filtering events through local preprocessing units is possible. Furthermore, the agents listen for incoming connections from the console server, on which command and control messages are exchanged.
- *Console*
  The collection of the components which form a central command-and-control entity is called the console. Again, a message dispatcher is needed which we call the *communication server*. It listens for incoming event message connection requests from distributed agents. After a successful authorization procedure, the server now listens on every established connection for incoming event messages. If a message arrives, it is fed into other console components. The communication server is also able to initiate a control connection to each agent. An event analyzer component examines incoming events and classifies the resulting system state. If the system state is classified as critical, a new event message with a higher priority is generated and the responder process is activated. In most cases,

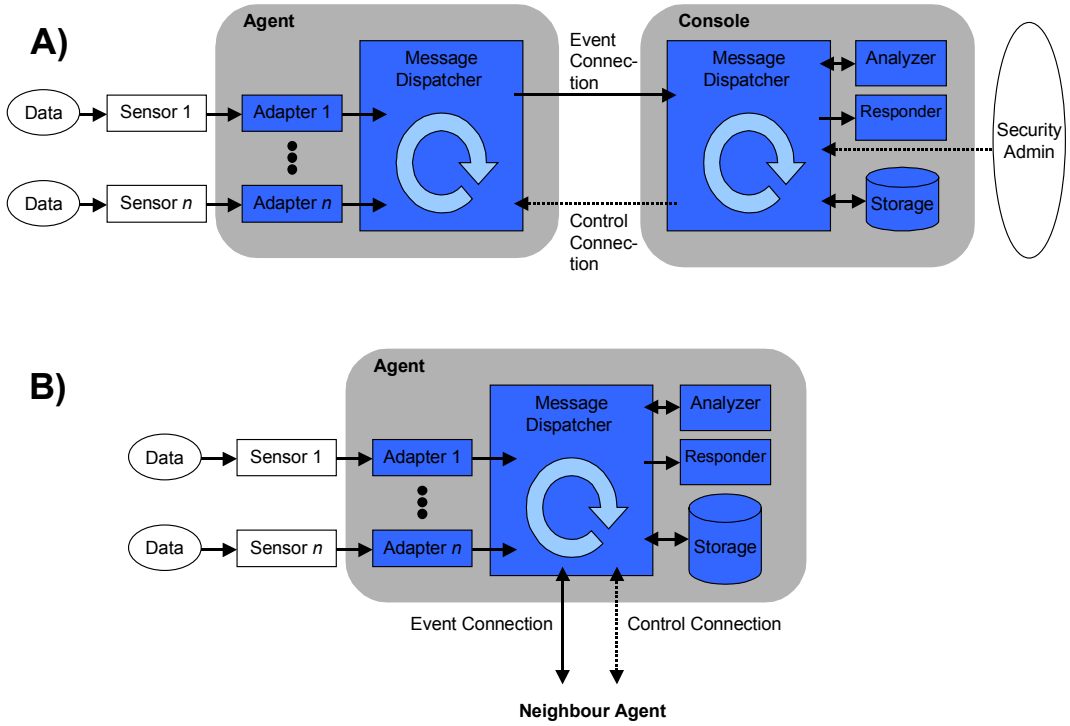

Figure 1: Two possible IDS architectures as instantiations of the proposed components: A) Hierarchical IDS, B) Cooperating Agents.

previously detected event messages have to be taken into account, so the analyzer needs to have access to the event database.

The concrete instantiation of the abstract components in a hierarchical IDS is illustrated in figure 1 A). Figure 1 B) shows an alternative architecture with autonomous client components where a centralized controlling entity is not included; this is not further looked at within this paper.

4. COMPONENT COMMUNICATION

There are two basic types of component communication involved in the described infrastructure: *event messages* and *control messages*. The former represent the occurrence of a potentially security-critical state of the system, while the latter are necessary for command-and-control tasks. Both types of messages might need to be transmitted over several different communication mechanisms, between different processes on one host and between hosts over a network connection.

4.1 Event Messages

Primarily, event messages are generated by the sensors, but generally all other IDS components shall be able to generate event messages as well, e.g in case of a system failure. The internal representation of the events may differ between components of the architecture, but it has to be based on one single data model to allow translation from one format to another. Additional requirements for transmitting event messages over a network are authentication and encryption services. Authentication is needed due to the possibility of man-in-the-middle-attacks, and encryption is necessary to avoid that possible attackers were informed about their detection status.

Due to interoperability reasons, a widely accepted data model for event messages shall be used, such as the recommendations of the IDWG (Intrusion Detection exchange format Working Group) of the IETF. Their data model is called IDMEF (Intrusion Detection Message Exchange Format [6]) and is defined as an XML Document Type Definition (DTD). In addition to event messages, so-called "Heartbeats" can be created to indicate to the receiving component that the sender is still alive.

To transport XML formatted IDMEF messages over the network, the IAP (Intrusion Alert Protocol [11]) has been defined. It provides a reliable service; the reception of a

message has to be confirmed after checking the content. Before establishing a communication channel, it is necessary to negotiate profiles for the communication capabilities. If they cannot be negotiated, the channel is not established. On a single IAP channel, a peer is exclusively in the role of a sender or of a receiver. For authentication and encryption services on the underlying TCP layer, IAP uses TLS (Transport Layer Security [8]). A more extensive discussion of the above protocols and data formats is far beyond the scope of this paper.

4.2 Control Messages

Control messages are needed to transmit component controlling information, such as

- Status requests and status reports,
- Start, stop and restart commands for processes, and
- Reconfiguration requests and acknowledgements.

The requirements for a protocol to transmit control messages are different from the event message protocol. Control messages are typically transported in the opposite direction of the event message flow. Since the protocols for event messages as described above are not specified for any other purpose, separate connections have to be established. The following requirements exist:

1. *Reliability of message transport*
   Reception of messages shall be acknowledged.
2. *Security services*
   The payload of the protocol shall be encrypted and authenticated due to the reasons mentioned in the previous section.
3. *Non-persisting connection*
   The connection shall only be established if it is required, because during normal operation control messages are not needed. ("Heartbeat" messages as defined in the IDMEF model, are sufficient to show that an entity is working properly).
4. *Small protocol overhead*
   Last, but not least, the processing overhead shall not be too big, because the event message protocols already contain a certain amount of overhead.

While looking for a protocol that meets the requirements, we identified different alternatives. The Simple Network Management Protocol (SNMP [4]) is frequently used for control purposes. It also provides an authentication service. But since reconfiguration requests for arbitrary sensor programs can include the content of whole configuration files (see section 5), a connectionless protocol is not suitable for handling the transport. A few more protocols (e.g. HTTP/S, BEEP [15]) would be able, but their protocol overhead is obviously too large, especially if the control connection has to be re-established for every message.

Proprietarily defined text messages, transmitted over a TLS-secured TCP connection, would provide an easy way to meet the requirements. The processing overhead is relatively small, especially if a TLS session can be reused. An extensible description of the syntax and the semantics of the protocol is required.

5. SENSOR ADAPTERS

One of the key issues of the described infrastructure is the easy integration of different sensor programs or processes. Due to the fact that sensors may produce output in arbitrary formats, it is necessary to have a specific translation module for each sensor. This translation module shall transform the events into one strictly defined format that is used for internal representation. Hence, necessary information has to be extracted from the sensor output, and a new according event representation has to be generated. For being able to transmit event messages over the network or to store events within a file, another translation module may be required. In the case of IDMEF events, the internal representation has to be translated to an XML formatted text, according to the DTD. The translation process within a sensor agent is illustrated in figure 2.

To manage arbitrary sensor programs, it is necessary to handle the program executables as well as all required configuration files. If a reconfiguration of a sensor becomes necessary, the controlling entity shall send a new

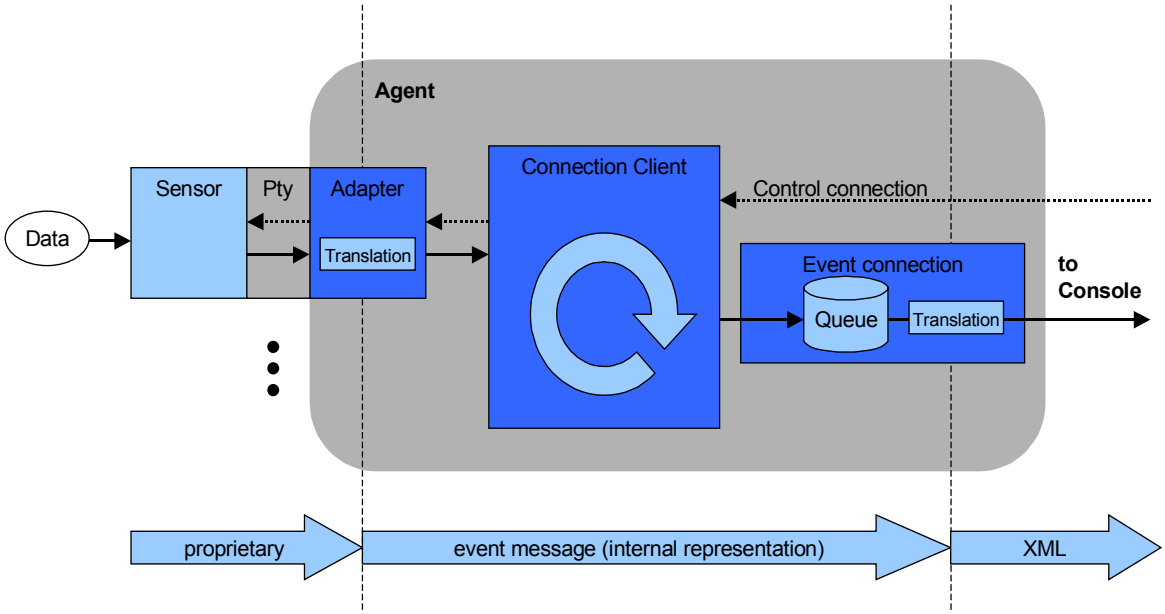

Figure 2: Translation of sensor data to XML-based IDMEF representation in a sensor agent.

executable or configuration data to the local message dispatcher, using the control message connection, and the adapter shall put the content to the correct file. After that, the sensor needs to be restarted. Implementing this requires an according extension of the control protocol.

6. BASIC IDS SELF-PROTECTION

A system protected by an IDS can only be as safe as the IDS itself. Therefore survivability issues are very important for a distributed IDS which aims at protecting potentially large networks.

Every component of the proposed infrastructure is supposed to create a high-priority event message if any sort of failure occurs. These event messages have to be submitted to the controlling entity which informs the security administrator or triggers countermeasures, if applicable. To avoid false positives (like for crashing sensors), the event messages have to be analyzed prior to a reaction.

6.1 Protecting IDS Connections

A possible threat against local message dispatchers concerns their network connection to other IDS entities. Attacks against communication integrity, confidentiality and authenticity are covered by a secure communication protocol like TLS [8]. We obtain a different situation if an attacker simply terminates the event or control message communication between the different components (e.g. by sending RST packets). This may also happen if there are other problems with the underlying network links. Therefore, the controlling entity has to decide if there is a potential denial-of-service attack or not.

Every dispatcher that has to submit event messages to another entity shall store its unsent messages on the local filesystem in case of a broken link (either due to a network error or to an active connection termination by an attacker) for a later state recovery. The affected message dispatcher has to re-establish a terminated connection immediately, if possible on an alternative link (different route, different port etc.). This is due to two reasons: On one hand, it is more difficult for the attacker to detect and destroy the alternative channel. On the other hand, if no re-connection can be established within a given time period, the controlling entity can be almost sure that there is an attack rather than a network problem.

6.2 Protecting IDS Components

There are several ways for an attacker to corrupt the IDS components on a penetrated host. Generally, there are two basic types of

attacks: *Termination* identifies the case in which the attacker terminates the execution of a component process to avoid being detected. *Blinding* means that the components keep on working, but do not have the possibility to detect the attacker.

6.2.1 Process Termination

The protection of a component against termination is possible if the running status of the component process is being checked continuously by another component. This component is called *observer*. If the observer detects a process termination, it sends an according high priority event message to the controlling entity that triggers an according reaction. It is not necessary that one component observes all other components on a single host, e.g. a sensor adapter might be observing the execution of its sensor whereas the local message dispatcher observes the attached adapters.

6.2.2 Blinding

Depending on the purpose and the nature of an IDS component, there might be a high number of possibilities to corrupt it without being notified. Especially for 3rd party sensors, there are no generic countermeasures; these attacks have to be prevented by the sensor itself or by external mechanisms.

A simple but effective way to blind a component is to tamper its configuration (or even its execution code) to make the IDS believe that everything works fine. For components which are restarted frequently, or which read their runtime configuration from the local storage device, configuration tampering can be detected by observing at least the following important file properties:

- Content checksum (digest)
- File ownership
- Modification time
- Access permissions.

Although external tools like Tripwire [2] or AIDE [12] perform this task very efficiently, their local configuration or database may also be tampered by the attacker. Therefore, we propose the following strategy:

When executing a component on a remote host for the first time or after a reconfiguration (within a clean environment), the above properties of the program executables (IDS components and 3rd party sensors) as well as of all affected configuration and data files are determined and submitted to the controlling entity which stores them within its clean and tamper-proof environment. During normal operation, on system startup and at regular intervals, these values are determined again and compared with the stored values, so unauthorized modifications can be detected reliably.

It has to be mentioned, that also runtime process manipulation may be misused for blinding IDS components, but detecting this needs further research.

7. IMPLEMENTATION

We implemented a hierarchical IDS infrastructure as described in section 3.2. When choosing the impementation platform, we had to take care of requirements (R13) and (R14) of section 2. Both constraints would have been met easily by a Java based approach. But Java has a few disadvantages, like dependency of a local runtime environment and properties of malicious mobile code. Even the popular Perl language has also disadvantages, as noted in [17]. Therefore, we decided to use C++, strictly using object oriented design techniques and a few platform-dependent implementation distinctions.

The following subsections describe implementation details, with respect to the requirements (R1) - (R12) of section 2.

7.1 Abstract Connection Framework

For easy maintaining and adjustments in the several protocol layers, we developed a multi-layer network abstraction framework. This framework generally implements the abstract representation of a *connection*, each with four basic operations: `open()`, `close()`, `read()` and `write()`. This is applicable for almost every class of connections one might

think of, like connection to files, databases, Unix pipes, shared memory, semaphores or network sockets. When creating a different class of connections for each OSI network layer, it is obvious that a higher layer connection operation makes use of the according operation of the next layer connection class below it. The framework has been implemented as a C++ class library. Each connection class on a higher network layer has been derived from a class on a lower level.

7.2 Sensor Adapters

To be consistent with the rest of the architecture, we modelled the sensor adapters as abstract connections, as described in the last section. In Unix and Unix-like environments, *pipe* connections are commonly used to control other programs, to feed their standard input channels and to read everything that comes from their standard output and error channels. Therefore, for using ordinary programs, sensor adapters are a certain type of pipe connections, with an integrated translation stage for the transformation of sensor output data to the internal event representation. For extracting the necessary information from the sensor program output, we use configurable standard *POSIX regular expressions*. Hence, requirements (R5) and (R6) are met.

In our current prototype, we have implemented adapters for the "snort" network IDS sensor [14], for the "logsurfer" log file analyzer [1] and other command-line based tools. The configuration management for the sensors is not yet supported.

7.3 Sensor Agent

A sensor agent consists of several sensor adapters and a message dispatcher. The dispatcher manages several connections and watches for state changes of their file descriptors by using the `select()` system call. Thus, we have almost no computational overhead for the distribution of messages (R4). An agent runs continuously without administrator interception, even if the connection to an according console cannot be established (R1). In this case, the event messages are held within a message queue that also can be written onto the local storage device (see section 6.1). When restarting, the event queue can then be recovered (R2).

The message dispatcher also observes the running status of all deployed sensors and generates an alert if necessary (see section 6.2). Additionally it determines the required properties of all affected files of the runtime environment and reports it to the console. These protection mechanisms partially meet (R3).

7.4 Communication Channels

The framework as described in section 7.1 was extended to handle the protocols which are conforming to the IETF recommendations for intrusion detection event transmission as described in section 4. This meets our requirement (R10). As described in section 6, an additional buffer for the event message objects has been integrated into the IDMEF connection class. Currently the IAP protocol is used for the transport of the event XML representation. The IAP implementation was originally based on the "libiap" part of "SnortNet" [10]. Every message has to be confirmed by the receiving peer. Only if the reception has been confirmed by the console, the message is removed of the agent's queue and will not be re-submitted.

The internal representation of event messages are C++ objects which reflect the IDMEF data model. For translation to the XML format and back, the "libidmef" library [16] from Silicon Defense, Inc. is used. The event message objects are used as parameters for the `read()` and `write()` operations of the IDMEF connection class; the according translation process is integrated. Hence, the programmer of the agents and console only deals with C++ objects and does not necessarily need to know how they are encoded, transmitted or decoded.

Control messages are simply encoded as proprietary Text messages which were sent over a TLS secured TCP connection. E.g. if an agent shall be instructed to report its current execution status, an according connection to the clients control message port is established by the console, the request command string is transmitted, and, after the client has submitted the status report, the connection is closed

again. Since the TLS context is shared between all connections between two peers, the control communication is very efficient (R4) without security loss.

7.5 Console Server

Our console currently consists of a message dispatcher, which additionally is able to process user commands, of a screen-based real-time console GUI and of a database module. The number of connected agents is only limited by the maximum number of allowed file descriptors, so the implementation supports scalability and therefore meets requirement (R7). The database module was implemented as an abstract database connection for the "MySQL" [3] database within our framework. Using the snort database structure, we are able to store the events submitted by several sensors and agents. Since we have chosen this database structure, it is possible to use a graphical user interface like "ACID" (Analysis Console for Intrusion detection Databases [7]), which can be used to present the results to the security administrator. Deploying these components meets requirement (R12). The first working prototype described in this paper has been implemented within 6 months, so requirement (R11) is obviously met.

8. INITIAL RESULTS AND FUTURE WORK

As a primary result of the open IDS sensor adapter design, we now are able to collect and store all relevant audit data in a centralized management console. The data collection is presented to the security administrator; this leads immediately to a shorter time period needed for manual inspection, because only one information source has to be analyzed instead of several databases and log files. Also, we obtain greater possibilities for concluding on possible correlations between security events coming from different sensors. Even applications which do not include dedicated audit log interfaces can now be connected to the distributed IDS.

The presented countermeasures against some common attack techniques for corrupting the distributed IDS components have been sucessfully tested. For an advanced attacker it was formerly possible to reconfigure a sensor silently in order to avoid being detected (e.g. deletion of configuration options which are necessary to detect root logins in the system log files). Due to our integrity observation of the operational environment, every unauthorized modification is now notified and an appropriate alert message is sent to the management console. The malicious termination of the network communication channel between agent and console leads now to the generation of alert messages on both entities, and the data which was supposed to be transmitted to the console is buffered on the agent´s storage device until a retransmission is possible.

By defining strictly used interfaces for abstract connections, there is the possibility to add more event processing or transmitting modules in both agents and console. Some future plans to extend the architecture to obtain a more fault-tolerant and attack-resistant IDS are as follows:

- *Integrate IDXP communication*
  The IAP [11] protocol is no longer maintained by the IDWG as an internet draft, but has been replaced by IDXP (Intrusion Detection Exchange Protocol [9]), which is a profile of BEEP [15]. For interoperability reasons, this will be integrated in our system.
- *IDS Self-Protection*
  As survivability of the IDS itself has become a raising issue, our main focus will be on developing new self-protection techniques and on integrating them into the infrastructure. As one example, an agent may use alternative communication channels in case of an attack against the connection to the console. (maybe to another, still properly working agent, or to an redundant console entity). Another example is the detection of IDS process manipulation during runtime.
- *Event Correlation Pre-Analyzer*
  For our next prototype, we plan to implement a simple analyzer stage, that detects predefined temporal event

correlations as an agent module. A trivial sample application is the detection of a brute-force password attack by the network sensor and a consequent successful login message found by a logfile analyzer.

9. SUMMARY

This paper has described our approach to design and implement a universal and secure intrusion detection sensor infrastructure as a basis for future developments of IDS self-protection techniques. After listing up all requirements for a secure distributed IDS, several generic IDS modules and communication components have been presented, as well as their concrete instantiation in a hierarchical architecture.

The possibility to add arbitrary 3rd party IDS sensors is given by a generic sensor adapter design which controls the sensor execution, manages its configuration and translates its output into a commonly used data format. Basic integrity protection techniques for the architecture components and their communication channels against possible attacks have been outlined.

A prototypical implementation of the architecture and the protection mechanisms has been discussed. Finally, initial results of the work have been presented, as well as possible directions for future research activities.

ACKNOWLEDGEMENTS

The author would like to thank M. Bussmann, C. Riechmann and R. Coolen for lots of valuable comments and S. Henkel for implementation workings.

BIBLIOGRAPHY

[1] Logsurfer Documentation Version 1.5. http://www.cert.dfn.de/eng/logsurf/, 1997.

[2] Tripwire Download Page. http://www.tripwire.org/downloads/index.php, 2001.

[3] The MySQL database. http://www.mysql.org/, 2002.

[4] J. D. Case, M. Fedor, M. L. Schoffstall, and J. Davin. RFC 1067: Simple Network Management Protocol. http://www.ietf.org/rfc/rfc1067.txt, August 1988.

[5] Mark Crosbie and Eugene H. Spafford. Active Defense of a computer system using autonomous agents. Technical report, The COAST Group, Department of Computer Science, Purdue University, West Lafayette, IN, Feb. 1995.

[6] D. Curry and H. Debar. Intrusion Detection Message Exchange Format - Data Model and Extensible Markup Language (XML) Document Type Definition. IETF Internet Draft draft-ietf-idwg-idmef-xml-03.txt, February 2002. IETF IDWG.

[7] R. Danyliw. ACID: Analysis Console for Intrusion Detection Databases. http://acidlab.sourceforge.net/, 2002.

[8] T. Dierks and C. Allen. RFC 2246: The TLS Protocol Version 1.0. http://www.ietf.org/rfc/rfc2246.txt, January 1999.

[9] B. Feinstein, G. Matthews, and J. White. The Intrusion Detection Exchange Protocol. IETF Internet Draft draft-ietf-idwg-beep-idxp-03.txt, September 2001. IETF IDWG.

[10] F. Yarochkin. 'SnortNet' - A Distributed Intrusion Detection System. http://snortnet.scorpions.net/, June 2000.

[11] H. P. Gupta, T. Buchheim, B. Feinstein, G. Matthews, and R. Pollock. IAP: Intrusion Alert Protocol. IETF Internet Draft draft-ietf-idwg-iap-05.txt, March 2001. IETF IDWG.

[12] R. Lehti. AIDE Download Page. http://www.cs.tut.fi/ rammer/aide.html, 2001.

[13] Phillip A. Porras and Peter G. Neumann. EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances. In *Proceedings 20th NIST*-NCSC National Information Systems Security Conference, pages 353–365, 1997.

[14] Martin Roesch. Snort - Lightweight Intrusion Detection for Networks. In *USENIX LISA '99 conference*, November 1999.

[15] M. Rose. RFC 3080: The Blocks Extensible Exchange Protocol Core. http://www.ietf.org/rfc/rfc3080.txt, January 1999.

[16] Silicon Defense Inc. LIBIDMEF download page. http://www.silicondefense.com/ idwg/libidmef/index.htm, 2002.

[17] Eugene H. Spafford and Diego Zamboni. Intrusion detection using autonomous agents. *Computer Networks*, 34:547–570, 2000.

[18] G. Vigna, R. A. Kemmerer, and P. Blix. Designing a web of highly-configurable intrusion detection sensors. In *Proceedings of the Workshop on Recent Advances in Intrusion Detection (RAID 2001), Davis, CA*, October 2001.