A Building Block Approach to Intrusion Detection

Mark J. Crosbie
Hewlett-Packard Company
Cupertino, CA 95014
mcrosbie@cup.hp.com

Benjamin A. Kuperman
CERIAS, Purdue University
West Lafayette, IN 47907
kuperman@cerias.purdue.edu

September 7, 2001

Abstract

This paper details the design and implementation of a host-based intrusion detection system (Hewlett-Packard's Praesidium IDS/9000) and a specialized kernel data source which supplies customized data to the IDS. Instead of the common attack-signature matching used in most other intrusion detection systems, IDS/9000 performs real-time monitoring of the system looking for misuse actions that are indicative of either attack or system policy violations. These misuse actions are called *building blocks*.

As part of the design and implementation, a new kernel data source was developed specifically to aid in intrusion detection. We describe the desired characteristics of an *Intrusion Detection Data Source* (IDDS) which is provided separately from the normal C2 audit subsystem. This new auditing subsystem provides customized audit records tailored to the needs of the intrusion detection system. Performance measurements are provided, and we also discuss some of the alternative uses of IDS/9000 that were discovered during the testing phase.

1 Introduction

The commonly exploited vulnerabilities in computer systems have changed little over the past few decades [1, 3, 5, 13, 18, 21, 23–25]. These vulnerabilities continue to occur despite the existence of mailing lists where such things are discussed [42], programming guides which explain general rules on how to avoid them [2, 41], or even schemes where they are classified and reported [6, 27, 38]. Buffer overflows; insecure handling of temporary files, lock files, and symbolic links; and other such problems continue to be reported with disturbing regularity even in supposedly "secure" software. Recent trends seem to indicate that these occurrences will not be decreasing in the near future [43]. Unfortunately, many of these flaws are undetected until after a software system has been released, and when the vulnerabilities are discovered, attack tools are often released shortly thereafter.

Traditional intrusion detection systems rely on a technique known as *signature matching* whereby an event or data field is compared against a list of known attacks. If a match is discovered, an alert is generated. This technique depends upon having an up-to-date list of signatures for known attacks, and a process by which new signatures can be added as new attacks are discovered.[1] Sometimes, minor variations of a known attack cannot detected by an IDS depending on the signature being used.

Although the attack mechanisms might be hard to generalize, experience has shown that most of them perform one or more of the following actions when successful:

- modify a system file
- unexpectedly change user privileges of a running process
- modify log files
- create a "setuid" backdoor program
- guess at passwords
- change a symbolic link during the execution of a program

We believe that building an intrusion detection system based on detecting these *building blocks* of attacks will allow us to have a handful of detection templates instead of the hundreds of attack signatures. These building blocks focus on the unauthorized *actions* that we are watching for instead of a laundry list of attacks. This technique allows us to detect unknown or unpublished variants of attacks, as well as new attacks that perform one of the above functions, because we are looking for the effects of a successful attack instead of attempting to determine when an exploitation of a specific vulnerability takes place.

In order to detect these building blocks, we need to have access to detailed audit information on system operation and a correlation engine that allows us to perform complex, stateful analysis of this information.

1.1 Related work

The basic model for an intrusion detection system was introduced by Denning in [9]. Much work has grown around this model, and research in intrusion detection has covered network based intrusions [14, 37], immunology based models [12], state machine models [22] and statistical analysis [19]. The approach we detail in this paper for detecting system misuse leverages some of the work performed by these authors; our core engine performs a state-based analysis of audit data. Analysis of the requirements for audit data content for intrusion detection has been performed by Price [39] and Bishop [3, 4]. We examined their conclusions before designing our data source for intrusion detection.

2 Available data sources for intrusion detection

Reliable detection of intrusions depends on the quality of the data provided for analysis. The old computing adage *garbage in, garbage out* applies to intrusion detection systems – they can only analyze the data made available to them. A host-based IDS has a variety of data sources available for its use. The syntax and semantics of data from each source differs dramatically. An IDS designer must understand

[1] The anti-virus community is built almost entirely upon this model.

the format of a data source, and more importantly the meaning of its contents, before the data can be used for detecting system misuse.

Some data sources that are usually available on UNIX systems are the syslog facility, the user login/logout records (wtmp), network packet traces, and a kernel system call audit trail facility [44]. Different versions and implementations of UNIX may have additional sources of data such as process accounting or quota systems.

2.1 The syslog facility as an IDS data source

The syslog facility provides a mechanism to transmit text messages to a system configured location (terminal, line printer, file, remote host) the selection of which is based on the subsystem label and priority of the particular message. These messages can originate in the system kernel, user processes, or even from a remote machine.

Since the syslog facility is available on almost all UNIX systems, it is frequently the first data source host-based IDS designers use when building their systems. Most of the messages logged by syslog are from system daemons and have a reasonably well defined structure (though subtle cross-platform variances in message formats are a problem). In addition, programmers can use the provided `syslog()` library function within an application to report information to the syslog facility.

However, the programmer is responsible for determining when their software has information to report, what information needs to be logged, and at what priority the message should be handled. Frequently, these messages are generated whenever system daemons start up, change configuration, encounter an error, or some other unusual behavior occurs. Programmers rarely consider the needs of an host-based IDS when deciding how and when to log data. The IDS designer is entirely dependent on the quality of the data logged by the programmers who wrote the system daemons.

Furthermore, the syslog file is written by a user space process (typically named `syslogd`). If this process is terminated, either inadvertently or maliciously, then no further syslog information will be saved. Syslog spoofing is a common activity, especially since early versions of the syslog daemon (syslogd) had buffer overflow vulnerabilities present. A remote attacker could force the syslog daemon to exit abnormally (a denial-of-service attack against the syslog facility), or to execute arbitrary code with privilege. [28–36]

2.2 User login records

A login record is generated and recorded to a file when a user successfully authenticates themselves to a UNIX system. Often, the logout of a user from a system is recorded, as are failures to authenticate. The record of these activities can form a valuable data source for an IDS designer. However, one must be aware of their limitations, which are similar to those exhibited by the syslog facility. The data recorded in these login/logout records may not be complete enough to fully reconstruct the sequence of activity that led to an intrusion.

2.3 Network packet traces

Network packets can be gathered from the locally connected network. Analysis of the packet headers can reveal attacks being directed from remote sites against local systems. Analysis of the packet contents at the protocol level is also possible, and may reveal protocol level attacks. However, inspection of network packets has been shown to have shortcomings [40], especially in cases where the IDS lacks context information about the network traffic. While our architecture will support the addition of network traffic data sources, our first implementation focussed on local host data sources.

2.4 Kernel audit data

A more comprehensive and reliable data source is kernel audit data. Most modern operating systems provide a system call level logging facility (often in order to comply with the U. S. Government's requirements for auditing [44, 45]). Unfortunately, when these audit systems were implemented, interacting with intrusion detection systems was not a concern for the vendors. Price's analysis of the audit subsystems present in a variety of UNIX platforms indicated that there were major shortcomings in almost all kernel audit subsystems which inhibit or prevent effective intrusion detection [39]. Additionally, the performance of an IDS which uses these facilities is significantly impaired because the data must be written to disk before it can be read by other processes.

3 A data source for intrusion detection

The IDS product team at Hewlett-Packard was aware of Price's work and decided to design a data source which was tailored exclusively for intrusion detection. Before laying out the design, the following principles were established:

1. *Configuration flexibility*: The data source must support enable/disable commands which can be issued at any time. When enabled, the data source must produce correct reports for all new processes as well as any existing ones.
2. *Unambiguous*: The contents of the data stream must be parsable by a simple grammar and each record must have a regular structure. The semantics of each record must be unambiguous (see [11] for details on the ambiguities in the Solaris BSM audit trail).
3. *Timeliness*: As our goal is to provide sub-second alerting when intrusions occur, the data source must deliver data to the intrusion detection system in a timely manner. Delays of even a few seconds could substantially impact the utility of an intrusion detection system. A batch processing system (where data is only generated at periodic intervals) would not satisfy our requirements.
4. *Minimal degradation of system throughput*: Data gathering by any data source must impose minimal overhead on the running system. If enabling the data source substantially degrades performance, or otherwise impairs normal user behavior, then it is unlikely that users will enable the system [10, 41].

5. *Atomicity*: A record from a data stream must stand alone and be semantically and syntactically correct. For instance, if state information about a file is stored separately from the file name, both are records are needed to correctly process the data stream. The loss of one record renders the other useless.

3.1 What did we build?

Given the above criteria, the IDS team chose to design a kernel data resident data source which would provide per-system call audit records into user space in a timely manner. Each invocation of a system call is audited and a record of the activity is placed into a circular buffer in the kernel. A user-space process reads the data from the buffer via a device driver interface. A device driver provides a clean interface between the kernel and the IDS. The semantics of device drivers are familiar to most UNIX programmers, following the standard file-based open-read-write-close paradigm.

Audit data is read from a device file `/dev/idds` and audit control commands are written to the kernel by performing `ioctl()` calls on the device file. A library is provided for user space applications to interface with the device driver (e.g. using `idds_open()` and `idds_read()` calls).

The audit system is enabled at system boot time, but audit data is only generated when the device driver is opened by a user-space process. There is no overhead imposed on the system until the IDS opens the audit device for read. Once the device is closed the audit system stops gathering data and flushes all audit records. Records are presented in a binary format to user space, and a library converts them to ASCII for further processing if required. The user-space library also provides an interface to enable and control the operation of the audit subsystem.

3.2 Why choose a kernel-resident data source?

All system state is represented in the kernel as a data structure. A kernel data source can provide data in a timely manner an IDS with a minimum of lookup operations. The number of data copying operations is minimized, and only one expensive kernel to user-space copy is required when data is read.

Time in the kernel can be considered frozen – the kernel can translate a filename into an inode, and gather information from that inode, safe in the knowledge that no other process is changing that inode at that moment in time. The audit system gathers as much data as possible in the kernel to avoid a race-condition between the creation of an audit record and a query made by the IDS for additional information. In the example case of a file deletion, no further information will be available after the system call has completed, so the desired meta-information about the file must be collected from the kernel inode.

A kernel resident audit system is more resilient to subversion by a determined attacker. To attack a running kernel, an attacker must gain sufficient privilege to load modules into the kernel, or to write to kernel memory via the `/dev/kmem` interface. While it is certainly possible to alter a running kernel to disable the audit system, such an attack has little chance of success. A well designed IDS will have generated an alert before the attacker even has a chance to alter kernel memory.

3.3 Characteristics of our intrusion detection data source

Our implementation of this kernel auditing subsystem is known as the *Intrusion Detection Data Source* or IDDS. IDDS is built into the HPUX kernel and it has the following features:

IDDS is a kernel resident subsystem and is enabled when the system boots. The kernel system call dispatch handler is modified to gather audit data for system calls. Extra code is added to each audited system call function in the kernel to gather data required for the IDS.

The per-process overhead of auditing is split between the process being audited and the IDS reading the records, with data delivery to the IDS performed on the context of the read call to the device driver.

Any system call which refers to a file will translate a pathname into an inode, and then the *meta-information* for the file (mode bits, owner uid, owner gid, file type, device, inode number) is read from the inode. The meta-information is crucial to our IDS and is included in every file-related audit record.

Relative pathnames are resolved to absolute pathnames and the current root directory is prepended to the pathname. If a file is accessed via a file descriptor the full pathname for the file is computed and included in the audit record. If the file is accessed via a symbolic link the full pathname for the target of the link is provided in the audit record, along with the meta-information for the link target.

The contents of the audit records allow for the deduction of before-and-after states without having to query the system. For example; any audit record for a system call which affects the filesystem contains the file meta-information about the file from before the call. The state of the file after the system call completes can be determined by examining the parameters to the call, the meta-information, and the return value from the system call.

The IDDS is focused on providing data for intrusion detection, and is not a full-featured audit system. It does not generate an audit log file, nor does it manage audit logs. It is the responsibility of the user-space application to create and manage audit log files if desired.

Any security tool exacts a performance penalty on the system it monitors. The designers of an IDS cannot know a priori whether a system administrator places an emphasis on system security or system performance. To avoid making a choice, IDDS configuration provides the administrator a toggle between security and performance. The device driver uses a circular buffer to store audit records to be read by the user-space IDS. When this buffer is full a decision has to be made when an audit record is generated: should the record be dropped or should the kernel wait until space is available in the buffer? If the audit record is dropped then the kernel can return to user-space context immediately. We term this *non-blocking mode*. Of course, a potential intrusion may be missed by the IDS, but no significant performance impact is observed.

If the kernel waits until space for the audit record is available in the circular buffer then the system call will sleep until the IDS reads enough records from the buffer to free up space. We term this *blocking mode*. No audit data will be lost in this mode, but the audited processes will notice an increase in system call response time if the IDS cannot read audit records as fast as they are generated. By default our IDS configures the kernel IDDS subsystem to use blocking mode, as our performance tests have shown that on average system call performance is minimally impacted.

One of our design guidelines for the IDS is to provide *layered filtering*. Data is reduced as soon as possible after it is generated to avoid the cost of copying unneeded data through the kernel to the IDS.

Determining what data can be filtered is not trivial, so our filtering is built in layers. Only those system calls needed to detect a particular exploit are enabled and the system calls are filtered based on UID. More complex filtering is performed at the analysis layer where more context is available from previous system calls. The audit system allows for filtering on system calls, system call outcome and by UNIX user id (UID). For example, this system can audit all `open()` system calls, only successful `open()` calls, or only unsuccessful `open()` calls by user id 123.

We believe that our IDDS design corrects and improves on past audit system designs in the context of intrusion detection, and addresses the issues raised in Price's thesis.

4 Analysis engine

Our analysis engine is built around the ECS (Event Correlation Services) correlation engine [15]. ECS was designed to provide a real-time analysis of event streams in the telecommunications field, and it has built-in support for correlation, filtering, data reduction, transient event suppression, and stateful analysis.

As mentioned in section 1, our approach is to detect the building blocks of attacks. To this end, we designed a set of *detection templates* in ECS each of which analyzes the system audit data for a specific attack building block. The initial set of detection templates we built included the following:

- Detection of the modification of files/directories
- Detection of the modification of files/directories by someone other than the file's owner
- Detection of changes to log files
- Detection of race condition attacks
- Detection of unexpected privilege escalation
- Detection of repeated failed login attempts
- Detection of repeated failed switch user (su) attempts
- Detection of the creation of SetUID files and programs
- Detection of the creation of world-writable files
- Reporting of user login/logout

Each template was tested against publicly available attacks where possible, and from our internal incident response database to verify that the templates behaved as expected.

The detection templates are stored as byte-code which can be dynamically loaded by the correlation engine on demand. This allows each template to be independently configured and updated. Through the use of dynamic loading, only those templates that are currently being used will be impacting system load.

The overall design of the IDS was based on a set of security requirements which are described in [8]. We use a "push" model for data input – there are data source processors (DSPs) that collect the audit data from the various sources mentioned above and feed them into the IDS.

TPC-C Measurement Test Cases

Test Case	Blocking Mode	Template Selection
1. Lowest overhead with loss of data	OFF	All templates less race condition
2. Lowest overhead with no loss of data	ON	All templates less race condition
3. Highest overhead, data loss likely	OFF	All templates including race condition
4. Highest overhead, no data loss	ON	All templates including race condition

Table 1: Enumeration of the various configurations used during the TPC-C benchmarking detailed in Table 2.

5 Performance Results

A host-based intrusion detection system will always exact some performance penalty on the system being monitored. Some IDS architectures offload the processing and analysis components to a centralized location to reduce the load on the monitored nodes. While this is an attractive option, it contradicted our goal of performing near real-time analysis. Therefore we had to analyze audit data on the local system and minimize the overhead that the analysis imposed on system activity.

We chose to measure performance as transaction throughput using the standard TPC-C benchmark [7]. We selected this benchmark as it represents the expected server workloads run on HP servers (decision support, data warehousing, database queries, Online Transaction Processing). Customers in these environments favor throughput over response time. We felt that a strong score in the TPC-C benchmark would indicate a low overall system performance impact.

The performance tests were performed using TPC-C Benchmark Profiling on a 8-way LandShark N-class with 32 GB Memory running on a 5000 Warehouses Sybase Database with 11.ACE HP-UX. The test cases used are shown in table 1. We tested the performance with and without the race-condition template because it was the most “expensive” of all our templates in terms of both CPU and memory usage. In blocking mode, the kernel will not drop audit records and will place the audited process on a sleep queue until the IDS can process all outstanding audit records. In non-blocking mode, the kernel will discard audit records if the IDS cannot process them at the same rate that they are generated.

Table 2 shows that on Test 1 and Test 2 (Test 3 skipped due to predictable outcome) IDS/9000 does not have any performance impacts on the TPC-C benchmark results; however, Test 4 (the one of the most concern) does show a range of performance impact from -1.0% to -28.0% of throughput.

It is worth noting that the load occurred on one processor, and did not spread across the entire system. The analysis process on the IDS consumed most of the processor resources. The load of the IDS is determined by the complexity and state maintenance of the templates. If a template maintains state information (e.g. the race condition template) then the load is increased.

TPC-C Performance Results

	Initial	Test 1	Test 2	Test 4	Delta
Run Type	Baseline	No IDS	Blocking-RC	Blocking+RC	Test 4 vs Baseline
Tools Run 1	62026.74	61733.27	61978.25	61122.06	-904.68 (-1.0%)
Tools Run 2	62113.17	61614.50	61839.95	61361.77	-751.40 (-1.0%)
NoTools Run 1	62206.33	61946.25	62221.34	44737.96	-17468.37 (-28.0%)
NoTools Run 2	62081.90	61980.62	62128.59	58057.82	-4024.08 (-6.0%)

Table 2: Performance tests of Praesidium IDS/9000 performed using TPC-C Benchmark Profiling on an 8-way LandShark N-class with 32 GB Memory running on a 5000 Warehouses Sybase Database with 11.ACE HP-UX. The values are TPC-C Compute Maximum Qualified with throughput in terms of 'tpmC' (transactions per minute C). *Tools Run* means TPC-C test run with performance data collecting tools invoked during the benchmark run. *NoTools Run* means it is a plain TPC-C benchmark run and there are no tools invoked during the run, this shows the real performance run.

6 Experiences using a building block based IDS

Once we built our host-based intrusion detection system, we tested it on a variety of systems. These ranged from low-end dual processor servers to high-end 32 CPU servers. The tests included interactive workloads from human users, kernel test suites, data warehousing applications and transaction processing benchmark suites. We hoped that running IDS/9000 in such a wide variety of environments gave us some perspective into how the product would behave under varying load conditions. While we did gather the performance data we desired, we also discovered that our tool was had utility in areas other than intrusion detection.

6.1 IDS/9000 as a system diagnostic tool

Some system daemons were misconfigured and performed operations which the default IDS/9000 configuration flagged as potential misuse. One of our templates detected any changes made to files and directories, and would trigger if directories considered “read-only” were modified. Normally directories such as `/opt` and `/sbin` are never modified, but we encountered software which wrote log files these read-only filesystems. Other software packages stored configuration data in `/opt`, which is considered a read-only filesystem in HPUX. Still other software packages insisted on creating lock files under the `/etc` directory, which resulted in frequent alerts. We notified the authors of the software of these problems, and included a filtering mechanism that will allow the user of the IDS to filter out these spurious alerts.

Another observation we made was that some software packages did not set a safe umask[2] mode and

[2] `umask` is an environment variable that controls which permission flags should be turned off in newly created files.

thus created world-writable files. One of our templates would generate frequent alerts when software daemons created lock files with world writable permissions.

6.2 Tracing and monitoring of administrators

While presenting the IDS/9000 product to a system administrator in our lab, he remarked how useful it would be in tracking what changes were made to configuration files by *authorized* administrators. Change control of configuration files is a challenge faced by any organization with hundreds of geographically remote servers and a team of human administrators.

The template which detects changes to files and directories provides data on what executable program and system call caused a file to change. For example, if a file was edited using the vi editor, the template would report that a user had modified a file using the `/bin/vi` editor, with the filename as an argument. If possible, the system call which modified the file is also reported; for example if the file's permission bits were changed, the alert text would report that the `chmod()` system call was used to change a file.

We feel that reporting the *means* by which a file has changed is as important as reporting the change itself. Such a level of detail in the alert is an improvement over existing file-integrity checking techniques (e.g. Tripwire [20], L5 [16], or `tcbck` in AIX [17]). For example, the administrator may decide that if the password file is modified by the `/bin/passwd` program then there is no cause for concern, but if `/etc/passwd` is changed by the `truncate()` system call then there may be cause for concern.

6.3 Providing information useful to system administrators

One added advantage of reporting the binary and system call used to modify a file is that a security administrator has more context to track down the source of false-positives. If an alert is generated every time a lock file is created, the name of the program creating the file, and its full pathname, is reported. The administrator can use this information to either ignore that alert in the configuration, or re-configure the offending program to behave correctly.

In another test case, the `sudo` [26] program was triggering our unexpected privilege change template. While we did not design a template to monitor sudo, it was reporting what user was running which command. We felt that this was useful data.

7 Conclusions

Detecting the actions produced by the building blocks of intrusions is a viable method of detecting computer misuse. By focusing on the behaviors which are the result of a successful system penetration, we were able to build a generalized, misuse based intrusion detection system. The IDS was able to detect known attacks and new attacks either recently published or developed during penetration testing.

The detection of the building block behavior is an improvement over the signature based systems because, unlike signature lists, building blocks do not need constant updates to be able to detect new attacks or variations of old attacks. The undesired behavior is detected regardless of the method by which it occurs.

The development of this detection system would not have been possible without the creation of IDDS. By being able to specify the exact data provided and the modes of operation, we were able to improve upon the audit data sources from past systems. Most importantly, this allowed the intrusion detection system to specify what data was required instead of being forced to rely upon what data was available.

8 Further details

The Praesidium IDS/9000 product is available for HPUX 11.0 and 11i systems at zero cost. It can be downloaded from `http://software.hp.com/`, referencing product code J5083AA. More details are also available at `http://www.hp.com/security/products/ids/`. It is also loaded on Application Release CDs and Operating Environment (OE) bundles.

References

[1] Taimur Aslam, Ivan Krsul, and Eugene H. Spafford. Use of a taxonomy of security faults. Technical Report TR-96-051, COAST Laboratory, Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-1398, September 1996.

[2] *A Lab engineers check list for writing secure UNIX code*. AUSCERT, rev.3c edition, May 1996.

[3] Matt Bishop. A Taxonomy of UNIX System and Network Vulnerabilities. Technical Report CSE-95-10, University of California at Davis, Department of Computer Science, University of California at Davis, Davis, CA 95616-8562, May 1995.

[4] Matt Bishop and David Bailey. A Critical Analysis of Vulnerability Taxonomies. Technical Report CSE-96-11, University of California at Davis, September 1995.

[5] Matt Bishop and Michelle Dilger. Checking for Race Conditions in File Accesses. *Computing Systems*, 9(2):131–152, Spring 1996.

[6] Center for Education and Research in Information Assurance and Security. The Cassandra tool, March 2001. URL `https://cassandra.cerias.purdue.edu/`.

[7] Transaction Processing Performance Council. TPC-C, 2001. URL `http://www.tpc.org/`.

[8] Mark Crosbie and Benjamin Kuperman. Experiences in Specifications: Learning to Live With Ambiguity. In *Proceedings of the 1st Symposium on Requirementes Engineering for Information Security*. CERIAS, Purdue University, March 2001.

[9] Dorothy E. Denning. An intrusion detection model. In *IEEE Transactions on Software Engineering, page 222*, 1987.

[10] Dorothy Elizabeth Robling Denning. *Cryptography and Data Security*. Addison-Wesley, Reading, Massachusetts, 1982.

[11] Chapman Flack and Mikhail J. Atallah. Better logging through formality. In *Proceedings of the Third International Workshop on Recent Advances in Intrusion Detection (RAID)*, Toulouse, France, October 2000.

[12] Stephanie Forrest, Steven A. Hofmeyr, and Anil Somayaji. Computer Immunology. *Communications of the ACM*, 40(10):88–96, October 1997.

[13] Simson Garfinkel and Gene Spafford. *Practical UNIX Security*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, second edition, 1991. ISBN 0-937175-72-2.

[14] R. Heady, G. F. Luger, A. B. Maccabe, and M. Servilla. The Architecture of a Network-level Intrusion Detection System. Technical Report CS90-20, University of New Mexico, University of New Mexico, Albuquerque, NM 87131, USA, 1990.

[15] Hewlett-Packard Company. HP OpenView Event Correlation Services. URL `http://www.openview.hp.com/products/ecs/index.asp`.

[16] *Hobbit*. L5. Posted to BugTraq, 15 September 1994. URL `ftp://ftp.cerias.purdue.edu/pub/tools/unix/sysutils/l5/`. File integrity checker.

[17] IBM. *tcbck - trusted computing base checker*, 1 May 1995. Commands Reference, Volume 5.

[18] Ivan Krsul. *Software Vulnerability Analysis*. PhD thesis, Department of Computer Sciences, Purdue University, 1998. URL `https://www.cerias.purdue.edu/techreports-ssl/public/98-09.pdf`.

[19] Harold S. Javitz and Alfonso Valdes. The sri ides statistical anomaly detector. In *IEEE Symposium on Research in Security and Privacy*, 1991.

[20] Gene H. Kim and Eugene H. Spafford. The design and implementation of tripwire: A file system integrity checker. In Jacques Stern, editor, *2nd ACM Conference on Computer and Communications Sec urity*, pages 18–29, COAST, Purdue, November 1994. ACM Press.

[21] Ivan Krsul, Eugene Spafford, and Mahesh Tripunitara. Computer vulnerability analysis. Technical Report COAST TR98-07, COAST Laboratory, Purdue University, West Lafayette, IN, May 1998. URL `ftp://coast.cs.purdue.edu/pub/COAST/papers/ivan-krsul/krsul9807.ps`.

[22] Sandeep Kumar. *A Pattern Matching Approach to Misuse Intrusion Detection*. PhD thesis, Purdue University, Department of Computer Sciences, 1995.

[23] Sandeep Kumar and Eugene Spafford. A Taxonomy of Common Computer Security Vulnerabilities based on their Method of Detection. (unpublished), June 1994.

[24] Carl E. Landwehr, Alan R. Bull, John P. McDermott, and William S. Choi. A taxonomy of computer program security flaws. *ACM Computing Surveys*, 26(3):211–254, September 1994.

[25] Richard R. Linde. Operating System Penetration. In *National Computer Conference*, pages 361–368, 1975.

[26] Todd Miller. Sudo (superuser do). URL `http://www.courtesan.com/sudo/`.

[27] MITRE. Common Vulnerabilities and Exposures (CVE), March 2001. URL `http://cve.mitre.org/`. A list of standardized names for computer vulnerabilities and exposures.

[28] MITRE. CVE entry CAN-1999-0171. Webpage at `http://cve.mitre.org/`, 22 January 2001. Denial of service in syslog by sending it a large number of superfluous messages.

[29] MITRE. CVE entry CAN-1999-0223. Webpage at `http://cve.mitre.org/`, 22 January 2001. Solaris syslogd crashes when receiving a message from a host that doesn't have an inverse DNS entry.

[30] MITRE. CVE entry CVE-1999-0063. Webpage at `http://cve.mitre.org/`, 22 January 2001. Cisco IOS 12.0 and other versions can be crashed by malicious UDP packets to the syslog port.

[31] MITRE. CVE entry CVE-1999-0099. Webpage at http://cve.mitre.org/, 22 January 2001. Buffer overflow in syslog utility allows local or remote attackers to gain root privileges.

[32] MITRE. CVE entry CVE-1999-0381. Webpage at http://cve.mitre.org/, 22 January 2001. super 3.11.6 and other versions have a buffer overflow in the syslog utility which allows a local user to gain root access.

[33] MITRE. CVE entry CVE-1999-0566. Webpage at http://cve.mitre.org/, 22 January 2001. An attacker can write to syslog files from any location, causing a denial of service by filling up the logs, and hiding activities.

[34] MITRE. CVE entry CVE-1999-0583. Webpage at http://cve.mitre.org/, 22 January 2001. vchkpw program in vpopmail before version 4.8 does not properly cleanse an untrusted format string used in a call to syslog, which allows remote attackers to cause a denial of service via a USER or PASS command that contains arbitrary formatting directives.

[35] MITRE. CVE entry CVE-1999-0831. Webpage at http://cve.mitre.org/, 22 January 2001. Denial of service in Linux syslogd via a large number of connections.

[36] MITRE. CVE entry CVE-1999-0867. Webpage at http://cve.mitre.org/, 22 January 2001. Kernel logging daemon (klogd) in Linux does not properly cleanse user-injected format strings, which allows local users to gain root privileges by triggering malformed kernel messages.

[37] Biswanath Mukherjee, Todd L. Heberlein, and Karl N. Levitt. Network intrusion detection. *IEEE Network*, 8(3):26–41, May/June 1994.

[38] National Institute of Statndards and Technology. ICAT Metabase, March 2001. URL http://icat.nist.gov/. A searchable index of reported computer vulnerabilities.

[39] Katherine M. Price. Host-Based Misuse Detection and Conventional Operating Systems' Audit Data Collection. Master's thesis, Purdue University, December 1997.

[40] Thomas H. Ptacek and Timothy N. Newsham. Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. Technical report, Secure Networks, Inc., January 1998.

[41] Michael D. Schroeder and Jerome H. Saltzer. A hardware architecture for implementing protection rings. *Communications of the ACM*, 15(3):157–170, March 1972. ISSN 0001-0782.

[42] SecurityFocus, 2001. URL http://www.securityfocus.com/. An online vulnerability database and official Bugtraq archive.

[43] SecurityFocus. Bugtraq vulnerability database statistics. Webpage at http://www.securityfocus.com/vdb/stats.html, 2001.

[44] U.S. Department of Defense. Trusted Computer Systems Evaluation Criteria. Technical Report CSC-STD-001-83, DoD Computer Security Center, Fort Meade, MD, August 1983.

[45] U.S. Department of Defense. A Guide to Understanding Audit In Trusted Systems. Technical Report NCSC-TG-001, Version 2, National Computer Security Center, Fort George G. Meade, Maryland 20755-6000, 1 June 1988. URL http://www.radium.ncsc.mil/tpep/library/rainbow/NCSC-TG-001-2.ps.