The Threat from the Net

As it stands today, the Internet is not secure, so the only option is to understand how attacks occur and how best to protect against them

In the United States in the late 19th century, when desperadoes rampaged through the Wild West, communication, commerce, and even basic trust in civil authority were threatened. Today's electronic highway is similarly threatened by a new breed of "highwaymen," called crackers, ranging from malicious pranksters to hardened terrorists. For the sake of public trust in the Internet, an infrastructure must be designed to support its safe use; systematic mechanisms and protocols must be developed to prevent breaches of security.

Since the Internet is an international collection of independent networks owned and operated by many organizations, there is no uniform cultural, legal, or legislative basis for addressing misconduct. Because the Internet has no central authority through which it can regulate the behavior of those using it, most organizations connected to the Internet have their own security policies. But these policies vary widely in their objectives and how they are put into effect.

Today most companies have a formal or informal information-security policy—a written or oral statement of objectives for ensuring that a system and the information in it meet with only appropriate treatment. Associated with this statement are those corporate and personal practices that must be implemented to reach the policy's goals. Typical policy objectives include protecting the confidentiality of private information, preventing unauthorized modification of data (that is, ensuring data integrity), and preserving the availability of system resources (such as computer time), in accordance with the needs and expectations of the system's users. In printed form, the policy and practices can range from a single page to manuals of several volumes.

Just as a legal system is designed to stop wrongdoers from harming those who live within its boundaries, a security policy prevents the unacceptable use of an information system's resources and data without impeding legitimate activity. The policy must protect not only the data stored on those company computers connected to the network, but the data contained in the communications relayed by the network as well; electronic mail passed along by network routers must be as sacrosanct as personnel records stored on the corporate mainframe.

A formal security policy may consist of a mathematical model of the system as a collection of all its possible states and operations, plus a set of constraints on when and how they may exist. But just as it is difficult to write laws that precisely define unacceptable behavior, it is hard to write security policies that formally and precisely express which activities are disallowed.

In current practice, security policies are usually stated informally, in ordinary language—which hobbles the task of translating their intent into a computerized form that automates enforcement. Imprecise translation, however, is not the only problem; automated security mechanisms may be configured incorrectly, too. In either case, the problem opens the system to malicious behavior.

Basic to any security policy is prevention of intrusion—that is, denial of access to a system's data or resources by someone not cleared for such access. Even an unintentional intrusion violates security.

As serious as an intrusion is, it is just the start of security problems. Determining what the intruder may have done once he or she gained access is usually more critical. As far back as 1980, the consultant James P. Anderson of Fort Washington, Pa., in his seminal report *Computer Security Threat Monitoring and Surveillance,* defined a still-useful list of the types of mischievous actions an intruder can carry out, which may be summarized as:

- Masquerading (impersonating an authorized user or a system resource, such as an e-mail server).
- Unauthorized use of resources (running a lengthy program that eats up computing cycles and so keeps others from running programs).
- Denial of service (by, say, deliberately overloading a system with messages to keep others from gaining access to it).
- Unauthorized disclosure of information (illicitly reading or copying an individual's personal information, such as a credit card number, or sensitive corporate data, such as business plans).
- Unauthorized alteration of information (tinkering with file data).

A single intrusion can result in a number of these problems.

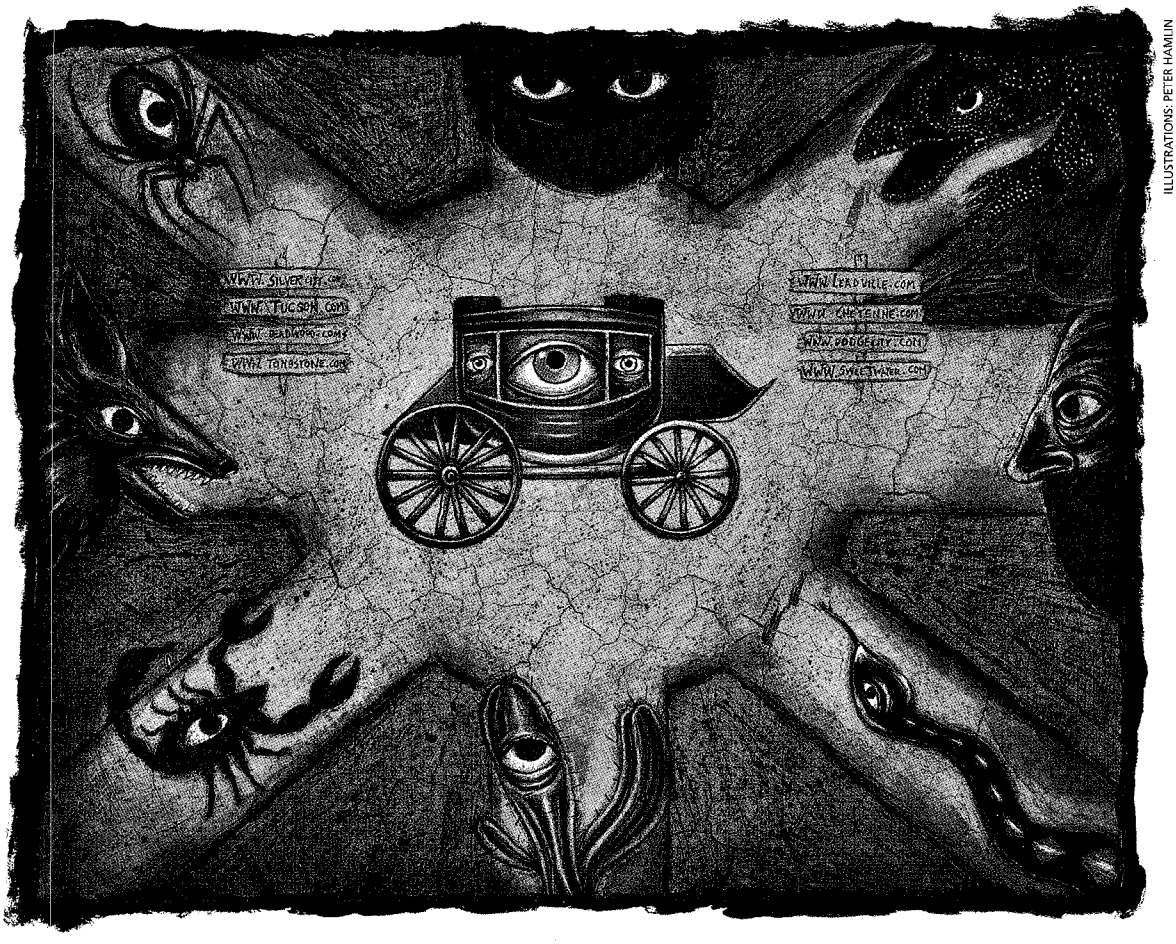
Ways to detect an intrusion and assess what the intruder did must be well thought out. For the most part, they will rely upon the ability of each system on the Internet to keep a log of events. The logs are invaluable for intrusion detection and analysis; indeed, they are basic to all post-attack analysis. Authors of the security policy must determine what to log (keeping in mind how the desired level of logging will affect system performance) and how the logs should be analyzed. The logs should note who has entered the system as well as what they have done.

Before a detailed examination is made of security methods, the issues affecting security enforcement warrant a broad overview.

An ounce of prevention, a pound of detection

The means of enforcing security policy involves either prevention or detection. Prevention is prophylactic—it

0018-9235/97/$10.00©1997 IEEE



ILLUSTRATIONS: PETER HAMLIN

seeks to preclude the possibility of malicious behavior. Detection, on the other hand, aims to discover and record any possibly malicious behavior as it occurs.

Among the protection mechanisms are access controls such as permission to access files, cryptography for safeguarding sensitive data such as credit card numbers, and authentication by asking for a password. All these are designed to ensure that only an authorized person can gain access to systems and alter information. Audit mechanisms, on the other hand, are investigative tools that detect and quantify malicious behavior. For example, some tools examine user activities on the system as they occur, while others check the records (called "audit logs") of system behavior.

One class need not be employed exclusively; in fact, most systems employ both. Audit can serve to review the effectiveness of access controls, while audit logs usually have the highest levels of access control to prevent a cracker from covering his or her tracks by altering them.

Even so, policies cannot be enforced exactly because of the limitations inherent in translating policies stated in everyday language into the software that enforces its intent. A case in point: the file protection mechanism of the Unix operating system can limit access to a file, but it cannot prevent any user who has permission to read a file from making a copy of it.

Besides the technological gap, there can be a gap between social policies and information security policies. People can usually distinguish between unintentional mistakes and malicious actions; computers cannot. There is even a gap between policies and actual user behavior: a system can be abused by careless authorized users.

The goal of protection mechanisms is to restrict a user's activities to those allowed by the security policy. A security policy might forbid any external users' viewing of information on an internal Web server, allow all internal users to view the information, but permit only certain corporate users to add to or change information on the server.

When a person tries to access a protected object—be it a text file, a program, or some hardware resource, such as a server—the system's access control mechanism determines whether that person is authorized to do so. If so, the person gains access; if not, the access is denied and an error message may be returned to the user. The decision is usually made during run-time at the beginning of each access. Alternatively, users may be given an electronic token, which they turn in at system start-up prior to making any accesses.

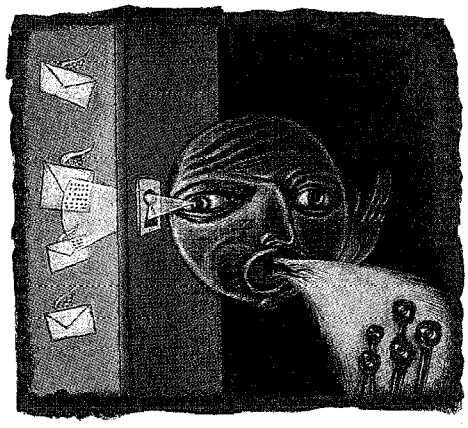
Creating and maintaining a security policy [Fig. 1] is an iterative process, during which the policy's authors must identify the organization's and users' security expectations, set them forth in a policy, enforce the policy, re-assess the system in light of policy violations or intrusions, and modify the original policy specification. During the re-assessment step of each iteration, both the policy and the protection mechanisms may be refined to address new attacks, close vulnerabilities, and update the policy to accommodate new user and organization requirements.

Protection's vulnerabilities

A cracker transgresses a security policy by exploiting the vulnerabilities in the system; if there were none, all attacks would fail. Vulnerabilities exist because the system's

MATT BISHOP, STEVEN CHEUNG,
& CHRISTOPHER WEE
University of California at Davis

designers, implementers, and administrators, when considering the problems would-be intruders might present, make assumptions and tradeoffs—about the environment in which the system will be used, the quality of data on which it will work, and the use to which it will be put. These assumptions stem from personal experience, beliefs about the environment in which the system operates, and the laws and cultural customs of the workplace. Vulnerabilities can be extremely subtle, existing in systems for years before being noticed or exploited; further, the conditions under which they can be exploited may be quite fleeting.



A good example of how assumptions breed vulnerability is to be seen in the development of the Unix operating system. Unix was created by programmers in a friendly environment, in which security mechanisms had to deal only with simple threats, such as one user accidentally deleting another's files. But as the Unix system's popularity grew, it spread into commercial realms, where the threats were very different.

For example, the original design of the Unix system has one all-powerful user (the "super-user"). Now in military and many other environments, the existence of such a user is a serious flaw. In fact, most attackers attempt to gain access in the guise of this user, so they can modify log-in programs or system libraries or even the Unix kernels, to let them return later. So the starting assumptions about security needs, reasonable though they were in the environment in which the Unix system was born, did not generalize well into other environments.

Vulnerabilities also arise when a use is made of systems that was not foreseen when they were built. Suppose a company decided that data from external World Wide Web servers should be barred from the company's network, say, to prevent unauthorized software from being sneaked into the system. To this end, the company could set up a firewall—software that can be configured to block specific types of communications between internal and external networks. To prevent Web traffic, the firewall might be configured to block communications using server port 80, which is the default port used by the Web's hypertext transport protocol (http) to transfer data. However, if someone outside the firewall purposely ran a World Wide Web server that accepted connections on port 25, the firewall would let communications to that server through on port 25. Because this kind of usage is not covered by the assumption (all http communications will go through the firewall at port 80), the site is vulnerable.

Another source of weakness is any flaw in the software system's implementation. A good example here are early server implementations that did not check input data. This allowed attackers to send messages to a Web server and have it execute any instructions in those messages.

Another good example is the initial implementation of Sun Microsystems Computer Corp.'s Java programming language, used to provide downloadable and executable programs called applets. To limit the dangers to the system receiving an applet, the designers restricted the actions the applet could perform, yet a number of implementation flaws allowed the little programs to breach those restrictions.

Also, it was the intent when designing Java to constrain each applet to connect back only to the system from which it was downloaded. To do this, an applet had to be written so that it identified the download system by its alias, or domain name (say www.xyz.com), not its absolute, or network, address (that is, 123.45.6.7), which is the actual address the Internet uses to locate the download system.

The problem, then, was that, when the applet asked to be reconnected to the download system, it had to request the download server's network address over the Internet by sending its domain name to a domain-name server, whose job it was to return the actual network address. If an attacker corrupted the domain-name-to-network-address translation tables in the domain server, it could "lie" and give a false network address.

Java's implementers trusted that the domain-name server's look-up table would be correct and reliable but, under fire from an attacker, it need not be. They later fixed this leak by having applets refer to all systems by addresses, not names.

Errors made when configuring the security system give rise to other vulnerabilities. For example, most World Wide Web servers allow their system administrators to use the address of the client asking for a page to control access to certain Web pages—such as those containing private company data. Should the system administrator mistype an address, or fail to restrict sensitive pages, the company security policy can be violated. Any time a system administrator or user must configure a security-related program—in the specific case noted by typing in a list of allowed-user addresses—a vulnerability exists.

Hardware vulnerabilities, usually more subtle, can also be exploited. Researchers have studied artificially injecting faults into smart cards by varying operating voltages or clock cycles so that the cryptographic keys inserted by the card's issuer could be discovered by comparing good and bad data. Evidently, while the "burning" of keys into hardware is supposed to protect them, it may not protect them well enough.

Vulnerabilities are not confined to end systems like servers. The computers, protocols, software, and hardware along the path to the server—that is, those that make up the Internet itself—have weak points, too. Consider the vulnerability of a router—a computer designed to forward data packets to other routers as those packets traverse the Internet to their final destination.

A router uses a routing table to determine the path along which the packet will be forwarded. Periodically, routers update each other's tables, making it possible to reconfigure the network dynamically as more paths are added to it. If, through design or error, a router were to announce that it were the closest one to all other routers, they all would send it all their packets. The misconfigured router would try to reroute the packets, but all routes would lead back to it. So the packets would never reach their destination—a perfect example of a denial-of-service attack and one that would bring the Internet to its knees.

Whom do you trust?

Central to the problem of vulnerability is the issue of what or whom to trust. Designers and engineers trust a system will be used in a certain way, under certain conditions; design teams trust that the other teams did their jobs correctly so that pieces fit together; program designers trust that the coders do not introduce errors; consumers trust a system will perform as specified. Vulnerabilities arise at every loose link in the chain of trust.

The vast scope of the Internet demands trust. Suppose Robin in Seattle wants to send a love letter via electronic mail to Sam in Tierra del Fuego. Robin types the letter on a computer and uses a mail program to send it to Sam, trusting that:

- The mail message contains the letter as typed, not some other letter.
- The mail program correctly sends the message from the local network to the next network.
- The message is sent on a path, chosen for efficiency by routers, over the Internet to Sam's computer.

• The destination computer's mail-handling program will receive the message, store it, and notify Sam that it has arrived.
• Sam will be able to successfully read the message using a mail-reading program.

For Robin's confidence to be well-placed, multiple pieces of hardware (including computers and dedicated routers) and the transport medium (be it twisted pair, fiber-optic cable, satellite link, or some combination thereof) must operate in the way intended. In addition, numerous pieces of software (including the mail programs, the operating systems, and the software that implements message transportation) must work correctly. In fact, the number of components in the network can become quite large and they must all interact correctly to guarantee that electronic mail is delivered safely. But if one of the components acts in some other way, Robin's trust is misplaced.

The man in the middle

Suppose that an attacker is competing with Robin for Sam's affections, and wants to intercept their e-mail *billet-doux*. If the messages traveling over the Internet can be modified en route, the message Sam receives need not be the one Robin sent. To do this, the attacker must change the router tables so that all e-mail messages between Robin's and Sam's computers are forwarded to some intermediate system to which the attacker has easy access. The attacker can then read the messages on this intermediate site, change their contents, and forward them to the original destination as if the intermediate site were legitimately on the message's path—a so-called "man in the middle" attack.

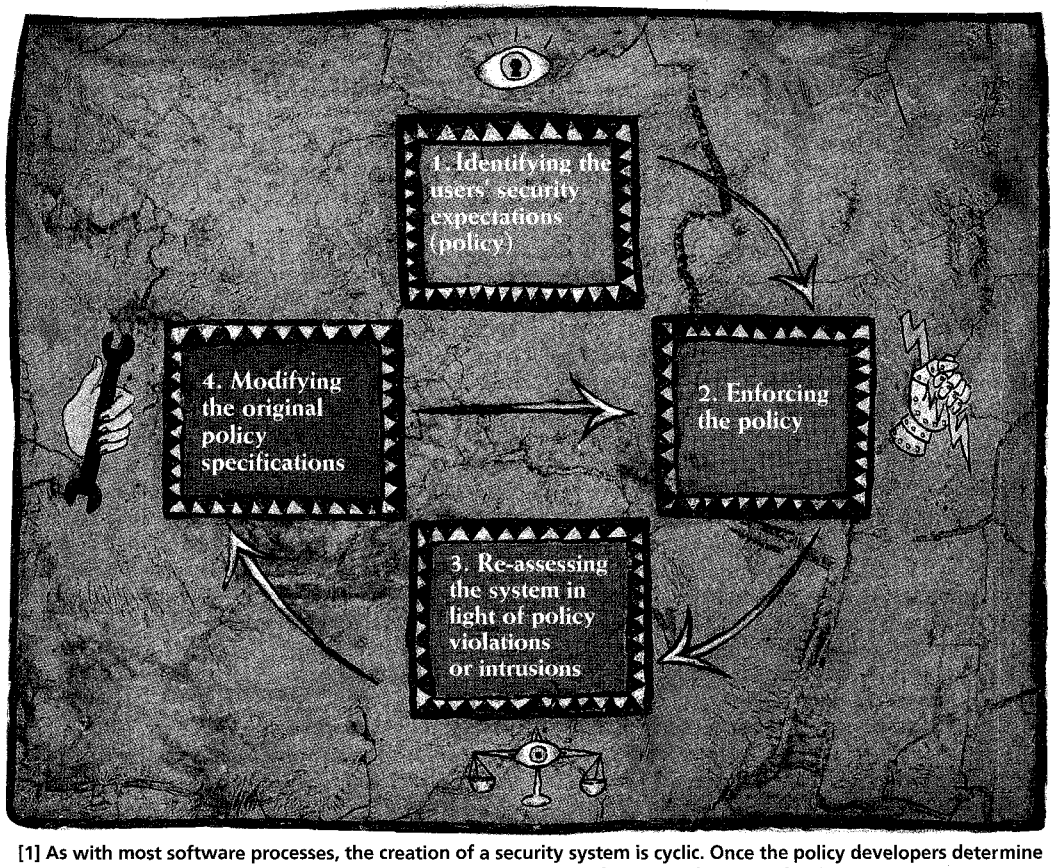
Using cryptography to hide the contents of messages, while often seen as the ultimate answer to this problem, is merely a part of the solution, because of a simple yet fundamental problem of trust: how do you distribute cryptographic keys? Public-key cryptographic systems provide each user with both a private key known only to that user and a public key that the user can distribute widely. With this scheme, if Robin wants to send Sam confidential mail, she enciphers a message using Sam's public key and sends the enciphered message to him [Fig. 2]. Only Sam, with his private key, can decipher this message; without that key, the attacker cannot read or change Robin's message.

But suppose the attacker is able to fool Robin into believing that the attacker's public key is Sam's, say by intercepting the unencoded e-mail message that Sam sent giving Robin the public key and substituting his own. Thus, Robin would encipher the message using the attacker's public key and send that message to Sam. The attacker intercepts the message, deciphers it, alters it, and re-encrypts it using Sam's real public key. Sam receives the altered message, deciphers it, and the romance goes sour.

The situation becomes even more complicated with the World Wide Web. Suppose Robin uses a Web browser to view a Web site in Germany. The German Web page, put up by an attacker, has a link on it that says: "Click here to view a graphic image." When she clicks on the link, an applet that scans her system for personal information (such as a credit card number) and invisibly e-mails it to the attacker, is downloaded along with the image. Here, Robin trusted the implied promise of the Web page: that only an image would be downloaded. This trust in implied situations ("this program only does what it says it does") is violated by computer programs containing viruses and Trojan horses. PC users spread viruses by trusting that new programs do only what they are documented to do and have not been altered, so they fail to take necessary precautions.

Auditing's objectives

Auditing, a way of finding such problems, has five main aims:
• To trace any system or file access to an individual, who may then be held accountable for his or her actions.

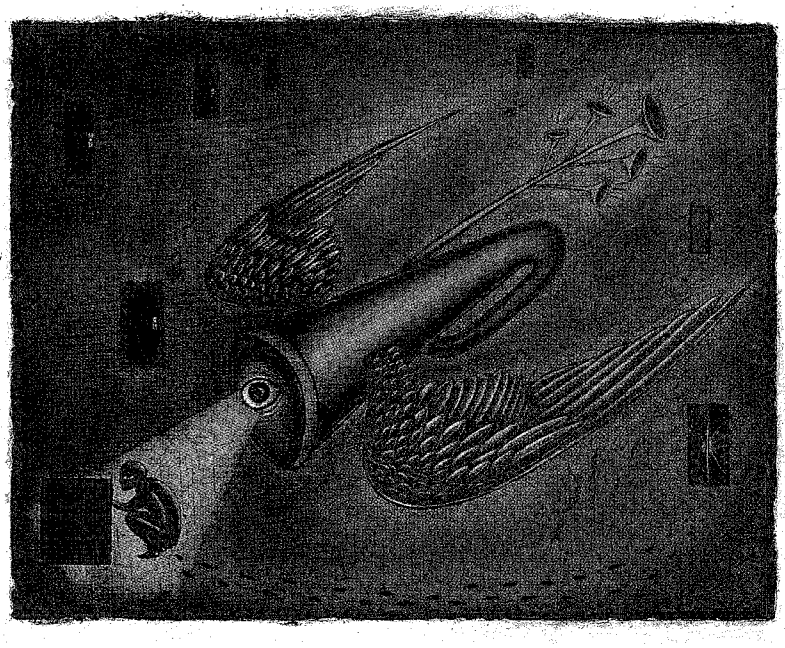


[1] As with most software processes, the creation of a security system is cyclic. Once the policy developers determine what the ultimate users of a system expect and need in the way of security, the cycle of enforcement, reassessment, and modification that makes up its life begins.

• To verify the effectiveness of system protection mechanisms and access controls.
• To record attempts that bypass the system's protection mechanisms.
• To detect users with access privileges inappropriate to the user's role within an organization.
• To deter perpetrators (and reassure system users) by making it known that intrusions are recorded, discovered, and acted upon.

While the goals of auditing are clear, they do not dictate that any particular audit scheme, or model, be followed, nor do they indicate how to perform the auditing. Thus current auditing consists of various *ad hoc* practices.

Auditing requires that audit events—such as user accesses to protected files and changes in access privileges—be recorded. A log is a collection of audit events, typically arranged in chronological order, that represents the history of the system; each logged event represents any change in the state of the system that is related to its security.



Because of the complexity of modern computer systems and the inability to target specific actions, audit logs can be voluminous. In fact, the logs are often so large that human analysis is quite time-consuming. It is therefore desirable to have tools that would cull entries of interest from the log. But development of such automated audit tools for all types of computer systems is hampered by three things: a lack of standard formats (such as ASCII or binary) and semantics (the order in which statements occur) for audit logs, and (as mentioned initially) the practice of stating security policies in an ordinary language that does not lend itself to automation.

While some tools have appeared to aid in log analysis, they are difficult to use. As a result, logs are usually inspected manually (often in a cursory manner), or possibly using some audit browsing tools that employ algorithms able to cluster together related data. All too often, they are not reviewed at all.

When the log is reviewed, the auditor compares the users' activities to what the security policy says that user may do and reports any policy violations. An auditor can also use the log to examine the effectiveness of existing protection mechanisms and to detect attempts to bypass the protection or attack the system. The identities of those behind attempts to violate the policy sometimes can be traced in the history of events, provided the audit log contains sufficient detail.

On networked computers, tracing the user may require an audit of logs from several hosts, some quite remote from the system where the intrusion occurred. Law enforcement agencies may want to use these logs as evidence when prosecution of the perpetrators is warranted, and this can spark jurisdictional and other legal disputes.

The Internet's basic design philosophy is to introduce new resources and capabilities at the end points of the network—the client and the server—so as to keep the infrastructure simple, flexible, and robust. The disadvantage of this philosophy is that the Internet Protocol requires only that the network make its best effort to deliver messages; it does not require that messages be delivered at all costs. Nor does it require that records of delivery be kept; as a result, logging on the Internet is merely a function of implementation, not a requirement of the protocols.

In a logging process known as packet sniffing, special software running on each node reads and logs data contained in the packets. Depending on the amount of traffic on the Net, sniffing can use up a lot of processing power and storage space. To minimize this resource drain, sometimes only the header portion of packets—which may contain such information as the packet's source, destination, and the number of packets making up the complete transmission—is logged and the message data in the packet is ignored. Deducing user behavior and the actions caused by the message from the relatively low-level information obtained by sniffing calls into play many extrapolations and assumptions.

Whereas there is no standard for all types of systems, most World Wide Web servers do use a standard audit log format, so audit tools have been developed for a wide range of Web servers. Also, there is something of a standard for electronic mail: e-mail often has the name of each computer encountered, and some further information, placed in the headers of the message as the mail moves over the Internet. These headers constitute a mini-log of locations and actions that can be analyzed to diagnose problems or to trace the route of the message.

Although prevention mechanisms are designed to prohibit violations of the security policy in the first place, a specter of accountability—the attacker's fear of being discovered—is raised by detection mechanisms and thus serves as a deterrent. An audit, then, may be thought of as a defense against attacks, too, albeit a reactive one, in which clues to the identity and actions of the intruder can be detected.

System check

Fortunately, several tools exist to help administrators check their systems' security For Unix systems, three popular tools are Satan, tripwire, and Cops; these are available free of charge at many sites on the Internet.

Satan is a World Wide Web–based program that analyzes systems for several known vulnerabilities exploitable only through network attack—such as the ability of a cracker to make available to any server files that are supposed to be restricted. It provides a Web browser interface, and allows scanning of multiple systems simply by clicking on one button. The browser presents a report outlining the vulnerabilities, and provides tutorials on the causes of each, how to close (or mitigate) the security flaw, and where to get hold of more information (such as the Computer Emergency Response Team, or CERT as it is popularly known, a group within Carnegie Mellon University's Software Engineering Institute in Pittsburgh that issues advisories about computer security).

Another means of verifying security is by checking up on the integrity of the system software—such as log-on programs and libraries—by seeing that the software has not somehow been altered without the administrator's knowledge. Tripwire is an integrity checking program that uses a mathematical function to compute a unique number ("hash") based on the contents of each file, be it a document or program. Each hash, along with the name of its corresponding file, is then stored for future refer-

ence. At random intervals, a system administrator reruns tripwire and compares the results of the new run with the results of the original one. If any of the hashes differ, the corresponding file has been altered and must be scrutinized more closely.

Cops examines the contents of configuration files and directories and decides if either their contents or settings threaten system security. For example, on Sam's Unix system, the contents of a configuration file might state that Robin need not supply a separate password to use Sam's system. This poses a double security problem at many sites, since anyone who obtains access to Robin's account also obtains access to Sam's system. Tripwire will not detect this problem, as it simply looks for files that changed—and the access control file does not change—but Cops will scan the configuration file, reporting that Robin does not need a password to log in to Sam's system, as part of its analysis of the configuration file's contents.

Intrusion detection

Intrusions can be detected either by manual analysis of logs for any suspicious occurrences or by automated tools that detect certain specific actions. Examples are unusual log-in times or unusual system characteristics, such as a very long run time for one supposedly simple program. Automated methods, of course, process lots of data more quickly and efficiently than humans could. The data comes from either logs or from the current state of the system [Fig. 3].

Human analysis entails looking at all or parts of the logs for a system with a view to uncovering suspicious behavior. The audit data may, though, be at such a low level, as previously mentioned, that events indicating an intrusion or attack may not be readily detectable as such. Here, detecting attacks may require correlating different sets of audit data, possibly gleaned from multiple logs; thus, a change in access privileges from the privilege log might be compared with the log-in log's record of the location from which the user who changed the privileges logged in. The data may span days or weeks and is often voluminous.

Another hindrance is that the person conducting the analysis must have special expertise, both in the hardware and software that constitute the system being audited and the particular way in which it is configured, to understand what may have happened and what actually did occur.

The previously mentioned consultant, James P. Anderson, also made the first serious study of how computers were being used to detect security violations. Modern computers have a capacity to analyze large amounts of data accurately, provided they are programmed to analyze the right data; to correctly detect intrusions, they must be told what to look for.

For this purpose, three methods have been established: anomaly detection, misuse detection, and specification-based detection. Among them, there is no one best approach to detecting intrusions; in practice, the particular combination of approaches used is tailored to an organization's specific needs.

Anomaly detection compares the current behavior of a person using a system to the historical behavior of the person authorized to use the system. The technique presumes that deviations from prior behavior—say, different log-in times or the use of different commands—are symptoms of an intrusion by an unauthorized person using a valid account. Similar reasoning suggests that a program altered to violate the security policy—that is, one changed by a virus so it now writes to other executables or to the boot program—will behave differently than the unaltered version of the program.

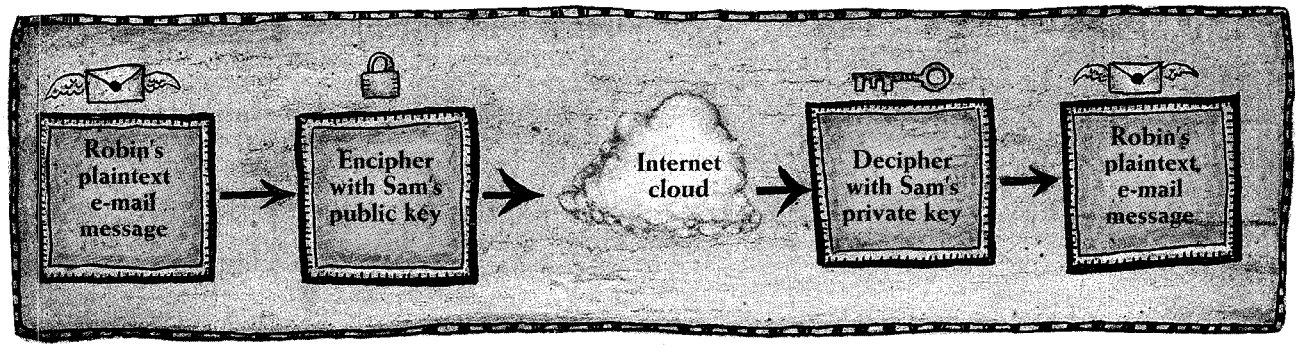
An intrusion detection system (IDS) based on anomaly detection must first be trained to know the expected behavior of each user, and there could easily be hundreds of users. This normalcy profile is built using statistical analysis of each user's use of the system and logical rules that define likely behavior for various types of users—programmers, sales managers, support personnel, and so on. Once a normalcy profile is established, the IDS monitors the system by comparing each user's activity to his or her normalcy profile. If some activity deviates markedly from the profile, then the IDS flags it as anomalous and, therefore, a possible intrusion.

Admittedly, a legitimate user can be flagged as an intruder (a false positive) because abnormal behavior is not necessarily an attack; for example, a legitimate user may become more proficient in using a program and thus employ commands not previously invoked. False negatives also occur when an intruder's actions closely resemble the normal behavior of the legitimate user whose log-in they have obtained. Finally, establishing the right time period over which to analyze the user's behavior and how often to retrain the IDS system affects its performance.

One anomaly detection system observes the interaction between a program and the operating system, and builds normalcy profiles of the short sequences of system calls normally made. Activity outside this is presumed to be part of an intrusion. For example, if an attacker tried to exploit a vulnerability in which unusual input, such as an e-mail message sent to a program rather than a person, caused a mail-receiving program to execute unexpected commands, these commands would be detected as anomalous and a warning given.

Unlike anomaly detection, in which normal user behavior is taught so that unusual behavior characteristic of an attack can be distinguished, misuse detection does not require user profiling. Rather it requires *a priori* specification of the behaviors that constitute attacks; if any observed behavior matches a specified attack pattern, the IDS warns the systems administrator.

The techniques used to describe the attacks vary. One method is to list events expected to be logged during an attack. A graph-based misuse detection IDS employs a set of rules that describe how to construct graphs based on network and host activity—for example, a graph of the connections between the systems involved in an attack, the time at which they became involved, and the duration of their involvement. The rules also describe at

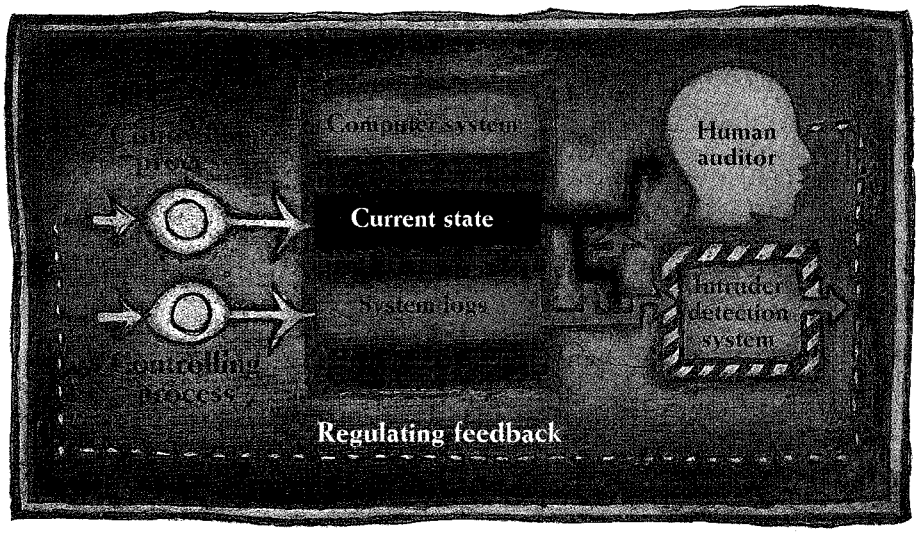


[2] In public-key cryptography, a user sends a public-key–encrypted message, as shown here, that can be decrypted only with the recipient's private key. Many think such a scheme makes communication secure. But an attacker can defeat it by artfully switching the public key.

what point such a graph is considered to represent an attack.

Another is to have an expert write a set of rules describing "felonious" behavior. For example, suppose an attacker gave unusual input to a mail-receiving program to change the way it operated. The expected system calls were "read-input; write-file," but the attacker's input would try to change the set to be "read-input; spawn-subprocess; overlay-program." The last two items in the altered set, which tell the system to execute another program, indicate an attack. Were the attacker to try to intrude using that technique, the misuse detection program would detect it.

The misuse detection method can be highly accurate, but, unlike anomaly detection, it cannot detect attacks that fall outside its prepared list of rules describing violations of security. In addition, it depends upon having an expert who is able to specify such rules.



[3] An intrusion detection system (IDS) processes information from both the computer system and its logs and reports any problems to a security auditor. The initial information can also be used to determine what other actions should be taken and what further information should be logged.

While anomaly and misuse detection catch security breaches by focusing on the attacker's behavior, specification-based detection describes breaches in terms of the system's expected behavior. Further, if system behavior has been specified accurately, there are no false alarms. The first step is to formally specify how the system should behave in all circumstances. Once fully profiled, the system is monitored and all its actions compared against the specification; any item of system behavior that falls outside what is specified as correct is flagged as a security violation.

One approach to specification-based detection uses a special policy-specification language to describe the security policy in terms of the access privileges assigned to each program in the system. This language indicates under what conditions certain system calls may be made, and it requires knowledge about privileged programs, what system calls they use, and what directories they access. Depending on the particular system for which the policy is being specified and the specification language used, creating specifications of this kind may require expertise, skill, and some time—although some effort might be automated using program analysis. But if the specifications do not cover all eventualities, false negatives (intrusion alarms) can occur.

Several companies and research groups have developed intrusion detection systems. The authors' group at the Computer Security Laboratory of the University of California, Davis, is designing and developing one such tool, called GrIDS, that will monitor both systems and network traffic, looking for actions indicating misuse. It also supports analysis of attacks conducted from more than one outside source, even when the attack is spread over a large number of systems.

Other, nonresearch systems are less ambitious, but are currently deployed. CMDF from Science Applications International Corp. (SAIC), San Diego, Calif., uses the anomaly approach by building a database of statistical user profiles and looking for deviations from that profile. NetRanger from WheelGroup Corp., San Antonio, Texas, and NetStalker from Haystack Labs Inc., Austin, Texas, detect attacks by comparing system actions to known exploitations of vulnerabilities.

Counterattack and damage assessment

Several responses to security violations are possible, particularly if the attack is detected while it is occurring, typically within a matter of seconds or minutes after an intrusion starts. The simplest reaction is to alert other people, while a more complex, automated detection system might respond autonomously to any violations of policy. The type of response selected depends on the degree of confidence that an attack is actually under way, and upon the nature and severity of the attack.

The first response by a security team to a reported attack is to gather the information needed to analyze the violation and decide how to respond further. Also, additional auditing—of more user accounts or more system resources—may be turned on, possibly only for those users involved in the violation or possibly, if the extent or nature of the violation of policy is not fully understood, for the entire system. Moreover, the system can turn defense into offense, fooling the attacker by countering his activities with misleading or incorrect information; the attacker can even be lured by the security team to a system designed on purpose to monitor intruders.

Another common response to a violation is to determine who is responsible. After that, legal action might be taken, or more direct responses (such as blocking further connections from the attacker's site or automatically logging the attacker off) may be appropriate. However, determining whom to hold accountable can be very difficult, since Internet protocols do not associate users with connections, and the attack might be laundered through multiple stolen accounts and might cross multiple administrative domains, as was the case with the attack described by Clifford Stohl in *The Cuckoo's Egg* (Doubleday, 1989). No formal support infrastructure exists to trace attacks that have been laundered in this way.

Once a violation has been detected, the attacked system needs to be analyzed to determine the immediate cause of the system's vulnerability and the extent of the damage. Knowing the vulnerabilities exploited by the attacker can often help to stop ongoing attacks and stop future ones. If the vulnerability cannot be fixed, knowing its causes helps determine what to monitor.

Security systems that detect deviations in a user's behavior can indicate only that a user may be an attacker, not what weak points were exploited to violate the security policy. Misuse detection systems catch exploitations of known vulnerabilities, but may give only a partial set of those exploited, because the activities that trigger the IDS may not be the root cause of an attack. That is, an attacker may at first use a means to violate the policy that goes undetected; only subsequent violations, based in part on the initial one, are reported.

Successful assessment depends upon the integrity of the audit data and the analysis programs used for the assessment, and a

sophisticated attacker may tamper with the audit data or disable or modify the analysis programs to hide the attack. Thus extra resources are needed to secure those data and programs.

For example, where security is of utmost importance, as in military and financial establishments, audit data may be written to write-only devices, such as write-once, read-many (WORM) optical storage disks, and analysis programs may be put on a dedicated machine that does not have ordinary user accounts or network connections and uses the vendor's distribution of the operating system.

Assessment can be approached using event-based or state-based analysis. In event-based analysis, the causal relationships in the events recorded in the log are tracked down. Parent-child processes are a good example: the Unix operating system records each process with an ID that identifies the process that spawned it and the user who started it. Moreover, some versions of Unix record these IDs with the corresponding events in the log.

With the aid of such information, the processes involved in unauthorized events can be pinned down. By tracing the parent-child process relationships, it is often possible to determine the vulnerabilities exploited and assess the damage caused by the attack. Then the user-process associations can be used to identify the user account(s) from which the violation of policy occurred.

The state-based approach constantly analyzes the current state of the system to see if it is secure in accordance with current requirements. A state includes the contents of configuration files and the rights of users to access various files.

Picking up the pieces

Using the information obtained through analysis, the system can be returned to a secure state—a process referred to as recovery. Recovery may mean a number of things. It may include terminating an on-going attack to stop further damage, replacing corrupted files with uncorrupted copies, fixing vulnerabilities to protect the system against future attacks, taking appropriate actions (such as notifying affected parties or aborting planned actions), and restarting system services that have been made unavailable.

Since systems are generally backed up periodically, a common technique used in recovery is rollback—that is, restoring a system to its state before the attack, using the backup files created before the intrusion occurred. A complete backup of all the files in the system may be effected, or else a selective backup in which only copies of recently modified files or critical files are saved. Different levels of backup may be combined—complete system backup once a week, say, and selective backups once a day—depending on the level of integrity a site wishes to maintain and the frequency with which files change significantly.

To reconstruct the pre-attack state of the system, it may be necessary to use the last complete backup plus any later selective backups. So the frequency of the backup is important because, during rollback, every change made since the last backup may be lost. For unchanging programs, backups may not be needed if the program distribution disks are on hand. Note that this rollback technique is useful even if complete damage assessment is not possible.

Another means of returning to a secure state is reconfiguration, in which the system is modified to bring it to a secure state by fixing all configuration files and, if needed, reinstalling all software. Reconfiguration is appropriate when one cannot roll back to a secure state, possibly because backups have not been done recently or the system has been in an insecure state since its inception.

Many vendors aid recovery by distributing "patches" or fixes for software once a vulnerability becomes known. Actually, this can be pre-emptive, because system administrators often receive program patches before the vulnerability has been exploited on their system. But sometimes a weakness cannot be fixed: perhaps the flaw is one of interaction between the software and another component, requiring modification of the operating system, or perhaps no fix is available. In such cases, administrators may be forced to disable the offending software or service. As an example, if an account's password has been compromised, its owner must change the password before it can be used again. Freezing the account before the password change can prevent future attacks through the compromised account.

A brighter future

As the need for security on the Internet increases, new mechanisms and protocols are being developed and deployed. But a system's security will always be a function of the organization that controls the system. So whether the Internet becomes more secure depends entirely upon the vendors who sell the systems and the organizations that buy them.

Ultimately, people will decide what, how, and how much to trust; and so security is a nontechnical, people problem, deriving its strength from the understanding by specifiers, designers, implementers, configurers, and users of what and how far to trust. ◆

To probe further

A seminal work that introduced the idea of using audit logs to detect security problems is James P. Anderson's *Computer Security Threat Monitoring and Surveillance* (James P. Anderson Co., Fort Washington, Pa., April 1980).

In "Decentralized Trust Management," *Proceedings of the IEEE Conference on Security and Privacy*, May 1996, pp. 164–73, M. Blaze, J. Feigenbaum, and J. Lacy discuss trust and illustrate the complexities of managing it in a distributed environment. Similarly, D. Denning discusses trust and the effect of misplacing it in "A New Paradigm for Trusted Systems," *Proceedings of the Fifteenth National Computer Security Conference*, October 1992, pp. 784–91.

A different take on security analysis is B. Cheswick's "An Evening with Berferd in Which a Cracker is Lured, Endured, and Studied," *Proceedings of the Winter 1992 USENIX Conference*, January 1992, pp. 163–74. This paper presents an encounter with an attacker who attempted to penetrate a Bell Labs system and was spotted. Rather than block the attack, the authors decided to allow the attacker access to a controlled environment to see what he or she would do.

Security problems in various Java implementations and in its design itself, and in downloadable code in general, are discussed in D. Dean, E. Felten, and D. Wallach's "Java Security: From HotJava to Netscape and Beyond," *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, May 1996, pp. 190–200.

Identifying intruders in the first place is very complex, and often impossible. A statistical technique to correlate two connections to see if they belong to the same session is proposed in S. Staniford-Chen and L. T. Heberlein's "Holding Intruders Accountable on the Internet," *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, May 1995, pp. 39–49.

Acknowledgments

Assisting intimately in writing this article were Jeremy Frank, who recently received his Ph.D. from the computer science department at the University of California, Davis, and works at NASA Ames Research, Moffett Field, Calif.; James Hoagland, a Ph.D. candidate at the University of California, Davis, who does research in computer and network security; and Steven Samorodin, a graduate student. Without their invaluable help, it would not have been possible to publish this work.

About the authors

Matt Bishop (M) is on the faculty of the Computer Security Laboratory of the department of computer science at the University of California, Davis, and does research in computer and network security.

Steven Cheung is a Ph.D. candidate at the same institution; his doctoral research concerns network intrusion detection.

Chris Wee is a postdoctoral researcher at Davis.

Spectrum editor: Richard Comerford