An Architecture for Intrusion Detection using Autonomous Agents[*]

Jai Sundar Balasubramaniyan, Jose Omar Garcia-Fernandez,
David Isacoff, Eugene Spafford, Diego Zamboni[†]
COAST Laboratory
Purdue University
West Lafayette, IN 47907-1398

E-mail: {balasujs,jgarcia,isacoff,spaf,zamboni}@cs.purdue.edu

Abstract

The Intrusion Detection System architectures commonly used in commercial and research systems have a number of problems that limit their configurability, scalability or efficiency. The most common shortcoming in the existing architectures is that they are built around a single monolithic entity that does most of the data collection and processing. In this paper, we review our architecture for a distributed Intrusion Detection System based on multiple independent entities working collectively. We call these entities Autonomous Agents. This approach solves some of the problems previously mentioned. We present the motivation and description of the approach, partial results obtained from an early prototype, a discussion of design and implementation issues, and directions for future work.

1. Background and motivation

We start by introducing some concepts that are used throughout this paper, as well as describing the limitations that we see in existing Intrusion Detection Systems, and why a distributed approach using autonomous agents can help in overcoming those limitations.

1.1. Intrusion Detection

Intrusion detection (ID) is defined [15] as "the problem of identifying individuals who are using a computer system without authorization (i.e., 'crackers') and those who have legitimate access to the system but are abusing their privileges (i.e., the 'insider threat')." For our work, we add to this definition the identification of *attempts* to use a computer system without authorization or to abuse existing privileges. Thus, our definition matches the one given in [8], where an intrusion is defined as "any set of actions that attempt to compromise the integrity, confidentiality, or availability of a resource."

We also use the broad categorization of models of intrusion detection described in [15]:

Misuse detection model: Detection is performed by looking for the exploitation of known weak points in the system, which can be described by a specific pattern or sequence of events or data.

Anomaly detection model: Detection is performed by detecting changes in the patterns of utilization or behavior of the system. This is the type of intrusion detection described in [5]. It is performed by building a model that contains metrics derived from system operation and flagging as intrusive any observations that have a significant deviation from the model.

An Intrusion Detection System (IDS) is a computer program that attempts to perform ID by either misuse or anomaly detection, or a combination of techniques. An IDS should preferably perform its task in real time [15].

IDSs are usually classified [15] as host-based or network-based. Host-based systems base their decisions on information obtained from a single host (usually audit trails), while network-based systems obtain data by monitoring the traffic of information in the network to which the hosts are connected.

Notice that the definition of an IDS does not include preventing the intrusion from occurring, only detecting it and reporting the intrusion to an operator.

[*] Portions of this work were supported by contract MDA904-97-6-0176 from the Maryland Procurement Office, and by sponsors of the COAST Laboratory. This paper was published in Proceedings of the Fourteenth Annual Computer Security Applications Conference.

[†] Main author and contact person for questions concerning this paper.

1.2. Desirable characteristics of an IDS

In [4], the following characteristics are identified as desirable for an IDS:

- It must *run continually* with minimal human supervision.
- It must be *fault tolerant* in the sense that it must be able to recover from system crashes and reinitializations.
- It must *resist subversion*. The IDS must be able to monitor itself and detect if it has been modified by an attacker.
- It must impose a *minimal overhead* on the system where it is running.
- It must be able to be configured according to the security policies of the system that is being monitored.
- It must be able to adapt to changes in system and user behavior over time.

As the number of systems to be monitored increases and the chances of attacks increase we also consider the following characteristics as desirable:

- It must be able to *scale* to monitor a large number of hosts.
- It must provide *graceful degradation of service* in the sense that if some components of the IDS stop working for any reason, the rest of them should be affected as little as possible.
- It must allow *dynamic reconfiguration*, this is, the ability to reconfigur the IDS without having to restart it.

1.3. Limitations of existing IDS

Many of the existing network- and host-based IDSs [9, 8] perform data collection and analysis centrally using a monolithic architecture. By this we mean that the data is collected by a single host, either from audit trails or by monitoring packets in a network, and analyzed by a single module using different techniques. Other IDSs [10, 20] perform distributed data collection (and some preprocessing) by using modules distributed in the hosts that are being monitored, but the collected data is still shipped to a central location where it is analyzed by a monolithic engine. A good review of systems that take both approaches is presented in [15].

There are a number of problems with these architectures:

- The central analyzer is a single point of failure. If an intruder can somehow prevent it from working, the whole network is without protection.
- Scalability is limited. Processing all the information at a single host implies a limit on the size of the network that can be monitored. After that limit the central analyzer becomes unable to keep up with the flow of information. Distributed data collection can also cause problems with excessive data traffic in the network.
- It is difficult to reconfigure or add capabilities to the IDS. Changes and additions are usually done by editing a configuration file, adding an entry to a table or installing a new module.
- Analysis of network data can be flawed. As shown in [18], performing collection of network data in a host other than the one to which the data is destined can provide the attacker the possibility of performing Insertion and Evasion attacks.

Other IDSs have been designed to do distributed collection and analysis of information. A hierarchical system is described in [22], and [27] describes a cooperative system without a central authority. These systems solve some of the problems mentioned.

1.4. Autonomous Agents

A software agent can be defined as [1]:

> . . . a software entity which functions continuously and autonomously in a particular environment . . . able to carry out activities in a flexible and intelligent manner that is responsive to changes in the environment . . . Ideally, an agent that functions continuously . . . would be able to learn from its experience. In addition, we expect an agent that inhabits an environment with other agents and processes to be able to communicate and cooperate with them . . .

In our context, we define an *autonomous agent* (henceforth *agent*) as a software agent that performs a certain security monitoring function at a host.

We term the agents as *autonomous* because they are independently-running entities (i.e., their execution is scheduled only by the operating system, and not by other process). Agents may or may not need data produced by other agents to perform their work, but they are still considered to be autonomous. Additionally, agents may receive high-level control commands from other entities. This high-level control does not interfere with our definition of agent autonomy.

An agent may perform a single very specific function, or may perform more complex activities.

1.4.1. How the use of Autonomous Agents can improve the characteristics of an IDS

Because agents are independently-running entities, they can be added and removed from a system without altering other components, therefore without having to restart the IDS. Furthermore, agents may provide mechanisms for reconfiguring themselves without having to restart. Additionally, agents can be tested on their own before introducing them into a more complex environment. An agent may also be part of a group of agents that perform different simple functions but that can exchange information and derive more complex results than any one of them may be able to obtain on their own.

Thus, we argue that an IDS whose data collection and analysis elements are agents solves all the problems mentioned in Section 1.3:

- If an agent stops working for any reason, one or two things may happen:
  - If the agent is independent and produces results on its own, only its results will be lost.
  - If the data produced by the agent was needed by other agents, that group of agents may be impeded from working properly.

  In any case, the damage is restricted to at most a set of agents.
- By organizing the agents in a hierarchical structure with multiple layers of agents reducing data and reporting it to the upper layers, the system can be made scalable. This idea is proposed in [3] and is also used in [22].
- The ability to start and stop agents independently of each other in the systems that are being monitored adds the possibility of reconfiguring the IDS (or parts of it) without having to restart it. If we need to start collecting a new type of data or monitoring for a new kind of attacks, the appropriate agents can be started without disturbing the ones that are already running. Similarly, agents can be stopped or reconfigured without having to restart the whole IDS.
- If an agent collects network information related to the host where it is running, we reduce the possibility of being subject to insertion and evasion attacks by reducing the number of mismatched assumptions that can be made.

Additionally, using agents as data collection and analysis entities provides the following desirable features:

- Because an agent can be programmed arbitrarily, it can obtain its data from an audit trail, by probing the system where it is running, by capturing packets from a network, or from any other suitable source. Thus, an IDS built from a collection of agents can cross the traditional boundaries between host-based and network-based IDSs.
- Because agents can be stopped and started without disturbing the rest of the IDS, agents can be upgraded to new versions, and as long as their external interface remains unchanged (or backward-compatible), other components need not even know that the agent has been upgraded.
- If agents are implemented as separated processes on a host, each agent can be implemented in the programming language that is best suited for the task that it has to perform.

1.5. Related Work

The ideas of doing distributed intrusion detection and of having different functions performed by different modules of the IDS are not new. The GrIDS project at UC Davis [22] employs modules running in each host to report information to engines that build a graph representation of activity in the network and use it to detect possible intrusions. GrIDS also provides mechanisms to allow third-party security tools to be used as data sources.

The NADIR system [10] performs distributed data collection by employing the existing *service nodes* in Los Alamos National Laboratory's Integrated Computer Network (ICN) to collect audit information, which is then analyzed by a central expert system.

Another approach is presented in [27], in which Cooperative Security Managers (CSM) are employed to perform distributed intrusion detection that does not need a hierarchical organization or a central coordinator. Each CSM performs as a local IDS for the host in which it is running, but can additionally exchange information with other CSMs. The architecture also allows for CSMs to take reactive actions when an intrusion is detected. Unclear aspects are the mechanisms through which CSMs can be updated or reconfigured, and the intrusion detection mechanisms that are used locally by each CSM.

The idea of employing widely distributed elements to perform intrusion detection, by emulating to some extent the biological immune systems, and by giving the system a sense of "self", has also been explored [7].

A distributed sensor system that performs central processing and that can be organized in a hierarchical fashion is described in [11]. This paper proposes a system that is almost identical to the original design of our system as done in [3]. It appeared several years later in the same conference, but [11] has little in the way of detail, and no citations to related work that would enable us to determine how their work may relate to ours.

The EMERALD project [17] proposes a distributed architecture for intrusion detection that employs entities called *service monitors* which are deployed to hosts and perform monitoring functions. They define several layers of monitors for performing data reduction in a hierarchical fashion. Monitors can be programmed to perform any function. The EMERALD project is work in progress, and we expect it to provide some interesting results.

The approach for using Autonomous Agents in ID that was the foundation for our work was proposed in [4, 3]. These papers introduced the idea of lightweight, independent entities operating in concert for detecting anomalous activity, prior to most of the approaches mentioned previously.

2. System architecture

We propose an architecture (which we call AAFID for *Autonomous Agents For Intrusion Detection*) for building IDSs that uses agents as their lowest-level element for data collection and analysis and employs a hierarchical structure to allow for scalability as described in Section 1.4.1.

2.1. Overview

A simple example of an IDS that adheres to the AAFID architecture is shown in Figure 1. This figure shows the three essential components of the architecture: agents, transceivers and monitors. We refer to each one of these components as AAFID entities or simply *entities*, and to the whole IDS constituted by them as an *AAFID system*.

An AAFID system can be distributed over any number of hosts in a network. Each host can contain any number of *agents* that monitor for interesting events occurring in the host. All the agents in a host report their findings to a single *transceiver*. Transceivers are per-host entities that oversee the operation of all the agents running in their host. They exert control over the agents running in that host, and they have the ability to start, to stop and to send configuration commands to agents. They may also perform data reduction on the data received from the agents. Finally, the transceivers report their results to one or more *monitors*. Each monitor oversees the operation of several transceivers. Monitors have access to network-wide data, therefore they are able to perform higher-level correlation and detect intrusions that involve several hosts. Monitors can be organized in a hierarchical fashion such that a monitor may in turn report to a higher-level monitor. Also, a transceiver may report to more than one monitor to provide redundancy and resistance to the failure of one of the monitors. Ultimately, a monitor is responsible for providing information and getting control commands from a user interface. This logical organization, which corresponds to the physical distribution depicted in Figure 1, is shown in Figure 2.

All the components export an API to communicate with each other and with the user.

In the following section we describe each component in greater detail.

2.2. Components of the architecture

2.2.1. Agents

An agent is an independently-running entity that monitors certain aspects of a host, and reports abnormal or “interesting” behavior to the appropriate transceiver. For example, an agent could be looking for a large number of **telnet** connections to a protected host, and consider the occurrence of that event as suspicious. The agent would then generate a report that is sent to the appropriate transceiver. The agent does not have the authority to directly generate an alarm. Usually, a transceiver or a monitor will generate an alarm based on information received from one or more agents. By combining the reports from different agents, transceivers build a picture of the status of their host, and monitors build a picture of the status of the network they are monitoring.

Agents do not communicate directly with each other in the AAFID architecture. Instead, they send all their messages to the transceiver. The transceiver decides what to do with the information based on agent configuration information.

Notice that the architecture does not specify any requirements or limitations for the functionality of an agent. Thus it may be a simple program or a complex software system (for example, an instance of IDIOT [2]). As long as the agent produces its output in the appropriate format and sends it to the transceiver, it can be part of the AAFID system.

Internally, agents are also allowed to perform any functions they need. Some possibilities are:

- Agents may learn or evolve over time using genetic programming, as suggested in [3], or other machine learning techniques.
- Agents may employ techniques to retain state between sessions, allowing them to detect long-term attacks or changes in behavior. Currently, the architecture does not specify any mechanisms for maintaining persistent state.
- Agents could migrate from host to host by combining the AAFID architecture with some existing mobile-agent architecture.

Agents can be written in any programming language. Some functionalities (e.g., reporting and communication) are common to all the agents, and can be provided through shared libraries or similar mechanisms. Thus, a framework implementation can provide most of the tools and mechanisms necessary to make writing new agents a relatively simple task.

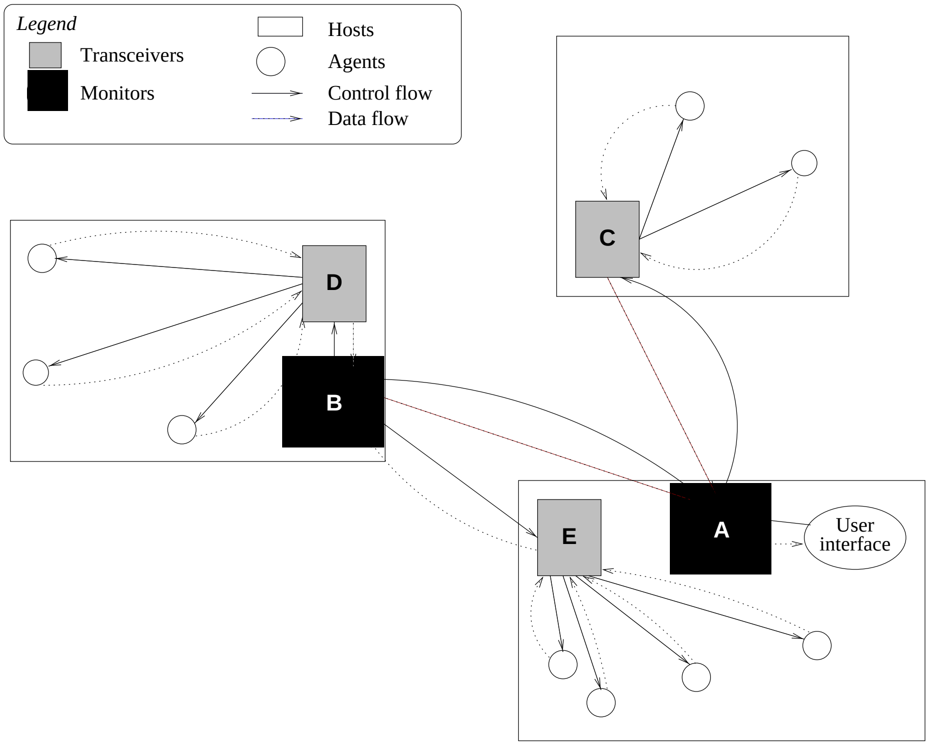

Figure 1. Physical layout of the components in a sample AAFID system, showing agents, transceivers and monitors, as well as the communication and control channels between them.

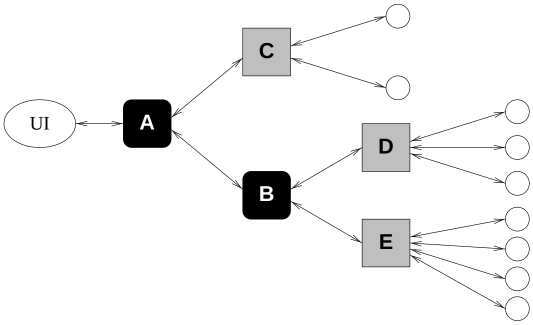

Figure 2. Logic organization of the same AAFID system showing the communication hierarchy of the components. The bidirectional arrows represent both the control and data flow between the entities. Notice that the logical organization is independent of the physical location of the entities in the hosts.

2.2.2. Transceivers

Transceivers are the external communications interface of each host. They have two roles: control and data processing. For a host to be monitored by an AAFID system, there must be a transceiver running on that host.

In its control role, a transceiver performs the following functions:

- Starts and stops agents running in its host. The instructions to start and stop agents can come either from configuration information, from a monitor, or as a response to specific events
- Keeps track of the agents that are running in its host.
- Responds to commands issued by its monitor by providing the appropriate information or performing the

requested actions.

In its data processing role, a transceiver has the following duties:

- Receives reports generated by the agents running in its host.
- Does appropriate processing on the information.
- Distributes information to other agents or to a monitor, as appropriate.

2.2.3. Monitors

Monitors are the highest-level entities in the AAFID architecture. They also have control and data processing roles that are similar to those of the transceivers. The main difference between monitors and transceivers is that a monitor can control entities that are running in several different hosts whereas transceivers only control local agents.

In their data processing role, monitors receive the reduced information from all the transceivers they control, and thus can do higher-level correlations, and detect events that involve several different hosts. Monitors have the capability to detect events that may be unnoticed by the transceivers.

In their control role, monitors can receive instructions from other monitors and they can control transceivers and other monitors. Additionally, monitors have the ability to communicate with a user interface and provide the access point for the whole AAFID system.

If two monitors control the same transceiver, mechanisms have to be employed to ensure consistency of information and behavior. The AAFID architecture does not currently specify the mechanisms for achieving this consistency.

2.2.4. User interfaces

The most complex and feature-full IDS can be useless if it does not have good mechanisms to allow users to interact with and control it. We have not looked in full detail into the user interface problem, although some issues are mentioned in Section 4.4.

The AAFID architecture clearly separates the user interface from the data collection and processing elements. A user interface has to interact with a monitor and it has to use the API that the monitor exports to request information and to provide instructions.

This separation allows different user interface implementations to be used with an AAFID system. For example, a Graphical User Interface (GUI) could be used to provide interactive access to the IDS, while a command-line based interface could be used in scripts to automate some maintenance and reporting functions.

2.3. Communication mechanisms

The transmission of messages between entities is a central part of the functionality of an AAFID system. Although the AAFID architecture does not specify which communication mechanisms are to be used, there is a minimum set of characteristics that we consider desirable. A more detailed discussion of the tradeoffs that have to be made can be found in Section 4.1.

We consider the following to be some important points about the communication mechanisms used in an AAFID system:

- Appropriate mechanisms should be used for different communication needs (for example, communication within a host versus inter-host communication).
- The communication mechanisms should be efficient and reliable in the sense that they should not add significantly to the communications load imposed by regular host activities, and provide reasonable expectations of messages getting to their destination quickly and without alterations.
- The communication mechanisms should be secure in the sense that they should be resistant to attempts of rendering it unusable by flooding or overloading, and provide some kind of authentication and confidentiality mechanism.

The topics of secure communications, secure distributed computation and security in autonomous agents have been already studied [6, 12], and possibly some previous work can be used in AAFID implementations.

2.4. Other ideas and possible components

In the course of designing our system architecture, we explored some alternate architectural components. We briefly discuss two such components. These components are not currently part of the AAFID architecture.

2.4.1. The Simple Network Management Protocol (SNMP)

The Simple Network Management Protocol (*SNMP*) [19] is a protocol designed to facilitate the exchange of management information between network devices. The SNMP model comprises a *Network Management System* (NMS) and *Managed Devices*. An SNMP Agent runs in each managed device, and an SNMP Manager operates in the machines from which the network is going to be monitored. The Management Information Base (MIB) is a database that specifies variables that are maintained by the agents, and that the manager can query or set [24].

The SNMP model can be used to implement the AAFID architecture. The transceivers can be implemented as SNMP agents, while the functionality of the monitor can be achieved by the SNMP NMS. The autonomous agents (not to be confused with the SNMP agents) can be given unique identifiers in a specially designed MIB. These object identifiers can provide access to a set of parameters within the MIB whose values represent the state of the autonomous agent and that can be retrieved or set by the NMS. The transceivers could communicate with the monitor by raising SNMP traps. The autonomous agents communicate with the transceiver by setting their corresponding data values.

Using SNMP to implement the AAFID architecture might be an interesting possibility. However, there are a number of significant issues that would have to be further investigated, including security, fault tolerance and ease of extension and deployment of the implementation.

2.4.2. Audit Router

System audit trails are an essential source of information for an IDS. However, there are problems that arise when many different entities try to access them simultaneously. It may be useful to have a mechanism that helps in distributing the information to the entities that need it.

The first and simplest scheme is to pass all the audit records to all the agents. The problem with this scheme is that every agent must process the whole audit trail, which is probably a waste of processing resources.

Another possibility is to embed the agents within a central audit server that passes appropriate records to appropriate agents. A version of this approach has successfully been used in the IDIOT IDS [14, 2]. One problem is that this model only supports the push mechanism of client-server interaction. This means that the server sends events to the agents as they become available. If an agent is not ready to receive events, those events are lost.

We propose the use of another mechanism that uses a central *audit router*. The central audit router maintains a database of agents and the audit classes that they require, and implements support routines such as buffer management. Agents need to register with the router and give it information regarding the types of audit records that they require. When doing so they receive a handle in return. The agent can then simply read from the handle whenever it is ready to receive new events. This technique implements the pull model of computing. The downside is that the audit router becomes more complex because of the need to manage buffers and other associated tasks.

This model could also support the push model by allowing agents to specify a callback function that can be invoked by the audit router when certain types of data are received.

2.5. Disadvantages of the AAFID architecture

We have identified several shortcomings in the AAFID architecture that we propose.

- In their control role, monitors are single points of failure. If a monitor stops working, all the transceivers that it controls stop producing useful information. This can be solved through a hierarchical structure where the failure of a monitor would be noticed by higher-level monitors, and measures would be taken to start a new monitor and examine the situation that caused the original one to fail. Another possibility is to establish redundant monitors that look over the same set of transceivers so that if one of them fails, the other can take over without interrupting its operation.
- If duplicated monitors are used to provide redundancy, mechanisms have to be used to ensure that redundant monitors will keep the same information, will obtain the same results, and will not interfere with the normal operation of the IDS.
- The AAFID architecture currently does not specify access-control mechanisms to allow for different users to have different levels of access to the IDS. This is an issue that will need to be addressed at the monitor, transceiver and agent levels as well as in the user interfaces.
- Detection of intrusions at the monitor level is delayed until all the necessary information gets there from the agents and transceivers. This is a problem common to distributed IDSs.

3. Implementations

We have developed two prototypes based on the AAFID architecture, and we are currently in the process of improving those implementations as well as developing new ones.

3.1. First prototype

The first prototype we built was programmed in a combination of Perl [26], Tcl/Tk [16] and C [13], and was intended as a proof of concept for the architecture. In this implementation, which we call AAFID$_1$, much of the behavior of the components was hard-coded and it was not extremely configurable. It used UDP as the inter-host communication mechanism and Solaris message queues as the intra-host communication mechanism. This prototype allowed us to:

- Show that the AAFID architecture could work for doing distributed detection of anomalous events.

- Gain some experience in writing agents that allowed us to identify important functionality that is needed for all agents.
- Identify some design issues that had to be improved. For example, this first prototype integrates the monitor and the GUI in a single program, which proved to be a limitation because it does not allow to organize monitors in a hierarchical fashion.

3.2. Second prototype

Drawing from the experiences obtained with the first prototype we developed a second one from scratch, which we call AAFID$_2$. This prototype is written exclusively in Perl, which has the advantage of making it easy to port to other architectures at the expense of some performance loss. The main objective of this implementation is to allow for extensive testing of the architecture, therefore emphasis has been made in its ease of use, configurability and extensibility. Some of the major contributions of this new implementation are:

- Increased portability because it is written completely in Perl.
- Implementation of an infrastructure that provides all the base services necessary for developing new entities.
- Definition of an internal API for developing new agents.
- Clear separation of communication mechanism internals and other platform-dependent elements.
- Clear definition of each entity as an object, and of the relationships between the different classes of objects.
- Different execution modes for entities, which facilitate developing, testing and debuggging new entities.
- Definition of an extensible message format that can be extended to represent different types of information.
- Separation of the monitor and the user interface.

This implementation is our current test bed for the architecture and is the one under which we are developing new agents and exploring new communication and data-reduction mechanisms. A much more detailed description of AAFID$_2$ can be found in [21].

3.3. Low-level implementations

The first two prototypes have helped us in refining the architecture as well as identifying needs and problems that have to be solved. However, not much emphasis has yet been placed in performance issues because the prototypes have been mostly implemented in high-level scripting languages such as Perl, which have large memory and CPU footprints. For this reason, as the architecture design starts to stabilize, we have started to work in porting the architecture to lower system levels. In particular, we are currently working on integrating components of the AAFID architecture into the Unix kernel. We are currently working on incorporating additional auditing and monitoring capabilities into the Linux kernel, and will possibly work also with Solaris, BSD/OS and Windows NT.

4. Experiences, comments and design issues

Through the experiences obtained with the design and implementations of the AAFID architecture (see Section 3 and [21]), we have identified a number of issues that should be subject of future work. They are discussed in this section.

4.1. Communication and Scalability

To reduce the overhead imposed by the IDS, the communication mechanisms employed have to be as efficient as possible.

We can classify the communication needs of the IDS in two major groups: intra-host communication (between processes inside a single host) and inter-host communication (between processes running in different hosts).

4.1.1. Intra-host communication

Although the general inter-host communication mechanisms may be used for communication within the same host, we think that intra-host communication should be optimized to make use of the fact that the entities involved are in the same host.

For our IDS architecture, we have identified two main types of intra-host communication that will take place:

- One-to-many communication, as in the case of the transceiver sending a message to several agents.
- Many-to-one communication, as when the agents send information to the transceiver.

Keeping these needs in mind, we have considered several different intra-host communication schemes:

Message queues. This mechanism makes use of the System-V IPC facilities for establishing message queues.

The primary shortcoming of message queues is that the maximum number of messages that can exist in the system at a given time is usually fairly low. A consequence of this resource limitation is that this method is vulnerable to denial-of-service attacks. Additionally, these restrictions place a practical limit on the number of agents that may be running simultaneously in the system.

An advantage of this approach is that this is fairly straightforward to implement and it also provides a mechanism to prioritize the messages so that agent messages can be processed immediately.

This approach was used for the first prototype, but it will most likely not be used in future versions of our system, mainly because of the lack of scalability and its vulnerability to attacks.

Shared memory. This scheme provides an efficient means of sharing data between two processes because data is not actually copied between processes.

This scheme is vulnerable to the same flavor of denial-of-service as the message queues, because the size of the shared memory space is fixed. This scheme also introduces a practical limit on the number of agents that can be run simultaneously in a system.

The shared memory scheme seems to be adequate for one-to-many communication. Given this perspective, a new problem would be how to implement reliable signaling for the transceiver to notify the agents that there is new information that they should receive.

Pipes. In traditional Unix implementations , a pipe is a unidirectional, first-in first-out, unstructured data stream of fixed maximum size [25]. Data is written to the end of the pipe and read from the front of the pipe. The data is removed from the pipe after it is read.

Another common type of pipe is a named pipe or FIFO (first-in, first-out) file. Although they behave very similar to a traditional pipe they differ in the way that they are created and accessed. One major difference is that a FIFO is bidirectional.

Either type of pipe could be used for communication between entities running on the same host. One of the disadvantages of either type of pipe is that there is an internal limit on the volume of data that can be put into a pipe if the reading process does not extract it.

Using pipes as a communication device also has several benefits. They are relatively simple to implement and every modern version of UNIX implements both types of pipes. Because pipes are accessed through file handles they have extreme flexibility in how the components of a distributed system can be used and interconnected.

Independently of the communication scheme used, the IDS needs access control in the communication channels. All the mechanisms mentioned have the ability of performing access control by the following means:

- For message queues and shared memory, the process that sets up the structure establishes its access modes.
- For regular Unix pipes, the pipe is only accessible to the process that creates it and its children.
- For named pipes, the access control is performed by the Unix file permissions.

In conclusion, although some of the schemes show some promise, they still have to be studied carefully before deciding on a particular one. It may be feasible to use more than one different communication scheme depending on the situation. The one-to-many and many-to-one distinctions are clear examples of where this may be possible.

4.1.2. Inter-host communication

The main characteristics that we would like to achieve in a communication scheme for an IDS are performance, reliability and security.

Performance. For the IDS to operate in real time, messages must be delivered as quickly as possible from one part of the system to another, but without overloading the network when many agents are running.

Reliability. Whether the messages sent from one host to another arrive correctly, in order and on time may be a major concern in an IDS, or it may not. The question is: can a single missing message from an agent make a drastic difference, such as the one between an intrusion being detected or not? If we can estimate the maximum amount of lost messages, we might also be able to give an acceptable estimate of the degradation in the service.

Security. Privacy and authentication are important needs for an IDS because some of the messages generated by the IDS may contain sensitive data about the hosts being monitored, and unauthorized entities should not be able to generate messages that are accepted by monitors and transceivers. Usually, cryptography is the solution to both problems. However, cryptography comes at a cost in performance and in overhead imposed to the systems.

The communication protocol used may also be subject to denial-of-service attacks in which an attacker makes it impossible or difficult for messages to get delivered.

The issues mentioned raise a number of questions on whose answers depend the specific approach that should be followed. Currently, we see two possible solutions:

- Use an existing protocol (such as UDP or TCP) in a way that takes into account its weaknesses to provide the functionality we need.
- Design a new protocol with the needs of the IDS in mind. Such a protocol may provide reliable transmission, low overhead, and security mechanisms.

After expressing our concerns regarding inter-host communication, particularly those related to performance and scalability, we come to a more fundamental question: do we really need to worry? In particular, will we ever get to a point when we have thousands of hosts communicating? It can be argued that if an appropriate hierarchical organization is used that may never happen. For example, if the system is structured such that only one subnetwork reports to a single monitor and those monitors in turn report to higher level monitors, the problem may not be as relevant as the previous discussion suggested. In this case, secure and reliable communication would be the priority.

Finally, the level of efficiency required from the communications protocol depends on the level of data reduction that can be achieved. If the data-reduction schemes are such that the amount of information that is actually sent through the network is limited, then efficiency may become a secondary concern.

4.2. Impact on host performance

In the first implementations of the AAFID architecture, all the entities are implemented as separate processes. However, much of the data that are being collected and analyzed are generated in the kernel (for example, user login information, process accounting and network connection establishment). This means that every time a system action has to be logged or analyzed, the information has to be transferred from kernel space to user space, causing a context switch, and increasing the load imposed on the system by the IDS. As the number of agents running on a host increases, the load overhead caused by them may start to impact normal use of the host.

One approach to reduce the overhead caused by the IDS is to write all the components in a compiled language, such as C, instead of in an interpreted language such as Perl. This would reduce the memory and CPU usage, but would not solve the context-switching problem, or the overhead derived from having many separate processes running.

A further step would be to use a language that supports multithreading, and implement each agent as a separate thread instead of a separate process. This may further reduce the per-agent overhead, but still would not address the context-switching problem.

The lowest level that we could achieve would be to integrate some of the components in the Unix kernel. Doing this would have several benefits:

- A context switch is prevented, because the agent would be running within the kernel itself.
- The information is registered and processed very close to where it is produced, reducing the possibility of it being modified by an attacker before it gets to the agent.
- It becomes harder for an intruder to tamper with the agents.

The transceivers could also be built into the kernel. This way, the data would never have to be transferred outside the kernel until the transceiver decides to send them to a monitor for further processing, or for making a notification.

The approach just described also has the following disadvantages:

- Building entities as kernel components essentially destroys the portability of the agents.
- An entity that misbehaves can do much more damage if it is running in the kernel.
- Entities in the kernel can have a large impact in the host behavior by slowing down fundamental operations (e.g. accesses to disk, memory and kernel data structures) or by disrupting timing in critical low-level operations (such as disk accesses).
- The resources that are available for entities in the kernel are very limited and may be insufficient for performing useful actions. For example, an agent that monitors IP packets may need a large amount of memory to be able to to keep enough state information to monitor all of the events necessary to detect a SYN-flood attack. Preliminary work indicates that it is probably not possible to develop distinct independent kernel agents that will perform any complex tasks unless they are tightly coupled into the kernel code itself.

4.3. Data processing and reduction

To make the agents as lightweight as possible, they should be little more than a forwarding element that sends data to the transceiver, which in turn merges the data coming from all the agents and forwards them to the appropriate monitor where everything is processed and the appropriate actions are taken. However, this technique can create a high amount of network traffic, which limits the scalability of the system.

The counterpart is to move computation load from the monitor to the transceivers and agents. This can be taken to the extreme of making each transceiver a local ID system on its own which communicates to the monitors as part of a larger global ID system. Unfortunately, this has an impact on the hosts being monitored because local computation will take away computing cycles from the real applications of the hosts.

A related problem is to decide where the state of the IDS is kept. In the centralized approach, all state is kept in the central monitor. Therefore, if that host is taken down or somehow compromised or destroyed, the state of all the hosts that depend on that monitor may be lost. On the other hand, if each host has its own processing engine, the state information is distributed in different hosts, making its complete loss much more difficult. The disadvantage is that building a consistent picture of the state of the whole IDS becomes more difficult.

We think that the best approach is to try to find a balance between the two extremes. More detailed performance studies may help in making a decision about how the computational load can be distributed between the agents, transceivers, and monitors to maximize the throughput and scalability of the system without imposing and excessive load on the hosts and the network.

4.4. User interface

Any IDS can be rendered useless if it does not have good mechanisms to allow users to control and monitor it. In our case, the user interface has to deal with a common problem: how to interface a high-speed, distributed, continuous-running computer system with the human user, which cannot quickly analyze large volumes of data and cannot be on-guard 24 hours a day. It is a problem that involves issues ranging from data formats and storage to GUI design, including communication, security and consistency.

We think that some of the fundamental issues are:

1. What data do the IDS entities need to provide to give the user a clear picture of the system.
2. How to efficiently, reliably and consistently get the information to the user.
3. How to present the information in a useful way. The interface has to be able to provide the user with multiple levels of detail in a manner that is as easy to use as possible.
4. How to allow the user to provide feedback and to control the entities in the system. Ideally, this has to be done in a way that is efficient, reliable, secure, auditable and manageable.
5. How to make the interface responsive. The user will want to be able to immediately see the effects of any changes made and to be told immediately when something of interest happens.
6. How to keep enough state to provide meaningful information to the user.

The issue of user interface is one that we have not studied in detail yet, and it is likely to be considered for future work.

5. Future work

These are some of the specific points we have identified as relevant for future work:

- Developing agents.
- Low-level implementations:
- Communication mechanisms.
- Developing transceivers and monitors.
- Semantics of the communication.
- Data reduction.
- Porting to other platforms.
- Deployment and testing.
- Global administration and configuration.
- Reliability and fault tolerance.

6. Conclusions

We propose an architecture for Intrusion Detection Systems called AAFID, which is based on independent entities called Autonomous Agents for performing distributed data collection and analysis. Centralized analysis is done on a per-host and per-network basis by higher-level entities called Transceivers and Monitors. The architecture allows for computation to be performed (and thus, for Intrusion Detection to happen) at any point where enough information is available.

We have demonstrated the feasibility of this architecture by the implementation of working prototypes.

The AAFID architecture allows data to be collected from multiple sources, thus being able to combine the best characteristics of traditional host-based and network-based IDSs. It apparently also allows to build IDSs that are more resistant to insertion and evasion attacks [18] than existing architectures, although no tests have been performed to support this claim.

Furthermore, the modular characteristics of the architecture allow it to be easily extended, configured and modified, either by adding new components, or by replacing components when they need to be updated. For example, it should be possible to modify the system to produce messages in CIDF format [23].

The AAFID architecture faces many of the problems that have been traditionally in the realm of distributed systems research, such as scalability, performance and security. Tradeoffs between efficiency, resource consumption and security have to be made, and although we may be able to use results from previous research to implement the mechanisms that AAFID needs, finding the appropriate balance in the ID context between the different factors is still an open area for research.

User interface is a big issue for future work. Most of the work that has been done in Intrusion Detection over the last few years focuses on how to perform the detections, but very little has been done in the way of presenting the information to the user, as well as how to allow the user to specify policies such that the IDS can understand and therefore enforce them.

References

[1] J. M. Bradshaw. An introduction to software agents. In J. M. Bradshaw, editor, *Software Agents*, chapter 1, pages 3–46. AAAI Press/The MIT Press, 1997.

[2] M. Crosbie, B. Dole, T. Ellis, I. Krsul, and E. Spafford. IDIOT—users guide. CSD-TR 96-050, COAST Laboratory, Purdue University, 1398 Computer Science Building, West Lafayette, IN 47907-1398, September 1996.

[3] M. Crosbie and E. Spafford. Defending a computer system using autonomous agents. In *Proceedings of the 18th National Information Systems Security Conference*, pages –, Oct 1995.

[4] M. Crosbie and G. Spafford. Active defense of a computer system using autonomous agents. Technical Report 95-008, COAST Group, Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-1398, Feb 1995.

[5] D. E. Denning. An Intrusion-Detection Model. *IEEE Transactions on Software Engineering*, 13(2):222–232, Feb. 1987.

[6] W. M. Farmer, J. D. Guttman, and V. Swarup. Security for mobile agents: Issues and requirements. In *Proceedings of the 19th National Information Systems Security Conference*, volume 2, pages 591–597. National Institute of Standards and Technology, October 1996.

[7] S. Forrest, S. A. Hofmeyr, and A. Somayaji. Computer Immunology. *Communications of the ACM*, 40(10):88–96, Oct. 1997.

[8] R. Heady, G. Luger, A. Maccabe, and M. Servilla. The Architecture of a Network Level Intrusion Detection System. Technical report, University of New Mexico, Department of Computer Science, Aug. 1990.

[9] L. Heberlein, G. Dias, K. Levitt, B. Mukherjee, J. Wood, and D. Wolber. A Network Security Monitor. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 296–304, May 1990.

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

[11] W. Hunteman. Automated information system — (ais) alarm system. In *Proceedings of the 20th National Information Systems Security Conference*. National Institute of Standards and Technology, October 1997.

[12] IEEE Journal on Selected Areas in Communications, May 1989. Special issue on Secure Communications.

[13] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, second edition, 1988.

[14] S. Kumar. *Classification and Detection of Computer Intrusions*. PhD thesis, Purdue University, West Lafayette, IN 47907, 1995.

[15] B. Mukherjee, T. L. Heberlein, and K. N. Levitt. Network intrusion detection. *IEEE Network*, 8(3):26–41, May/June 1994.

[16] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, MA, USA, 1994.

[17] P. A. Porras and P. G. Neumann. EMERALD: Event monitoring enabling responses to anomalous live disturbances. In *Proceedings of the 20th National Information Systems Security Conference*, pages 353–365. National Institute of Standards and Technology, 1997.

[18] T. H. Ptacek and T. N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, Inc., January 1998.

[19] M. Rose. *The Simple Book: an introduction to management of TCP/IP based internets*. Prentice Hall, NJ, 1993.

[20] 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 *Proceedings of the 14th National Computer Security Conference*, pages 167–176, Oct. 1991.

[21] E. Spafford and D. Zamboni. A framework and prototype for a distributed intrusion detection system. Technical Report 98-06, COAST Laboratory, Purdue University, West Lafayette, IN 47907-1398, May 1998.

[22] 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 *Proceedings of the 19th National Information Systems Security Conference*, volume 1, pages 361–370. National Institute of Standards and Technology, October 1996.

[23] S. Staniford-Chen et al. Common intrusion detection framework. WWW page at `http://seclab.cs.ucdavis.edu/cidf/`.

[24] W. R. Stevens. *TCP/IP Illustrated*, volume Volume 1—The Protocols of *Professional Computing Series*. Addison-Wesley, 1994.

[25] U. Vahalia. *UNIX Internals: The New Frontiers*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1996.

[26] L. Wall, T. Christiansen, and R. L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., second edition edition, September 1996.

[27] G. B. White, E. A. Fisch, and U. W. Pooch. Cooperating security managers: A peer-based intrusion detection system. *IEEE Network*, pages 20–23, January/February 1996.