Host-Based Intrusion Detection

Giovanni Vigna, *Reliable Software Group*
Christopher Kruegel, *Technical University Vienna*

INTRODUCTION

Intrusion detection (Crothers, 2002; Schultz, Endorf, & Mellander, 2003) is the process of identifying and responding to suspicious activities targeted at computing and communication resources. An intrusion detection system (IDS) monitors and collects data from a target system that should be protected, processes and correlates the gathered information, and initiates responses when evidence of an intrusion is detected. Depending on their source of input, IDSs can be classified in to network-based systems and host-based systems.

Network-based intrusion detection systems (NIDSs) collect input data by monitoring network traffic (e.g., packets captured by network interfaces in promiscuous mode). Host-based intrusion detection systems (HIDSs), on the other hand, rely on events collected by the hosts they monitor.

HIDSs can be classified based on the type of audit data they analyze or based on the techniques used to analyze their input. We chose a characterization based on the type of audit data and, in the following, present the two most common classes: operating system–level intrusion detection systems and application-level intrusion detection systems. For each class, we describe how audit data is gathered and what type of techniques are used for its analysis.

OPERATING SYSTEM–LEVEL INTRUSION DETECTION

Host-based IDSs in this class use information provided by the operating system (OS) to identify attacks. This information can be of different granularity and level of abstraction. However, it usually relates to low-level system operations such as system calls, file system modifications, and user logons. Because these operations represent a low-level event stream, they usually contain reliable information and are difficult to tamper with, unless the system is compromised at the kernel level.

In the following, some OS-level auditing data-gathering mechanisms are presented. Then, different analysis techniques that use this type of information are described.

Audit Data Gathering

Auditing is a mechanism to collect information regarding the activity of users and applications. To be useful, audit mechanisms have to be both tamperresistant and nonbypassable. The OS is usually regarded as a trusted entity because it controls access to resources, such as memory and files. Therefore, most existing audit mechanisms are implemented within the OS.

OS audit data is not designed specifically for intrusion detection. Therefore, in many cases, the audit records produced by OS-level auditing facilities contain irrelevant information and sometimes also lack useful information. As a result, IDSs often have to access the OS directly to gather required data.

In past years, researchers have attempted to identify what type of information should be provided to an IDS to be able to detect intrusions effectively. For example, Lunt (1993) suggested the use of IDS-specific audit trails. Daniels and Spafford extended this initial idea and identified the audit data that OSs need to provide to support the detection of attacks against the transmission control protocol/Internet protocol (TCP/IP) stack (1999).

The availability of OS-level auditing mechanisms depends on the operating system considered. For example, Sun's operating systems (first SunOS and later Solaris) provide auditing information through the basic security module (BSM). The BSM is a kernel extension that allows one to log events at the system call level. Different auditing levels can be specified, and, in addition to system calls, security-relevant higher-level events can be generated as well (e.g., login events). Unfortunately, auditing can be disabled by the `root` user, making this particular facility vulnerable to abuse by an attacker who gains administrative privileges on the monitored host.

BSM produces audit records that are stored in audit files in a binary format (to be more space efficient). The contents of such an audit file can be printed in human-readable format using the `praudit` tool. Figure 1 shows an example of records contained in a BSM audit file, as printed by the `praudit` tool. In this example, the records represent the execution of commands performed by invoking the `execve` system call.

```
Thu Aug 10 22:01:29 2004 -> UID:root EUID:root RUID:root - From machine:log1
execve() + /usr/bin/sparcv7/ps + cmdline:ps,-ef + success
Thu Aug 10 22:01:50 2004 -> UID:root EUID:root RUID:root - From machine:log1
execve() + /usr/bin/tail + cmdline:tail,/etc/system + success
Thu Aug 10 22:11:18 2004 -> UID:root EUID:root RUID:root - From machine:log1
execve() + /usr/bin/pwd + cmdline:pwd + success
Thu Aug 10 22:11:20 2004 -> UID:root EUID:root RUID:root - From machine:log1
execve() + /usr/bin/ls + cmdline:ls,-l + success
Thu Aug 10 22:11:33 2004 -> UID:root EUID:root RUID:root - From machine:log1
execve() + /usr/bin/ls + cmdline:ls,-l + success
```

Figure 1: BSM audit records.

Similar information is provided by other auditing facilities for different OSs. For example, for the Linux OS, SNARE, and LIDS provide kernel-level auditing information.

SNARE (system intrusion analysis and reporting environment) wraps system calls in routines that gather and log information about processes that execute security-relevant system calls. It also supports simple pattern-matching operations on the audit records produced, which can be used as a rudimentary form of intrusion detection. A graphical tool that allows for the filtering of the collected information is provided as well.

LIDS (linux intrusion detection system), despite its name, is not an intrusion detection system per se. Instead, it provides, in addition to its auditing capabilities, an access control layer that complements the standard UNIX access control mechanisms. This access control layer allows one to specify access control for files, processes, and devices. In particular, LIDS does not grant complete control to the `root` user. Therefore, it is possible to guarantee the protection of critical system parts (e.g., the kernel) even when the `root` account has been compromised. Access control is managed with the help of *capabilities*. Examples of such capabilities include `CAP_LINUX_IMMUTABLE`, which protects files or complete file systems from being overwritten when marked as "immutable," and `CAP_NET_ADMIN`, which prevents tampering with the network configuration (e.g., prevents route table entries from being changed, and prevents firewall entries from being tampered with). Other capabilities are provided to control the insertion and removal of kernel modules, raw disk/device I/O, and a range of other system administration functions.

Another place where audit and security information is stored are operating system log files. For example, almost all UNIX systems offer the *syslog* logging facility. The syslog facility is accessible through an API that sends a log message to `syslogd`, the logging daemon. Each log entry is composed of the identity of the logging process (usually the program name), the entry's level (i.e., the importance of the message), its facility (i.e., the source of the message), and the actual textual message. Q1

Unfortunately, the syslog system has a number of shortcomings. For example, it logs textual messages with arbitrary formats, and, as a result, automated analysis of syslog output is very difficult. Also, the syslog facility encourages the use of multiple log files as a method for classifying events. Therefore, classifications are arbitrary and static, and related events are often sent to different log files. As a result, important context information might be lost. Finally, this facility provides limited notification and response mechanisms (e.g., sending mail to operators or administrators). Other event-logging implementations (e.g., syslog-ng, SDSC-syslog) exist that have addressed some of the limitations. Usually, these implementations support the `syslog()` function for backward compatibility but feed the syslog-generated messages into a more flexible logging system.

Microsoft Windows also provides an auditing system that can be leveraged to perform host-based intrusion detection. The auditing facility produces three event logs, namely the *system log*, the *security log*, and the *application log*.

1. The system log (`SYSEVENT.EVT`) contains events pertaining to Windows services and drivers. It tracks events during system startup, as well as hardware and controller failures (e.g., services that hang upon starting). In a networked setting, there will often be "browser" events in this log, because the machines on the network vote on which one will maintain the browser list.
2. The security log (`SECEVENT.EVT`) tracks security-related events such as logons, logoffs, changes to access rights, and system startup and shutdown, as well as events related to resource usage, such as creating, opening, or deleting files. Note that by default the security log is turned off and has to be explicitly enabled by the administrator.
3. The application log (`APPEVENT.EVT`) is used for events generated by applications. For example, a database program might record in this log a file access error or a problem with the application configuration. The events to be recorded are determined by the developer. This log can grow quite large in size when certain applications such as MS SQL Server or MS Exchange Server are running.

All three logs can be viewed using the native Windows Event Viewer and accessed via the Windows32 API. In addition, there are a number of third-party applications available to examine event logs or to collect log events from multiple Windows machines. The operating system offers built-in mechanisms to search and filter events using several different criteria (e.g., time, source, or category).

Misuse-Based Approaches

Misuse detection systems contain a number of attack descriptions (or *signatures*) that are matched against the stream of audit data and compared to look for evidence that one of the modeled attack is occurring (Ilgun, Kemmerer, & Porras, 1995; Lindqvist & Porras, 1999).

Misuse detection systems usually provide an *attack language* that is used to describe the attacks that have to be detected. These languages provide mechanisms and abstractions for identifying the manifestation of an attack. Well-known examples of detection languages for host-based intrusion detection systems are P-Best (Lindqvist & Porras, 1999), which is the rule-based component of SRI's EMERALD, UCSB's STATL (Eckmann, Vigna, & Kemmerer, 2000), which is used by the STAT Toolset, and RUSSEL (Mounji, 1997), which is the language used by ASAX (Habra, Le Charlier, Mounji, & Mathieu, 1992).

All these languages provide a number of basic mechanisms to describe sequences of events and maintain some sort of intermediate state between different event matchings. In the following, we present in more detail the STATL language as an example of a language that is able to model complex attacks.

The STATL language provides constructs to represent an attack as a composition of *states* and *transitions*. States are used to characterize different snapshots of a system during the evolution of an attack. Obviously, it is not feasible to represent the complete state of a system (e.g., all volatile memory, file system); therefore, a STATL scenario uses variables to record just those parts of the system state needed to define an attack signature (e.g., the value of a counter or the ownership of a file). Each transition has an associated *action*, which is a specification of the events that cause the transition to be taken (i.e., the scenario moves into a new state). Examples of actions are the opening of a file or the execution of an application. The space of events that are relevant for an action is constrained by a *transition assertion*, which is a filter condition on events that may match the action. For example, an assertion may require that a file be opened with a specific mode (e.g., readonly) or that an application being executed is part of a predefined set of security-critical applications.

It is possible that several occurrences of the same attack are active at the same time. Thus, a STATL attack scenario has an operational semantics in terms of a set of *instances* of the same scenario *prototype*. The scenario prototype represents the scenario's definition and global environment, and the scenario instances represent individual attacks currently in progress.

The evolution of the set of instances of a scenario is determined by the type of transitions in the scenario definition. A transition can be *consuming*, *nonconsuming*, or *unwinding*. A nonconsuming transition is used to represent a step of an occurring attack that does not prevent further occurrences of attacks from spawning from the transition's source state. Therefore, when a nonconsuming transition is taken, the source state remains valid and the destination state becomes valid, too. For example, if an attack has two steps that are the creation of a link named
Q2 "`-i`" to a SUID shell script and the execution of the script through the created link, then the second step does not invalidate the previous state. That is, another execution of the script through the same link may occur. Semantically, the firing of a nonconsuming transition causes the creation of a *new* scenario instance. The original instance is still in the original state, whereas the new instance is in the state that is the destination state of the fired transition. In contrast, the firing of a consuming transition makes the source state of a particular attack occurrence invalid. Semantically, the firing of a consuming transition does not generate a new scenario instance; it simply changes the state of the original one. Unwinding transitions represent a form of "rollback" and are used to describe events and conditions that invalidate the progress of one or more scenario instances and require the return to an earlier state. For example, the deletion of a file may invalidate a condition needed for an attack to complete, and, therefore, the corresponding scenario instances may be brought back to a previous state, such as before the file was created.

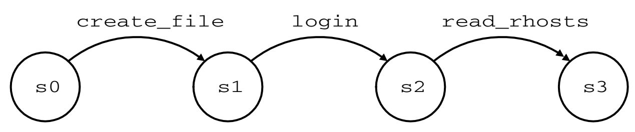

Figure 2: *ftp-write* state transition diagram.

The STATL language is used to describe scenarios in a host-based intrusion detection system called USTAT. USTAT uses Sun Microsystems' BSM as a source of audit Q3
data. For example, consider an *ftp-write* attack, where an attacker uses the *ftp* service to create a bogus `.rhosts` file in a world-writable *ftp* daemon home directory. Using the created file, the attacker is then able to open a remote session using the *rlogin* service without being required to supply a password. A generalization of this attack is that an attacker creates a bogus `.rhosts` file in any other user's home directory and then uses it to be allowed to login without providing a password. This generalization of the *ftp-write* attack is depicted schematically in the state transition diagram in Figure 2. The different types of arrows are used to denote different types of transitions: a solid arc with a single arrowhead denotes a nonconsuming transition and a solid arc with a double arrowhead denotes a consuming transition. A text-based STATL specification of the attack is given in Figure 3.

The sequence of events detected by this scenario is that a file is created (or written to) by a nonroot user who does not own the directory containing the file, and then the login program runs and reads the suspicious file. `WRITE`, `EXECUTE`, and `READ` are abstractions of BSM-specific event types. The predicate `match_name`() and procedure `userid2name`() are helper functions that perform matching and user ID translation.

Anomaly-Based Approaches

Anomaly-based techniques (Barbera & Jajodia, 2002; Denning, 1987; Ghosh, Wanken, & Charron, 1998; Javitz & Valdes, 1991) follow an approach that is complementary to misuse detection. In their case, detection is based on models of normal behavior of users and applications, which are called *profiles*. Any deviations from established profiles are interpreted as attacks.

The main advantage of anomaly-based techniques is that they are able to identify previously unknown attacks. By defining an expected, normal behavior, any abnormal

```
use bsm, unix;
scenario ftp_write
{
  int user;
  int pid;
  int inode;

  initial state s0 { }

  transition create_file (s0 -> s1)
    nonconsuming
  {
    [WRITE w] : (w.euid != 0) && (w.owner != w.ruid)
    { inode = w.inode; }
  }

  state s1 { }

  transition login (s1 -> s2)
    nonconsuming
  {
    [EXECUTE e] : match_name(e.objname, "login")
    {
      user = e.ruid;
      pid = e.pid;
    }
  }

  state s2 { }

  transition read_rhosts (s2 -> s3)
    consuming
  {
    [READ r] : (r.pid == pid) && (r.inode == inode)
  }

  state s3
  {
    {
       string username;
       userid2name(user, username);
       log("remote user %s gained local access", username);
    }
  }
}
```

Figure 3: Example attack scenario specified in STATL.

action can be detected, whether it is part of the threat model or not. The advantage of being able to detect previously unknown attacks is usually paid for with a high number of false positives (i.e., legitimate events are classified as malicious).

In the past, a number of host-based anomaly detection approaches have been proposed that build profiles using system calls (Forrest, Hofmeyr, Somayaji, & Longstaff, 1996; Wagner & Dean, 2001). More specifically, these systems rely on models of legitimate system call sequences issued by the application during normal operation. During the detection process, every monitored sequence that is not compliant with previously established profiles is considered part of an attack.

One technique to create the necessary models of legitimate system call sequences is the analysis of system call invocations during normal program execution. That is, the anomaly detection system “learns” normal behavior by monitoring system call traces of legitimate application runs. These systems, which do not rely on any a priori assumptions about applications, are thus called learning-based anomaly detectors.

An example of a learning-based anomaly detector that is based on system call analysis is described by Forrest et al. (1996). During the learning phase (also called training phase), this system collects all distinct system call sequences of a certain specified length. During detection, all actual system call sequences are compared to the set of legitimate ones and an alarm is raised if no match is found. The approach has been further refined by Lee, Stolfo, and Chan (1997) and Warrender, Forrest, and Pearlmutter (1999), where the authors study similar models and compare their effectiveness to the original technique. In Sekar, Bendre, Bollineni, and Dhurjati (2001), a deterministic system call automaton for a program is learned by associating each system call with its corresponding program counter. This model, however, does not take into account context information, which denotes the history of function calls stored on the program stack, and may miss attacks because of this imprecision. An extension that includes the context provided by the program call stack was described by Feng et al. (2003).

Another group of system call–based anomaly detection systems (Kruegel, Mutz, Valeur, & Vigna, 2003; Provos, 2003) focus on the analysis of system call arguments instead of using sequence information. In (Kruegel et al., 2003), the input to the detection process consists of an ordered stream $S = \{s_1, s_2, \ldots\}$ of system call invocations recorded by the OS. Every system call invocation $s \in S$ has a return value r^s and a list of argument values $< a_1^s, \ldots, a_n^s >$. For each system call, a distinct profile is created. This profile captures the notion of a “normal” system call invocation by characterizing “normal” values for one or more of its arguments.

The expected "normal" values for individual arguments are determined with the help of models. A model is a set of procedures used to evaluate a certain feature of a system call argument, such as the length of a string. A model can operate in one of two modes, learning or detection. In learning mode, the model is trained and the notion of "normality" is developed by inspecting examples. Examples are values that are considered part of a regular execution of a program and are either derived directly from a subset of the input set S (learning on the fly) or provided by previous program executions (learning from a training set).

In detection mode, the task of a model is to return the probability of occurrence of a system call argument value based on the model's prior training phase. This value reflects the likelihood that a certain feature value is observed, given the established profile. The assumption is that feature values with a sufficiently low probability (i.e., abnormal values) indicate a potential attack. To evaluate the overall anomaly score of an entire system call, the probability values of all models are aggregated.

An example of a model is the *string length* model. Usually, system call string arguments represent canonical file names that point to an entry in the file system. These arguments are commonly used when files are accessed (`open`, `stat`) or executed (`execve`). Their length rarely exceeds a hundred characters and they mostly consist of human-readable characters.

When malicious input is passed to programs, it is often the case that this input also appears in arguments of system calls. For example, consider a format string vulnerability in the log function of an application. Assume further that a failed open call is logged together with the file name. To exploit this kind of flaw, an attacker has to carefully craft a file name that triggers the format string vulnerability when the application attempts and subsequently fails to open the corresponding file. In this case, the exploit code manifests itself as an argument to the `open` call that contains a string with a length of several hundred bytes.

The goal of the string length model is to approximate the distribution of the lengths of a string argument and detect instances that significantly deviate from the observed normal behavior. To characterize normal string lengths, the mean $\hat{\overline{X}}$ and the variance $\hat{\sigma}^2$ of the real string length distribution are approximated by calculating the sample mean $\overline{X}$ and the sample variance σ^2 for the lengths $l_1, l_2, \ldots, l_n$ of the argument strings processed during the learning phase. Then, in the detection phase, the actual values of system calls parameters are compared to the established profiles to determine if the observed value is within the range of legitimate values.

Another example of a model is the *character distribution* model. This model captures the concept of a "normal" string argument by looking at its character distribution. The approach is based on the observation that strings have a regular structure, are mostly human-readable, and almost always contain only printable characters. A large percentage of characters in such strings is drawn from a small subset of the 256 possible 8-bit values (mainly from letters, numbers, and a few special characters). As in English text, the characters are not uniformly distributed but occur with different frequencies.

This model learns the "normal" character distribution during a training phase. For each observed argument string, its character distribution is stored. The "normal" character distribution is then approximated by calculating the average of all stored character distributions. Then, during detection, a statistical test is used to determine the probability that the character distribution of an argument is an actual sample drawn from its established profile.

In contrast to signature-based approaches, the character distribution model has the advantage that it cannot be evaded by some well-known attempts to hide malicious code inside a string. In fact, signature-based systems often contain rules that raise an alarm when long sequences of 0×90 bytes (the `nop` operation on Intel $\times$86-based architectures) are detected. An intruder may substitute these sequences with instructions that have a similar behavior (e.g., `add rA,rA,0`, which adds 0 to the value in register A and stores the result back to A). By doing this, it is possible to prevent signature-based systems from detecting the attack. Such sequences, nonetheless, cause a distortion of the string's character distribution, and, therefore, the character distribution analysis still yields a high anomaly score.

The models described previously are just examples of how it is possible to create a learning-based anomaly detection system. There are a number of possible variations on this scheme and this field is still the object of active research.

A somewhat different approach is followed by RAD (Apap et al., 2002), a system that uses as input the registry access events on MS Windows hosts. RAD uses an attack-free history of accesses to the Windows registry to build a statistical model of the normal behavior of applications with respect to registry interaction. The model is then used to detect malicious applications that perform anomalous operations on the registry. A major drawback of this approach is that malicious software can damage the operating system without modifying the registry at all. Therefore, this system can only detect a subset of the possible attacks.

In general, evasion is a problem of all intrusion detection systems but it becomes more relevant in the case of anomalous detection techniques. The reason is that it is difficult to define models that prevent an intruder from performing attacks that stay within the limits of what is considered "normal." Such attacks, often called *mimicry attacks*, can be possible because of design problems (Tan & Maxion, 2002; Tan, Killourhy, & Maxion, 2002) or because of the poor quality of the input event stream. For example, user modeling based on command line analysis is well known for being prone to evasion attacks, in which commands and application binaries are renamed (or replaced) by an attacker to create a session that conforms perfectly to the established normal profiles (Maxion & Townsend, 2002; Wang & Stolfo, 2003; Shonlau et al., 2001). In general, if the auditing mechanism relied upon by an intrusion detection system can be bypassed or modified by the attacker, then the design of the system will detect only attackers who are not aware of the existence of the intrusion detection system or who are not careful enough to cover their tracks.

Specification-Based Approaches

Whereas learning-based anomaly detection systems build models by monitoring application traces, specification-based approaches define models a priori without using dynamic program information. In their case, the models are written manually or built by statically analyzing application code.

An early specification-based technique that was based on written specifications for events in distributed systems was presented by Ko, Ruschitzka, and Levitt (1997). It was later been refined by Bernaschi, Gabrielli, and Mancini (2002) and Chari and Cheng (2002), in which the focus was moved to system calls. Another system, called Janus (Goldberg, Wagner, Thomas, & Brewer, 1996), creates a restricted environment (called sandbox) for processes in which all system call invocations are intercepted and verified with respect to a manually written specification. The idea is to limit the potential damage that an attacker can cause after successfully compromising a process. Yet another approach, which is related to system call policies, are software wrappers (Fraser, Badger, & Feldman, 1999; Ko, Fraser, Badger, & Kilpatrick, 2000). Software wrappers define policies based on state machines that operate in kernel space. Whenever a system call is invoked, a number of wrappers are called to check whether the system call itself and its arguments are permitted. Because the wrappers dispose of state, it is possible to base decisions on a series of system calls.

The use of static analysis techniques to determine system call models was introduced by Wagner and Dean (2001) and Wagner and Soto (2002). In this approach, a call-graph model based on automata is used to characterize the expected system call sequences. The initial approach was extended by Giffin, Jha, and Miller (2004), who present an alternative, more efficient model to represent “legal” sequence call sequences. The price that has to be paid is the need to insert additional “checkpoint” system calls into the program, which is realized via binary rewriting. Another system that uses static analysis to extract a model of acceptable system calls is presented by Feng et al. (2004). In this case, the call stack information is used to better model the context in which normal system call are executed.

Techniques that use specifications are usually not as prone to reporting false alarms as their anomaly-based cousins. That is, given a complete and accurate policy, these systems perform very well. Unfortunately, the task to produce such a policy for realistic applications and scenarios is not trivial.

APPLICATION-LEVEL INTRUSION DETECTION

An important source of audit data for host-based intrusion detection systems is the information provided directly by applications. In the traditional sense, this data is read from log files or other similar sources. However, other techniques were developed where the integration between the IDS and the monitored application is tighter. Application audit data is rich, reliable, and very focused. Therefore, it is easy to determine which program is responsible for a particular event. On the downside, application data is also very specific and different applications have to be dealt with on an individual basis by the HIDS.

In the following section, we present application-level audit data-gathering techniques. Then, different analysis techniques that are based on this type of information are described.

Audit Data Gathering

Most operating systems use centralized log files to provide a central repository for both operating system and application audit data. Besides the operating system log files (discussed in the previous section), audit information is also found in application-specific log files. Of particular interest are error logs, because malicious activity often causes side effects that are detected by an application’s error-checking routines.

Application log files have the advantage that they can contain very detailed information. However, their format differs significantly among programs and intrusion detection systems need to be adapted to each individual application. Another disadvantage is the fact that, by the time the information is written to the log, the application has completed the operation in question. Thus, the IDS cannot act preemptively. In addition, the information available is often limited to a summary of a complete transaction. Consider, for example, a Web request that is logged in the common logfile format (CLF) as follows (taken from Almgren & Lindqvist, 2001):

```
10.0.1.2 - - [02/Jun/1999:13:41:37 -0700]
 "GET /a.html HTTP/1.0" 404 194
```

This log entry describes a request from the host with IP address 10.0.1.2 that asked for the document `a.html`, which, at that time, did not exist. The server sent back a response containing 194 bytes. The log entry might not contain all the information an IDS needs for its analysis. Were the headers too long or otherwise malformed? How long did it take to process the request? How did the server parse the request? What local file did the request get translated into? In some applications, logging can be customized and can contain much more information. Nevertheless, in most systems not all internal information that is needed to understand the interpretation of an operation is available for logging. Furthermore, by enabling all log facilities, the risk of running out of storage space for the logs or incurring performance degradation is increased.

To remove some of the limitations of log files with regard to audit data collection, application-integrated systems have been proposed (Almgren & Lindqvist, 2001). In such systems, the IDS monitors the inner workings of the application and analyzes the data at the same time as the application interprets it. This offers an opportunity to detect (and possibly stop) malicious operations before their execution. By tightly integrating the IDS with the application, more information can be accessed (e.g., local information that is never written to a log file). Furthermore, one can expect an application-integrated monitor to generate fewer false alarms, because it does not have to guess the interpretation and outcomes of malicious operations. For example, a module in a Web server can see the entire HTTP request, including headers. The module can determine to which file within the local file system the request was mapped, and it can also determine if this Q4 file will be handled as a CGI program (which is not visible in either the network traffic or the log files), even without parsing the configuration file of the Web server.

To successfully integrate an IDS into an application, the application must provide a suitable interface. To this end, some applications provide APIs or appropriate hooks for call-back routines. In case the application is open-source, another possibility is to extend the application with suitable IDS functionality.

Taking the idea of application-integrated audit data collection even further, honeypots were introduced. Spitzner (2004) defines a honeypot as "an information system resource whose value lies in unauthorized or illicit use of that resource." In general, a honeypot is a host or network intentionally configured with known vulnerabilities that are deliberately exposed to a public segment of the network so as to invite an intrusion attempt. Honeypots are useful in studying the behavior of attackers and they are a way to delay and distract intruders away from more valuable targets.

Honeypots are traditionally classified as low-interaction or high-interaction honeypots, which specifies the level of activity an attacker is allowed to perform. Low-interaction honeypots have limited interaction, and they normally work by emulating services and operating systems. For example, an emulated file transfer protocol (FTP) service listening on port 21 may just emulate a FTP login, or it may support a few additional FTP commands. The advantage of low-interaction honeypots is their simplicity. These honeypots tend to be easier to deploy and maintain, because the attacker never has access to a fully functional operating system to attack or harm others. The main disadvantage of low-interaction honeypots is that they log only limited information and are designed to capture known activity. Also, it is easier for an attacker to detect a low-interaction honeypot. No matter how good the emulation is, a skilled attacker can eventually detect their presence. A well-known open-source low-interaction honeypot is honeyd (Provos, 2004).

High-interaction honeypots are different from low-interaction honeypots in that they attempt to fully emulate the functioning of real OSs and applications. High-interaction honeypots offer two advantages over low-interaction honeypots. First, they can capture extensive amounts of information and so make it possible to learn the full extent of the intruder's behavior, be it the installation of new rootkits or the establishment of internet relay chat (IRC) sessions. Second, because high-interaction honeypots make no assumptions about how an attacker will behave, they provide an open environment that captures all activity and so make it possible to trap and learn from unanticipated behavior. On the downside, high-interaction honeypots can be more complex to deploy and maintain.

Misuse-Based Approaches

One of the simplest signature-based systems that monitors application audit data is Swatch. Swatch, the *Simple Watch* daemon, is a program for UNIX system logging and was written to monitor messages as they are written to a log file via the UNIX `syslog()` utility. The idea is to keep system administrators from being overwhelmed by large quantities of log data. The tool monitors log files, filters out unwanted data, and takes one or more user-specified actions based upon patterns in the log. Swatch can monitor information as it is being appended to the log file and alert system administrators immediately to serious system problems as they occur. The patterns are specified as regular expressions.

A very similar system is LogSentry, which extends the monitoring to other system log files such as the ones produced by Psionic's PortSentry and HostSentry, system daemons, Wietse Venema's TCP Wrapper and Log Daemon packages, and the Firewall Toolkit by Trusted Information Systems (TIS).

Another tool that also works by monitoring system log files but that allows the specification of more complex attack scenarios is logSTAT (Vigna, Valeur, & Kemmerer, 2003). Using the same STATL language previously introduced, logSTAT applies the state transition analysis technique to the contents of syslog files.

WebSTAT (Vigna, Robertson, Kher, & Kemmerer, 2003), a tool related to logSTAT, extends the state transition analysis to application-specific log files, in particular, the Web server log files created by Apache. Several attack scenario have been implemented that include a malicious Web crawler scenario, a pattern-matching scenario, a repeated failed-access scenario, and a buffer-overflow detection scenario.

One interesting attack scenario included with WebSTAT is the cookie-stealing scenario. Cookies are a state management mechanism for HTTP (defined in RFC 2965, Kristol & Montulli, 2000) that is often used by Web application developers to implement session tracking. The cookie-stealing scenario detects if a cookie used as a session ID is improperly utilized by multiple users. This is often a manifestation of a malicious user attempting to hijack the session of a legitimate user to gain unauthorized access to protected Web resources.

The scenario begins by recording the issuance or initial use of a session cookie by a remote client by mapping the cookie to an IP address. In addition, an inactivity timer is simultaneously set. Subsequent use of the session cookie by the same client results in a reset of the timer, whereas a cookie expiration or session timeout results in the removal of the mapping for that cookie. If, however, a client uses the valid session cookie of another client, then an attack is assumed to be underway and an alarm is raised.

The cookie-stealing scenario is interesting because it underlines the need for *state* to detect certain classes of attacks. Most intrusion detection systems are *stateless*, meaning that each event is treated independently of others. However, certain attacks only manifest themselves as multiple steps in which each individual step is not intrusive *per se*. In the cookie-stealing scenario, the use of a cookie by each client appears benign. Only by detecting that two different clients share a single cookie can malicious behavior can be identified.

As mentioned in the previous section, systems that operate on application log files have only a limited view of the operations performed by an application. This shortcoming is addressed by application-integrated systems. Almgren and Lindqvist have developed an integrated monitor to detect Web-based attacks against the Apache Web server (2001). The tool is directly attached to the Apache request pipeline, which consists of several stages that a client request runs through. Each stage possesses so-called hooks or callbacks that are used by the monitor to give feedback to the server as to whether it should continue executing the request.

The presented approach makes evasion techniques less effective, because the view of the intrusion detection system and the view of the server application are tightly integrated. On the other hand, a disadvantage of this approach is that by "in-lining" intrusion detection analysis, the performance of the Web server is affected. In addition, the proposed solution is tailored to a specific Web server (in this case, Apache) and cannot be easily ported to different servers.

Anomaly-Based Approaches

An application-based anomaly detection system creates a profile of application behavior based on normal application activity. This activity is usually expressed as operations that an application performs. A profile can be created by observing traces of normal activity or by specifying all operations that an application is allowed to perform. This section deals with learning-based systems that establish a description of normal behavior by monitoring actual program executions. The following section discusses approaches in which profiles are specified a priori, based on policies that determine acceptable behavior.

An example of a system that monitors application behavior to create a profile of normal behavior is DIDAFIT (detecting intrusions in databases through fingerprinting
Q5 transactions; Lee, Low, & Wong, 2002). This system works by fingerprinting access patterns of legitimate database transactions and using them to identify database intru-
Q6 sions (in particular, SQL injection attacks). The work addresses the problem of learning the set of legitimate fingerprints from database trace logs that contain the SQL statements of benign transactions. To this end, the authors developed algorithms that perform useful generalization of the training set. That is, the system summarizes SQL statements into more general fingerprints and is capable of deriving possibly legitimate fingerprints that are missing from the training data. In addition, it can identify possibly malicious (so called "high-risk") SQL statements even in the training set.

Another example is a system that analyzes the interaction of Web clients with Web-based applications (Kruegel & Vigna, 2003). More precisely, this system analyzes client queries that reference server-side programs, and it creates models for a wide-range of different features of these queries. Examples of such features are access patterns of server-side programs or values of individual parameters in their invocation. Similar to the system call–based analysis presented previously, the tool derives automatically the parameter profiles associated with Web applications (e.g., the length and structure of request parameters). In addition, the relationship between queries and the access patterns of applications over time are also monitored.

Changes in access patterns can indicate attacks. When an application is usually accessed infrequently but is suddenly exposed to a burst of invocations, this increase could be the result of an attacker probing for vulnerabilities or the result of an exploit that has to guess parameter values. A single determined attacker can evade detection by executing his actions slowly enough to keep the frequency low. However, most tools used by less skilled intruders execute brute force attacks without stealthiness in mind. Also, when the knowledge of a vulnerability becomes more widespread, many attackers independently attempt to exploit the vulnerability and raise the total frequency to a suspicious level.

Another pattern focuses on the order in which programs are accessed. Web-based applications are often composed of a set of server-side programs, which, together, implement the application functionality. For example, a shopping application may have a login program to authenticate a user, a program to access a catalog, a program to add items to a virtual cart, and a program to perform checkout/payment. The nature of a Web-based application may impose a well-defined ordering over the invocation of its component programs. For example, a user has to first login before being able to perform any other transaction. Unusual order of program accesses can indicate malicious behavior, such as an attacker attempting to bypass a login check and access a privileged program directly.

Specification-Based Approaches

The specification of application behavior is usually done at the system call interface (as described previously). However, there have been also suggestions to formally specify the normal behavior of an application by defining the input and output data that it exchanged with its users.

For example, in Cheung and Levitt (2002), the speci- Q7 fication language VDM is used to create formal specifications that characterize the normal behavior of domain name service (DNS) clients and servers. The aim was to define a security goal of the DNS service, which states that a name server should only use DNS data that is consistent with data from name servers that manage the corresponding domain (i.e., authoritative name servers). Based on these specifications and to enforce the security goal, a DNS wrapper was implemented that examines the incoming and outgoing DNS traffic between name servers and resolvers. To detect messages that violate the security goal, cooperation with the authoritative name servers is required. Whenever a violating message is detected, it is dropped. Using their wrappers, the authors were successful in detecting DNS cache poisoning and spoofing attacks. Of course, the approach can be extended to specify the format and content of messages exchanges by other network service daemons.

Similar to wrappers for system calls, there are application-based techniques that verify arguments of shared library function calls (Balzer & Goldman, 1999). Using mediators, it is possible to prevent library functions from being called, to modify their arguments, and to adjust their return values.

RELATED TECHNIQUES

A family of host-based tools that are considered related to intrusion detection systems are *integrity checkers*. The task of these tools is to detect whether certain monitored files were tampered with. Although these systems are not intrusion detection systems in the classic sense, they are often used to identify the activities of an intruder after a successful compromise. A well-known integrity checker is Tripwire (Kim & Spafford, 1994). Tripwire is a program that monitors key attributes of files that should not change, including binary signature, size, and expected change of size. To do so, file information and file content hashes are stored in a custom database. Similar to Tripwire, RedHat's packet management system `rpm` can also be used for integrity-checking tasks.

The limit of file system integrity checkers is the fact that the analysis can only be reliably performed in an offline fashion. That is, the file system has to be mounted by an operating system that is known to be not compromised. The reason is that integrity checkers rely on operating system routines to access the file system. If the operating system itself is modified on a compromised host, the checkers could be provided with incorrect information that can lead them to conclude that no modifications have occurred. Once the kernel is infected, it is very hard to determine if a system has been compromised without the help of hardware extensions such as the trusted platform module (TPM).

A common occurrence after a successful intrusion is the installation of a *rootkit* (Black Tie Affair, 1989). A rootkit is a collection of "easy-to-use" tools that help an intruder to hide her presence from the system administrator (e.g., log editors), to gather information about the system and its environment (e.g., network sniffers), and to ensure access at a later time (e.g., backdoored servers). Often, these rootkits include modified versions of system-auditing programs. These modified programs do not return any information to the administrator that involves specific files and processes that are used by the intruder.

Rootkits that simply replace or modify system files or intruders that manually change files can be detected by file integrity checkers. Recently, however, kernel-level rootkits have emerged that can bypass file integrity checks by modifying the kernel directly. Such rootkits (e.g., knark or Adore; Stealth, 2003) operate within the kernel by modifying critical data structures such as the system call table. No modification of program binaries to conceal malicious activity takes place anymore.

To detect kernel-level rootkits, a number of detection tools have been developed that could also be considered host-based intrusion detection systems. The most basic techniques used by these systems include searching for modified kernel modules on disk, searching for known strings in existing binaries, or searching for configuration files associated with specific rootkits. The problem is that when a system has been compromised at the kernel level, there is no guarantee that these detection tools will return reliable results. This is also true for signature-based rootkit detection tools such as chkrootkit that rely on operating system services to scan a machine for indications of known rootkits.

To circumvent the problem of a possibly untrusted operating system, rootkit scanners such as kstat, rkscan, and St. Michael follow a different approach. These tools are either implemented as kernel modules with direct access to kernel memory, or they analyze the contents of the kernel memory via `/dev/kmem`. Both techniques allow the programs to monitor the integrity of important kernel data structures without the use of system calls. For example, by comparing the system call addresses in the system call table with known good values (taken from the `/boot/System.map` file), it is possible to identify hijacked system call entries.

Another group of related tools, which have had the most commercial success, are virus scanners. A computer virus is a piece of software designed to make additional copies of itself and spread from location to location, typically without user knowledge or permission. Virus scanners are tools that scan a binary for the occurrence of viruses (code fragments) that are known to perform malicious actions. With the tremendous increase of Internet virus activity, the importance of scanners has increased dramatically. As a result, virus scanners are now virtually ubiquitous, especially on Microsoft Windows platforms.

HOST-BASED IDSs VERSUS NETWORK-BASED IDSs

Host-based IDSs have both advantages and disadvantages when compared with network-based intrusion detection systems. One advantage is that HIDSs can access semantically rich information about the operations performed on a host, whereas NIDSs that analyze network traffic have to reassemble, parse, and interpret the application-level traffic to identify application-level actions. This is even more evident when application-level traffic is encrypted. In this case, a network-based monitor has to be equipped with the key material needed to decrypt the traffic; otherwise, the application-level information is not accessible. Also, the amount of information that HIDSs have to process is usually more limited, because the rate at which events are generated by the OS and applications is smaller than the rate at which network packets are sent over busy links. A third advantage is that HIDSs are less prone to evasion attacks because it is more difficult to desynchronize the view that the intrusion detection system has of the status of a monitored application with respect to the application itself. Finally, a host-based intrusion detection system has a better chance of performing a focused response because the process performing an attack can sometimes be easier identified and terminated.

On the other hand, host-based IDSs suffer from a set of limitations. A major disadvantage of HIDSs with respect to NIDSs is that the compromise of a host may allow an attacker to disable or tamper with the auditing system or even to disable the intrusion detection system altogether. This problem is caused by the all-or-nothing approach to privilege management followed by most operating systems. Once a process obtains administrative privileges, it is able to change any aspect of the system, including the kernel configuration and the code stored in programmable hardware. The other major disadvantage of HIDSs is that the intrusion detection process may affect substantially the performance of the operating system, and, as a consequence, of the applications running on a monitored host. A single network-based IDS can monitor a number of host without affecting their performance. In addition, a NIDS that monitors many hosts may detect attacks that span multiple hosts. The evidence associated with such attacks might be unavailable when monitoring only a single host. The limited view of host-based systems can be compensated for by combining information gathered from different host-based sensors at a central location. When data from different hosts is available, it is possible to correlate host-based data to get a more global view (Kruegel, Valeur, & Vigna, 2004). However, for this correlation process, a dedicated infrastructure is required. One of the first systems that used correlation of data produced by multiple host-based sensors was DIDS (Snapp et al., 1991). Finally, there is a maintenance cost associated with the deployment and maintenance of host-based IDSs, which tends to be higher than the cost associated with NIDSs because of the need to install an IDS on each protected host and of the heterogeneity of most environments (different operating systems, and different versions of the same operating system).

FUTURE TRENDS

In the past, host-based IDSs have not been considered a viable solution to detect intruders because of their performance overhead and their inability to deliver reliable information in the case of a compromise. In addition, the problem of the large number of false positives produced by anomaly detection systems made it impossible to perform effective intrusion response (e.g., killing a process that is misbehaving) without the risk of hurting legitimate users of the system.

Recently, the interest in this class of intrusion detection systems has been boosted by increasingly more efficient detection techniques and the introduction of new hardware-based mechanisms that guarantee the integrity of a system even in the face of the compromise of a privileged account. This trend is also supported by firewall technology, which, in the past few years, has gradually moved from the gateway of a network to each single host (e.g., the popular ZoneAlarm tool for Windows). These new host-based firewalls are actually host-based intrusion detection systems that analyze network data.

The next generation of HIDSs will probably integrate host-based firewall technology, as well as other technologies, such as virus scanners and integrity checkers. The resulting intrusion detection system will monitor several event streams (e.g., network traffic directed to the host and system calls executed by applications) and will be able to correlate evidence that belongs to different domains. These hybrid systems will be able to provide effective detection and very focused response at a reasonable performance cost.

Another trend in host-based intrusion detection is the progressive integration of intrusion detection mechanisms into the kernel of the operating systems. Currently, the OS is extended to gather auditing information but the actual intrusion detection process is performed outside the kernel, in user space. By integrating intrusion detection within the kernel itself, it is possible to perform much more effective response, blocking suspicious operations before they cause any actual harm. This type of systems may be able to foil a wide range of attacks in an effective manner.

There are still many challenges in developing systems that are tightly integrated with the operating system kernel. The critical nature of the kernel leaves very little room for mistakes and the failure of a single kernel component can render the whole system unusable. Therefore, kernel-level solutions have to be implemented following a high-quality development processes to meet the expectation of the user in terms of reliability and performance.

CONCLUSIONS

For the plast decade, network-based intrusion detection systems have clearly dominated host-based systems. The ease of maintenance and the possibility to monitor several targets with a single IDS installation has tipped the scales toward the network-based solution. However, the increasing use of very fast network links and encrypted connections have change the situation. The quality of audit data that is available at the operating system and application levels, the increasing security awareness of end users, and the improved accuracy of host-based techniques have all contributed to a higher acceptance of such detection mechanisms.

This chapter discussed host-based intrusion detection systems and related techniques such as file integrity checkers and virus scanners. The main sources of audit data (operating system and application) were introduced and different approaches to analyze the data were presented. In addition, we analyzed the advantages and limitations of host-based solutions with regard to network-based techniques and outlined possible future developments in the field.

GLOSSARY

Anomaly-Based Intrusion Detection Intrusion detection techniques that rely on models of normal system behavior to identify intrusions.

Audit A procedure used to validate that controls are in place and adequate for their purposes. This includes recording and analyzing activities to detect intrusions or abuses into an information system.

Audit Data Data produced by an audit procedure.

File Integrity Checker A system that verifies that the attributes and contents of files have not been modified by unauthorized subjects.

Honeypot A host or network with known vulnerabilities deliberately exposed to a public network.

Host-Based Intrusion Detection An intrusion detection system that uses audit data produced by the operating system and applications.

Intrusion Detection System A system that tries to identify attempts to hack or break into a computer system or to misuse it.

Intrusion Detection Evasion An attempt to perform an attack such that it is not detected by the intrusion detection system.

Mimicry Attack An evasion attack in which the intruder attempts to make the attack appear like legitimate behavior.

Misuse-Based Intrusion Detection Intrusion detection techniques that rely on specifications (signatures) of attacks to identify intrusions.

Network-Based Intrusion Detection An intrusion detection system that uses audit data extracted from the network.

Operating System Call The services provided by the operating system kernel to application programs and the way in which they are invoked.

Program Context The context at a certain point in the program's execution is the history of function calls stored on the program stack at this time.

Rootkit A collection of tools that allows a hacker to mask the fact that the system is compromised and to collect additional information.

CROSS REFERENCES

See *Computer Viruses and Worms; Intrusion Detection Systems Basics; Network-Based Intrusion Detection Systems; Security Policy Guidelines; The Use of Agent Technology for Intrusion Detection.* Q8

REFERENCES

Almgren, M., & Lindqvist, U. (2001). Application-integrated data collection for security monitoring. In *Recent advances in intrusion detection (RAID)* (LNCS, pp. 22–36). Davis, CA: Springer.

Apap, F., Honig, A., Hershkop, S., Eskin, E., & Stolfo, S. (2002). Detecting malicious software by monitoring anomalous windows registry access. In *Proceedings of the Fifth International Symposium on Recent Advances in Intrusion Detection (RAID)*.

Balzer, R., & Goldman, N. (1999). Mediating connectors: A non-bypassable process wrapping technology. In *19th IEEE International Conference on Distributed Computing Systems*.

Barbera, D., & Jajodia, S. (Ed.). (2002). *Applications of data mining in computer security*. Kluwer. Q9

Bernaschi, M., Gabrielli, E., & Mancini, L. V. (2002). REMUS: A security-enhanced operating system. *ACM Transactions on Information and System Security, 5*(36). Q10

Q11 Black Tie Affair. (1989). Hiding out under UNIX. *Phrack Magazine, 3*(25).

Chari, S., & Cheng, P., (2002). BlueBoX: A policy-driven, host-based intrusion detection system. In *Symposium on Network and Distributed System Security (NDSS)*, San Diego, CA.

Cheung, S., & Levitt, K. (2002, June). A formal specification based approach for protecting the domain name system. In *International Conference on Dependable Systems and Networks*, New York, NY.

Crothers, T. (2002). *Implementing intrusion detection systems: A hands-on guide for securing the network*. Wiley. Q12

Daniels, T., & Spafford, E. (1999). Identification of host audit data to detect attacks on low-level IP vulnerabilities. *Journal of Computer Security, 7*(1), 3–35.

Denning, D. (1987, February). An intrusion detection model. *IEEE Transactions on Software Engineering, 13*(2), 222–232.

Eckmann, S., Vigna, G., & Kemmerer, R. (2000, November). STATL: An attack language for state-based intrusion detection. In *ACM Workshop on Intrusion Detection Systems*, Athens, Greece.

Feng, H., Giffin, J., Huang, Y., Jha, S., Lee, W., & Miller, B. (2004, May). Formalizing sensitivity in static analysis for intrusion detection. In *IEEE Symposium on Security and Privacy*, Oakland, CA.

Feng, H., Kolesnikov, O., Fogla, P., Lee, W., & Gong, W. (2003). Anomaly detection using call stack information. In *IEEE Symposium on Security and Privacy*.

Forrest, S., Hofmeyr, S., Somayaji, A., & Longstaff, T. (1996, May). A sense of self for UNIX processes. In *IEEE Symposium on Security and Privacy* (pp. 120–128), Oakland, CA.

Fraser, T., Badger, L., & Feldman, M. (1999). Hardening COTS software with generic software wrappers. In *IEEE Symposium on Security and Privacy* (pp. 2–16), Oakland, CA.

Ghosh, A., Wanken, J., & Charron, F. (1998, December). Detecting anomalous and unknown intrusions against programs. In *Annual Computer Security Application Conference (ACSAC)* (pp. 259–267), Scottsdale, AZ.

Giffin, J., Jha, S., & Miller, B. (2004, February). Efficient context-sensitive intrusion detection. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, California.

Goldberg, I., Wagner, D., Thomas, R., & Brewer, E. (1996). A secure environment for untrusted helper applications. In *sixth Usenix Security Symposium*, San Jose, CA.

Habra, N., Le Charlier, B., Mounji, A., & Mathieu, I. (1992, November). ASAX: Software architecture and rule-based language for universal audit trail analysis. In *European Symposium on Research in Computer Security (ESORICS)*, Toulouse, France.

Ilgun, K., Kemmerer, R., & Porras, P. (1995). State transition analysis: A rule-based intrusion detection system. *IEEE Transactions on Software Engineering, 21*(3), 181–199.

Javitz, H. S., & Valdes, A. (1991). The SRI IDES statistical anomaly detector. In *IEEE Symposium on Security and Privacy*.

Kim, G., & Spafford, E. (1994). The design and implementation of tripwire: A file system integrity checker. In *second ACM Conference on Computer and Communications Security (CCS)* (pp. 18–29), Fairfax, VA.

Ko, C., Ruschitzka, M., & Levitt, K. (1997). Execution monitoring of security-critical programs in distributed systems: A specification-based approach. In *IEEE Symposium on Security and Privacy* (pp. 175–187), Oakland, CA.

Ko, C., Fraser, T., Badger, L., & Kilpatrick, D. (2000). Detecting and countering system intrusions using software wrappers. In *Usenix Security Symposium*.

Kristol, D., & Montulli, L. (2000). *HTTP state management mechanism* (RFC 2965). Q13

Kruegel, C., Mutz, D., Valeur, F., & Vigna, G. (2003, October). On the detection of anomalous system call arguments. In *Eighth European Symposium on Research in Computer Security (ESORICS)*, (LNCS, pp. 326–343). Gjøvik, Norway, Springer-Verlag. Q14

Kruegel, C., Valeur, F., & Vigna, G. (2004). *Intrusion detection and correlation: Challenges and solutions*. Norwell MA: Springer Verlag.

Kruegel, C., & Vigna, G. (2003, October). Anomaly detection of Web-based attacks. In *ACM Conference on Computer and Communication Security (CCS)* (pp. 251–261). Washington, DC: ACM Press. Q15

Lee, S., Low, W., & Wong, P. (2002). Learning fingerprints for a database intrusion detection system. In *Seventh European Symposium on Research in Computer Security (ESORICS)*.

Lee, W., Stolfo, S., & Chan, P. (1997, July). Learning patterns from unix process execution traces for intrusion detection. In *AAAI Workshop: AI Approaches to Fraud Detection and Risk Management*.

Lindqvist, U., & Porras, P. A. (1999, May). Detecting computer and network misuse with the production-based expert system toolset (P-BEST). In *IEEE Symposium on Security and Privacy* (pp. 146–161). Oakland, California.

Lunt, T. (1993). Detecting intruders in computer systems. In *Sixth Annual Symposium and Technical Displays on Physical and Electronic Security*.

Maxion, R., & Townsend, T. (2002, June). Masquerade detection using truncated command lines. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)* (pp. 219–228). Washington, DC.

Mounji, A. (1997). *Languages and tools for rule-based distributed intrusion detection*. Unpublished doctoral dissertation Facultés Universitaires Notre-Dame de la Paix Namur, Belgium.

Provos, N. (2003). Improving host security with system call policies. In *12th Usenix Security Symposium*. Washington, DC.

Provos, N. (2004). A virtual honeypot framework. In *13th Usenix Security Symposium*.

Schonlau, M., DuMouchel, W., Ju, W., Karr, A., Theus, M., & Vardi, Y. (2001). Computer intrusion: Detecting masquerades. *Statistical Science, 16*(1) 57–74.

Schultz, G., Endorf, C., & Mellander, J. (2003). *Intrusion detection and prevention*. New York: McGraw-Hill Osborne Media. Q16

Sekar, R., Bendre, M., Bollineni, P., & Dhurjati, D. (2001, May). A fast automaton-based method for detecting anomalous program behaviors. In *IEEE Symposium on Security and Privacy*. Oakland, CA.

Snapp, S. R., Brentano, J., Dias, G., Goan, T., Heberlein, T., Ho, C., et al. (1991). DIDS (distributed intrusion detection system)—Motivation, architecture, and an early prototype. In *14th National Computer Security Conference*.

Spitzner, L. (2004). *Honeypots: Tracking hackers*. Bostan, MA: addison-Wesley.

Stealth. (2003, August). Kernel rootkit experiences and the future. *Phrack Magazine, 11*(61).

Tan, K., & Maxion, R. (2002). "Why 6?" Defining the operational limits of stide, an anomaly-based intrusion detector. In *Proceedings of the IEEE Symposium on Security and Privacy* (pp. 188–202). Oakland, CA.

Tan, K. M. C., Killourhy, K. S., & Maxion, R. A. (2002). Undermining an anomaly-based intrusion detection system using common exploits. In *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection* (pp. 54–73).

Vigna, G., Robertson, W., Kher, V., & Kemmerer, R. A. (2003). A stateful intrusion detection system for World Wide Web servers. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC 2003)* (pp. 34–43).

Vigna, G., Valeur, F., & Kemmerer, R. A. (2003). Designing and implementing a family of intrusion detection systems. In *Proceedings of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2003)*.

Wagner, D., & Dean, D. (2001, May). Intrusion detection via static analysis. In *IEEE Symposium on Security and Privacy* (175–187). Oakland, CA: *May 2001*. IEEE Press. Q17

Wagner, D., & Soto, P. (2002). Mimicry attacks on host-based intrusion detection systems. In *Ninth ACM Conference on Computer and Communications Security (CCS)* (pp. 255–264).

Wang, K., & Stolfo, S. (2003). One class training for masquerade detection. In *Proceedings of the ICDM Workshop on Data Mining for Computer Security (DMSEC)*.

Warrender, C., Forrest, S., & Pearlmutter, B. A. (1999). Detecting intrusions using system calls: Alternative data models. In *IEEE Symposium on Security and Privacy* (pp. 133–145).

Author Queries

Q1: Au: define API?

Q2: Au : define SUID

Q3: Au: define USTAT+BSM?

Q4: Au: define CGI?

Q5: Au: check editing?

Q6: Au: define SQL?

Q7: Au: check year?

Q8: Au: Pls provide publisher, pub's location pgs. URL and Retrieved date of References.

Q9: Au: provide location?

Q10: Au : give pages?

Q11: Au: is "Black Tie Affair" part of title? Author, even? Give pages?

Q12: Au: give Wiley's location?

Q13: Au: For all RFSs, provide URL and retrieval date?

Q14: Au: give Springer's location for Refs. 27+28?

Q15: Au: give ACM's location?

Q16: Au: give McGraw-Hill location?

Q17: Au: give IEEE'S location + pages?