Enabling Automated Detection of Security Events that affect Multiple Administrative Domains

Jed Pickel
Carnegie Mellon University
Information Networking Institute

Roman Danyliw
Carnegie Mellon University
Information Networking Institute

Enabling Automated Detection of Security Events that affect Multiple Administrative Domains

by Jed Pickel and Roman Danyliw

Abstract

Current security event analysis techniques are myopic in focusing only on single administrative domains. With the increasing security dependencies among Internet-connected networks there are few mechanisms in place to share and analyze security event data on a larger scale. Existing information sharing strategies do not scale well, as they rely on human-intensive operations such as telephone calls and email messages. There exists a need for an infrastructure that crosses administrative boundaries and enables automated data sharing, coordination, and response to security events. By providing an architecture and prototype for collecting data originating from multiple administrative domains, this research begins to tackle Internet the security dependency problem using automated technology rather than human effort.

Table of Contents

List of Tables

List of Figures

List of Examples

Preface

When the first nodes of the ARPANET were connected in 1969, researchers quickly realized the power of using computers to share and exchange information. Their emphasis was on defining and implementing protocols that achieved interoperation [R14]. At the time, they may not have realized their work was building a foundation for a new economy; but, they did understand the importance of establishing standard protocols as a prerequisite to growth and progress. While standards enabled the Internet to grow, many commercial vendors saw TCP/IP as a nuisance add-on that had to be glued onto their own proprietary networking solutions [R14].

There are interesting parallels to observe between computing history and emerging trends in the Internet security industry. In the early days of computing there was little interoperability and few data exchange standards; a significant percentage of the available technology was proprietary and inoperable. Since then the industry has evolved from independent, vendor-specific solutions to a mindset of global interoperability where "the network is the computer"®[1]

The Internet security industry today is still in its infancy and we are likely to see history repeat and the industry to grow similar to the Internet. Market dynamics are driving consumers to demand interoperation and open standards. In the current state, security administrators are forced to grow their own tools in order to have any semblance of interoperation across multiple vendor products. Also, there is a growing demand to exchange data and knowledge about security incidents but no technology or standard exists to help automate this process.

The research performed in this thesis intends to address these issues and take a step forward in building standards that can enable a new world of possibilities for the Internet security industry.

1. Why read this thesis?

This thesis covers emerging topics of interest to early adopters and vendors of intrusion detection technology, incident response teams, or any entity interested in exchanging intrusion data.

The research discussed in this thesis is part of an ongoing effort to build Internet-wide

intrusion detection infrastructure. Collaboration across organizations is essential for security. Building a standard intrusion data representation and exchange format will enable "neighborhood watch" capabilities to emerge on the Internet.

2. Intended Audience

We expect readers will have some familiarity with intrusion detection, certificate-based authentication, and a good understanding of the current state of Internet technology.

3. Definitions

There is no standard dictionary for the Internet security industry and there are many widely used terms that have different meaning depending on the context. Terms uses throughout this document that can have more than one meaning are defined in the *Glossary*.

4. Organization

Chapter 1

This chapter sets the stage by exploring the motivations for this research and providing a detailed statement of goals and objectives.

Chapter 2

There are many projects and technologies related to the topics covered in this thesis. This chapter provides a general overview of the these related works.

Chapter 3

Hypothesis and methods of research are discussed within this section to detail the strategy utilized to accomplish the research goals and objectives.

Chapter 4

The overall architecture, requirements, and functionality of the components are discussed.

Chapter 5

The specific implementation of the prototype is examined in this chapter.

Chapter 6

Based on the result of the prototype, this section discusses design decisions and presents a tradeoff analysis.

Chapter 7

This chapter highlights the results of the research, conclusions, the projected next steps, and future direction for the projects started in this thesis.

Appendix

The appendix includes a rich set of information including source code, schema diagrams, and sample configuration files.

Notes

1. Although this is the marketing slogan for SUN Microsystems, it is a very accurate description for the mentality of computing in general today.

Chapter 1. Introduction

Before covering the goals, scope, and assumptions for this research some background information is required to frame the problem and understand the motivations of researching this topic.

1.1. Background

Topics covered in this thesis cross multiple segments of the Internet security discipline. Intrusion detection, incident response, and incident reporting have traditionally been considered separate tasks, but this separation has only made sense because there is little technology to link them together. This section begins to build the case for linking and finding automated ways of interoperation between these separate yet related information security tasks.

1.1.1. Intrusion Detection Technology

The goal of intrusion detection is to positively identify all true attacks and negatively identify all non-attacks [R7]. The general process an intrusion detection system (IDS) takes to accomplish this goal is as follows:

1. receive input data from one or more sources
2. optionally process or normalize data
3. examine data and possibly correlate with previous data to identify known attacks or filter data known not to be an attack
4. optionally store data for later inspection or correlation
5. output data identified as an attack or potential attack (ie. log files, notifying a management console or another intrusion detection system, an instant message or alert to an administrator)

Details of this process, and the current state of the art for intrusion detection practices

and technology are available in the a document authored by the Software Engineering Institute entitled "State of the Practice of Intrusion Detection Technology" [R7].

While current intrusion detection solutions have the ability to successfully identify attacks inside their specific organizations, none have the capability of sending or receiving data outside the *administrative domain* where they are deployed. Also, current products do not have a mechanism by which to automatically report to an incident response team or the attack source.

The notion of sharing security information is not merely limited to Internet security. In the United States, neighborhoods often use an awareness program called a "neighborhood watch" [1] as a mechanism for keeping neighborhoods more physically secure. The basic premise behind this program is that neighbors will communicate amongst themselves and with the proper authority in the event that they witness suspicious activity (e.g. perhaps someone breaking into a house or a strange person walking the streets). Advances in intrusion detection technology and adoption of standards will enable similar interactions and allow collaborative security processes to occur automatically across administrative domains.

1.1.2. Incident Reporting

Although intrusion detection systems do not have native data sharing mechanisms, there exist best practice processes for sharing data during or in the aftermath of an attack. Incident reporting is the process where attack sources, intermediary sites, incident response teams, and other proper authorities are notified of an attack.

> **Excerpt from RFC1281 [R15]**
>
> The Internet is a cooperative venture. The culture and practice in the Internet is to render assistance in security matters to other sites and networks. Each site is expected to notify other sites if it detects a penetration in progress at the other sites, and all sites are expected to help one another respond to security violations. This assistance may include tracing connections, tracking violators and assisting law enforcement efforts.

According to the "Incident Reporting Guidelines" document [R16] from the CERT Coordination Center, the preferred methods for sending incident reports are electronic

mail, telephone hotline, or fax. The numerous problems with these methods are covered in detail later in this chapter.

1.1.3. Computer Security and Incident Response Teams (CSIRTs)

Over the past 12 years, numerous CSIRTs have formed to address the issue of coordination and communication in response to security incidents. Response teams provide a coordinated and organized method of data sharing in their sphere of influence. This coordination may include the detection, prevention, and handling of security incidents; understanding the current state of security; and identifing trends in activity within their constituency. Because the Internet is a cooperative network, there does not exist one entity with the authority or responsibility for it's security. Instead, authority is scattered across logical domains. Table 1-1 highlights a few of the existing response teams in the government, military, university, and corporate sectors.

Table 1-1. Examples of CSIRTs and their constituencies

Response Team	Constituency
AUSCERT	Australia (sites in .au domain)
CERT(R) Coordination Center (CERT/CC)	The Internet
Cisco-PSIRT	Commercial - Cisco Customers
DFN-CERT	German sites
DOD-CERT	Department of Defense systems
Global Integrity (REACT)	Commercial and government customers
OSU-IRT	The Ohio State University
OxCERT Oxford University IT Security Team	Oxford University

Information is shared between response teams formally through conferences and professional organizations such as FIRST (Forum of Incident Response and Security Teams), as well as informally through mailing lists, email messages, and telephone conversations. In their paper "International Infrastructure for Global Security Incident

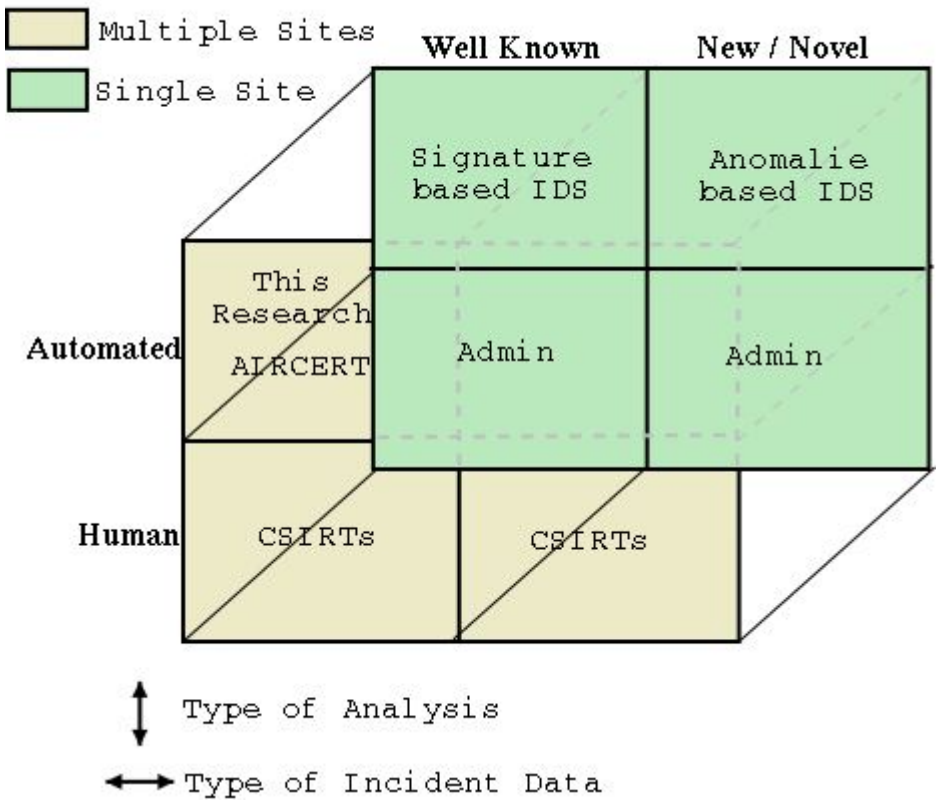
Response", West-Brown and Kossakowski highlight that "we need standards agreement to enable us to make sense of the data we have and to give us a common understanding of the issues" [R17]. This lack of standards is one of the significant challenges incident response teams have faced. With both the volume and complexity of security incidents increasing, these teams must work to find mechanisms to communicate and interoperate effectively. To date there are not any established standards for exchanging data about incidents.

1.1.4. What is an Incident?

There is little consensus on the meaning of the word "incident". For the purposes of this thesis we will consider an incident "a collection of data representing one or more related attacks"[R7]. Such data can originate from an intrusion detection system, a human analysis of a data set, or a report from an external entity.

Different types of incidents have different analysis and exchange requirements. Table 1-2 represents the most appropriate mechanism for analyzing, exchanging, and responding to different types of incidents.

Figure 1-1. Incident Type vs. Handling Matrix



Thinking of incident data in these categories is useful because instances of known activity do not require detailed human analysis; however, new or novel activity require a significant amount of human analysis to understand the technical details, estimate the scope, and determine potential risk. This thesis is primarily targeted at learning how to automatically handle well understood attack activity.

1.1.5. Convergence of Incident Response and Intrusion Detection

Intrusion detection systems and CSIRTs share many of the same goals. Fundamentally,

a CSIRT can be viewed as an intrusion detection system. Both collect and process data to identify trends and events that require attention, and attempt to provide alerts when attacks or suspicious activity is identified. CSIRTs have traditionally been defined by the fact that they do not scale to a large number of incidents, while IDSes typically only have a limited perspective of the data collected. Nevertheless, this distinction is beginning to blur as as intrusion detection systems become more capable of visualizing enterprise level security issues and response teams begin utilizing automated mechanisms to exchange and data.

1.1.6. Internet Security Challenges

The following sub-sections highlight the problems with Internet infrastructure security and the current practices of handling security incidents.

1.1.6.1. Security Awareness

In haste to reap the benefits the Internet has to offer, computers are often connected without considering the risks of electronic attack. Ultimately, this lack of security awareness leads to a global security problem by ensuring there is a constant pool of vulnerable machines available for compromise. CERT states, "It has been our experience that the first time many organizations start thinking about how to handle a computer security incident is *after* an intrusion has occurred" [R18].

1.1.6.2. Security Dependency

Another often overlooked aspect of Internet connectivity is that the security of one machine is direct related to the security of all other machines on the Internet. The past 12 years have brought many examples of widespread high impact attacks affecting large portions of the Internet. Many are familiar with the 1988 Morris Worm[2], the widespread denial of service attacks of 1998[3], the 1999 the Melissa Virus[4], and the distributed denial of service attacks in February of 2000[5]. Each of these incidents clearly highlights the security interdependency concept. A successful infrastructure attack requires exploitation of vulnerability across multiple machines within multiple domains.

Combining the nature of the Internet, a network with no central control or authority, with the fact that current security technologies are focused on protecting only the

perimeter within which they are deployed, the need for an infrastructure to quickly and automatically share data about ongoing attacks becomes apparent. Such infrastructures could enable automatic collaborative coordination and response to attacks affecting a large portion of the Internet.

1.1.6.3. No standard representation of intrusion data

Problems with incident data collection and processing, and interoperation of intrusion detection systems are based on the fact that there is no standard for representing incident data. Human intelligence is required to interpret and understand the essence of an incident report.

1.1.6.4. No standard for exchange of intrusion data

In addition to an inadequate standard representation, there is no standard protocol to exchange incident data. Currently, email, telephone calls, and fax are the accepted transport protocols for exchanging incident data. However, these techniques do not scale.

1.1.6.5. Human effort in handling incidents

Because there are no standard mechanisms for automatically exchanging intrusion data, human effort is required in every step of the process.

Interpretation of an incident report begins when an email-based incident report is first examined. Because of the wide variety of ways data can be structured within email messages, some processing such as decrypting (e.g. PGP), decoding (e.g. uuencoded, base64), or passing messages through an interpreter (e.g. MIME, HTML, MS Word) may be necessary even before being able to read an incident report. Once the incident report is in a readable format, a human must interpret the data. This process could include interpreting the meaning of the text provided, parsing log files, examining source code, or analyzing binary files.

Current incident reporting processes are too expensive to scale because humans are the critical component.

It is not just the CSIRTS which are overwhelmed with processing the reports. The system administrators who report the incidents often do not have the resources to submit security incidents. CERT/CC has found that the most common reason that for not reporting an incident is that the process is simply too cumbersome and time consuming.

1.1.6.6. Human error and inconsistency

Humans are far from perfect; using human effort to represent, exchange, extract, and record data can lead to typographical errors or misinterpretation of intrusion data. Furthermore, the lack of a a standard natural language vocabulary for representing attack data only further exacerbates the problem.

1.1.6.7. Other

Fear that exchanging incident information will compromise or implicate weak security practice is another significant impediment to security data exchange. Many companies, government agencies, and organizations treat any data collected as a result of a successful or failed attack as highly sensitive. Even worse is not reporting data that implicates compromise at other sites. This behavior is tantamount to seeing someone being mugged or attacked and doing nothing about it. The net result of this behavior is quite favorable to the *black hat* community since vulnerabilities tend to persist longer before fixes are available, signatures to detect attacks take longer to become incorporated into intrusion detection systems, and statistics about security trends are more difficult to estimate. Fear of reporting or exchanging incident data leads to less global security awareness and a more vulnerable global network infrastructure. The end result is that site administrators are forced to make risk management decisions without supporting data.

1.2. Objectives

The thesis explores the feasibility of creating infrastructures for automated reporting, exchange, and analysis of incident data across administrative domains. The design,

implementation, and testing of a prototype system comprises the bulk of the research.

1.3. Scope, Limitations and Assumptions

Limiting the scope and identifying assumptions was critical in defining an attainable research goal. From the onset, the following restrictions were placed on the thesis.

- The algorithm or technology used to detect attacks is assumed to function properly and all information provided by sensors is assumed to be correct.
- Apprehension to report or exchange incident data is assumed to be entirely a social issue without technical merit; thus, solving this social problem is not considered in the scope of this research.
- The prototype architecture is designed to cross *administrative domain*s; however, internationalization issues are not addressed.

Notes

1. see http://www.sherriffs.org/crime_prevention.htm for more details
2. On November 2, 1988 the Morris Worm was launched and quickly had a significant impact across a large portion of the Internet. This was the first major security wake up call for the Internet. See http://www.worm.net
3. see http://www.cert.org/summaries/CS-98.02.html for more information
4. see http://www.cert.org/advisories/CA-1999-04.html
5. see http://www.cert.org/advisories/CA-2000-01.html

Chapter 2. Technology Review

Internet security has been a topic of research since the existence of the Internet. Many of the problems addressed in this thesis have been identified or addressed by other research projects. Examining existing research helped determine scope for our research.

2.1. Standard Formats

Recognizing that standard formats are a prerequisite for technology adoption in the intrusion detection and incident reporting industries is not a new idea. "If ID systems are to evolve into a robust, mature market, open standards must be developed to address this issue" [R19]. Below are some existing contributions in the security data exchange standardization efforts.

2.1.1. CIDF

The basic premise behind the Common Intrusion Detection Framework is "standardizing formats, protocols, and architectures to co-manage intrusion detection and response systems" [R20]. This effort has made substantial progress in defining a language (CISL: Common Intrusion Specification Language), API, and architecture. While this standardization effort has produced some excellent work, it has been hampered by a lack of adoption by intrusion detection vendors and remains largely a research effort.

2.1.2. IDWG

To address the lack of commercial support in the CIDF, researchers involved in that project created the Intrusion Detection Working Group (IDWG) within the IETF. The following excerpt from the IDWG charter explains the goals and purpose of the group [R22].

> The purpose of the Intrusion Detection Working Group is to define data formats and exchange procedures for sharing information of interest to intrusion detection and response

> systems, and to management systems which may need to interact with them. The Intrusion Detection Working Group will coordinate its efforts with other IETF Working Groups.
>
> The outputs of this working group will be:
>
> 1. A requirements document, which describes the high-level functional requirements for communication between intrusion detection systems and requirements for communication between intrusion detection systems and with management systems, including the rationale for those requirements. Scenarios will be used to illustrate the requirements.
> 2. A common intrusion language specification, which describes data formats that satisfy the requirements.
> 3. A framework document, which identifies existing protocols best used for communication between intrusion detection systems, and describes how the devised data formats relate to them.

The Intrusion Detection Message Exchange Format (IDMEF) is the XML-based encoding format proposed by the group. To avoid the pitfalls of CIDF, work by Joe McAlerney at Silicon Defense has attempted to transition this proposed standard into the Snort [C4] intrusion detection system.

The IDWG is also defining protocols for transmission and communication of IDMEF messages. The Intrusion Alert Protocol IAP is the current proposed protocol.[R24]

The differentiating factor between the IDWG and the research in this thesis lies in the layer of abstraction at which data is being standardized. The IDWG is concerned with standardizing analysis results from current intrusion detection systems. On the other hand, this thesis is exploring the representation of raw network data.

2.1.3. IODEF

The Incident Object Description and Exchange Format (IODEF) is a new standards effort intended to "create a uniform incident classification framework that fully describes an incident" [R32]. This effort is driven largely by participants from the incident response community to enable easier data exchange when reporting incidents, and to create a baseline for enabling trend analysis. Because both the IODEF and IDWG efforts are focused on standardization of higher level analysis, they can make use of the work in this thesis to begin establishing a standard representation of the

underlying raw data.

2.1.4. tcpdump and libpcap

The libpcap library[C8] is an API used to capture network traffic. Because of its widespread use, this library is important to recognize as a defacto standard. Several network based intrusion detection systems, network sniffers, and network data processing tools can inter-operate and exchange data simply because they use the standard libpcap format files for storing raw network traffic.

Tcpdump is a tool that provides an easy interface to the functionality of the libpcap library and is capable of displaying network data in human readable format. The human readable output from tcpdump is a standard way of representing network data when reporting intrusive, anomalous, or suspicious traffic.

2.1.5. Other work

Finding standard ways to represent intrusion and incident data is not a new concept. There have been a number of other papers and research projects that are related to this topic. One of the most expansive was a joint research project between CERT and Sandia to define "A Common Language for Computer Security Incidents" [R25]. This effort was not intended to provide a comprehensive dictionary, but a minimum set of high level terms and their relationships.

The focus of this thesis thus far has been on languages to report and represent data after an intrusion or incident has been detected. Kemmerer, Vigna, and Eckmann point out, "it is possible to identify at least three classes of attack languages: exploit languages, report languages, and detection languages" [R26]. For the purpose of this thesis we will concentrate only on report languanges.

2.2. Intrusion Detection Infrastructure

As networks, intruders, and intrusion detection systems become more sophisticated the demand for data sharing between intrusion detection components increases. There are

many ongoing efforts to address this issue. In general these efforts differ from the research in this thesis as they are focused on a single *administrative domain*; whereas, this thesis is focused on addressing the issues of inter-operation across many domains.

2.2.1. AAFID

AAFID [R27] takes a different approach from the traditional monolithic intrusion detection systems. This novel architecture uses multiple autonomous agents working cooperatively to detect intrusions whereby improving the survivability of the system. This project shares many similarities with our research when facing trade-offs in architecture, scaleability, performance, and security.

2.2.2. AIDE

The Automated Intrusion Detection Environment (AIDE), funded by the DoD Advanced Concept Technology Demonstration program, entails building an intrusion detection infrastructure to inter-operate with existing components (firewalls, intrusion detection systems, routers, etc) rather than create a standard for new products. Data from these components is normalized and propagated through a hierarchical architecture where events can be correlated. The central difficulty with this process is normalization. With lack of standard methods for data representation, the same data may have entirely different meanings when coming from different types of systems; nevertheless, normalization is currently the most practical method until standards emerge.

Following is an excerpt from the SANS web site: [R28]

> The need exists for the fusing of data from multiple disparate sensors into a single display to enhance intrusion and anomalous behavior detection. This fusing of the data enables the AIDE system to integrate firewall, Router and Intrusion Detection System log files, prioritize alert signatures, analyze and correlate data in a near real-time, display this data into a comprehensive, uncluttered graphical view along with providing a longer term evaluation of attempted penetrations. The AIDE system incorporates a three-tier hierarchical encrypted communication system to allow local, regional, and global levels to share event data. This feature allows correlation across multiple sites to assist in detection of concerted intrusion efforts.

2.2.3. Emerald

Event Monitoring Enabling Responses to Anomalous Live Disturbances (EMERALD) is a set of tools designed to handle detecting intrusions across large infrastructures [R29]. The most applicable piece of work is ASIC (Assessing Strategic Intrusions Using CIDF) which attempts to correlate CIDF reports to "discern large-scale patterns of attack, and infer the intent of the adversary."

2.2.4. FIDNET

As part of Presidential Directive 63, the General Service Administration (GSA) intends to create the Federal Intrusion Detection Network (FIDNET) to monitor federal networks. FIDNET will gather and analyze information from sensors distributed among federal agencies [R30]. The GSA will not have a direct administrative control of any of the sensors, and will only aggregate information from them. The current status of this project is unknown.

2.2.5. IDIAN

More tangentially related is research from NAI Labs on the Intrusion Detection Intercomponent Adaptive Negotiation (IDIAN) protocol [R42]. IDIAN is an attempt for IDSes to dynamically adapt to intrusion stimuli and architectural changes (the addition of new components) through intercommunication. IDIAN is an IDS-to-IDS information exchange protocol rather than an aggregation, analysis, and dissemination architecture for intrusion information. Likewise, IDIAN only intends to interconnect IDSs in a distributed (but centrally controlled) network.

2.2.6. Shadow

Shadow is a collection of tools initially developed by the US Navy that enable an administrator to build an intrusion detection capability using tcpdump and other free software. The effort has grown and is now referred to as CIDER (Cooperative Intrusion Detection Evaluation and Response)[R33]. SHADOW uses traffic analysis to identify anomalies; therefore, the system is an excellent complement to the signature based

approach used in this thesis.

2.2.7. SnortNet

Snortnet [R34] is a project started by Fyodor Yarochkin that enables administrators to build distributed intrusion detection systems. The project shares a number of similarities with this thesis but does not include methods to handle exchange of data across administrative domains.

2.3. Incident Reporting Infrastructure

"The need for an international forum to respond to ID security incidents was recognized in the early 1990s and resulted in the formation of the Forum of Incident Response and Security Teams" [R17]. Additionally, members of the response team community have recognized the need for automated mechanisms for reporting, analyzing, and handling incidents.

2.3.1. AIRCERT

A significant portion of the technology developed for this thesis was jointly developed as part of the AIRCERT (Automated Incident Reporting to CERT) project [R36]. The purpose of the AIRCERT project is to enable administrators to automatically report malicious activity to the CERT Coordination Center with no human interaction. The hypothesis is that collection and aggregation of such sensor data will enable automated analysis to identify security trends. The project is currently in a prototype phase where issues of scalablility and useabilty are being tested.

2.3.2. AUSCERT: Automated Report Processing

AUSCERT has developed a mechanism for automatically receiving and processing incident reports via email [R37]. The intent of this collection effort is to gather scan and probe data in order to enable automated cross-correlation and visualization of the

state of security within the .au domain.

2.3.3. GIAC

The Global Incident Analysis Center (GIAC) is operated by the SANS institute. Their mission is to provide up to date reports of malicious activity submitted by their international community of system administrators [R41]. They are currently undergoing research to determine methods of automating the collection and presentation of reports.

2.3.4. INCIDENT.ORG

The INCIDENT.ORG project [R38] did pioneering work in 1999 by aggregating data from ipchains firewall logs in realtime across a distributed network. The initial release of the project was successful and provided correlation and trending for the constituency of reporting hosts. As the project grew, it faced issues with scalability and false positives. The problems were derived from the fact that data sources were required to log only unexpected and anomalous traffic, but there was no common understanding among participating sites of how to do this. The lesson to be learned from this project is the difficulty in aggregating data from anomaly based sensors. This is one of the reasons we focused our research on a signature based model.

2.3.5. Other IR Infrastructure Projects

There are other related incident reporting projects. There is a project called Voyeur from the Computer Sciences Corporation, yet we were unable to find any documentation and information with more details. Another recently announced project is www.dshield.org. That project is involved in aggregation and discovery of active attack sources.

2.4. Other Related Projects

There are a few other projects worth noting which are not easily classifiable.

2.4.1. Mailing Lists and Web Sites

In addition to response teams and the projects mentioned above, mailing lists and web sites are useful for building collaborative intrusion detection capability. The "incidents" mailing list [R40] at securityfocus.com is a prime example of a successful mailing list for exchanging incident information. Likewise, the GIAC website [R41] from SANS demonstrates an an online community for exchanging incident data.

These forums provide an effective method of communication and collaboration; however, they face the same problems described in chapter 1 as they rely on human interaction for data flow and analysis.

2.4.2. SnortSnarf

Developed by Silicon Defense, SnortSnarf [R39] is an analysis and presentation tool similar in functionality to the ACID tool developed as part of this thesis. SnortSnarf can be used to process Snort [C6] text log files and present the data more conveniently through web pages.

2.4.3. Event Notification

A number of open source projects have formed around building publisher subscriber messaging and queuing architectures. During early research phases some were considered as possible components for developing the prototype system. Notably, we considered openqueue[1], xmlblaster[2], and siena[3]. Nevertheless, we decided not to use any of them because they were in early development stages at the time.

Notes

1. http://openqueue.sourceforge.net
2. http://www.xmlblaster.org
3. http://www.cs.colorado.edu/users/carzanig/siena/

Chapter 3. Methods of Research

This chapter covers the strategies and methods used to pursue our research objectives.

3.1. Open source software

One of the motivations for this thesis is the lack of open standards in the Internet security industry. Given that the majority of open standards on the Internet are driven by open source software projects, the use of open source for this thesis was imperative.

The defining property of open-source is that source code is readily available to examine, modify, and extend as required. With code in the open, no architectural components are "black-boxes"; that is to say, all functionality can be scrutinized not only for inappropriate or malicious behavior (i.e. trojan-horse), but also for possible design flaws or implementation errors. In addition to increased trust and reliability, public scrutiny brings increased portability, useability, and quality.

3.2. Reuse of code

Existing open source software projects provide the bulk of the functionality for the prototype system. Therefore, the majority of the implementation work was in providing mechanisms for data to flow between existing technologies.

3.3. Working with the CERT/CC

Throughout this research we worked with the CERT Coordination Center in designing and implementing a prototype for automated incident data collection that culminated in AIRCERT[1]. Working with CERT provided an excellent environment for testing and developing this research in a real operational environment.

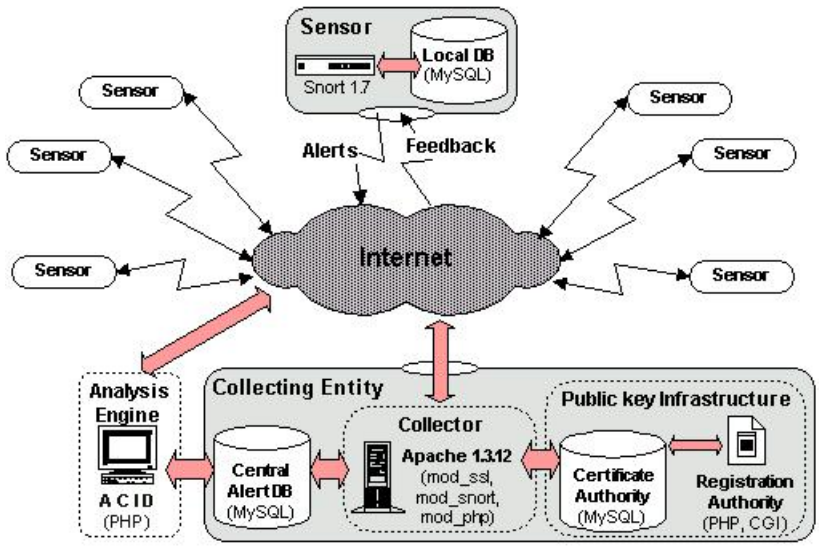
Notes

1. http://www.cert.org/kb/aircert

Chapter 4. Architecture

The architecture to collect incident information consists of four crucial components: a sensor, collector, backing store, and an analysis engine (see Figure 4-1).

Figure 4-1. Prototype Architecture



4.1. Sensor

A sensor detects security-related events and reports them to a central collector. Any device capable of supporting remote logging and the necessary alert encoding format (i.e. SNML, see Section 4.3.2) can be a sensor. Intrusion detection systems make effective sensors since they are specifically designed to detect events of security interest. However access-control or transit devices such firewalls and routers will also serve well. In order for the architecture to accomplish the stated goals, a large number of sensors, regardless of their type, must be deployed Internet-wide.

Two requirements defined by COAST for a "good intrusion detection system" also apply to sensors [R6].

- *It must run continually without human supervision.* A primary goal of the architecture is to enable automated incident reporting to solve many of the current bottlenecks with human reporting. Therefore, the detection activity of the sensor must entail very little human intervention.
- *On a similar note to above, it must resist subversion.* Not only are the monitored host possible targets, the sensor may be as well. Great effort must be put into both the implementation and deployment to ensure that the intended detection and logging functionality of the sensor is not subverted.

In addition, the constraints of the prototype system introduce the following additional requirements:

- *Sensors must be able to communicate with the collectors.* Communication with the collector dictates that the sensor has the ability to encode its alert data into the proper format as well as understand the collector supported transport protocol. Furthermore, the security related events detected by the sensor must be transmitted to the collector in a "timely" manner, but not necessarily in real-time.
- *Sensors must be able to contend with sporadic disconnects from the collector server.* The reality of Internet connectivity dictates that communication with a collector server will not be possible under certain circumstances. The inability to communicate may be the result of any number of factors: unbearably high network latency, denial of service (DoS) against the collector server, or disconnection from the Internet. The sensor should be able to cache alerts for a "reasonable" period of time for retransmission to the collector when communicate becomes possible again.
- *Participation in the prototype infrastructure must not diminish the usefulness, performance, or capacity of the local sensor.* The deployed sensors do not merely collect data for the purposes of this prototype system. Rather, each sensor has some locally significant data collection responsibility that cannot be impeded by the additional incident reporting. It follows that the additional remote reporting also must not affect the false positive or negative rate of the underlying detection engine.
- *Sensors must support a distinction between a local and remote configuration.* Events of concern from the scope of the local domain are different than those from a cross-administrative domain perspective. For example, a particular system administrator may want to be informed of all port-scans against his hosts. However, such detail is too specific when collecting and analyzing event across the Internet. Thus, there must exist granularity in the type of reporting that is done remotely,

independent of local logging. However, all remote configurations should be subject to local examination and approval.

- *Sensors must support locally defined sanitization parameters.* Suspicious events logged by the sensor may contain sensitive data from the heart of an internal network (e.g. IP addresses, contents of a packet payload which could contain email or passwords). As a consequence, organizations may be apprehensive about reporting complete alert data to an organization outside of their control. Allowing granularity in the type of information that is reported will allay privacy and confidentiality concerns. However, in order to aid in analysis, the implementation of these sanitization facilities should be done in such a fashion that it is possible to distinguish between data that is missing or incomplete and that which was intentionally sanitized.
- *Compromise of single sensor must not undermine the integrity of the entire system.* Since each sensor will be deployed across administrative domains, the compromise of a sensors is inevitable. Nevertheless, a compromised sensor should not increase the ease with which any other architectural component can be subverted.
- *Sensors should be easy to configure.* Minimal knowledge should be required to configure a sensor minimal and the configuration process should be automated as much as possible.

4.2. Collector

A collector server serves as the aggregator and logging entity for alerts collected from sensors deployed across administrative domains. The collector is a network server responsible for accepting an alert stream, validating its correctness, and finally parsing and writing each data element into a database.

The primary purpose of the collector is to eliminate direct communication between sensors and the database. Within a single domain, it may be acceptable for sensors to log directly to a database; however, this type of logging does not scale across administrative domains. The single driving constraint lies in that neither the components in a multi-administrative domain infrastructure, nor the communication paths between them can be trusted. Thus, strong cryptography and authentication must be employed on the channel. It follows, that it is both infeasible (e.g. administration, performance) and insecure to give every sensor in the architecture direct access (e.g. an

account) to a database. Furthermore, the process of aggregating alerts extends beyond merely storing them in the database. A collector allows additional logic to be applied and operations to be performed on the data before it is stored.

The following requirements are imposed on the collector server:

- *The process of aggregating and processing alerts must highly scalable.* A collector server will potentially have to communicate with a multitude of sensors that can be transmitting simultaneously. Therefore, its architecture must have a low overhead per transaction, and allow for a large number of concurrent transactions.
- *The collector must be a bastion host.* As a central component of the prototype architecture, the collector will be a highly visible target. Great effort needs to be placed into ensuring the quality of the data flowing into database and the graceful degradation of service as the load increases. With regards to ensuring quality data, it should not be possible for any alert to be processed without emphatically verifying from which sensor it arrived. Furthermore, all malformed, crafted, or invalid alerts should be immediately discarded. Finally, it should not be possible attack the collector's availability at the application layer.
- *Each collected alert must be uniquely identifiable.* Each sensor can uniquely identify every logged alert locally. However, there is no guarantee that this identifier is globally unique across all sensors. The collector server, as a central logging entity, must generate a unique identifier for each alert it processes and stores. Discarded alerts need not have unique identifiers.

4.3. Communication

4.3.1. Channel

The sensor and collector communicate over a logical, secure channel across the Internet. This channel uses in-band signaling such that both data and control information are transported over the same link. The sensor submits alerts to the collector and in turn receives back status information from the collector.

The channel has very strict security and performance requirements.

- *All entities communicating on the channel must be authenticated.* All authentication must be mutual such that both entities can verify each other's identities. It should not be possible for anonymous communication to occur.
- *The channel must be private and tamper-proof.* The confidentiality of all the data must ensured to all but the two end-points of communication. No intermediaries, even entities in the same administrative domain as one of the end-points (e.g. routers), should be able to read the traffic. Furthermore, it should not be possible for an attacker to modify the traffic without detection.
- *The transport protocol must lend itself to high-volume transactions.* In order to support the necessary volume, the underlying protocol must have an efficient connection initialization and tear-down sequence.

4.3.2. Data encoding

Each sensor type has its own internal characterization of alerts. However, in order for each of them to be submitted to a central authority, they must first be normalized into a common representation. Without a common representation, the collector server would have to understand and decode as many formats as there are sensor types. This decoding would bloat the server to an unmaintainable program as well as a severely impacting performance.

The alert encoding format must have the following properties:

- *Alert encoding must be generated efficiently.* Alerts can be triggered at an extremely high volume. The process of converting the native alert representation to the common encoding format must be able to support this rate. Furthermore, the computational and memory overhead to perform this conversion should be minimal since few assumptions can be made about the available resources of the sensor.
- *The encoding language must be easily extensible.* The encoding language will inevitably have to evolve to allow for new sensor types. All efforts should be made to leave the language as flexible as possible to accommodate new types of data.
- *Machine parsing of encoded alerts must be efficient.* The collector server will have to process a huge number of alerts. The process of extracting and validating the data from the alerts must be relatively efficient to allow for the necessary scale.

- *The encoding must be human-readable.* While the ultimate goal of the encoding scheme is to facilitate automated processing, an analyst should still be able to view the alert stream and still understand what data is being sent.

4.3.3. Feedback protocol

The "feedback" protocol will be the rudimentary mechanism by which status on the processing of an alert is returned to the sensor by the collector. In a sense, it is also a C2 (command and control) protocol that allows a central authority (collector) to change the behavior of the remote and subservient entities (sensors). The requirements imposed on the encoding language also apply to the feedback protocol.

4.4. Backing store

The backing store is a database (permanent store) in which alerts detected by the sensor are stored for later analysis.The database schema implemented in this store is a representation of the common alert encoding format.

There are two levels of backing store in the prototype architecture.

- The *Local database* is situated in the same administrative domain as the sensor. It is exclusively controlled by the sensor operator and is the location where alerts would be logged without any extensions created by the prototype. The presence of this database (and the information stored in it) is transparent to the goals of the prototype.
- The *Central database* is the repository of alerts aggregated from sensors across administrative domains by the collector server. There is no direct connectivity between the central database and the sensors or the sensors' operators. The central database is not merely a larger, centralized instance of a local database. Rather, the central database contains a superset of the data found in the local database. This additional data is meta-information about how the alerts were collected.

The backing store has the following requirements imposed on it:

- *Access to the stored alerts must be efficient.* The database will grow to an immense size over time. However, there is only value in the data if it can be analyzed effectively. It should be possible to efficiently (i.e. random access) extract alerts.
- *The owner of no submitted data can be ambiguous.* The database structure must always be able to identify the sensor which submitted every particular data element.

4.5. Analysis Engine

The analysis engine examines the alerts collected in the database for cross-administrative domain events, attack trends, and will ultimately use this alert data to assess the state of security on the Internet. The results of this analysis will be the added value given to the prototype participants for submitting their data.

Chapter 5. Implementation

5.1. Sensor Implementation: Snort

The system prototype was implemented using the network intrusion detection system Snort as the sensor. The author of Snort, Martin Roesch, describes it as follows [C4]:

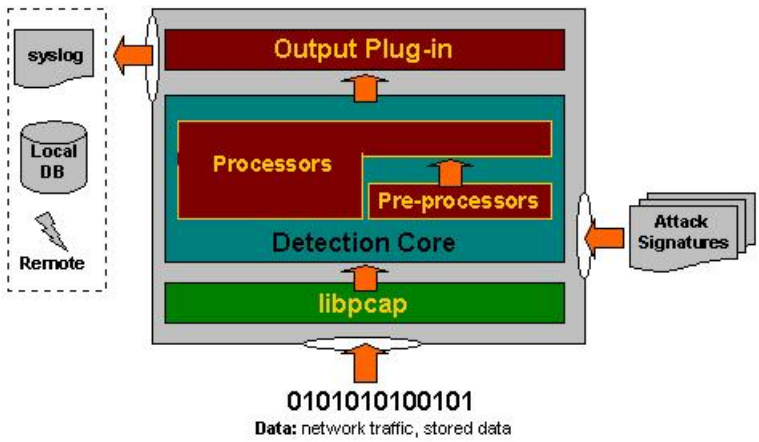
> Snort is a lightweight network intrusion detection system, capable of performing real-time traffic analysis and packet logging on IP networks. It can perform protocol analysis, content searching/matching and can be used to detect a variety of attacks and probes, such as buffer overflows, stealth port scans, CGI attacks, SMB probes, OS fingerprinting attempts, and much more. Snort uses a flexible rules language to describe traffic that it should collect or pass, as well as a detection engine that utilizes a modular plug-in architecture. Snort has a real-time alerting capability as well, incorporating alerting mechanisms for syslog, a user specified file, a UNIX socket, or WinPopup messages to Windows clients using Samba's smbclient.
>
> Snort has three primary uses. It can be used as a straight packet sniffer like tcpdump, a packet logger (useful for network traffic debugging, etc), or as a full-blown network intrusion detection system.

5.1.1. Snort Architectural Overview

Snort is designed with a multi-stage, piped architecture that reads and compares network data against a set of attack signatures. Any signatures matches are logged to

the output facility (see Figure 5-1).

Figure 5-1. Snort Architecture



5.1.1.1. Input

Snort has the ability to read packet data directly off the wire in real-time using the libpcap [1]. In addition, raw network data can be captured and stored with the packet-sniffer tcpdump [2] and later inputted back into Snort for off-line analysis. This latter scenario is highly desirable in circumstances when deploying an IDS is not possible due to an inordinate amount of traffic.

5.1.1.2. Detection core

The detection core compares attack signatures [C6] (see Example 5-1) from a configuration file and the input data stream for possible matches. The functionality of detecting attacks is separated into stages handled by a preprocessor and a processor.

Optionally, preprocessors may accept the raw data stream and converts it into a common representation to prevent attack obfuscation. This normalization process includes TCP stream reassembly, IP fragment reassembly, and HTTP escape code decoding. Without this abstraction step, a different signature would have to be written

for every possible representation of an attack; and even then could not be detected at all variants. For example, in order to foil older IDSes, intruders would explicitly fragment their exploit across extremely small IP fragments. Thus, when compared on a single packet basis, no malicious activity could be detected by the IDS. However, when the IP stack reassembled all the fragment packets, the exploit was still intact [R9].

Another unintended but widely employed detection technique uses preprocessors to identify attacks that are difficult to characterize with mere signatures. These novel preprocessor modules include a port-scan detection engine and a network traffic anomaly detector (e.g. Spade [C5]).

The processor in turn accepts the normalized data from the pre-processor and does all the necessary comparisons between the data and signatures to find matches.

Example 5-1. Sample Snort Rule

```
alert tcp any any -> $HOME 143 (msg:"IMAP Buffer Overflow!";
                                content:"|9090 9090 9090 9090|";
                                   depth: 16; offset: 5;
                                content:"|E8 C0FF FFFF|";
                                   depth: 10; offset: 200;)

The above Snort rule detects an "IMAP buffer overflow"
triggered under the following circumstances
    - TCP protocol
    - destination port of 143
    - payload contains the NOP sequence of "9090 9090 9090 9090"
      starting with the 5th-byte in payload and matching the
      next 16-bytes (bytes 5-31 of the payload) AND
    - payload contains the byte sequence of "E8 COFF FFFF"
      starting 200th-byte into payload and searching in the next
      10-bytes (bytes 200-10of the payload)
```

5.1.1.3. Output plug-ins

Output plug-ins are the facility by which alerts triggered by the processor are logged. Snort supports a number of output targets such as syslog, flat-file, raw TCP socket, WinPopup, and database.

5.1.2. Required Snort additions

While Snort provided the raw detection facilities, significant changes were still needed to facilitate the goal of incident reporting in a larger infrastructure.

- *Alert encoding:* The native Snort logging formats were inadequate for a distributed infrastructure since most were not scalable for high volume data logging and analysis. Among the existing candidates, only database logging would scale in size, but it did not lend itself to easy or secure network transport across administrative domains. Therefore, a separate output plug-in (spo_xml) was developed to encode alerts in XML (see Section 5.3 for details).
- *Remote logging infrastructure:* Snort had no support for secure remote logging over which alerts could be sent. Therefore, the output plug-in designed to encode alerts in XML (spo_xml) also had extensive network communication functionality added to facilitate secure and efficient communicate with the collector server (see Section 5.2 for details).
- *Simultaneous local and remote configuration:* Snort provided no distinction in the signature set or logging options for those events logged locally and remotely. Since these configurations could possibly be different, and the prototype is required not to disrupt the existing local functionality, developing support for different classes of logging was necessary. While in development of the prototype, Andrew Baker (andrewb@hiverworld.com) contributed this exact functionality to the Snort code-base obviating the need to develop this code independently. With this contribution, completely different rule sets or even those that overlap can log to different output facilities. For example, all detectable alerts could be logged to a local database, but only buffer-overflow and certain mail-virus alerts would be logged and reported remotely.
- *Sanitization:* With the remote logging introduced, client-configured sanitization for the source or destination address, and the payload contents was implemented directly into the XML encoding plug-in. This functionality allows the sensor to report as much information as the organization feels is appropriate.

5.2. Collector Implementation: Apache

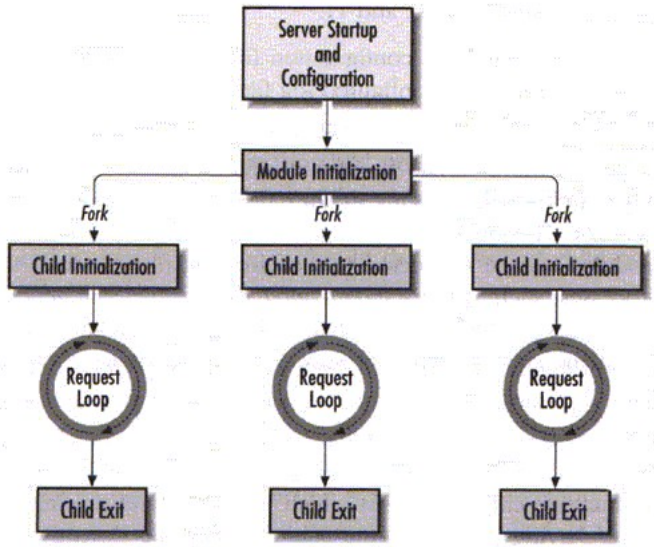
The collector server was implemented using the Apache Server which is a "robust, commercial-grade, featureful, and freely-available source code implementation of an HTTP (Web) server." [C3]

5.2.1. Apache Architecture

Apache is designed with a highly modular architecture and implemented as a pre-forking server. The prototype made use of this extensible architecture using the module API to develop `mod_air`. The prototype component, `mod_air`, extends the Apache core and builds the prototype functionality directly into the web server daemon (httpd).

A pre-forking, multi-process implementation is the major reason behind Apache's great scalability (note: the pre-forking architecture only applies to UNIX. Under Windows, Apache runs as a multi-threaded process). At start-up, Apache spawns a master process (httpd) to handle the reading and processing of the configuration file (see Figure 5-2). Next, all modules, including `mod_air`, are also initialized in the Module Initialization phase. Later, this master process fork()s multiple instances of itself in the Child Initialization phase. These newly forked children are collectively referred to as the child pool. After each child spawns and initializes, the server enters the Request Loop phase waiting to service client requests. The child pool is exclusively responsible for handling all connections from the sensors. The master process that spawned the children never handles requests. In order to increase parallelism if the load on the server should grow, the master process will grow the child pool (i.e. fork() more children to serve the requests). In turn, when the load subsides, these excess children will be killed. Several "spare" children are always be kept in order to service sudden spikes in the number of requests [C2].

Figure 5-2. Apache Life-cycle



Source: Writing Apache Modules with Perl and C [R45]

5.2.2. Apache Additions

Developed as an Apache module, the collector component hooks the incident reporting functionality into the Request Loop replacing the "typical" web server behavior. Instead of serving a web page or spawning a CGI process for a client request, Apache will invoke `mod_air`.

This selective behavior can be further understood with a closer examination of request handling by the Request Loop. Within Apache, all of the server functionality is highly modular. The core server is only responsible for the intricacies of handling raw network connections and passing data from these connections to the appropriate module for processing. It is the responsibility of the invoked modules to extend and fulfill the real server functionality.

Communication with sensors occurs over SSL/TLS. Initially, the core server will accept the sensor’s TCP connection and invoke `mod_ssl`, a module that will handle the

negotiation of an SSL connection. With an SSL connection established, the actual request must be identified and serviced with the appropriate handler. This identification usually occurs based on the extension of the requested file or the type of request made (e.g. POST, GET). Each handler is a different module (e.g. PHP extension invokes mod_php, a CGI script invokes mod_cgi). POST requests for files with the `.air` extension will result in Apache ignoring the default handler (mod_cgi) and forcing `mod_air` to process the request.

`Mod_air` executes in three stages when processing and logging an alert. Each of the three steps of authentication, throttling, and validation must be completed in this order, but it is possible for any step to abort without an alert being logged to the database.

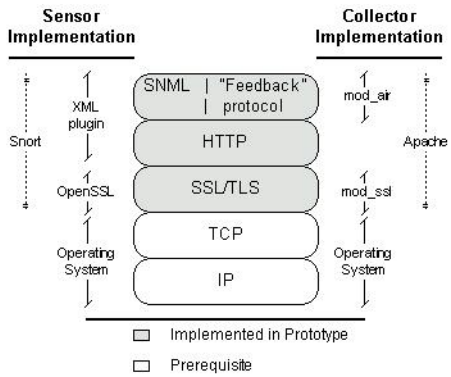
- The *Authentication* phase uses an X.509 certificate [R10] (presented by the sensor during the SSL connection) to explicitly identify the sensor with which the collector is communicating. If a sensor's certificate is invalid or unknown, the corresponding alert is dropped.
- The *Connection throttling* phase prevents the server from being flooded with alerts from a particular sensor. This functionality involves creating a shared-state among all the Apache processes (i.e. the child pool) using shared memory segments. An entry is made in a hash table when the sensor communicates with the collector for the first time. Thereafter, the arrival time of every alert is noted in this corresponding table entry allowing the number of arrived alerts within a given time window to be counted. Should this number exceed a certain threshold, the alert is dropped.
- The *Validation* phase involves processing the submitted alert. Using the libxml 2.0.0[3] library, a SAX parser examines the XML encoded alerts to validate the correctness of the data and writes the alerts into a database. Should the XML be malformed (i.e. incorrect XML as defined by the DTD) or invalid (i.e. lacking the proper data), the alert will be dropped. Prior to writing an alert to the database, the collector assigns a globally unique identifier using a sensor identifier and a sequence number. A globally unique sensor identifier is created by taking a combination of the sensor's IP address, interface name, and BPF parameters.

Regardless of success, `mod_air` returns status on the processing of the alert via a feedback protocol (see Section 5.3) to the sensor.

5.3. Communication Implementation

Communication between the sensor and collector requires a number of protocols to facilitate the necessary requirements (see Figure 5-3). All hosts communication is done over TCP/IP; however, in order to instill the necessary security properties, TLS 1.0 (SSL) must also be layered on top. Next, HTTP is used to encapsulate the custom. Using HTTP imposes request-reply communication semantics between the sensor to the collector. The sensor initiates communication with an HTTP POST request [R11] passing alerts (encoded via SNML) and the collector returns status (with the feedback protocol) via an HTTP reply.

Figure 5-3. Prototype Protocol Stack



All attempts were made to re-use as much code as possible for the well-defined protocols. Obviously, the operating system provided the TCP/IP stack. The OpenSSL[4] library and API were used to add TLS functionality to Snort (i.e. the sensor) and HTTP support was written by hand. `Mod_ssl`[5] developed by Ralf Engelschall endowed the server with TLS support, while Apache was the code-base for HTTP. Only the encoding format and feedback protocol were newly developed to fulfill the needed prototype functionality.

5.3.1. Simple Network Markup Language

The encoding format was implemented with the Simple Network Markup Language

(SNML), a rigorous Document Type Definition (DTD) which defines the encoding of any IP network data in XML (see Appendix D). An SNML document consists of two sections: detection information and stream contents (see Example 5-2). The detection information is the meta statistics about the triggering of an alert: timestamp, signature, and sensor statistics (name, interface BPF parameters). The stream content is the network packets (i.e. packet header and payload) which triggered the alert.

Example 5-2. Sample SNML document

```
<report>
  <event version="1.0">
    <sensor encoding="hex" detail="full">
      <interface>eth0</interface>
      <ipaddr version="4">10.0.0.1</ipaddr>
      <hostname>samplesensor.net</hostname>
    </sensor>
    <signature>PING-ICMP Destination Unreachable</signature>
    <timestamp>2000-11-27 03:12:51-04</timestamp>
    <packet>
      <iphdr saddr="10.0.0.1" daddr="10.0.0.2"
             proto="1" ver="4" hlen="5" tos="192" len="140"
             id="27894" ttl="255" csum="63475"
        <icmphdr type="3" code="3" csum="61863">
          <data>000000004500005C9CE900007E3149F18003040D801251730800
                FF25020086DA00000000000000000000000000000000000000
                00000000000000000000000000000000000000000000000000
                00000000000000000000000000000000450000390200000
                40068CFB8002514518405F1D</data>
        </icmphdr>
      </iphdr>
    </packet>
  </event>
</report>
```

5.3.2. Feedback protocol

After processing the alert, the collector replies to the sensor with a text-based "feedback" protocol. The status is returned in the form of a code number and a text

description modeled very similar to HTTP return codes (e.g. `404 File not found`). However, unlike HTTP return codes, more than one code can to be returned with every reply (see Example 5-3). There are three categories of possible messages (see Appendix E for a complete listing): authentication (3xx) status returns information on the success of identifying the sensor; input validation (4xx) status identifies whether the alert could be processed; and throttle (5xx) status signals any application-layer congestion on the channel.

Example 5-3. Sample "Feedback" protocol result

```
HTTP/1.1 200 OK
Date: Sat, 30 Sep 2000 18:04:49 GMT
Server: Apache/1.3.12 (Unix) mod_air/0.8.4 PHP/4.0.2
        mod_ssl/2.6.4 OpenSSL/0.9.5a
Connection: close
Content-Type: text/html

mod_air/0.8.4 300 AUTH_CLIENT_OK
mod_air/0.8.4 500 THROTTLE_OK
mod_air/0.8.4 400 INPUT_COMMIT_OK (1)
mod_air/0.8.4 200 OK
```

5.3.3. Public Key Certificates

Implementing mutual authentication with TLS/SSL requires that every component in the system have an X.509 certificate. A self-signed CA issues all certificates, but the specifics of distribution to each of the sensors occurs outside the normal sensor-collector communication.

The CA communicates with both the sensor and the collector. Prior to being able to report alerts, every organization with a sensor is required to visit a web-site to register. This registration is an out-of-band mechanism for collecting "real" (non-prototype related) contact information and to validate to whom certificates are issued. Once the sensor is issued a certificate and submits alerts, the collector server also contacts the CA to validate the sensor’s certificate against a CRL.

5.4. Backstore Implementation: MySQL

The local and central database in the prototype system were implemented with MySQL 3.22.32 [6] using ISAM tables [C7].

5.5. Analysis Engine Implementation: ACID

The analysis engine was implemented as the Analysis Console for Intrusion Databases (ACID) tool. ACID is a PHP application [7] that enables a security analyst to perform real-time analysis operations on the alert database. Its core functionality includes the following:

- A *search interface* for finding alerts matching practically any stored field as a criteria. This criteria can include meta-data on the detection of the alerts (e.g. arrival time, sensor, signature time) or protocol fields from the packet (e.g. source/destination address/port, flags, payload). These queries can be made arbitrarily complex to satisfy almost any parameters.
- *Alert Groups* facilitate more complex analysis by allowing a logical grouping of alerts on which analysis can be done. It is a quick way to combine multiple searches, annotate an alert or group of alerts, or to save query results for later examination.
- *Alert purging* allows for the deletion of alerts from the database. This functionality is ideal for removing known false-positives.
- A number of *aggregating statistics and graphs* can be generated to get a perspective on the type of alerts in the database or to glean statistics from a particular query result: percentage of traffic for each protocol, sensor, or IP address; unique alerts or IP addresses detected.

Notes

1. libpcap is BPF-compatible library which captures network traffic. http://www.tcpdump.org
2. TCPdump is packet-sniffer. http://www.tcpdump.org

3. libxml2 is a SAX XML library. http://www.xmlsoft.org.
4. OpenSSL is an open-source SSL and cryptography library based on Eric Young's SSLeay library. http://www.openssl.org
5. mod_ssl is a module which adds SSL version 2-3 and TLS 1.0 into Apache using OpenSSL.
6. MySQL is an open-source SQL92 compatible database. http://www.mysql.org
7. PHP is an open-source server-side scripting language. http://www.php.net

Chapter 6. Analysis and Interpretations

6.1. Sensor Analysis

Snort was chosen as the first prototype sensor in part because of an open source license (GPL) that freely distributed its highly extensible code. The Snort plug-in architecture made it trivial to insert functionality without making significant modifications to the core code-base. From the non-technical perspective, the nascent Snort community (as with many open-source projects) is growing, highly active and energized. Therefore, the users were willing to run the prototype code and return instructive feedback on our efforts. Furthermore, the community has reached a critical size such that enough useful data could be collected to validate the infrastructure.

Nevertheless, Snort is a relatively immature product, not yet two years old. As a consequence, it has several shortcomings in comparison to more long-lived commercial products. These limitations are evident in some architectural constraints:

- *Single-threaded core:* Snort currently is implemented as a single threaded process. This architecture is of concern since it may be possible for alerts to be missed during high loads. For example, if the logging functionality does not execute in a timely fashion (e.g. network latency), the output plug-in will remain blocked waiting for the operation to complete. This waiting precludes the detection of new alerts. While this deficiency remains a real issue, efforts are currently underway in the Snort development community to add multi-threading in the next major release. This would allow all the major Snort components (i.e. input/output plug-ins, detection core) to run independently of each other.
- *Stateless:* One of the defining weaknesses of Snort is that alerts are detected only on the basis of a single packet. While Snort performs fragment reassembly, analyzing protocol streams is much more effective in detecting malicious behavior. In order to address this limitation, Chris Cramer of Duke University (cec@ee.duke.edu) has written a preprocessor into Snort that will do full stream reassembly. While this code is currently not ready for production use, it will bring Snort to the next level of utility whereby making the Snort detection engine comparable to commercial IDSes.
- *Alert characterization language:* Seminal to detecting attacks is the ability to

characterize them with signatures. Unfortunately, Snort's description language is not very flexible. It has incomplete support for boolean operators between criteria and no conditional statements (e.g. if ... then ... else). Again, just as shortcomings in the architecture, Snort is maturing to address this issue. There is currently development in the Snort community to improve boolean operators support in the next major release.

- *High false positives:* Another consequence of a lack of flexibility in the signature language is a high false positive rate. The particular signature to detect an attack may not be specific enough to characterize merely the desired exploit. Instead, legitimate traffic may sometimes also trigger the alert. In other cases, alerts may be legitimately triggered (i.e. due to attack traffic), but the attack may not be applicable. For example, an IIS specific exploit could be launched against a host that does not even have web services enabled. The former issue of legitimate traffic triggering alerts can only be helped with a more feature rich signature language. However, the latter issue of inapplicable alerts can be somewhat mitigated with the `nmap`[1] and `snortrules`[2] tools. Using `nmap`, a list of services running on each host can be identified. In turn, `snortrules` can determine the attack signatures that are targeted at monitoring services which are not present on the network and remove them from the configuration.

Snort is also not immune from the typical IDS tradeoffs associated with the being primarily a signature-based, network IDS. Using signature-based detection limits Snort to detecting only known attacks; while being a NIDS dictates that Snort is only capable of analyzing network data and recognizing those attacks that can be identified through packet analysis.

6.2. Collector Analysis

The choice of using Apache as the implementation of the collector server was influenced by the selection of HTTPS as the alert submission protocol. While a minimal HTTP server could have been written, Apache provided a number of advantages.

- *Quality code base*: Using Apache precludes the need to "re-invent the wheel." The Apache source code is available, well documented, and portable across almost every UNIX and Windows platform. Apache not only could provide the HTTP protocol

code, but outside contributions to the Apache Server Project have also added SSL extensions via `mod_ssl` to support HTTPS. Furthermore, open source licensing has allowed the Apache code to be highly scrutinized and tweaked. Thus, the prototype was able leverage the highly optimized socket and connection management code-base.

- *Extensible*: Apache provides a clean module API through a series of function callbacks that allows custom functionality to be hooked into the server. Likewise, using dynamically shared objects (DSO), a technique of dynamically loading libraries, allows these custom modules to be compiled and build independently whereby allowing easy distribution [C1].

From a design perspective, this close relationship between the collector implementation and the transport protocol of HTTP initially seems problematic, especially since this transport infrastructure has not been fully validated. Should the transport protocol ever be replaced, the current collector code-base runs the risk be being obsolete due to an Apache module implementation. Despite these concerns, very little of the collector implementation is transport protocol specific. Rather, not having to write this portion of this code was one of the primary appeals of using Apache. Thus, replacing the transport protocol would only require implementing the corresponding raw connection management, not the core collector functionality of alert processing.

6.3. Channel Analysis

The communication channel between the sensor and collector needs to guarantee confidentiality, integrity, and authentication. There were two possible OSI layers at which to ensure these properties: network (layer 3) or transport (layer 4-5) layer. The two likely candidates were IPSec [R2] and SSL/TLS [R1] respectively. However, since not all OS kernels (IP stacks) support IPSec, it was quickly discarded in favor of TLS which would maximize portability and allow for the widest possible sensor deployment.

6.3.1. Transport protocol analysis

HTTP was chosen as the protocol to submit alerts because it is extremely lightweight and requires only a minimal footprint into the sensor to implement. Likewise, the stateless nature of HTTP solved many of the design issues related to sensor

connectivity. One of the early concerns in designing the communication semantics dealt with whether a sensor should always retain an open connection to the collector; specifically, as to whether this architecture could scale to a large number of sensors. Using the HTTP semantics of request-reply eliminates constant connections and obviates the need to identify timed-out links. These semantics also have the highly desirable property of minimizing the number of open connections on the collector. Heartbeats, messages sent explicitly to the server to inform the continued presence of sensor, were also initially considered. However, as the number of administrative domains increases the status of individual sensors becomes less relevant. Finally, using HTTP is highly desirable since it is a widely deployed and well understood protocol.

One of HTTP's major advantages, simple communication semantics, presented some difficulties. The most obvious constraint was that HTTP (and Apache) is stateless, but enforcing various connection quotas (thresholds on the acceptable amount of traffic from a sensor) and database-access policies does require state across connections. However, keeping extensive statistics at the server and implementing IPC mechanisms between the Apache processes easily circumvented this stateless-ness.

6.3.2. Data encoding analysis

Analyzing the prototype protocols reveals an inconsistency in the encoding scheme. Communication from the sensor to the collector is in XML, but the communication back to the sensor is in normal text. Such a design is quite deliberate. While generating XML is a relatively straightforward operation requiring only string concatenation, parsing XML is a much more logic intensive task typically offloaded to a dedicated parser. Thus, the sensor can easily create the XML with a small footprint, but processing it would have required an additional XML parsing library. It an effort to keep the sensor as lightweight as possible (i.e. minimize external library dependencies), the collector communicates with the sensor with text-based messages that can be easily processed with standard string handling functions.

6.4. Backing Store Analysis

Early in the design it was obvious that a database was the only possible candidate of a

backing store for alerts. Despite being very fast, simple text file logging was deemed to be wholly inadequate as it would not scale to the proper size, and random access searching was simply not feasible. Examining the open-source database candidates drew only two possibilities: MySQL and PostgreSQL[3].

Comparing the two candidate databases reveals that MySQL lacks several very key features in comparison to PostgreSQL. The production release of MySQL does not yet support transactions (although a beta version of transaction supported MySQL on Berkeley tables does current exist). Likewise, there is no lock granularity when accessing the tables. Only table level locking is available. Finally, MySQL does not support views or sub-selects with its SQL implementation.

Since, PostgreSQL supports all these deficiencies in MySQL, it would seem like the obvious candidate for the prototype. Nevertheless, MySQL was still used in the implementation for the sole reason that it is much faster. The MySQL database engine is optimized for fast writes (INSERTs) which lends itself to the sheer volume of alerts which must be handled. This speed consideration outweighs any other limitations. However, selecting was MySQL was not really trade-off since most of its limitation could be mitigated with additional logic and a creative use of IPCs. Aside from the speed, MySQL also provided much richer data types which makes the alert storage more efficient.

6.5. Analysis Engine Analysis

ACID’s implementation in PHP made it easy to develop and flexible to deploy. PHP provided a very easy construct in which to rapidly create a MySQL-aware application. Furthermore, as a web-scripting language, PHP precluded the need to write any sophisticated user interface since one was already inherited from the browser. This browser paradigm was also popular with users because of the high portably across platforms and low resource requirements on the client.

The reaction to ACID when deployed in the community was positive, although surprising. In addition to fulfilling its technical role, ACID turned out to be an excellent marketing vehicle to advertise the prototype and to entice community interest in upgrading their software to run the prototype code. It is also this community deployment which originally uncovered a database scalability issue. ACID would not run in a timely manner (e.g. long latency to run queries) when used against a database

with a large number of alerts (i.e. 300,000 alerts) or during analysis operations concurrent to high-volume alert logging. Through this instructive feedback, the database bottlenecks were able to be identified and corrective modifications applied to the database schema.

6.6. Performance

The collector server and the associated infrastructure code in the sensor have morphed through various incarnations to arrive at the current prototype. The prototype has been heavily optimized to perform the specialized task of generating, sending, processing, and logging alerts.

Since the sensor was already built and the prototype functionality was merely added, established metrics on "acceptable" performance existed. Evaluating the sensor's performance was a matter of comparing the the execution time of an unmodified version of Snort against a patched version with the remote reporting enabled. This testing revealed that the prototype largely succeeded in having only a minimal effect on the sensor's core functionality.

Analyzing the performance of the collector server was more involved and resulted in an iterative process since no baseline performance metrics existed. In the initial design, the collector server was implemented as a Common Gateway Interface (CGI) application. Under high loads, performance issues became evident when using this paradigm. The inherent problem with CGI was that a new instance of the application must be spawned for each request. Thus, for every batch of alerts, the overhead of spawning a new process is incurred (e.g. Apache formatting the environment with the required variables, kernel context switches).

The Apache module interface was explored in an effort minimize the penalty incurred by spawning a new process with every request. A module implementation allows the custom functionality to run directly inside the server process instead of as an external process. `Mod_air`, the implementation of the collector server, is essentially designed as a minimal POST request handler (which replaced `mod_cgi`, the default Apache code for handling POST requests) with XML parsing and DB logging ability. Superficial testing yielded roughly a 50% performance gain with a module interface implementation of the same code over the CGI paradigm.

After an efficient architecture for the collector server was established, the

communication performance was investigated. Using the original module implementation under a testing scenario, the latency between the attack detection at the sensor and the alert logging at the collector took an average of 427 ms (see Figure G-1 in Appendix G). An analysis of the time spent performing each of the operations (see Figure G-2) revealed that more than 90% of this latency was due to cryptographic key generation to support the SSL connection. Therefore, cryptography, not the alert processing, was the major system bottleneck. The central problem lay in the fact that the semantics of communication required creating and tearing down a network connection with every batch of alerts sent. With each connection, a new cryptographic key needs to be generated.

In an attempt to reduce the alert processing latency, an SSL-specific optimization technique was added. The prototype made use of session caching whereby each endpoint (sensor and collector) "remembers" the key that was used between them in the previous session. This caching technique eliminates the need to generate a new key for every connection and allows the expensive SSL key generation operation to be amortized over a number of connections. Implementing this technique lowered the average latency to merely 42 ms (see Figure G-1).

A re-analysis of the time spent in each operation with session caching revealed that network latency, as well as, collector server processing time were now the most expensive operations (see Figure G-2). Effectively, the cost of cryptography had been eliminated in the average case. Even with session caching, session initiation accounted for about 25% of the total latency, but in real numbers this percentage was only 10 ms. This optimization increased the speed of the prototype by an order of magnitude. With these results, it follows that the performance benefits of HTTP (clear-text transport) instead of HTTPS (encrypted transport) is effectively moot.

Several other performance motivated choices were made with regard to inter-process communication (IPC) in the collector server. Instead of using disk-based shared memory via memory mapping (`mmap`), a shared state was established using System V kernel semaphores and shared memory segments. Using this implementation of IPC eliminated any reliance on the efficiency of a disk cache for fast access. In addition, there are several other transport protocol based performance "tweaks" such as HTTP keep-alive semantics and `mod_ssl` fast SSL re-negotiations which were not fully investigated, but it is believed that they could yield very minor additional benefits.

6.7. Scalability

The current implementation of the architecture functions as expected in a single collector-to-many sensor scenario sustaining up to 22 alerts/second. The major strain on the infrastructure was the sheer quantity of alerts, independent of which sensors submitted them. Thus, getting 10,000 alerts from a single sensor rather than one alert from 10,000 sensors will yield comparable performance. This result is a function of the state-less semantics of the collector server (HTTP) which require a connection from the sensor to the collector to be established, processed, and torn-down for every batch of alerts.

6.8. Portability

There can be no expectation of a common platform when deploying an infrastructure across the Internet (and administrative domains). Every organization could potentially be running a different configuration of hardware, software, or network topology. As a result of this heterogeneity, any proposed design could only make use of open and well-defined protocols to ensure maximum compatibility.

There is an obvious difficulty in implementing software in such a diverse environment. Each of the components must not only work on the platform it was developed but also with all other major operating systems. In order to facilitate this portability, all efforts were made to use only ANSI-C in the prototype and avoid all platform dependent assumptions. Furthermore, all the underlying libraries were explicitly chosen for the highest portability.

Table 6-1. Prototype components portability

Architecture	Component	Linux	OpenBSD	Solaris	Win32
Sensor	Snort	X	X	X	X
	DB logging (MySQL)	X	X	X	X
	HTTPS logging (OpenSSL)	X	X	X	X

Architecture	Component	Linux	OpenBSD	Solaris	Win32
Collector	Apache 1.3.14	X	X	X	X
	mod_ssl	X	X	X	X
	mod_air	X			
	libxml 2.0.0	X	X	X	X
Backing Store	MySQL 2.23.22	X	X	X	X
Analysis	PHP 4.0.2	X	X	X	X
	ACID 0.9.5	X	X	X	X

Table 6-1 illustrates that the prototype was largely successful in ensuring portable across platforms. Only the `mod_air` component remains limited to UNIX variants due to the use System V IPC mechanisms (semaphores [R12] and shared memory [R13]) to achieve a shared state in Apache. However, migrating to a more portable IPC mechanisms such as Ralf S. Engelschall `mm`[4] would be trivial to implement.

6.9. Security

The entire prototype infrastructure was explicitly designed and implemented with security concerns in mind.

6.9.1. Authentication and Authorization

Each of the communicating components of the system is identified via an X.509 certificate. There is no inherent trust relationship between components and mutual authentication is always employed. Such a scheme not only prevents the obvious attack from a rogue sensor submitting bogus alerts, but also precludes the possibility of sensors submitting alerts to a rogue collector. Each sensor is explicitly configured to submit alerts to a particular collector.

Validating a sensor's credentials is a two step process. Initially, `mod_ssl`, the SSL functionality of Apache, will verify the integrity of the certificate and confirm that the issuing certificate authority is acceptable. When these criteria are not met, the connection is terminated even before the prototype routines are invoked. If the

certificate is deemed acceptable, `mod_air`, the alert processing module, is called. `Mod_air` further verifies the certificate by checking it against a certificate revocation list (CRL) to ensure that the certificate is still valid, as well as whether any other limitations have been placed on the user. Only after this two-phase process is a sensor considered authenticated. The authorization to send an alert to the collector is implicit. All sensors which have a valid certificate are also authorized to submit alerts.

One of the issues that always arise when using certificates is the difficulty of implementing the CRL. The real-time process of connecting to another CA to ask for validation on a certificate is very expensive and rarely implemented in Internet-scale applications. As a consequence, certificates are given an indefinite lifetime because their revocation is almost never verified. However, the architectural design of this system mitigates this issue by virtue of the fact all certificates are issued by a single, self-signed certificate under the same administrative control as the collector server. Thus, there is only one CRL that needs to be checked, and updates to the revocation database are immediately used.

Deploying a collector server and certificate issuer in the same administrative domain yields several benefits. The user community that can communicate with the collector server is closed and well defined. Those entities that want a certificate must register through the collector server's CA. This increases security and lowers the authorization complexity because the mere ability to make an SSL connection to the collector is contingent on having a proper certificate. If arbitrary certificates from trusted CAs such as Verisign [5] and Thawte[6] were accepted, any entity could at least connect to the server. In such a scenario, another level of authorization beyond a mere certificate would be required.

A known user community can also increase the speed of authentication. All certificates can be initially screened on having the proper certificate issuer (CA) without being passed to the automated incident response prototype (`mod_air`). Therefore, only valid requests (and requests from sensors which were only allowed to communicate) are ever passed up to the prototype for further examination.

6.9.2. Confidentiality

The encryption used by TLS provides a confidential "pipe" between the sensor and collector ensuring that the privacy of every alert is maintained. The underlying cryptography employs only strong algorithms: RSA for key negotiations and RC4 for

bulk encryption. Likewise, sufficient key size [R43] are used (1024-bit RSA, and 64-bit RC4) to ensure that brute-force as well as mathematically based cryptanalysis attacks (e.g. differential, know-plaintext [R44]) are computationally infeasible.

6.9.3. Integrity

The cryptography used by TLS also ensures that the integrity of all data in the communication channel between the sensor and collector. Using strong hashing algorithms (SHA, MD5), a message authentication code (MAC) is generated for every packet rendering man-in-the-middle attacks ineffective. Thus, an attacker will not be able to modify an alert stream sent to the collector without the changes being detected.

In order to preclude rouge sensors (with valid certificates) from sending bogus or garbage alerts, heavy input validation is done on every SNML encoded alert. Since SNML is an XML document, there exist two techniques to verify its validity: check for well-formed-ness and DTD validation. Well-formed XML dictates that all open tags have a corresponding close tags. The collector server only accepts well-formed alerts. The more rigorous testing of DTD validation where each alert document is actually compared against the document definition is explicitly avoided for performance reasons. In order to achieve the equivalent effect of DTD validation, hard-coded logic is used to validate the data-type of elements, verify that mandatory entities are present, etc. While more efficient, such an implementation necessitates a more complex parsing algorithm and a source-code level change (sometimes quite extensive) with every change in the SNML DTD.

6.9.4. Availability

The availability of the collector server to accept and aggregate alerts from various sensors is of paramount concern. In order help manage the high-loads from many sensors, several load-balancing techniques can be adopted from web technology. Round-robin DNS can be used to spit the load among several identical collector servers. Furthermore, priority (and additional load-balancing) can be introduced on the sensors via a layer-7 switch (application switch). Priority may be important in order ensure that "more reliable, strategically place, or representative" sensors get preferential treatment.

The collector servers will be high profile targets for denial of service (DoS) attacks.

However, attacks at layer 3 and 4 of the OSI model (e.g. SYN floods [R3], Smurfing [R4], and dDoS) are outside the scope of this system. The effect of these attacks can be mitigated with varied degrees of effectiveness with known techniques [R5]. Furthermore, the use of challenge-response messages in TLS/SSL eliminates the possibility of a replay-based flood attack.

At the application layer (layer-7), the prototype provides explicit protection against DoS; a legitimate sensor flooding a collector with alerts. The collector assigns each sensor a quota of alerts that can be sent per threshold of time. Thus, even if a rouge, but properly authenticated sensor, attempts to flood the collector, these excessive alerts will be dropped as soon as the sensor's alert quota has been exceeded.

6.9.5. Security failure consequences

As per the requirements, the implementation architecture attempts to contain the effect of subversion as increasingly significant components are compromised. As implemented, the following are the effects on the system if a particular architectural component is compromised:

- A *compromised sensor* forces all post-compromise data from the single sensor to be considered suspect. The attacker who subverted the sensor will have the sensor's private key whereby allowing him to send bogus, forged alerts. Nevertheless, the data of no other sensor is affected. As the number of sensors grows, this type of security breach is inevitable because the sensor runs in a different policy domain each with potentially variable regard for "good" security practices.
- A *compromise in the issuer (root) certificate* would involve the exposure of the certificate authority's private key. Since, the CA's key is used to generate all certificates in the system, the attacker could forge any certificate whereby impersonating any other sensor. This impersonation is possible even without compromising the sensor in question. When the issuer's private key is disclosed, the post-compromise logged alert data from all the sensor is suspect. This type of security breach would undermine the entire system and would require every sensor to be reissued a certificate. The private key of the certificate issuer must be tightly secured.
- A *compromised collector server* forces all post-compromise logged data stored in the

database from all sensors communicating with this server to be suspect. Likewise, the confidentiality of all stored data is lost. Since each collector server has a database credential (username, password) which gives it the privilege to read all records and add new ones into the database, the attacker can use this login information to completely bypass the collector server and directly access the database.

- A *compromise of the backing store (database)* forces all stored data to be suspect. With this level of subversion, the attacker has violated the confidentiality of the alerts (i.e. can read all records); and can modify or delete any alert. Compromising the database is tantamount to undermining the entire system.
- A *compromise of the analysis engine* is also equivalent to compromising the database itself. The effective credentials used by the analysis component allows all the malicious activity possible under a subverted database.

In many of the previously identified scenarios, only the data generated after the time of compromise is declared suspect. While this distinction seems quite evident in the theoretical, practical consideration make drawing a conclusion about the integrity of the data difficult. It may be all but impossible to clearly determine when an architectural component was subverted. Therefore, it may be required to err on the side of caution and rollback to a last known "safe" state.

Another important issue to consider is the implications of compromises to the organization administrating the collector server. If any of the significant system components are subverted (e.g. collector, database, analysis engine), this will be a severe blow to the trust relationship held with the reporting organizations (those with sensors). Such a compromise could impede the future ability of this organization to collect data.

6.10. Reliability

The prototype infrastructure has been used for several months in a mixed test (high alert volume) and production (low volume DSL connections) environment with success.

6.11. Affordability

The entire prototype can be build from open-source components. Thus, all software can be acquired at no cost. However, this does not preclude the need for adequate hardware and a skilled staff in information security, network design, and database administration.

6.12. Extensibility

As a prototype, the components were largely designed as a proof-of-concept. Therefore, the specific implementation was carefully crafted such that extensions could be made. However, there is an understanding that as a work in progress, community review will probably require significant changes to the original code-base. It may also prove true that as public scrutiny is applied, certain system components will change dramatically and the originally implementation may need to be entirely rewritten.

6.13. Usability

The implementation went to great length to automate all pieces of the system. The greatest difficulty of using and installing all the components lies in acquiring and building all the underlying libraries.

Notes

1. http://www.insecure.org
2. http://www.andrew.cmu.edu/~rdanyliw/snort/index.html
3. http://www.postgresql.org
4. http://www.engelschall.com/sw/mm/
5. http://www.verisign.com
6. http://www.thawte.com

Chapter 7. Conclusions

Significant progress has been made in automating the process of collecting security event data across administrative domains.

7.1. Deficiencies

There are several shortcomings in the current prototype implementation.

- *Unidirectional communication semantics:* Due to an HTTP-based implementation, communication between the sensor and collector server can only be sensor-initiated. The collector server can convey information back to the sensor only by passing it back in the response message to any sensor submitted alert.
- *Database scalability:* In environments with multiple simultaneous readers and writers, MySQL does not scale well due to a locking granularity of an entire table. The consequences of this shortcoming are most evident when using the ACID tool to process the database in real-time in parallel to the collector also logging alerts. These concurrent operations can result in either component being blocked waiting to access the database; typically, the blocked process will be the lower priority reader, ACID.
- *Database design:* The collector and local database are not fully normalized. Specifically, the full text string of the detected signature is stored in each record instead of as a numeric key which is a foreign key into a lookup table as dictated by Third Normal Form (3NF) [R31]. This design flaw was purposely left unresolved in the collector database in order to maintain strict compatibility with the local sensor database. All efforts were made to only add tables and fields, rather than remove field in order to preserve a database schema compatibility that would allow the same analysis tools (e.g. ACID) to be used for both databases.

7.2. Current Status

The prototype has been completed and has enjoyed over a month of extensive testing. An architecture where multiple sensors send more than two alerts per second to a

collector server has been validated.

7.3. Future Work

7.3.1. Collector Server Architecture

The prototype demonstrates that a multi-sensor, single collector server architecture is possible. However, when considering scalability, performance, and security factors, such a topology is not desirable for Internet-wide usage. Rather, multiple collector servers, which have the capability to exchange data with each other, seems like a more viable solution. It can be easily envisioned that large organization may want to run their own collector server that must communicate with another collector.

There are at least two viable topologies in which to deploy collectors: hierarchy and mesh (peer-to-peer). A hierarchical collector architecture would work very similar to the current DNS infrastructure of central authority. It would designate a clear relationship between the collectors and the flow path of reported alerts, but would allow collectors to be single points of failure. A mesh topology would create a peer-to-peer relationship between the collector servers decentralizing authority, but would require defining the relationship between the collectors.

The semantics of communication for inter-collector communication would also need to be defined when multiple collectors are introduced. Would there be "master" collectors to whom other collectors report to as sensors? Would there be any granularity in what information gets passed between collectors; a meta-alert definition language? Furthermore, the issue of how to store alerts if collectors report to each other must be addressed. Could the same alert be stored at two collectors?

Finally, the complexity of authentication and authorization will be greatly compounded with a multi-collector model. At a minimum, multiple self-signed certificate authorities will likely be required. Likewise, designing a richer authorization schema will have to be investigated to enforce access control when sensors can submit more alerts to more than one collector.

7.3.2. Deploying prototype-aware sensors

The current prototype uses only Snort sensors to gather alerts. However, building a Snort-centric infrastructure is inadequate. The Internet community uses a wide variety of other IDSes whose alerts should also be aggregated. Furthermore, limiting alert collection to merely IDSes is shortsighted since many other types of devices and applications produce interesting security events which should also be analyzed (e.g. firewalls, routers).

In expanding the base of sensors in the reporting infrastructure, it will not always be possible (e.g. COTS) or desirable (e.g. for performance reasons) to modify the output facilities of the sensors to log alerts remotely. Instead, a "translator architecture" may need to be considered that will act as an intermediary between a sensor and collector, converting the sensor generated propriety-encoded alerts into the abstracted SNML encoding format.

Independent of which sensors are ultimately used, the type of organization and network in which they are deployed will directly affect the "quality" of the data collected. In order to assess the security of the Internet, a representative sample of the population must be running the sensor software.

7.3.3. Detecting Events

Critical to detecting attacks is the underlying signature set. The prototype will be faced with a number of challenges due to the size and heterogeneous environment in which it must operate.

7.3.3.1. Common Alert Naming Scheme

As alerts are collected from a gambit of sensors, a unique naming convention must be applied to known attacks. It must be ensured that when the same attack is detected by different sensors that these alerts are recognized as merely multiple instances of the same event. Otherwise, analysis will be imprecise and ineffective. There are several ways to reference known vulnerabilities (e.g. CVE[1], CERT[2] advisory numbers, Bugtraq[3] IDs). The challenge lies in migrating these existing efforts into IDS signatures and ensuring that these conventions remain timely.

7.3.3.2. Defining and tailoring the signature-set

A precise list of which events should be identified and studied remains an open question; analyzing at the granularity of port-scans on an Internet-scale is unfeasible. Furthermore, precise signatures with no false positives will need to be investigated.

More ambitious than merely tweaking a static signature-set would be a dynamic set distributed through some pre-defined infrastructure. This infrastructure would enable the automatic updating of new signatures as new attacks arise or different analysis criteria are identified.

7.3.3.3. Building an Abstract Signatures Language

As the number of sensor-types grows, so does the number of different signature languages that must be supported. Re-implementing the same attack description in a variety of proprietary languages is an undesirable situation in the large. Instead, a common representation for attack signatures would alleviate this difficulty. The responsibility would lie with sensor vendors to implement an attack description standard.

7.3.4. Assessing the encoding format and protocol flexibility

The prototype introduces a new encoding format and protocol: SNML and the feedback protocol, respectively. Both must be analyzed to ensure that they are capable of encapsulating all the necessary data and events in the most efficient manner. This extensibility is crucial to allow SNML to encode new sensor data such as cryptographically signed alerts (allowing for non-reputable data) which will also be added in the future.

7.3.5. Storing Alerts

The prototype has shown that MySQL will probably not be the best implementation of the central database. What remains unclear is what open-source or COTS product will

be able to replace it.

Independent of the actual backing store implementation, the database scheme must also be further analyzed for extensibility, performance, and scalability.

7.3.6. Analysis Possibilities

The issues surrounding the data collection have largely been examined in the prototype. However, once all this data is stored, analysis techniques that can examine this amount of data must be investigated. Since the data set will contain multi-administrative domain alerts, new approaches may be required.

ACID ties directly into the future analysis work. A security architecture will have to be added to identify users and delegate authorization. This addition allows system participants to view their own data, as well as how their data fits into larger trends. This strategy is one of many techniques possible to provide participants with timely feedback.

Another participant feedback approach would draw on a publisher/subscriber paradigm whereby users identify events that concern them via a "subscription". When the event (or series of events) matches the prescribed subscription, the participant is notified. It is only through adequate and useful feedback that the sensor community will grow.

7.4. Final Thoughts

Believing that current security event analysis techniques are myopic in focusing only on single administrative domains, our initial interest was to study methods of correlation and analysis to automatically detect attacks that impact multiple administrative domains. At the onset, there was no technology available to enable the collection of these prerequisite datasets in a reasonable way. Therefore, we changed our thesis focus to concentrate on enabling such data collection efforts.

By providing mechanisms for the collection of data originating from multiple administrative domains, this research begins to tackle the security dependency problem using automated technology rather than human effort. Having laid the foundation with the collection of structured data sets, analysis techniques can be employed to enable

automatic detection of Internet-wide security events and identification of existing trends. Furthermore, the fruits of this analysis may provide effective risk management strategies by uncovering predictive indicators for looming threats.

Notes

1. http://cve.mitre.org
2. http://www.cert.org
3. http://www.bugtraq.org

Appendix A. Snort Database plug-in documentation

A.1. README.database file included with snort

```
I. Summary

The database output plug-in enables snort to log to Postgresql, MySQL,
or any unixODBC database. This README contains information about how
to set up and configure your database for use with snort and how to
configure the database plugin.

There is a web site at "http://www.incident.org/snortdb" that will
always have the most up to date information and documentation about
this plug-in. Questions or comments about the database plugin can be
directed to Jed Pickel <jed@pickel.net> or to the snort-users mailing
list.

Database logging for snort would not be possible without the help,
contribution of code, comments, and debugging from many people. Listed
here are some of the folks that have been very helpful in making this
happen.

  Credits:
     Todd Schrub <tls@cert.org>
        * author of initial code for MySQL
        * helped design initial database architecture

     Roman Danyliw <rdd@cert.org> <roman@danyliw.com>
        * submitted structure for storing options
        * all kinds of help with design and code

     Yen-Ming Chen <yenming.chen@foundstone.com>
        * helped with initial database design and testing
        * author of the first analysis applications based on
          snortdb

     Geoff the UNIX guy <galitz@uclink.berkeley.edu>
        * developer of applications based on snortdb

     Bill Marquett <wlmarque@hewitt.com>
        * snortdb power user
        * bug squasher

     Mike Anderson <mike@src.no>
        * provided a lot of useful feedback on the database format

     George Colt <colt@ojp.usdoj.gov>
        * contributed fix for machines that need libm to link to
          mysqlclient library.
```

```
II. Database Setup

To get this plug-in working you must have a database set up and
configured properly. Take the the following steps to get things
working.

   1) Install MySQL, Postgresql, or (unixODBC + some other RDBMS)
        MySQL      => http://www.mysql.org
        Postgresql => http://www.postgesql.org
        unixODBC   => http://www.unixodbc.org

   2) Follow directions from your database vendor to be sure your
      RDBMS is properly configured and secured.

   3) Follow directions from your vendor to create a database for
      snort.
         MySQL example
         % echo "CREATE DATABASE snort;" | mysql -u root -p

   4) Create a user that has privileges to INSERT and SELECT
      on that database.
      example
         - First create a user - for this example we will use "jed"
         - now grant the right privileges for that user
         > grant INSERT,SELECT on snort.* to jed@localhost;

   5) Build the structure of the database according to files supplied
      with snort in the "contrib" directory as the user created in
      step 4.

      Do this while in the snort source directory.

      For MySQL
      % mysql < ./contrib/create_mysql

      For Postgresql
      % psql < ./contrib/create_postgresql

      If you are using unixODBC, be sure to properly configure and
      test that you can connect to your data source (DSN) with isql
      before trying to run snort.

      For RDBMS other than MySQL and Postgresql that are accessed
      through ODBC you will need to create the database
      structure yourself because datatypes vary for different
      databases. You will need to have the same column names and
      functionality for each column as in the mysql and
      postgresql examples. The mysql file is the best example to
      follow since it is optimized (given that mysql supports tiny
      ints and unsigned ints). I intend to document this process
      better in the future to make this process easier.

      As you create database structure files for new RDBMS mail
      them in so they can be included as part of the distribution.


III. Plugin Configuration
```

```
You must add some information to the snort configuration file
to enable database logging. The configuration file distributed
with snort has some sample configuration lines.

The configuration line will be of the following format:

   output database: [log | alert], [type of database], [parameter list]

Arguments:

   [log | alert] - specify log or alert to connect the database
       plugin to the log or alert facility. In most cases you will
       likely want to use the log facility.

   [type of database] - You must supply the type of database. The
       possible values are mysql, postgresql, and unixodbc.

   [parameter list] - The parameter list consists of key value
       pairs. The proper format is a list of key=value pairs each
       separated a space.

       The only parameter that is absolutely necessary is "dbname".
       All other parameters are optional but may be necessary
       depending on how you have configured your RDBMS.

       dbname - the name of the database you are connecting to

       host - the host the RDBMS is on

       port - the port number the RDBMS is listening on

       user - connect to the database as this user

       password - the password for given user

       sensor_name - specify your own name for this snort
             sensor. If you do not specify a name one will be
             generated automatically.

       encoding - Because the packet payload and option data is
             binary, there is no one simple and portable way to
             store it in a database. BLOBS are not used because they
             are not portable across databases. So I leave the
             encoding option to you. You can choose from the
             following options. Each has its own advantages and
             disadvantages:

           hex: (default) Represent binary data as a hex string.

                storage requirements - 2x the size of the binary

                searchability..... - very good

                human readability... - not readable unless you
                                         are a true geek
                                       requires post processing

           base64: Represent binary data as a base64 string.
```

```
                storage requirements - ~1.3x the size of the binary

                searchability..... - impossible without post
                                       processing

                human readability... - not readable
                                       requires post processing

            ascii: Represent binary data as an ascii string. This is
                the only option where you will actually loose data.
                Non ascii data is represented as a ".". If you choose
                this option then data for ip and tcp options will
                still be represented as "hex" because it does not
                make any sense for that data to be ascii.

                storage requirements - Slightly larger than the
                                       binary because some characters
                                       are escaped (&,<,>)

                searchability..... - very good for searching for
                                       a text string
                                     impossible if you want to
                                       search for binary

                human readability... - very good

        detail - How much detailed data do you want to store? The options
                are:

                full: (default) log all details of a packet that
                      caused an alert (including ip/tcp options and
                      the payload)

                fast: log only a minimum amount of data. You severely
                      limit the potential of some analysis
                      applications if you choose this option, but
                      this is still the best choice for some
                      applications. The following fields are logged
                      - (timestamp, signature, source ip,
                      destination ip, source port, destination
                      port, tcp flags, and protocol)

   The configuration I am currently using is MySQL with the database
   name of "snort". The user "jed@localhost" has INSERT and SELECT
   privileges on the "snort" database and requires a password of
   "xyz". The following line enables snort to log to this database.

   output database: log, mysql, dbname=snort user=jed host=localhost password=xyz

IV. Changelog:

2000-10-05: Created README.database and removed documentation from
               spo_database.c
2000-10-03: Added sensor_name configuration option
2000-09-29: Added configuration option enabling user to connect
               the plugin to the alert or log facility
            Changed name from spo_log_database to spo_database
            Removed all old references to the log facility
            Fixed a logic error that prevented messages from
```

```
                the portscan preprocessor to be logged.
2000-08-24: Fixed the full logging of tcp fields
            Added encoding and detail to sensor table
            Added escaping for the ascii character '
            Added hex binary logging support
            Added detail and encoding to sensor table
            Slightly changed data table to make more sense
            Added encoding option so you can select hex, base64,
               or ascii for logging binary data
            Added the "detail" option so you can choose between
               full and fast logging.
2000-08-23: A lot of code cleanup.
            Added linked list to store queries before they are
               executed.
            Added all tcp, udp, and icmp fields
            Added support for tcp and ip options
            Added support for logging the packet payload
2000-08-14: Added usage, very verbose error messages and other
               small fixes. No real functional changes. This update
               is focused on making the plugin easier to install
               and configure.
2000-06-06: Multiple instantiations is now working
2000-06-06: Added restart and cleanexit functions
2000-06-02: Bugfixes, better error reporting
2000-05-09: Bugfixes, documentation fixes, and added some
            better error reporting
2000-04-13: Bugfixes
2000-04-03: Updated database structure
2000-03-28: Added unixODBC support
            Added MySQL support
            Changed database structure
2000-03-08: Added new table "sensor" and a new field to event
            table to represent the sensor
2000-03-08: Added locking on inserts to eliminate concurrency
               problem
2000-03-08: Changed "type" and "code" in icmphdr to int2 instead
               of char
2000-03-01: Added extra argument to RegisterOutputPlugin
2000-02-28: First release
```

A.2. Function documentation

`SetupDatabase()`

Purpose: Registers the output plugin keyword and initialization function into the output plugin list. This is the function that gets called from InitOutputPlugins() in plugbase.c.

`DatabaseInit(u_char *)`

Purpose: Calls the argument parsing function, performs final setup on data structs and links the preproc function into the function list.

```
ParseDatabaseArgs(char *)
```

Purpose: Processes the preprocessor arguements from the rules file and initialize the preprocessor's data struct.

```
void FreeQueryNode(SQLQuery * node)
```

Purpose: Free the datastructure containing a linked list of buffered queries.

```
SQLQuery * NewQueryNode(SQLQuery * parent)
```

Purpose: Append another query to the data structure containing a linked list of buffered queries.

```
Database(Packet *, char * msg, void *arg)
```

Purpose: The is the main callback function that handles inserting alerts into the database when an event is detected by Snort.

```
Insert(char * query, DatabaseData * data)
```

Purpose: Database independent abstraction function for SQL inserts

```
Select(char * query, DatabaeData * data)
```

Purpose: Database independent function for SQL selects that return a non zero int.

```
Connect(DatabaseData * data)
```

Purpose: Database independent function to initiate a database connection

```
Disconnect(DatabaseData * data)
```

Purpose: Database independent function to close a connection

```
void DatabasePrintUsage()
```

Purpose: Used to print usage of the plugin when incorrect arguments are supplied.

```
void SpoDatabaseCleanExitFunction(int signal, void *arg)
```

Purpose: Disconnect from database and free memory when snort exits.

```
void SpoDatabaseRestartFunction(int signal, void *arg)
```

Purpose: Disconnect from database and free memory when snort restarts.

Appendix B. Database Schema

There are two database schemas implemented in the prototype architecture: an alert storage schema (see Section B.1) and the certificate authority schema (see Section B.2).

B.1. Snort and Collector Schema

Figure B-1. Snort and Collector database ER diagram

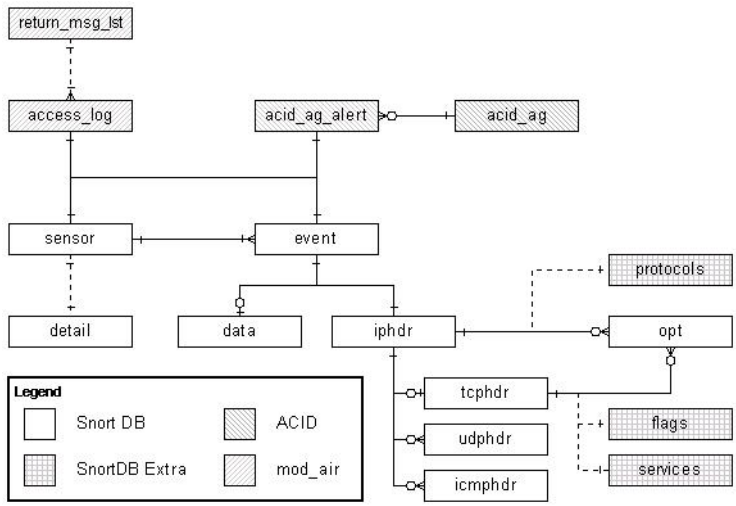


Table B-1. Snort and Collector table schema

Table	Component	Description
`sensor`	Snort	Sensor name
`event`	Snort	Meta-data about the detected alert
`data`	Snort	Contents of packet payload
`iphdr`	Snort	IP protocol fields

Table	Component	Description
`tcphdr`	Snort	TCP protocol fields
`udphdr`	Snort	UDP protocol fields
`icmphdr`	Snort	ICMP protocol fields
`opt`	Snort	IP and TCP options
`detail`	Snort	(lookup table) Level of detail with which a sensor is logging
`protocols`	SnortDB extra	(lookup table) Layer-4 (IP encoded) protocol list
`services`	SnortDB extra	(lookup table) TCP and UDP service list
`flags`	SnortDB extra	(lookup table) TCP flag list
`acid_ag`	ACID	Meta-data for alert groups
`acid_ag_list`	ACID	Alerts in each alert group
`access_log`	mod_air	Collector server to sensor transaction log
`return_msg_list`	mod_air	"Feedback" protocol return codes

B.2. Certificate Authority Schema

Figure B-2. Certificate Authority database ER diagram

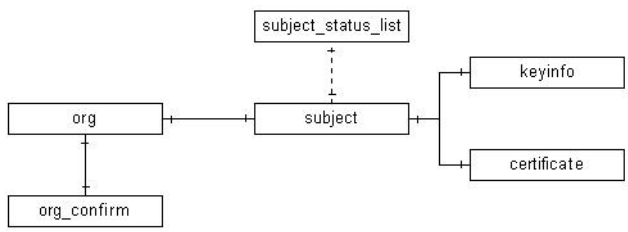


Table B-2. CA table schema

Table	Component	Description
`subject`	CA	Meta-data about each X.509 certificate
`keyinfo`	CA	Public Key of certificate
`certificate`	CA	PEM encoded X.509 certificate
`subject_status_list`	CA	(lookup table) Status code list of a certificate
`org`	CA	Registration information from each participating organization
`org_confirm`	CA	Temporary table to store un-confirmed accounts

Appendix C. Snort XML plug-in documentation

C.1. README.xml file included with snort

```
Snort XML Output Plug-in

I. Summary

The XML plug-in enables snort to log in SNML - simple network markup
language aka (snort markup language) to a file or over a network.  The
DTD is available in the contrib directory of the snort distribution
and at: http://www.cert.org/DTD/snml-1.0.dtd. You can use this plug-in
with on one or more snort sensors to log to a central database and
create highly configurable intrusion detection infrastructures within
your network. The plugin will also enable you to automatically report
alerts to the CERT Coordination Center, your response team, or your
managed IDS provider.

This plugin was developed by Jed Pickel and Roman Danyliw at the CERT
Coordination Center as part of the AIRCERT project.

Be aware that the SNML DTD is in its early phases of development and
is likely to be modified as it undergoes public scrutiny.

See http://www.cert.org/kb/snortxml for the most up to date
information and documentation about this plugin.

II. Configuration

You must add some information to the snort configuration file
to enable database logging. The configuration file distributed
with snort has some sample configuration lines.

The configuration line will be of the following format:

   output xml: [log | alert], [parameter list]

Arguments:

   [log | alert ] - specify log or alert to connect the xml
       plugin to the log or alert facility.

   [parameter list] - The parameter list consists of key value
       pairs. The proper format is a list of key=value pairs each
       separated a space.

       file     - when this is the only parameter it will log to
                  a file on the local machine.  Otherwise, if
                  http or https is employed (see protocol), this is
                  the script which is to be executed on the remote
```

```
           host.

protocol - The possible values for this field are

           http  - send a POST over HTTP to a webserver
                   (required: a [file] parameter)

           https - just like http but ssl encrypted and
                   mutually authenticated.
                   (required: a [file], [cert], [key]
                    parameter)

           tcp   - A simple tcp connection. You need to
                   use some sort of listener
                   (required: a [port] parameter)

           iap   - An implementation of the Intrusion Alert
                   Protocol (This does not work yet)

host     - remote host where the logs are to be sent

port     - The port number to connect to
           (default ports are)
           http  80
           https 443
           tcp   9000
           iap   9000

cert     - the client X.509 certificate to use with https
           (PEM formatted)

key      - the client private key to use with https
           (PEM formatted)

ca       - the CA certificate used to validate the https
           server's certificate (PEM formatted)

server   - the file containing a list of valid servers with
           which to communicate. It is used so that Snort can
           authenticate the peer server. Each server is
           identified by a string formed by concatenating
           the subject of the server's X.509 certificate.
           This string can be created by:

             % openssl x509 -subject -in <server certificate>

           Typically only someone deploying the HTTPS will have
           to perform this task (since they have access to the
           server certificate). This entitity should publish
           this subject string for configuration inside each
           snort sensor.

sanitize - The argument is a a network/netmask combination for
           an IP range you wish to be sanitized. Any IP address
           within the range you specify will be represented as
           "xxx.xxx.xxx.xxx". Also, for sanitized alerts, no
           packet payload will be logged. You can use the
           sanitize parameter multiple times to represent
           multiple IP ranges.
```

```
encoding - Packet payload and option data is binary and
           there is not one standard way to represent it as
           ASCII text. You can choose the binary encoding
           option that is best suited for your environment.
           Each has its own advantages and disadvantages:

      hex: (default) Represent binary data as a hex string.

           storage requirements - 2x the size of the binary

           searchability..... - very good

           human readability... - not readable unless you
                                    are a true geek
                                  requires post processing

      base64: Represent binary data as a base64 string.

           storage requirements - ~1.3x the size of the binary

           searchability..... - impossible without post
                                    processing

           human readability... - not readable
                                  requires post processing

      ascii: Represent binary data as an ascii string. This is
             the only option where you will actually loose
             data.  Non ascii data is represented as a ".". If
             you choose this option then data for ip and tcp
             options will still be represented as "hex" because
             it does not make any sense for that data to be
             ascii.

           storage requirements - Slightly larger than the
                                  binary because some characters
                                  are escaped (&,<,>)

           searchability..... - very good for searching for
                                    a text string
                                  impossible if you want to
                                    search for binary

           human readability... - very good

detail - How much detailed data do you want to store? The options
         are:

           full: (default) log all details of a packet that
                 caused an alert (including ip/tcp options and
                 the payload)

           fast: log only a minimum amount of data. You severely
                 limit the potential of some analysis
                 applications if you choose this option, but
                 this is still the best choice for some
                 applications. The following fields are logged
                 - (timestamp, signature, source ip,
```

```
                      destination ip, source port, destination
                      port, tcp flags, and protocol)

Examples:

output xml: log,   file=output
output xml: log,   protocol=https host=air.cert.org file=alert.snort
                   cert=mycert.crt key=mykey.pem ca=ca.crt server=srv_list.lst


     PROTOCOL     tcp        http       https      iap
    ========================================================
    | file   ||    no    | required | required |    no    |
    |========||==========|==========|==========|==========|
    | host   || required | required | required | required |
    |========||==========|==========|==========|==========|
    | port   || required | optional | optional | optional |
    |========||==========|==========|==========|==========|
    | cert   ||    no    |    no    | required | optional |
    |========||==========|==========|==========|==========|
    | key    ||    no    |    no    | required | optional |
    |========||==========|==========|==========|==========|
    | ca     ||    no    |    no    | required | optional |
    |========||==========|==========|==========|==========|
    |server  ||    no    |    no    | required |    no    |
    |========||==========|==========|==========|==========|
    |sanitize|| optional | optional | optional | optional |
    ========================================================

III. Change Log

    08/25/2000 : Added "encoding" and "detail" configuration options
    08/14/2000 : Initial Release

IV. TODO

  * still need to get IAP working
  * add expiration dates/CRL into server certificate validation
  * support multiple HTTPS servers
  * real queue management on alerts: batching, using congestion
    feedback
```

C.2. Function documentation

C.2.1. Callback functions

```
void LogXml(Packet *, char * msg, void * data)
```

Purpose: This function is called when an event is detected by the Snort detection engine that is configured to log to XML.

```
XmlInit(u_char *)
```

Purpose: Called when Snort first initializes. This routine contains all of the initialization code for the XML plugin.

```
XmlExit(int signal, void *arg)
```

Purpose: This routine is called when snort exits. It cleans up by freeing memory and flushing any buffered data.

```
XmlRestart(int signal, void *arg)
```

Purpose: Similar to XmlExit but this is called when snort receives a HUP signal.

C.2.2. Initialization Routines

```
SetupXml()
```

Purpose: Register the output plugin with snort

```
ParseXmlArgs(char *)
```

Purpose: Process arguments supplied by user in configuration file.

C.2.3. XML Generating Functions

```
Tag * newtag(char * name)
```

Purpose: create a new XML tag

```
Tag * addtag(Tag *parent, Tag *child)
```

Purpose: connect a child tag to a parent

```
Tag * addopt(Tag * tag, char * name, char * value)
```

Purpose: add options and values to an XML tag

```
Tag * addvalue(Tag * tag, char * val)
```

Purpose: inserts a value for an XML tag

```
Tag *snml(XmlData *d, Packet *p, char *msg)
```

Purpose: simple network markup language encoder

```
char *tag2string(char *buf, int size, Tag * ptr, int depth)
```

Purpose: convert datastructures to XML

C.2.4. Networking Code

```
send_data(XmlData * data)
```

Purpose: Write data to a file or over the network

```
BrokenNetConnection(int signal)
```

Purpose: Print errors on a broken connection

```
send_data_network(XmlData *d, char *output)
```

Purpose: Send XML-formatted output onto the wire.

C.2.5. SSL Functions

```
init_snort_ssl_ctx(XmlData *d)
```

Purpose: Initialize an TLSv1 context

```
EVP_PKEY * load_key(char *key_fname)
```

Purpose: Loads a PEM encoded RSA Private key from disk into memory

```
X509 * load_crt(char *ca_fname)
```

Purpose: Loads a PEM encoded .CRT file from disk into memory

```
VerifyServerCertificate(X509 *current, char *valid_server,
X509 *issuer)
```

Purpose: Validates a certificate based on an issuer certificate

```
SSLServerResponse * ProcessSSLServerResponse(char
*rbuf)
```

Purpose: Process response from server

C.2.6. Other

```
snort_return_msg_index(int msg_code)
```

Purpose: check return codes after sending alert

```
int srcSanitized(XmlData *d, Packet *p)
```

Purpose: check to see if the user wants the source IP address to be sanitized

```
int dstSanitized(XmlData *d, Packet *p)
```

Purpose: check to see if the user wants the destination IP address to be sanitized

```
void flush_data(XmlData *d)
```

Purpose: flush an alert queue

```
void freetag(Tag * root)
```

Purpose: free memory associated with internal XML datastructures

Appendix D. SNML DTD

```
<!ENTITY version "1.0">

<!ELEMENT event (sensor, signature, timestamp, packet)>

<!- The sensor field contains information that can be used to
     uniquely identify the source of where this event was
     detected. It always contains a "hostname" and optionally a
     filter. And you have the option of including a file (the
     file being the source of data), or an ip address and
     network interface. ->
<!ELEMENT sensor (file|(ipaddr, interface?)), hostname, filter?>

<!- The signature is just free-
form text. In snort it is the string
     contained in the "msg" variable ->
<!ELEMENT signature #PCDATA>

<!- The timestamp must conform to ISO-8601 standard.
     e.g. ISO-8601: 1999-08-04 00:01:23-05 ->
<!ELEMENT timestamp #PCDATA>

<!- A packet can be logged without being decoded if you use
     "raw" mode. The only time you would re-
ally want to do this is
     if you were receiving a packet containing proto-
cols you don't
     have the ability to decode. ->
<!ELEMENT packet (raw|iphdr)>

<!- IP address only. Anything else is rejected. This means no
     domain names. The version attribute is the version of IP
     address (Should be 4 or 6).->
<!ELEMENT ipaddr #PCDATA>
<!ATTLIST ipaddr
          version   CDATA  #REQUIRED
>

<!- This field contains an ordinary hostname ->
```

```
<!ELEMENT hostname #PCDATA>

<!- This contains a file name with a full path ->
<!ELEMENT file #PCDATA>

<!- This field contains an ordinary hostname ->
<!ELEMENT hostname #PCDATA>

<!- Contains a string representing a network interface
     ie. eth0, ppp0, hme0, etc ->
<!ELEMENT interface #PCDATA>

<!-
A string representing a tcpdump filter that is normally passed
     in on the command line.
     ie. "not net 10.1.1.0/24" ->
<!ELEMENT filter #PCDATA>

<!- raw contains a base64 representation of a binary packet ->
<!ELEMENT raw #PCDATA>

<!- IPv4 header
     saddr   = source IP address      -
IP address  IP  (192.168.1.2)
     daddr   = destination IP address -
IP address  IP  (192.168.1.2)
     ver     = version of ip          - 1 byte INT (0 - 15)
     hlen    = header length in 32 bit words
                                      - 1 byte INT (0 - 15)
     tos     = type of service        - 1 byte INT (0 - 255)
     len     = total length of the packet
                                      - 2 byte INT (0 - 65535)
     id      = identification         - 2 byte INT (0 - 65535)
     flags   = fragment flags         - 1 byte INT (0 - 7)
     off     = fragment offset        - 2 byte INT (0 - 65535)
     ttl     = time to live           - 1 byte INT (0 - 255)
     proto   = protocol               - 1 byte INT (0 - 255)
     csum    = checksum               - 2 byte INT (0 - 65535)
     ->
<!ELEMENT iphdr (tcphdr|udphdr|icmphdr), option?)>
<!ATTLIST iphdr
```

```
          saddr     CDATA  #REQUIRED
          daddr     CDATA  #REQUIRED
          ver       CDATA  #REQUIRED
          hlen      CDATA  #IMPLIED
          tos       CDATA  #IMPLIED
          len       CDATA  #IMPLIED
          id        CDATA  #IMPLIED
          flags     CDATA  #IMPLIED
          ttl       CDATA  #IMPLIED
          off       CDATA  #IMPLIED
          ttl       CDATA  #IMPLIED
          proto     CDATA  #REQUIRED
          csum      CDATA  #IMPLIED
>

<!- TCP header information
     sport   = source port            - 2 byte INT (0 - 65535)
     dport   = destination port       - 2 byte INT (0 - 65535)
     seq     = sequence number        - 4 byte INT (0 -
4294967295)
     ack     = acknowledgment number  - 4 byte INT (0 -
4294967295)
     off     = data offset            - 1 byte INT (0 - 15)
     res     = reserved field         - 1 byte INT (0 - 63)
     flags   = represents TCP flags   - 1 byte INT (0 - 255)
     win     = window                 - 2 byte INT (0 - 65535)
     csum    = checksum               - 2 byte INT (0 - 65535)
     urp     = urgent pointer         - 2 byte INT (0 - 65535)
     ->
<!ELEMENT tcphdr data, option?>
<!ATTLIST tcphdr
          sport     CDATA  #REQUIRED
          dport     CDATA  #REQUIRED
          seq       CDATA  #IMPLIED
          ack       CDATA  #IMPLIED
          off       CDATA  #IMPLIED
          res       CDATA  #IMPLIED
          flags     CDATA  #REQUIRED
          win       CDATA  #IMPLIED
          csum      CDATA  #IMPLIED
          urp       CDATA  #IMPLIED
```

```
>

<!- UDP header information
     sport   = source port            - 2 byte INT (0 - 65535)
     dport   = destination port       - 2 byte INT (0 - 65535)
     len     = length field of UDP header
                                      - 2 byte INT (0 - 65535)
     csum    = checksum               - 2 byte INT (0 - 65535)
     ->
<!ELEMENT udphdr data>
<!ATTLIST udphdr
          sport     CDATA  #REQUIRED
          dport     CDATA  #REQUIRED
          len       CDATA  #IMPLIED
          csum      CDATA  #IMPLIED
>

<!- ICMP header
     type    = icmp type              - 1 byte INT (0 - 255)
     code    = icmp code              - 1 byte INT (0 - 255)
     csum    = checksum               - 2 byte INT (0 - 65535)
     id      = identifier             - 2 byte INT (0 - 65535)
     seq     = sequence number        - 2 byte INT (0 - 65535)
     ->
<!ELEMENT icmphdr data>
<!ATTLIST icmphdr
          type   CDATA #REQUIRED
          code   CDATA #REQUIRED
          csum   CDATA #IMPLIED
          id     CDATA #IMPLIED
          seq    CDATA #IMPLIED
>

<!- This field contains a representation of data.
     The format attribute must be either base64 or ascii
     depending on how the data is logged. ascii data is
     generally the data with portions that can not be
     represented in standard characters substituted with a
     period.
 ->
<!ELEMENT data #PCDATA>
```

```
<!ATTLIST data
          format (base64|ascii) #REQUIRED
```

Appendix E. Module AIR (mod_air) documentation

E.1. Configuration

1. Uncompress the mod_air distribution.
2. `./configure`
3. `./make`
4. `./make install`
5. The existing Snort DB must have its schema updated in order to support the additional functionality of the collector (mod_air).

   ```
   % cat ./database/create_mod_snort_support | mysql <Snort DB name>
   ```

6. Configure the following variable in the Apache configuration file (`httpd.conf`):

   ```
   LoadModule air_module         libexec/mod_air.so

   # Sets mod_air to listen for any re-
   quests for files with the .air
   # extension
   AddHandler air .air

   # Log file name
   AirTraceFile /tmp/alert

   # Critical error log
   AirErrorFile /tmp/mod_snort_error

   # Sets the mod_air debug level.  Values range
   #between 0 .. 7 where
   #   7 : debug level
   # ... :
   #   0 : only server-critical
   AirLogLevel 7
   ```

```
# Configures the alert quota of the sensors.
# 'AirThrottleInterval' sets the time window in which
# to assign the quote, while 'AirThrottleThreshold'
# sets the number of alerts which can be received
# in that unit of time.
AirThrottleInterval 1
AirThrottleThreshold 10


# Identifies the Central database name and
# connection information
#  - AirDBName     : alert database name
#  - AirDBHost     : host on which the database resides
#  - AirDBPort     : port over which to connect to the database
#  - AirDBUser     : username with which to login
#  - AirDBPassword : username's password
AirDBName     snort_log
AirDBHost     localhost
AirDBPort     3306
AirDBUser     root
AirDBPassword mypassword

# Identifies the Certificate Authority
# (CA) database name and connection information
#  - AirCADBName   : CA database name
#  - AirCAHost     : host on which the database resides
#  - AirCAPort     : port over which to connect to the database
#  - AirCAUser     : username with which to login
#  - AirCAPassword : username's password
AirCADBName ca_db
AirCAHost   localhost
AirCAPort   3306
AirCAUser   root
AirCAPassword mypassword
```

7. For each sensor, generate an RSA key and certificate signing request (CSR), and submit it to the `certgen` utility. It will return a a PEM encoded X.509 certificate with no password using the same issuer as the HTTPS server.

```
% openssl genrsa -out sensor.key 1024

% openssl req -new -key sensor.key -out sensor.csr

% ./certgen < sensor.csr > sensor.crt
```

E.2. Function documentation

Table E-1. mod_air source tree

File	Description
`README, FAQ`	Documentation included in distribution
`Makefile.in, configure.in`	The scripts necessary to build mod_air
`mod_air.h`	Global variable, data structure, and function declaration
`mod_air.c`	Apache module API callback routines
`mod_air_ca.c`	Certificate Authority (CA) and authentication API
`mod_air_ipc.c`	Inter-Process Communication (IPC) - semaphore and shared memory segments
`mod_air_throttle.c`	Throttle-table management
`mod_air_log.c`	Logging facilities
`mod_air_xml.h`	XML-specific variable and data structure definitions
`mod_air_xml_sax.c`	libxml callback functions
`mod_air_xml_db.c`	XML-to-DB INSERT routines
`mod_air_xml_util.c`	XML parsing helper-utilities
`create_mod_snort_support`	Modifies and creates the database tables to support mod_air
`certgen.c`	Accepts a CSR and returns a certificate
`sslpost.c`	Utility which simulates sensor behavior by taking a XML-alert and sending it to a collector

E.2.1. Apache Callbacks

```
void air_init(server_rec *s, pool *p)
```

Purpose: (Apache callback) module initializer, called once per server record in the

Apache configuration

Performs the basic initializations for the global data structures that are shared among Apache daemons:

- validates the parameters from httpd.conf
- inits the semaphore for DB access
- Snort DB and CA DB connection handles setup
- log file handles are opened
- throttle table shared memory created

The function is called once per every server_rec before the child pool is spawned. For example, if you have an HTTP and HTTPS server are running from the same httpd process, then at a minimum there will be two server records (i.e. two calls to this function). This routine is run from the parent process of all the httpd processes which are fork()ed.

Argument:

- s => server record
- p => memory pool

```
void * create_dir_mconfig(pool *p, char *dir)
```

Purpose: (Apache callback) A per-directory configuration structure initializer. This routine also inits the global shared variables

Argument:

- p => module memory pool
- dir => directory for which callback was issued

```
int air_handler(request_rec *r)
```

Purpose: (Apache callback) content handler of each SNML alert.

This routine is the entry point for all the processing done by the module. Initially all request (across all the processes) are blocked on a common semaphore waiting to run the critical section which will process and commit the submitted alert (via POST). To process an alert the following steps are followed. Note: it is possible to abort at any stage and return an error.

- return MIME type
- block on access to mutex (all httpd blocking)
- authentication: check client certificate
- alert processing: check congestion/throttle
- alert processing: SAX parsing
- alert processing: commit alert(s) to DB
- return status to client
- release mutex

Argument: r => request record from which to read the data

```
int air_fixup(request_rec *r)
```

Purpose: (Apache callback) fix-up. Last chance to add information to the environment or to modify the request record. Check to make sure this is not a sub-request.

Argument: r => request record

Returns: DECLINED => status on whether this module should handle the request

```
void air_child_init(server_rec *s, pool *p)
```

Purpose: (Apache callback) Child initializer.

Performs the basic initializations for the httpd child:

- attach to the shared memory throttle table
- increment the number of existing children

When this routine is called for the first time (for the first child), the throttle table values are initialized.

The function is called when an instance of the httpd daemon is spawned (forked()) from the parent http process. There will be many instances of httpd and therefore calls to this function.

Argument:

- s => server record of associated child
- p => child's memory pool

```
void air_child_exit(server_rec *s, pool *p)
```

Purpose: (Apache callback) Child exit. Performs the deallocations required before an instance of the child exits.

- decrement the number of outstanding children
- detach from any shared memory
- close our handle to any log files

Since Apache does not have a callback for a module exit (despite the fact that there is a module init), we need to always check if this particular child is the last one to exit. It is the last child's responsibility to deallocate all the global data structures (database semaphores/mutexes, throttle table).

The function is called for every instance of the httpd daemon when it receives a signal to terminate from the parent httpd process. It should be called for every child which child_init().

Argument:

- s => server record of associated child
- p => child's memory pool

E.2.2. Apache Callback helpers

```
int read_content (request_rec *r, char *data, long length)
```

Purpose: reads the POST data from stdin handling all the details of Apache data chunking

This is the crucial code which reads the POST data from stdin. It was unceremonially 'liberated' and modified from mod_cgi.c in the Apache core.

Argument:

- r => request record for which to read the data
- data => pointer to store POST data
- length => expected length of POST data

Returns: # of bytes read

E.2.3. Certificate Authority API

```
int auth_snort_client(X509 *client_cert,
                      unsigned *long subject_id)
```

Purpose: Accepts an X.509 certificate (typically passed by the client), and validates its status against the CA DB.

Argument:

- client_cert => X.509 certificate to validate
- subject_id => sets this to be the id # of the certificate

Returns: status of the certificate

- 1 => authenticated
- 0 => recognized, but revoked certificate
- -1 => unknown unknown certificate

E.2.4. Inter-Process Communication

```
void semaphore_init (int *semid, key_t semkey)
```

Purpose: creates and initializes a System V kernel semaphore

Argument:

- semid => ID # of the newly created semaphore
- semkey => unique key number with which to create a semaphore

```
void semaphore_wait(int semid)
```

Purpose: blocks waiting on the semaphore to be available (P)

Argument: semid => ID # of of semaphore to block on

```
void semaphore signal(int semid)
```

Purpose: signals semaphore to be available (V)

Argument: semid => ID # of of semaphore to signal

```
void semaphore_destroys(int semid)
```

Purpose: destroys a System V semaphore

Argument: semid => ID # of of semaphore to destroy

E.2.5. Logging Facilities

```
unsigned long snort_log_access(time_t when, request_rec *r,
                               long cert_id, long throttle,
                               long auth, long commit,
                               long num_alerts, long sid,
                               long cid)
```

Purpose: Log the arrival/processing of an alert. Writes a record into the access log about the current connection. A record is made for every connection attempt.

Argument:

- when => time when alert(s) arrived
- r => request record of the HTTP connection over which the alert arrived
- cert_id => ID of the X.509 certificate of the client
- throttle => throttle status code
- auth => authentication status code
- commit => commit status code
- num_alerts => # of alerts in the POST
- sid => sensor ID on which the alert was detected
- cid => last event ID of the alert

Returns: ID of the newly inserted row in the access log database, or -1 if there is an error

```
void snort_log_malformed(time_t when,
                         unsigned long access_log_id,
                         long cert_id, unsigned long sid,
                         char ip_address, char xml_input)
```

Purpose: Logs any malformed XML that cannot be parsed. Used to detect any possible attacks exploitable through crafting the XML input (e.g. buffer overflow)

Argument:

- when => when alert(s) arrived
- access_log_id => ID # of entry in the access_log
- cert_id => ID of the X.509 certificate of the client
- sid => sensor ID on which the alert was detected
- ip_address => IP address of the client which sent the alert
- xml_input => raw XML of the alert

```
char * snort_return_msg_string(int msg_code)
```

Purpose: return code translation; code # => text description

Argument: msg_code => numeric code of return message

Returns: text equivalent of the return message code

```
void snort_response_string_print(request_rec *r, int_msg_code)
```

Purpose: returns a message to the client

Argument:

- r => request record of the connection
- msg_code => numeric code of message

```
void air_log_msg(int severity, char format, ...)
```

Purpose: logs a message into the trace file

Argument:

- severity => indicates the logging level of the event [0,7]
- format => format string of message
- ... => variable-length parameter list of message

```
void FatalError(const char format, ...)
```

Purpose: Write a fatal error message to stderr and dies

Argument:

- format => format string of message
- ... => variable-length parameter list of message

```
void ErrorMessage(const char format, ...)
```

Purpose: Write a non-fatal error message to stderr

Argument:

- format => format string of message
- ... => variable-length parameter list of message

E.2.6. Connection Throttling

```
int check_throttle(unsigned long sid)
```

Purpose: returns throttle information on a particular certificate (user)

Argument: sid => certificate ID for which to return throttle info

Returns: throttle status

- THROTTLE_OK : "no" congestion
- THROTTLE_CONGESTION: congestion
- THROTTLE_QUENCH: exceeded allowable # of alert per time
- THROTTLE_DENIED: throttle unable to be determined, denied

```
void InitThrottleTable()
```

Purpose: (Throttle Table) Initializes

Argument:

- client_cert => X.509 certificate to validate
- subject_id => sets this to be the id # of the certificate

Returns: status of the certificate

- 1 : authenticated
- 0 : recognized, but revoked certificate
- -1 : unknown unknown certificate

```
void DeallocateThrottleTable()
```

Purpose: (Throttle Table) Deallocates throttle table

```
int RemoveOldAlertInfo(SensorEntry *alert_history, time_t cur-
rent_time)
```

Purpose: (Arrival Queue) Evaluates which packets are still in the relevant time window defined by the ’current_time’ and ’alert_throttle_duration’ and removes all those alerts that are older.

Argument:

- alert_history => entry in ThrottleTable to consider
- current_time => alert arrival time

Returns: number of alerts which were dropped because they were no longer in the current time window

```
void InsertNewAlertInfo(SensorEntry *alert_history,
                        time_t current_time)
```

Purpose: (Arrival Queue) Adds a newly arrived alert time into the associated sensor’s alert arrival queue.

Argument:

- alert_history => entry in ThrottleTable to consider
- current_time => alert arrival time

```
void CreateAlertEntry(SensorEntry *alert_history)
```

Purpose: (Arrival Queue) Deals with the situation where there is an overflow in the arrival queue. This routine ’makes’ space by dropping the oldest entry (i.e. at front) and makes it available for enqueuing

Argument: alert_history => entry in ThrottleTable to consider

```
time_t AlertHistoryHead(SensorEntry *alert_history)
```

Purpose: (Arrival Queue) Returns the front (next to be dequeued) arrival time of a particular sensor in the ThrottleTable

Argument: alert_history => entry in ThrottleTable to consider

Returns: head of the arrival queue of a particular ThrottleTable entry

```
time_t AlertHistoryDequeue(SensorEntry *alert_history)
```

Purpose: (Arrival Queue) Dequeues a new alert arrival time from the front of a particular arrival queue of a sensor in the ThrottleTable

Argument: alert_history => entry in ThrottleTable to consider

Returns: newly dequeued arrival time from the queue

```
void AlertHistoryEnqueue(SensorEntry *alert_history,
                         time_t value)
```

Purpose: (Arrival Queue) Enqueues a new alert arrival time to the end of a particular arrival queue of a sensor in the ThrottleTable

Argument:

- alert_history => entry in ThrottleTable to consider
- value => alert arrival time to add into the queue

```
int AlertHistoryFull(SensorEntry *alert_history)
```

Purpose: (Arrival Queue) Checks and returns whether the queue of alert arrival times (alert history) for a particular sensor entry in the ThrottleTable is full.

Argument: alert_history => entry in ThrottleTable to consider

Returns: boolean of whether the queue is full

```
void PrintAlertHistory(SensorEntry *alert_history)
```

Purpose: (Arrival Queue) Prints the queue of alert arrival times (alert history) for a particular sensor entry in the ThrottleTable

Argument: alert_history => entry in ThrottleTable to consider

```
unsigned long htbl_hash(unsigned long sid)
```

Purpose: (Hash Table) Hashes a key into a hash value

Argument: sid => certificate ID (key into the ThrottleTable)

Returns: hashed value of key

```
unsigned long htbl_rehash(unsigned long sid)
```

Purpose: (Hash Table) Re-Hashes a key into a hash value. Typically invoked after a collision.

Argument: sid => certificate ID (key into the ThrottleTable)

Returns: hashed value of key

```
SensorEntry * htbl_get(unsigned long sid)
```

Purpose: (Hash Table) Retrieves the corresponding entry from the ThrottleHash table based on a certificate ID.

Collision resolution is addressed with nonlinear re-hashing; that is, re-hashing a another hash to find an alternate location in the table. This algorithm is undesirable but is implemented since other, better technique (external chaining) will require using dynamically allocated memory which cannot be used because of shared memory limitation/complexity

Argument: sid => certificate ID (key into the ThrottleTable)

Returns: pointer to the corresponding sensor's entry in the table OR NULL if too many collisions occurred

```
unsigned long htbl_hash_function(char *raw_key)
```

Purpose: (Hash Table) This function takes an arbitrary length string and hashes it returning a long. The underlying algorithm is taken from the hash code used in the UNIX ELF format for object files.

Argument: raw_key => value to hash

Returns: hashed equivalent of 'raw_key'

E.2.7. XML Processing

```
int xmldb_main(unsigned long id, char *data, int size,
               int *success_commits, int *throttle_state,
               long *sid, long *cid)
```

Purpose: Entry point into the SAX parsing of an alert:

- clean the XML of illegal characters
- validate that have SNML XML header
- invoke libxml2 SAX parser
- < callbacks get invoked >
- return status of XML parse

Argument:

- id => X.509 serial number id of the sensor sending the alert
- data => XML alert stream
- size => length of XML alert stream
- success_commits => returns back number of alerts written successfully from the XML stream
- throttle_state => returns back throttle information for connection

Returns: a status code on the processing the XML stream

E.2.8. XML SAX Callbacks

```
void startDocument(void *ctx)
```

Purpose: (SAX callback) Called at the beginning of a XML document

Argument: ctx => current parser context

```
endDocument(void ctx)
```

Purpose: (SAX callback) Called at the end of a XML document

Argument: ctx => current parser context

```
void startElement(void *ctx, const xmlChar *name,
                  const xmlChar **atts)
```

Purpose: (SAX callback) Called when an element is encountered.

Argument:

- ctx => current parser context
- name => name of triggering element
- atts => array (1-dim) holding the element attributes. The structure of the array is as follows:
  - atts[i] : name of field, e.g. ’len’
  - atts[i+1] : value of field, e.g. ’32’

```
void endElement(void *ctx, const xmlChar *name)
```

Purpose: (SAX callback) Called when an end element is encountered.

Argument:

- ctx => current parser context
- name => name of triggering element

```
void characters(void *ctx, const xmlChar *ch, int len)
```

Purpose: (SAX callback) Called when characters between entities are encountered.

Argument:

- ctx => current parser context
- ch => characters
- len => length of characters

E.2.9. XML-to-DB Abstraction

```
void WriteAlert(alert_instance *alert)
```

Purpose: Execute SQL statements generated by processing of alerts into a MySQL database.

- Validate that have correct number of statements
- write ’event’ table
- write ’iphdr’ tbl, ’option’ tbl (ip options)
- write layer-4 tbl (tcphdr, udphdr, icmphdr), ’option’ tbl (tcp options)
- write ’data’ tbl (payload)

Argument: alert_instance => current context of parser

```
void InitDatabase(alert_instance *ctx)
```

Purpose: Determines the correct sensor ID of the current alert (via the sensor table in the DB). If this is a new sensor, it is added to the database.

This routine is called at the beginning of every alert

Argument: ctx => current context of parser

```
int GetNextCID(alert_instance *ctx)
```

Purpose: Given a sensor ID, returns the next available event ID

Argument: ctx => current context of parser

Returns: next available event ID (cid) for a given sensor (sid)

```
void LogEvent(alert_instance *ctx)
```

Purpose: Writes a row into the ’event’ table.

This routine must be called for each alert processed.

Argument: ctx => current context of parser

```
void LogEventHeader(alert_instance *ctx, int protocol,
                    const char **header)
```

Purpose: Writes a row into a protocol header table (iphdr, tcphdr, udphdr, icmphdr)

This routine must be called at a minimum twice for each alert processed (1x for iphdr, 1x for the layer 4-protocol)

Argument:

- ctx => current context of parser
- protocol => for which protocol to write protocol header (IP, IPPROTO_TCP, IPPROTO_UDP, IPROTO_ICMP)
- header => array (1-dim) holding the protocol header values. The structure of the array is as follows:
  - atts[i] : name of field, e.g. 'len'
  - atts[i+1] : value of field, e.g. '32'

```
void LogEventHeaderOptions(alert_instance *ctx, int protocol)
```

Purpose: Writes a row into a the protocol options table ('option'). The option data is stored in the 'ctx'.

This routine must be called once for every option (ip or tcp)

Argument:

- ctx => current context of parser
- protocol => for which protocol to write protocol header (IP, IPPROTO_TCP, IPPROTO_UDP, IPROTO_ICMP)

```
void LogEventPayload(alert_instance *ctx)
```

Purpose: Writes a row into a the payload table ('data')

Not all alerts will have payload, and it is possible to configure the sensor not to send payload at all.

Argument: ctx => current context of parser

```
int Insert(char *query)
```

Purpose: (DB Abstraction) Inserts a SQL statement into a MySQL database

Argument: query => SQL statement to execute

Returns: success of running the SQL statement

- 1 : success
- 0 : failure

```
int Select(alert_instance ctx, char  query)
```

Purpose: (DB Abstraction) Runs a SELECT query against a MySQL database which will result in a single row being returned. From this row, a single numeric field will be extracted and returned.

This routine is primarily used to retrieve the unique ID of a row with a particular criteria.

Argument:

- ctx => current parser context
- query => SQL statement to execute

Returns: single field found by the query

- > 0 : query result
- 0 : failure

```
MYSQL * Connect(char t_dbname, char t_host, char t_port,
                char t_user, char t_password)
```

Purpose: (DB Abstraction) Obtains a handle to a MySQL database

Argument:

- t_dbname => database name
- t_host => database host

- t_port => database port on host
- t_user => database username
- t_password => database password for username

Returns: handle to the database

- > 0 : good DB handle
- 0 : failure

E.2.10. Alert Parsing Helpers

```
void ContextInit(alert_instance *alert, unsigned long id)
```

Purpose: Initializes the alert context. Sets the global variables of the parser related to the entire XML stream.

This function should be called only once per XML stream.

Argument:

- alert => parser context to initialize
- id => X.509 serial number of sensor sending the alert

```
void ContextReInit(alert_instance *alert)
```

Purpose: Initializes the alert context between every individual alert in the stream. May be called multiple times per single XML stream (ideally once per every alert).

Argument: alert => parser context to initialize

```
void ContextFullFree(alert_instance *alert)
```

Purpose: Deallocates and frees the alert context.

This function should only be called once per XML stream.

Argument: alert => parser context to deallocate

```
void ContextFullFree(alert_instance *alert)
```

Purpose: Deallocates the alert context between individual alerts. Frees the specific variables local to a single alert in the stream.

May be called multiple times per single XML stream (ideally once per alert).

Argument: alert => parser context to deallocate

```
unsigned int IPOctets2Int(unsigned int o0, unsigned int o1, un-
signed int o2,
                          unsigned int o3)
```

Purpose: Given 4 octets of an IP address, returns a 32-bit integer

Argument:

- o0 : XXX.x.x.x : byte 3 of the IP address
- o1 : x.XXX.x.x : byte 2 of the IP address
- o2 : x.x.XXX.x : byte 1 of the IP address
- o3 : x.x.x.XXX : byte 0 of the IP address

Returns: 32-bit representation of an IP address from 4 octets

```
void CleanXMLStream(char *stream, int len)
```

Purpose: Takes an XML string and cleans out all characters deemed to be illegal.

Argument:

- stream => XML string to clean
- len => length of 'stream'

```
char * SQLClean(char *stream)
```

Purpose: Takes a string and cleans out all characters that would be illegal in a SQL statement. This routine presupposes that CleamXMLStream() was already run on the string.

Argument: stream => SQL string to clean

Returns: cleaned SQL string

```
int CheckSQLOverflow(char *sql, int buffer_size)
```

Purpose: Verifies whether a generated sql string is malformed. Essentially, we assume that if the string is the same size as the buffer length (a condition which should never occur), then some piece of data is overflowing the buffer.

Argument:

- sql => SQL string to check
- buffer_size => length of SQL string

Return: boolean of whether the SQL statement has overflowed

```
void AddTag(alert_instance *ctx, char *tag)
```

Purpose: (Tag stack) Adds a tag to the top of the stack

Argument:

- ctx => alert parser context
- tag => tag to add

```
void RemoveTag(alert_instance *ctx)
```

Purpose: (Tag stack) Removes the top element from the tag stack

Argument: ctx => alert parser context

```
int CheckTag(alert_instance *ctx, int num, ...)
```

Purpose: (Tag stack) Verify that the top tag of the stack has the correct depth as well as is embedded (tags prior) in the correct order.

Never compare the bottom of the stack (this is the dummy tag), or the very top (the current element being verified)

Argument:

- ctx => alert parser context
- num => number of tags to compare
- ... => (variable number of parameters) strings against which to compare previous tag. First parameter = bottom of stack, last parameter is (top of stack) - 1

Return: boolean on whether the top tag is correct

```
char * top(alert_instance *ctx)
```

Purpose: (Tag stack) return the top of the tag stack

Argument: ctx => alert parser context

Return: top element of tag stack or NULL if empty

```
void RemoveAllTag(alert_instance *ctx)
```

Purpose: (Tag stack) removes all tags from tag stack

Argument: ctx => alert parser context

```
void PrintTags(alert_instance *ctx)
```

Purpose: (Tag stack) print the tag stack

Argument: ctx => alert parser context

Appendix F. Feedback protocol specifications

This appendix documents all possible codes that the collector server can return with the feedback protocol.

There are two classes of messages sent back to a sensor by the collector: status and error. As the name implies, status messages are informational or non-fatal error messages. Multiple status messages can be returned per each connection to the collector. On the other hand, error messages are fatal messages caused by a graceful abort. Only the first error message will be returned to the sensor; it will be the last message in the the message stream.

The collector performs the following actions:

1. Authentication
2. Throttle control
3. Input processing

As a consequence of this task ordering, messages will appear in the following order: AUTH_*, THROTTLE_*, INPUT_*, OK. It is possible that not all actions will be performed due to an error.

F.1. OK (1xx)

```
STATUS 100 OK
```

Indicates a successful authentication and logging of all alerts into the database.

F.2. Authentication codes (3xx)

```
STATUS 300 AUTH_CLIENT_OK
```

Indicates that the client has been authenticated.

`ERROR  301 AUTH_CLIENT_DENIED`

Indicates that the client has been authenticated, but its certificate has been revoked. The corresponding alerts sent with this credential are dropped.

`ERROR  302 AUTH_CLIENT_UNKNOWN`

Indicates that the client certificate was signed by the collector CA, but is not in the CA database. This condition should never happen and indicates problems with the certificate registration authority. The corresponding alerts sent with this credential are dropped.

`ERROR  303 AUTH_CLIENT_IGNORED`

Indicates that the CA could not be contacted to validate the client certificate. The corresponding alerts sent with this credential are dropped.

F.3. Input Processing codes (4xx)

`STATUS 400 INPUT_COMMIT_OK`

Indicates that the alerts were successfully logged to the backing store.

`ERROR  401 INPUT_COMMIT_ERROR`

Indicates that some (unknown) error occurred and one or all commits to the backing store failed. Since transactions are not implemented, there could be consistency issues in the backing store.

`ERROR  402 INPUT_EMPTY`

Indicates that alerts were allegedly received but they appear to have no substance (i.e. no content).

`ERROR  403 INPUT_MALFORMED`

Indicates that the XML parser (libxml) believes that the alert is not well-formed XML

`ERROR  404 INPUT_INVALID`

Indicates that the validation logic believes that the XML does not conform to the SNML DTD. This error is similar to ERROR 406, but was most likely triggered due to maliciously crafted XML: infinite depth, processing routines called in wrong order, crucial data committed or illegal.

```
ERROR  405 INPUT_PARSER_ERROR
```

Indicates that an internal error occurred in the XML parser (libxml).

```
ERROR  406 INPUT_INCOMPLETE
```

Indicates that the validation logic found missing fields when generating the write command (INSERT) for the database.

```
ERROR  407 INPUT_DB_READ_FAIL
```

Indicates that the collector could not read a record from the database (probably when trying to get a sensor ID).

```
ERROR  408 INPUT_OVERFLOW
```

Indicates that the collector has received a data stream in excess of what can be stored. This usually indicates an attempt to perform a buffer-overflow.

```
ERROR  409 INPUT_IGNORED
```

Indicates that the collector received an alert, but was not able to process it due to an internal server difficulty (e.g. high load).

F.4. Throttling codes (5xx)

```
STATUS 500 THROTTLE_OK
```

Indicates that alerts were accepted and the sensor is within the accepted throttle/commit rate.

```
STATUS 501 THROTTLE_CONGESTION
```

Indicates that the alerts were accepted, but the sensor is approaching the maximum threshold of allowable alerts per unit time. This threshold is configurable at the collector.

`STATUS 502 THROTTLE_QUENCH`

Indicates that the alerts were dropped since the number of allowable alerts in a time window were exceeded by the client.

`STATUS 503 THROTTLE_DENIED`

Indicates that there is either a corrupt or full throttle table since the required connection entry cannot be retrieved. Any connection receiving this error will be throttled and its alerts dropped.

`STATUS 504 THROTTLE_IGNORED`

Indicates that an internal error has accord with the throttle table. Any connection receiving this error will be throttled and its alerts dropped.

Appendix G. Detailed Performance Analysis

The following configuration defines the testing parameters used to analyze single sensor-to-collector performance.

- *Architecture:* `Snort, Logging server (Apache), and DB server (MySQL) all on same host`
- *Host:* `Pentium II 266, 64MB RAM - Linux`
- *Snort version:* `1.7-beta0`, first release of spo_xml
  - *cmd line:* `./snort -c snort.rules -i lo -d`
  - *logging:* `output xml: log, host=128.2.243.68 port=443 file=alert.snort cert=./post.crt key=./key.pem ca=./ca.crt server=AIR_CERT_Collector`
  - *Alerts per Connection:*`1`
  - *CipherSpec:* `TLSv1, EDH-RSA-DES-CBC3-SHA`
  - *Trigger event:* `nc -G4 -g 127.0.0.1 -g 127.0.0.1 -g 127.0.0.1 -g 127.0.0.1 127.0.0.1 44`
- *Apache version:* `Apache/1.3.12 (Unix) mod_ssl/2.6.4 OpenSSL/0.9.5a mod_snort/0.8.1`
- *MySQL version:* `3.22.32`, connected to via a local UNIX socket
- *Network:* `1000 packets`

Figure G-1. Alert processing time comparison

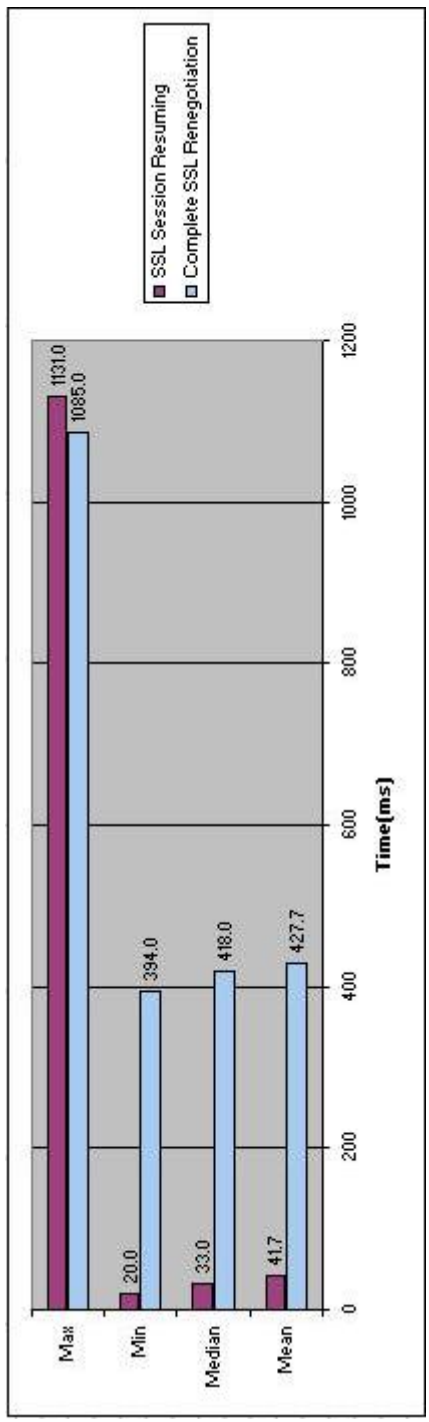
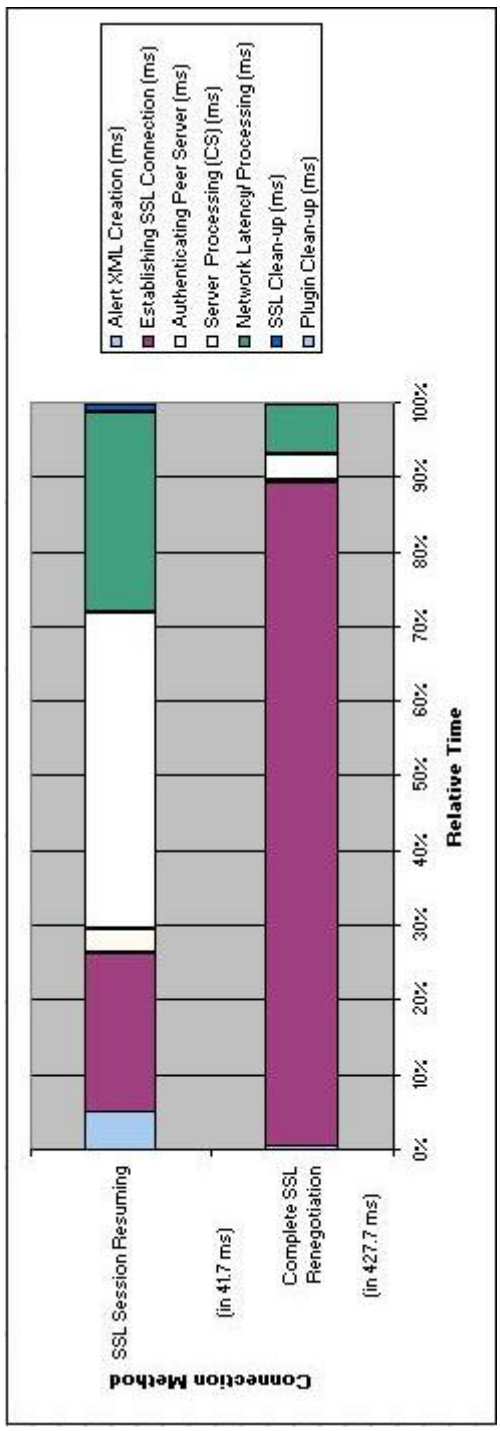


Figure G-2. Processing an Alert: Percent time in each operations



Glossary

Given that there is not an established set of terms for the Internet security industry these terms are defined to ensure the reader will understand terms used in the context of this thesis.

administrative domain

A network or networks under the sovereign control of a single individual or organization.

alert

A representation of an a security event and the supporting data. Alerts are either stored in a database or encoded as XML documents.

black hat

An attacker with malicious intent; the "bad guy". This word was chosen because the other alternatives (e.g. attacker, intruder, hacker, cracker) tend to have multiple overloaded meanings.

incident

A collection of data representing one or more related attacks. Attacks may be related by attacker, type of attack, objectives, sites, or timing [R7].

false negative

Malicious activity that was not detected as suspicious. This type of failure is viewed as very severe because an actual attack was missed.

false positive

Activity that is flagged as suspicious when in reality they are normal. This error is typically the result of imprecise or misconfigured thresholds or attack signatures. Too many false positives generate extreme numbers of IDS alerts whereby

obscuring the legitimate events of concern, as well as causing a human tendency to ignore future, legitimate instances of this alert as also erroneous.

security event

A security event is an instance of malicious activity.

signature

A representation or description of a particular type of security event.

white hat

A security professional with intent to improve security rather than exploit weaknesses; the "good" guy.

Bibliography

Due to the mercurial nature of the security domain, much of the information is first published (and sometimes only) on the Web. Thus, many of the references may be modified or invalid in the future. If there are any questions about these entries, please email roman@danyliw.com or jed@pickel.net.

References

[R1] T. Dierks and C. Allen, *RFC2246: The TLS protocol version 1.0*, January 1999.

[R2] S. Kent and R. Atkinson, *RFC2401: Security Architecture for the Internet Protocol*, November 1998.

[R3] CERT/CC, *CERT Advisory CA-1996-21 TCP SYN Flooding and IP Spoofing Attacks*, http://www.cert.org/advisories/CA-1996-21.html, 1996.

[R4] CERT/CC, *CERT Advisory CA-1998-01 Smurf IP Denial-of-Service Attacks*, http://www.cert.org/advisories/CA-1998-01.html, 1998.

[R5] CERT/CC, *Distributed-Systems Intruder Tools Workshop*, http://www.cert.org/reports/dsit_workshop-final.html, Pittsburgh, November 2-4, 1999.

[R6] COAST, *Intrusion Detection*, http://www.cerias.purdue.edu/coast/intrusion-detection/detection.html.

[R7] *State of Practice of Instrusion Detection Technologies*, CERT/CC, Julia Allen, et. al., January 2000.

[R8] Aurobindo Sundaram, *An Introduction to Intrusion Detection*, ACM Crossroads, April 1996.

[R9] Thomas H. Ptacek and Timothy N. Newsham, *Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection*, Secure Networks Inc., January 1998, http://www.clark.net/~roesch/idspaper.html.

[R10] R. Housley, W. Ford, W. Polk, C. Allen, and D. Solo, *RFC2459: Internet X.509 Public Key Infrastructure Certificate and CRL Profile*, January 1999.

[R11] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, and et. al., *RFC2068: Hypertext Transfer Protocol - HTTP/1.1*, June 1999.

[R12] Michael Lemmon, *System V Semaphores*, http://www.nd.edu/~lemmon/courses/UNIX/15/15.htm.

[R13] Guy Keren, *Unix Multi-Process Programming and Inter-Process Communication (IPC)*, http://users.actcom.co.il/~choo/lupg/multi-process/multi-process.html.

[R14] Barry M. Leiner, Vinton G. Cerf, David D. Clark, Robert E. Kahn, Leonard Kleinrock, Daniel C. Lynch, Jon Postel, Larry G. Roberts, and Stephen Wolff, *A Brief History of the Internet*, August 4, 2000, http://www.isoc.org/internet-history/brief.html.

[R15] R. Pethia, S. Crocker, and B. Fraser, *Guidelines for the Secure Operation of the Internet*, November, 1991, http://info.internet.isi.edu/in-notes/rfc/files/rfc1281.txt.

[R16] Cory Cohen, *Incident Reporting Guidelines*, 1998, http://www.cert.org/tech_tips/incident_reporting.html.

[R17] Klaus-Peter Kossakowski and Moira West-Brown, *International Infrastructure for Global Security Incident Response*, June 4, 1999, http://www.cert.org/inter_infra/inter_infra.pdf.

[R18] CERT/CC, *CERT/CC Overview, Incident and Vulnerability Trends*, http://www.cert.org/present/cert-overview-trends/sld225.htm.

[R19] Maureen Stillman, *What's Next for IDSs?*, April 1999, Information Security Magazine.

[R20] Clifford Kahn, Phillip Porras, Stuart Staniford-Chen, and Brian Tung, *A Common Intrusion Detection Framework*, July 15, 1998, Journal of Computer Security.

[R21] Brian Tung, *CIDF Interoperability Experiment Report (September 1999)*, September 1999, http://www.gidos.org/demo/september99.html.

[R22] Michael Erlinger and Stuart Staniford-Chen, *IDWG Charter*, October, 2000, http://www.ietf.org/html.charters/idwg-charter.html.

[R23] Dave Curry, *Intrusion Detection Message Exchange Format Extensible Markup Language (XML) Document Type Definition*, July 6, 2000, http://www.ietf.org/internet-drafts/draft-ietf-idwg-idmef-xml-01.txt.

[R24] Gupta, *Intrusion Alert Protocol - IAP*, March 31, 2000, http://www.ietf.org/internet-drafts/draft-ietf-idwg-iap-01.txt.

[R25] John Howard and Tom Longstaff, *A Common Language for Computer Security Incidents*, October, 1998, http://www.cert.org/research/taxonomy_988667.pdf.

[R26] Steve T. Eckmann, Giovanni Vigna, and Richard A. Kemmerer, *STATL: An Attack Language for State-based Intrusion Detection*, http://www.cs.ucsb.edu/~kemm/netstat.html/documents.html.

[R27] Jai Sundar Balasubramaniyan, Jose Omar Garcia-Fernandez, David Isacoff, Eugene Spafford, and Zamboni Zamboni, *An Architecture for Intrusion Detection using Autonomous Agents*, ftp://coast.cs.purdue.edu/pub/COAST/papers/diego-zamboni/zamboni9805.ps.

[R28] *AIDE abstract*, http://www.sans.org/NS2000/techcon.htm.

[R29] *EMERALD Project Description*, http://www.sdl.sri.com/emerald/project.html.

[R30] *Federal Intrusion Detection Network (FIDNET) Homepage*, http://www.fedcirc.gov/fidnet/.

[R31] Peter Rob and Carlos Coronel, *Database System: Design, Implementation, and Management*, Boyd & Fraser, 1995.

[R32] Andrew Cormack, Jan Meijer, and Yuri Demchenko, *Incident Taxonomy and Description Working Group Charter*, http://www.terena.nl/task-forces/tf-csirt/i-taxonomy/.

[R33] *The SHADOW Intrusion Detection System*, http://www.nswc.navy.mil/ISSEC/CID/.

[R34] Fyodor Yarochkin, *SnortNet - A distributed IDS approach*, http://snortnet.scorpions.net/snortnet.pdf.

[R35] Jed Pickel, *Automating Incident Reporting*, May, 1999, FIRST Conference Proceedings 1999.

[R36] *AIRCERT*, http://www.cert.org/kb/aircert/.

[R37] *AUSCERT - Automated Report Processing*, http://www.auscert.org.au/Information/Auscert_info/probe.html.

[R38] Jed Pickel, *INCIDENT.ORG Project*, http://www.incident.org/.

[R39] Silicon Defense, *SnortSnarf*, http://www.silicondefense.com/snortsnarf/.

[R40] *SecurityFocus Incidents Mailing List*, http://www.securityfocus.com/.

[R41] *Global Incident Analysis Center*, http://www.sans.org/giac/.

[R42] *IA Architecture, Modeling, and Mangagement*, Richard Feiertag, Stuart Staniford-Chen, Karl Levitt, Mark Heckman, and Dave Peticolas, http://www.pgp.com/research/nailabs/ia-architecture/demonstration.asp.

[R43] RSA Security, *RSA Labs FAQ: What key size should be used?*, http://www.rsasecurity.com/rsalabs/faq/4-1-2-1.html, 2000.

[R44] Bruce Schneier, *Applied Cryptography*, Wiley, 1996.

[R45] Linclon Stein and Doug MacEachern, *Writing Apache Modules with Perl and C*, O'Reilly, 1999.

Implementation Component Documentation

[C1] Ralf S. Engelschall, *Apache 1.3 Dynamic Shared Object (DSO) support*, http://www.apache.org/docs/dso.html, April 1998.

[C2] DevShed.com, *Professional Apache: Improving Apache's Performance*, http://www.devshed.com/Books/ProApache, May 18, 2000.

[C3] Apache Software Foundation, *What about the Apache Server Project*, http://httpd.apache.org/ABOUT_APACHE.html.

[C4] Martin Roesch, *What is Snort?*, http://www.snort.org/what_is_snort.htm.

[C5] James Hoagland and Stuart Staniford, *SPICE: The Stealthy Portscan and Intrusion Correlation Engine*, SiliconDefense, http://www.silicondefense.com/spice.

[C6] Martin Roesch, *Writing Snort Rules*, http://www.snort.org/writing_snort_rules.htm.

[C7] MySQL, *MySQL Manual: Chapter 8.3: ISAM Tables*, http://www.mysql.com/documentation/mysql/commented/manual.php?section=ISAM.

[C8] Tcpdump and Libpcap, *TCPDUMP Public Repository*, http://www.tcpdump.org.