SAINT: A Security Analysis Integration Tool*

Diego M. Zamboni
Computer Security Area
Dirección General de Servicios de Cómputo Académico
Universidad Nacional Autónoma de México
Apdo. Postal 20-059, 01000 México D.F., México
`diego@conga.super.unam.mx`

Abstract

This paper presents the design of SAINT, a tool being developed at the National Autonomous University of México that will allow integrated analysis of information gathered from various sources, such as security tools and system logs. By simulating events occurring in the systems, and collected from the different sources, SAINT will allow detection, or even prevention of problems that may otherwise go undetected due to lack of information about them in any single place. SAINT's modular and extensible architecture make it feasible to add new modules for processing new data types, detecting new kinds of problems, or presenting the results in different formats.

1 Introduction — The Problem

As part of the ongoing computer security activities at the National Autonomous University of México (UNAM), the use of various security tools has been promoted as one of many ways of increasing Unix system security. Until now, only freely available tools have been used, mainly because they cover most of the needs in this particular academic and research environment.

The main set of tools used consists of COPS [FS90], TCP-Wrappers [Ven92], Passwd+ [Bis95], Crack [Muf], TripWire [KS93, KS94a, KS94b] and SATAN [FV], although other tools (like Tiger [SSH93], S/Key [Hal94, HA94] and the logdaemon suite [Ven]) are also used.

Experience has shown that, when need arises to diagnose a problem, the solution often comes after collecting information from more than one source, including, but not restricted to, the tools mentioned above. For example, to trace a suspicious `su` access to **root**, it may be necessary to match a `wtmp` record with a `sulog` entry. To further trace it back to its origins, it may be necessary to match the `wtmp` record with a TCP-Wrappers log entry, go to other systems and repeat the log analyzing and matching until all the needed data is collected. The information is available, in many cases, but it is scattered all over several systems and in different formats, and it has to be collected and analyzed manually to get something more useful than just a collection of facts.

Therefore, the problem can be summarized in the following points:

- To achieve acceptable levels of security, it is necessary —among many other things, of course— to use several different tools, each one of them working in something specific (and, many times, even duplicating actions).
- Each one of these tools generates data on its own, and in different formats.
- To have a more complete view of what is happening, the system and/or security administrator has to read several reports and logs generated by the tools, often over a period of time.
- The correlations and matching between related items in the different logs has to be done manually by the administrator.

*Originally published in the Proceedings of the 1996 SANS (System Administration, Networking and Security) Conference, Washington D. C., May 12–18, 1996.

- In Mexico (and other non-English speaking countries, for sure), the fact that all the generated information is in English poses yet another problem. Although English is the *lingua franca* in computing, it is still a barrier for people (including many Unix system administrators) using computers in México.

This can, and does, lead to mis-utilization of the tools, which just sit there collecting mountains of data that nobody never uses. Recently, some tools have been released that allow easier viewing of generated data (most notably CIAC's Merlin [CIA]), but the problem still remains of making an understandable whole of the seemingly chaotic set of reports and log files.

That is why SAINT's idea was born: to make a system that allows integrated analysis of data collected from various sources, and tries to extract interesting information to be presented to the administrator in an easy to read format.

This paper presents the design of SAINT, which is still under development at UNAM's Computer Security Area[1].

2 Related work

Log file analysis is not new. In fact, it has been used for many years. In the simpler end, there are tools like Swatch [HA92, HA93] and TkLogger [Hug] which primarily do regular expression matching against log files, searching for certain patterns and doing something when they are found. These tools are useful for looking for very specific things, but since the search they perform is essentially stateless, their usefulness is restricted to looking for specific things that may indicate problems.

On the higher end, there is a tool called ASAX (Advanced Security audit trail Analyzer on uniX) [MCZH95, HCMM92], which uses a rule-based language (called RUSSEL) to process audit trails generated by a number of systems. In a distributed environment, ASAX runs local "evaluator" processes on each monitored host, which submit their local results to a master server, which in turn processes the consolidated data. Although the model is general enough to be ported to any type of system, the current implementation is oriented towards SunOS 4.1 with C2 security features, and uses PVM [GBD+94] as the communication mechanism between distributed processes.

ASAX is a very powerful package, and its rule-base analysis makes it able to detect complex event sequences that may indicate problems. However, its same complexity makes it difficult to use in a very heterogeneous environment like ours. The recommended (C2) audit mechanisms are not in place in most of our systems, and compiling ASAX in very different versions of Unix proved difficult.

3 What is SAINT?

SAINT provides the framework for performing the following functions:

1. Cross-analysis of reports and logs generated by various security tools, as well as system logs, in several Unix systems. The goal is trying to detect things (or sequences or patterns of things) that may indicate problems of any kind.
2. If it is possible, obtain information about likely causes of detected problems.
3. Warning generation when appropriate (the most clear case would be when a flagrant security problem is detected, but there are many other situations where opportune notifications are very useful).
4. If possible, suggest available solutions to detected problems.
5. In its first version, presentation of all the results in Spanish.

The main goals when designing SAINT were:

Make it extensible. It should be easy to add new modules to the system, to make it aware of new kinds of available information (for example, a new tool), or to modify or improve its analysis capabilities.

[1] More information about UNAM's Computer Security Area can be found at `http://www.super.unam.mx/asc/`

Make it configurable. Security is not the same for everyone, and the user should be able to specify, in a more or less detailed fashion, what is important for him or her, and what is not.

Make it easy to use. One of the security tool curses that SAINT is trying to avoid is "you need to be an expert to use this". Once it is in place (and it shouldn't be too difficult to do that too), it should be easy to use and to review the generated results.

Make it portable. If it is going to succeed, SAINT must be usable in heterogeneous environments with as little changes as possible. Building on the experience of other tools like COPS, SATAN and Merlin, as well as on the experience of the people working on it, most of SAINT (if not all) is being written in Perl5 [WS92, Wal] (why not admit it, SATAN also inspired the name).

4 What is SAINT *not*?

SAINT does not try to be a full-featured Intrusion Detection System (IDS), although SAINT reports can be used to detect intrusions. IDS technology is by now far ahead of SAINT's design. Advanced work on this topic is being done, among others, by Crosbie and Spafford [CS95a, CS95b], Kumar and Spafford [KS94c, KS95] and Kumar [Kum95].

SAINT is intended just as an information analysis tool. This point made, let's proceed with SAINT's design description.

5 What does SAINT do?

SAINT's operation can be divided in four big phases:

1. Data collection and homogenization.
2. Event sorting.
3. Event analysis.
4. Results presentation.

Each of this phases is described in detail next.

6 Data collection and homogenization

SAINT needs to process data produced by various tools and, possibly, various computer systems. To facilitate further processing, the first stage in SAINT's execution consists in getting the data from all the sources that will be used, and converting it to a common data format that makes it easier for the next stages to process.

This task presents the following problems:

1. The information produced by each tool comes from different places, and it is in different formats. The process of converting to the common data format is different for each data type.
2. Data formats used by some tools may change in future versions of the same tool.
3. It must be possible to add support for new tools (or new versions of existing tools) without modifying SAINT.
4. The common data format must contain all the information that may be useful for processing.

For the first 3 points, the given solution is using different program modules for each tool. These modules are independent programs that are used by SAINT (according to its configuration file) for getting and processing each tool's data.

This gives the following advantages:

- Each module is specialized in only one data type, thus it may be very simple and easy to test.
- Each module can be written in the language that is most appropriate for the kind of processing to do, as long as it complies with SAINT's interface specifications.
- Each module can get the data from wherever it is appropriate. It may get it from a file, from the network or from another program. This process is transparent to SAINT, since it is performed internally by each module.
- Modules can be easily added, replaced or modified to deal with new tools or new versions of existing tools, to correct problems or to improve processing efficiency.

The design of the common data format had to meet the following goals:

Completeness: It must include all the information it gets, only in another format. Its output must be in the form of "events", that is, discrete pieces of information as concise and simple as possible, to simplify further processing.

Extensibility: It must allow representing any data type, so the appearance of new tools doesn't force a change in the format.

Simplicity: It must be easily processable by a program. This implies that the fields must be easily distinguishable, and that the contents of each field must belong, as far as possible, to a finite and predictable data set.

The common data format is composed by lines of text representing events, and each line includes the following fields:

Event type: A keyword that allows classification of the event. These are assigned arbitrarily, according to the events the system must recognize. Some of the event types currently recognized are:

port_connect

telnet

ftp

rlogin

rsh

su_root

su_user

reboot

Most of these types are self-explanatory and need no further discussion. Perhaps it is worth explaining *su_root*, *su_user* and *port_connect*. The first refers to an `su` command executed to get **root** access, and the second refers to the same command used to get access to a non-**root** account. These events were separated since `su`'s to **root** normally require a much stricter analysis.

Port_connect represent a connection to any TCP/IP port in the system. This can be useful, for example, to detect port scans, failed login attempts, etc.

It is worth noting that new data collection and analysis (see "Event Analysis" below) modules can define new event types, as long as they deal correctly with them.

Event date and time: Since the analysis performed will be chronological, it is important that each event has a time stamp, wherever possible. If the original data doesn't include date and time, one of two solutions may be provided:

1. The corresponding module may be able to estimate the time when the event happened (for example, by interpolating from other known data). This is the preferred solution.

2. Leave the field empty, in which case the event will be considered as having happened before all the others.

Event source system: When the event involves an access through the network, this field identifies the originating system.

Event destination system: The destination system for network events.

Event source user: For some events it is possible to find out which user generated it, and the information can be useful in the analysis.

Event destination user: It may also be useful to know which user "received" the event, for example, a `telnet` session or an `su`.

General purpose field: It was considered appropriate to have a field whose meaning depends on the event that is being registered. Some examples of what this field may contain are:

- Process ID for the daemon providing a service.
- Flag indicating success or failure of an operation.
- Relevant environment variables.
- File name in an FTP transfer.
- Command executed by `rsh`.
- Terminal name for an interactive session.
- Reason for reboot.

Original message: It is impossible to foresee and classify all the information that, at any given moment, may be necessary to analyze an event. For this reason, it was considered convenient to include in the common data format the original message from which the event was detected. This will allow to perform on this field any further analysis that might not be originally considered. In this field, all non-printing characters should be substituted by their common escape-character sequences (for example, `\n` for a newline or `\t` for a tab) or their ASCII code representation in octal (for example, `\014`).

The format used for each record in the common data format is as follows:

```
type|date_time|orig_sys|dest_sys|orig_usr|dest_usr|gen|msg
```

Where the fields appear in the order they were previously mentioned. The vertical bar character (|) was chosen because it is rare in most of the reports we have seen. In case it appears in a field, it should be escaped (\|) before storing it.

7 Event sorting

The analysis performed on the events will be necessarily chronological, since this facilitates the detection of relationships between them. For this reason, once the events have been collected and converted to the common data format, they will be sorted according to their time stamps.

Events whose second field (date and time) is empty should be left at the beginning of the list, in the order they appeared in the event collection stage.

It is important to mention a possible problem here: if the clocks in the systems being analyzed are not correctly synchronized (more than a few seconds difference), this stage could probably leave events in a different order than they originated. This could affect the analysis results.

8 Event analysis

This is the stage that does SAINT's central work. Based on the data collected and processed in the previous stages, an analysis must be done that allows detection of relationships among events and trying to decide if these events are part of the systems' normal operation or represent possible problems.

Two approaches were considered for this analysis in the design: parsing and event simulation.

8.1 Parsing

This technique implies doing a parse of the event sequence, trying to detect normal or abnormal patterns. Each event would be a token, and the grammar would detect special token sequences.

This approach was discarded for the following reasons:

- Relevant event sequences may be too complex, and may not even be predictable.
- Adding support for new tools or new problem detection modules would imply, very probably, code modifications. This greatly diminishes the expandability capabilities planned for SAINT.
- In real life, more than an event sequence can occur simultaneously. This would make the resulting grammars excessively complex or even impossible to develop.
- Many event sequences may be similar or even exactly the same, and yet represent different things. Deterministic and finite-lookahead grammars have very hard times dealing with these kind of things.

8.2 Event simulation

This technique consists of doing a simulation, inside the program, of the events that affect each computer system, and the consequences they have. This is different from parsing mainly in that the possible event sequences are not predefined, only how the system reacts to events under certain circumstances.

A simulation is based on elements that respond to certain stimulations and interact between them. In this case, the elements of the simulation would be:

Systems: These are the central elements of the simulation, since they are where most events are generated and received. Each system remains in its current state until an event happens that it is programmed to react to. Based on its current state, a system can trigger certain actions (such as reporting a possible problem).

Security domains: In this case, security domains refer to groups of systems that share some characteristics related to security. The most common security domain is a group of systems that "trust" each other (at least as defined by the SAINT configuration).

The network: It is through the network that systems interact among each other. In this case, the network is considered only as a passive transport element, since we are not considering it as being able to generate events on its own. Thus, it is not considered in the simulation.

Given this view of things, the simulation technique seems appropriate for doing event analysis.

The method used for implementing it seems to suggest itself. Object-oriented programming allows us to define objects (in this case, systems and domains), each of which can have its own characteristics, have an internal state, can communicate with other objects, and can react to messages received from other objects. This is everything we need for the simulation, so we decided to use some kind of object-oriented programming technique for implementing it.

Here we faced another problem. Most of the "traditional" object-oriented languages (C++, Objective-C, Smalltalk, etc.) are not portable enough to allow us to easily use SAINT in a wide variety of systems. The choice was Perl5. In its latest incarnation, Perl includes object-oriented capabilities. This, along with Perl's great flexibility, portability, and the fact that it is free, made the decision clear. Most, if not all, of the original SAINT modules are being written in Perl5.

It was considered that the object classes used in the simulation must be the following:

Systems: These objects must contain a representation of all the factors that may affect, at any given moment, the system operation. The state of a Unix system, for our initial purposes, can be summarized with the following elements:

- Active sessions of any kind (`telnet`, `rlogin`, `ftp`, etc.) Each session has associated information about the user it belongs to, assigned terminal, date and time, etc.

- Existing processes. Not all the processes are relevant in the analysis, but those that are significant (for example, those triggered by events in the simulation) must be tracked. Each process has associated information about the user it belongs to, associated terminal, privileges, date and time, etc.
- Files. Again, not all the files are relevant, but some of them are worth tracking during the analysis. Each file has associated information about user and group it belongs to, size, access permissions, pathname, date and time, etc.

Besides this, a system and its reaction to events are defined by the following characteristics:

- System name.
- Trusted systems and domains. Normally this information is inherited from the security domain it belongs to (see below), but it may be changed for a particular system. Trusted systems and domains may be given special privileges, like establishing sessions with administrative privileges.
- Privileged users, which are allowed to do privileged administrative tasks.
- Special times. Some periods of time may be defined as "maintenance" periods, where the system may be rebooted or some special programs may be run.
- Event selection mechanism. This is the entry point for events directed to the system. This mechanism, based on the event characteristics, sends it to the appropriate event reaction routines.
- Event reaction routines. Each system has built in, as part of its definition, the capability of reacting to certain events. According to the type of event and the system's current state, the reaction may be ignoring it, modifying the system state, issuing a warning message or a notification to other object, etc.

  These routines, along with the event selection mechanism, define the events a system will react to and the problems it will be able to detect. To be able to add SAINT support for new problems, new tools, or new system types, it may be necessary to add or modify reaction routines. Here, the object-oriented approach also comes in handy. Using inheritance and Perl's great flexibility in interpreting code, SAINT can make the needed additions or modifications.

These are the basic elements of a system object. Some kinds of systems (for example, depending on the operating system version) may react differently to events. For this, new classes could be defined that inherit from the base system object class and modify their behavior (or even the data representing the internal state) according to the specific characteristics of the new system. New analysis modules could also add data fields or even reaction routines to increase the object's event-reaction capabilities.

Security domains: The state of a security domain in any given moment may be defined by the following:

- Connections established to it from the "outside" (systems that don't belong to the domain).
- Connections established from the domain to the outside.
- The state of each of the systems that belong to it. This state is maintained, independently, by each system object.

A domain is also defined by the following characteristics:

- Domain name (for reference).
- Systems that belong to it, represented by their respective objects.

8.3 Performing the simulation

Once object classes are defined the steps for performing the simulation are:

1. Create the object instances representing all the systems and domains participating in the simulation.

2. In chronological order, send each object the events in which it may be involved. Each object will receive the events through its event selection mechanism, and will react to it according to its internal state and its event reaction routines.

An object may react to an event with one or more of the following actions:

- Modify its internal state to reflect a change introduced by the event.
- Send a message (in the form of an event) to another object, to communicate something to it.
- Emit an output to be presented as a result of the simulation. All the output produced by the objects will be passed to the next stage for generating the final report.

  To facilitate communication between stages, a common format was also defined for the results of the simulation. Called the common result format for avoiding confusion with the common data format mentioned in section 6, it contains the following fields:

  Event type: Possibly using the same classification mentioned before, plus other types defined for describing problems.

  Entity reporting the event: The system or domain that emits the message.

  Event priority: Each reported event must be classified according to its severity. The currently defined levels are:

  0 Informative. Something notable enough to be reported, but not representing a particular problem.

  1 Warning. An event that is somehow uncommon, but not directly representing a problem.

  2 Alert. A possible problem that must be immediately investigated.

  3 Emergency. A clear problem.

  It must be observed that the priority level is assigned by the object reporting the event, so its determination is left to that object's "opinion".

  Date and time: Date and time of the event that triggered the message.

  Event summary: A short description of the event, normally including the event type, affected system or domain, and any other relevant information.

  Descriptive message: A detailed description of the event, which will be included in some formats of the final report.

  The format of the common report format is the following, where the fields appear in the order they were mentioned above:

```
type|entity|priority|date_time|summary|description
```

8.4 Central data repository

During the simulation, objects may need global information they don't have stored internally. The most obvious application is knowing which other objects exist (something like a DNS server), for communicating with them. For this reason, one or more objects may exist that serve as data repositories which other objects may query for information.

9 Results presentation

The last stage in SAINT's execution is the interpretation of the messages produced during the simulation, and its presentation in a human-readable format.

Having the result presentation in a separate module makes it simple to modify the result format without modifying the program. Some possible output formats are:

1. Chronological report.
2. Reports by event severity.
3. Reports by system or domain.
4. Reports by event type.
5. Reports restricted to a specific time interval.
6. Reports formatted in ASCII.
7. Reports automatically stored and/or mailed.
8. Reports formatted in HTML, to be viewed with a WWW browser.

As can be seen, this stage determines both the contents and the form of the presented reports. The idea is to have separate modules for each kind of report, inserting the proper module in the execution chain according to what the user asked for. This means that all the reporting modules receive exactly the same information (the messages produced by the event analysis stage), and present it according to their particular functionality. This stage is SAINT's front-end to the user.

10 Control module

It is necessary to have a central module that controls the rest, so the information flows correctly and the results are appropriately generated.

The task of this main module is pretty straightforward since most of the work is distributed among other modules. Its activities can be summarized as follows:

1. Interpretation of command-line arguments.
2. Reading the configuration file.
3. Based on the information collected in the previous steps, set the appropriate parameters and read any additional information (for example, load additional code modules).
4. Execute, one by one, the programs implementing the four SAINT stages. The flow of information between the stages is established using Unix pipes.

11 Configuration file

Practically all of SAINT's behavior is configurable. This is needed if we expect the tool to be usable in several different work settings, with different needs, policies and configurations.

All the operational parameters are set in the configuration file, called `saint.cf` by default, and located in the same directory as the SAINT executable.

The configuration file is a text file. In fact, it consists of Perl5 code, but most of it are variable assignments and other simple statements, so it is easily readable and modifiable even by someone not experienced in Perl programming.

Some of the information that may be definable through the configuration file includes:

- Information about the systems that will be simulated, such as their types, names and characteristics.
- Support modules to be loaded for supporting different tools, problem-detection capabilities, system types, etc.
- Normal working schedule. This information can be used by the modules to detect anomalous behavior (for example, a secretary establishing a network connection at 3 in the morning). This information can be set in general terms and specifically by domain, system or user.

- System maintenance schedule. At these times, some activities (for example, rebooting a system) may be considered less suspicious than during other times.

- Users authorized to use **root**. Normally, a non-authorized user using **root** is a clear indication of problems.

- Electronic addresses from which users normally access systems.

- Security domain definitions.

- Trust relationships between systems and domains.

- Distrust relationships, so problematic systems and domains can be immediately identified.

- Default module for result presentation.

It must be kept in mind that the format of the configuration file is very free-form. Since most of the SAINT modules will be handled through Perl packages, this file will consist primarily of package variable assignments or procedure calls. Each package will define its own interface with the outside world, through which the configuration file can set that package's parameters.

12 Some thoughts about SAINT

Probably the most important thing to keep in mind when using SAINT, as when using any other security and system administration tool, is that it is not magic. Actually, SAINT is only a data collection and analysis tool, so the following factors must be carefully considered:

- SAINT cannot invent information. All the data SAINT uses comes from other tools and places. Information it is not given cannot be taken from anywhere, although SAINT modules may be intelligent enough to deduct certain pieces. On the other side, the more tools SAINT has to feed it, the more information it will have to make more precise analyses.

- SAINT can be only as trustworthy as the information it receives. If the data it gets for analysis has already been modified by an intruder to hide his tracks, SAINT will never detect him.

Not discussed in this paper are matters about privacy and integrity of the information being fed to SAINT. Work is being done in these aspects too, investigating methods to increase the reliability of the data. Some possibilities include using PGP to encrypt traveling data, using `syslog` to send data to a central system as it is being generated, and others.

Although SAINT was designed with security analysis in mind, its structure allows for it to be used for analyzing any kind of information. It could be used for monitoring the correct functioning of other subsystems such as NFS, NIS and printing systems. Its name can even be easily changed to "System Administration Analysis Integration Tool."

SAINT is not intended as a monolithic do-it-all system. Instead, it tries to define a general infrastructure that allows modules of any level of complexity to be interconnected and to work in coordination. SAINT modules can (and probably will) be very complex, but the general structure of the system tries to be as simple as possible, and allows for modules to be defined to deal with any kind of problems or data.

Although this paper talks about SAINT's ability to detect "problems", SAINT modules can as well be developed for detecting "things that are not normal". Under this scheme, anything that is not explicitly considered normal should be reported. This may be preferred by some security-critical sites.

SAINT allows us to use tools and information that is, in most cases, already available in our Unix systems, which is important due to the already installed base of security tools and logging facilities. The fact that most of it is being written in Perl makes it very portable, and no complex mechanisms are used for collecting the information. Most of it is transferred via the `syslog`(3) facility, by e-mail, or using FTP.

13 Current state of SAINT

The design presented in this paper is currently being refined and implemented as part of a student's bachelor's thesis. When it is in a usable state, SAINT will be released under GNU's General Public License, and will be available under `ftp://ftp.super.unam.mx/pub/security/tools/`.

The author is no longer involved in the development of SAINT. Please forward all requests to `saint@ds5000.super.unam.mx`, or check SAINT's WWW page at `http://www.super.unam.mx/asc/`.

14 The future of SAINT

In the first release of SAINT, most of the functionality described in this paper will be implemented. However, there are many things that can be considered for future work. For example:

- Generation of reports in different languages. Although support for multiple languages will be included from the beginning, the first version will generate reports in Spanish. However, English modules will soon follow.
- Generation of information that can be used with other tools. For example, a SAINT module could suggest, according to the configuration file, the contents for TCP-Wrapper access control files. This would allow not only detecting problems, but preventing them.
- Graphical interface. SATAN (by Dan Farmer and Wietse Venema) set a precedent in using a WWW browser for presenting results with a hypertext interface, making it enormously easier to read and interpret the results. This same kind of interface could be implemented for SAINT as one of the front-end modules.
- Real-time monitoring. In the current design, SAINT is intended to be periodically executed over data collected during a time interval. However, it could be possible to collect and analyze data as it is being generated. In fact, only the data collection and homogenization modules would have to change.
- Distributed processing. Interesting work is being currently made in the field of intelligent agents for security monitoring and intrusion detection [CS95a, CS95b]. It probably wouldn't be too difficult to allow SAINT to distribute processing, so the simulation stage becomes more of a real-time monitoring by an independent agent located in each system, and reporting to a central system where the results are consolidated and reports generated.

References

[Bis95] Matt Bishop. Passwd+. In *Course W16: UNIX Security Tools: Use and Comparison,* presented at Network Security '95 in Washington, D. C., pages 20–33. Network Security Institute, November 1995. Package available at `ftp://ftp.super.unam.mx/pub/security/tools/passwd+.tar.gz`.

[CIA] U. S. Department of Defense Computer Incident Advisory Capability (CIAC). Merlin. Available at `ftp://ftp.super.unam.mx/pub/security/tools/merlin.tar.gz`.

[CS95a] Mark Crosbie and Gene Spafford. Active defense of a computer system using autonomous agents. Technical Report CSD-TR-95-008, COAST Laboratory, Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-1398, February 1995. Available at `http://www.cs.purdue.edu/homes/spaf/tech-reps/9508.ps`.

[CS95b] Mark Crosbie and Gene Spafford. Defending a computer system using autonomous agents. Technical Report CSD-TR-95-022, COAST Laboratory, Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-1398, March 1995. Available at `http://www.cs.purdue.edu/homes/spaf/tech-reps/9522.ps`.

[FS90] Daniel Farmer and Eugene H. Spafford. The COPS security checker system. In *Proceedings of the Summer 1990 Usenix Conference*, pages 165–170. Usenix, June 1990. Available at `http://www.cs.purdue.edu/homes/spaf/tech-reps/993.ps`.

[FV] Dan Farmer and Wietse Venema. SATAN documentation (Security Administrator Tool for Analyzing Networks). Included in the SATAN package distribution, available at `ftp://ftp.super.unam.mx/pub/security/tools/satan.tar.gz`.

[GBD+94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. PVM 3 user's guide and reference manual. Manual ORNL/TM-12187, Engineering Physics and Mathematics Division, Mathematical Sciences Section, Oak Ridge National Laboratory, September 1994. Available at `http://www.netlib.org/pvm3/ug.ps`.

[HA92] Stephen E. Hansen and E. Todd Atkins. Centralized system monitoring with Swatch. In *Proceedings of the 3rd UNIX Security Symposium*, pages 105–117. Usenix, September 1992.

[HA93] Stephen E. Hansen and E. Todd Atkins. Automated system monitoring and notification with swatch. In *Proceedings of the LISA VII Systems Administration Conference*, pages 145–155. Usenix, 1993.

[HA94] N. Haller and R. Atkinson. On Internet authentication. RFC 1704, Network Working Group, October 1994. Available at `ftp://nic.ddn.mil/rfc/rfc1704.txt`.

[Hal94] Neil Haller. The S/KEY one-time password system. In *Proceedings of the ISOC Symposium on Network and Distributed System Security*, February 1994. Article and related programs available at `ftp://ftp.super.unam.mx/pub/security/tools/skey/`.

[HCMM92] Naji Habra, Baudouin Le Charlier, Abdelaziz Mounji, and Isabelle Mathieu. ASAX: Software architecture and rule-based language for universal audit trail analysis. In *Proceedings of ESORICS '92: European Symposium on Research in Computer Security*. Springer-Verlag, November 1992. Available in the ASAX distribution at `ftp://coast.cs.purdue.edu/pub/tools/unix/asax/`.

[Hug] Doug Hughes. TkLogger. Program available at `ftp://coast.cs.purdue.edu/pub/tools/unix/tklogger.tar.Z`.

[KS93] Gene H. Kim and Eugene H. Spafford. The design of a system integrity monitor: Tripwire. Technical Report CSD-TR-93-071, COAST Laboratory, Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-1398, November 1993. Available at `http://www.cs.purdue.edu/homes/spaf/tech-reps/9371.ps`.

[KS94a] Gene H. Kim and Eugene H. Spafford. Experiences with Tripwire: Using integrity checkers for intrusion detection. Technical Report CSD-TR-94-012, COAST Laboratory, Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-1398, February 1994. Available at `http://www.cs.purdue.edu/homes/spaf/tech-reps/9412.ps`.

[KS94b] Gene H. Kim and Eugene H. Spafford. Writing, supporting, and evaluating Tripwire: A publically available security tool. Technical Report CSD-TR-94-019, COAST Laboratory, Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-1398, March 1994. Available at `http://www.cs.purdue.edu/homes/spaf/tech-reps/9419.ps`.

[KS94c] Sandeep Kumar and Eugene H. Spafford. An application of pattern matching in intrusion detection. Technical Report CSD-TR-94-013, COAST Laboratory, Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-1398, June 1994. Available at `http://www.cs.purdue.edu/homes/spaf/tech-reps/9413.ps`.

[KS95] Sandeep Kumar and Eugene H. Spafford. A software architecture to support misuse intrusion detection. Technical Report CSD-TR-95-009, COAST Laboratory, Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-1398, March 1995. Available at `http://www.cs.purdue.edu/homes/spaf/tech-reps/9509.ps`.

[Kum95] Sandeep Kumar. *Classification and Detection of Computer Intrusions*. PhD thesis, Purdue University, August 1995. Available at `ftp://coast.cs.purdue.edu/pub/COAST/papers/kumar-intdet-phddiss.ps.Z`.

[MCZH95] Abdelaziz Mounji, Baudouin Le Charlier, Denis Zampuniéris, and Naji Habra. Distributed audit trail analysis. In *Proceedings of the ISOC 95 Symposium on Network and Distributed System Security*, 1995. Available in the ASAX distribution at `ftp://coast.cs.purdue.edu/pub/tools/unix/asax/`.

[Muf] Alec D.E. Muffet. *Crack Version 4.1: A Sensible Password Checker for Unix*. Manual included in the distribution of Crack, available at `ftp://ftp.super.unam.mx/pub/security/tools/crack.tar.gz`.

[SSH93] Dave Safford, Douglas Lee Schales, and David K. Hess. The TAMU security package: An on-going response to internet intruders in an academic environment. In *Proceedings of the Fourth USENIX UNIX Security Symposium*, pages 91–118. Usenix, October 1993. Program available at `ftp://ftp.super.unam.mx/pub/security/tools/TAMU/`.

[Ven] Wietse Venema. Logdaemon package. Program and documentation available at `ftp://ftp.super.unam.mx/pub/security/tools/logdaemon.tar.gz`.

[Ven92] Wietse Venema. TCP Wrapper: Network monitoring, access control, and booby traps. In *Proceedings of the 3rd UNIX Security Symposium*, pages 85–92. Usenix, September 1992. Program available at `ftp://ftp.super.unam.mx/pub/security/tools/tcp_wrappers.tar.gz`.

[Wal] Larry Wall. Perl5 documentation. Included in the Perl5 package, available at `ftp://ftp.super.unam.mx/pub/languages/perl/`.

[WS92] Larry Wall and Randal L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., 103 Morris Street, Suite A, Sebastopol, CA 95472, first edition, March 1992.