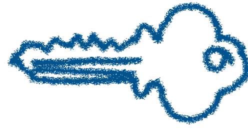


A Software Platform for Testing Intrusion Detection Systems

NICHOLAS PUKETZA, MANDY CHUNG, RONALD A. OLSSON, and BISWANATH MUKHERJEE, University of California, Davis

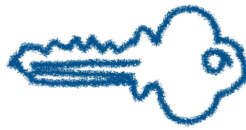
The steady growth in research on intrusion detection systems has created a demand for tools and methods to test their effectiveness. The authors have developed a software platform that both simulates intrusions and supports their systematic methodology for IDS testing.

Traditional computer system security efforts were aimed at rendering systems invulnerable to attack. However, because of system complexity, configuration and administration errors, and abuse by "authorized" users, this goal is unlikely to be realized[1] for most systems. For this reason, the emphasis on detecting intrusions is increasing.

Intrusion detection systems monitor system activities to identify unauthorized use, misuse, or abuse. IDSs offer a defense when your system's vulnerabilities are exploited and do so without requiring you to replace expensive equipment. As the use of IDSs increases, the need for tools and methodologies for testing and evaluating IDSs is also growing.

We have developed a software platform that simulates intrusions and tests IDS effectiveness. Our platform lets you create scripts that simulate both typical and suspicious activity. It also has a record-and-replay feature that helps you create scripts quickly and

0740-7459/97/$10.00 © 1997 IEEE



easily. We created this software platform to support our testing methodology, which we have discussed at length elsewhere[2] and review briefly here. Our testing methodology includes techniques and approaches that we adapted from the general software testing field. Because neither the methodology nor the software platform is specific to a particular IDS, they can be used to test and compare several different IDSs. IDS developers can use the platform and methodology to supplement their own testing approaches. The software platform can also be employed in testing other applications that require simulation of computer users, especially multiple cooperating users.

TESTING METHODOLOGY

As the boxed text, "Intrusion Detection Systems" on page 45 describes, IDSs detect anomalies and misuse. We designed our methodology to focus on misuse detection. In general, a misuse detection system consists of two key components: a database of intrusion signatures, which are encapsulations of the identifying characteristics of specific intrusion techniques; and a pattern-matching mechanism that searches for the intrusion signatures in user activity records (such as audit trails).

Because window interfaces are ubiquitous, concurrent intrusions are likely to occur.

To formulate our testing method, we first identified IDS performance objectives.

♦ *Broad detection range.* The IDS should be able to detect many types of intrusions.

♦ *Economy in resource usage*. The IDS should function without monopolizing system resources such as main memory, CPU time, and disk space.

♦ *Resilience to stress.* Stressful system conditions, such as a very high level of computing activity, should not impair IDS function.

Selective simulations. As the first step in our methodology, you select test cases—simulated user sessions—for the testing experiments. Although some of the tests require you to simulate "normal" sessions, most of the test cases are simulated intrusions. Which intrusions to simulate is a key problem. The best approach is to select test cases based on your organization's computer security policy, which should define what is and what is not an intrusion.

You can derive some test cases from published descriptions of well-known attacks. You can also develop site-specific test cases based on your security policy and private, onsite information. To ensure that your test cases cover many different kinds of attacks, you can use intrusion classifications[2] to guide your test case selection.

You should consider also that system intrusions can take one of two forms: sequential or concurrent. In a sequential intrusion, a single person issues a single sequence of commands from a single terminal or workstation window. In a concurrent intrusion, one or more intruders issue sequences of commands from several terminals, computers, or windows. The command sequences work cooperatively to carry out an attack. For example, an intruder can open multiple windows on a workstation and connect to a target computer from each window. Multiple intruders can try to conceal an intrusion attempt by distributing the suspicious behavior among themselves. Because window interfaces are ubiquitous, concurrent intrusions are likely to occur and an IDS should be able to detect them. Therefore, we recommend that you include simulated concurrent intrusions among your test cases.

After you select test cases, you can develop scripts that simulate the different intrusive behaviors.

Testing experiments. The rest of our methodology consists of using the simulation scripts in a variety of testing experiments. We have developed a set of detailed procedures, several of which are adaptations of the "higher-order" testing methods described by Glenford Myers.[3] Most of the procedures include the same basic steps. You create or select a set of test scripts and establish the desired conditions in the computing environment (such as the level of "background" computer activity). You then activate the IDS and execute the test scripts. Finally, you analyze the output.

We have divided the test procedures into three categories, which correspond directly to the three performance objectives described earlier.

♦ *Intrusion identification* tests measure the IDS's ability to distinguish intrusions from normal behavior.

♦ *Resource usage* tests measure how many system resources the IDS consumes. You can use the results of these tests to make decisions, such as whether it is practical to run a particular IDS in a particular computing environment.

♦ *Stress* tests check how the IDS is affected by "stressful" conditions in the computing environment. An intrusion that the IDS would ordinarily detect might go undetected under such conditions. For example, one stress test evaluates how the IDS performs on a heavily loaded computer.

When detection fails. If the IDS fails to detect a simulated intrusion, you attempt to find the source of the failure. Given the main IDS components in misuse detection—an intrusion database and a pattern-matching mechanism—and the IDS dependence on the computer's system activity logs (such as audit trails), an IDS may fail to detect an intrusion during testing because:

INTRUSION DETECTION SYSTEMS

An intrusion detection system is a system that attempts to identify intrusion attempts by both unauthorized users and insiders who abuse their privileges.[1] For example, an intrusion has occurred when

- an employee browses through confidential performance reviews,
- a system administrator modifies system files to permit an unauthorized user access to either system or user information,
- an intruder accesses or modifies other users' files or information,
- an intruder modifies routing tables in a network to upset message delivery, or
- an intruder installs a "snooping program" on a target computer to examine sensitive data (such as user passwords) contained in network traffic.

As these examples show, intrusions can lead to lost or altered data and can deny service to legitimate users. Intrusions can also cost an organization money and the trust of their employees and the public at large.

Interest in the research and development of IDSs has been growing over the last several years. Biswanath Mukherjee and his colleagues describe several IDSs in "Network Intrusion Detection,"[1] including the National Security Agency's Midas, AT&T's ComputerWatch, SRI International's Intrusion-Detection Expert System, Los Alamos National Laboratory's Wisdom and Sense, UC Davis' Network Security Monitor, and the Distributed Intrusion Detection System (DIDS) developed jointly by UC Davis, Lawrence Livermore National Laboratory, Haystack Laboratory, and the US Air Force.

IDSs use two general approaches: anomaly detection and misuse detection.

Anomaly detection is based on the premise that an intruder's behavior will differ noticeably from that of a typical user.[2] Anomaly detection works by establishing "profiles" of typical user activity such as login time, CPU usage, favorite editor, disk usage, and login session length. The IDS then uses these profiles to monitor current user activity and compare it with past user activity. Whenever a user's current activity deviates from past activity "significantly" (beyond some predefined tolerance level), the activity is considered to be anomalous, and hence suspicious.

A limitation of anomaly detection is that it depends on consistency in the behavior of users. In some environments, legitimate users may frequently change their behavior. For those users, it is difficult to create profiles that are flexible enough to tolerate legitimate variations in behavior, yet sensitive enough to detect intrusions.

In misuse detection, the IDS's goal is to recognize "specific, precisely representable techniques of computer system abuse."[3] The IDS detects intrusions by searching user-activity records for intrusion "signatures"—encapsulations of the identifying characteristics of specific intrusion techniques. Anomaly detection lets you detect masqueraders or legitimate users abusing their privileges without requiring knowledge of security flaws in the target system.[2] Misuse detection, on the other hand, can identify intrusive behavior even when it appears to conform with established patterns of use. For this reason, several IDSs employ both an anomaly detection component and a misuse detection component.

REFERENCES

1. B. Mukherjee, L.T. Heberlein, and K.N. Levitt, "Network Intrusion Detection," *IEEE Network*, May 1994, pp. 26-41.
2. D.E. Denning, "An Intrusion Detection Model," *IEEE Trans. Software Eng.*, Feb. 1987, pp. 222-232.
3. S. Kumar and E.H. Spafford, "A Software Architecture to Support Misuse Intrusion Detection," *Proc. 18th Nat'l Information Systems Security Conf.*, National Inst. Standards and Technology, Washington, DC, 1995, pp. 194-204.

- the system activity logs do not supply enough information for the IDS to detect the intrusion,
- the intrusion database does not contain a signature representing the intrusion, or
- the pattern-matching mechanism fails to recognize a match between a database signature and a record in the system activity logs.

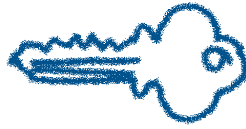
You can address the first problem by reconfiguring or adding to the system's logging component. This might require changes to the IDS to accommodate changes in the content or format of the incoming information it analyzes. The second problem is usually easy to correct: an IDS typically includes a mechanism for adding to its database of signatures. However, the language used to describe attack signatures might be inadequate to describe the intrusion. In that case, significant language development work might be needed. You can fix the third problem by debugging the pattern-matching mechanism.

You should continue the testing process until all detection failures are eliminated or at least explained; you can then use other security mechanisms to cover IDS weaknesses.

Limitations. The primary limitation of our IDS testing methodology is that it tests the IDS only against known attacks. In many environments, known attacks occur more frequently than new attacks because their details are often distributed widely (via newsgroup postings, for example), while the number of people who know the details of a truly new attack is probably small.

Still, sites that value security highly are clearly interested in detecting new attacks. Fortunately, the limitation of our methodology is not as severe as it initially seems. New attacks are often similar to known attacks; if an IDS can detect most of a thorough set of test cases (known attacks), the IDS will probably succeed in detecting new attacks as well. However, evaluating the thoroughness of the test case selection method is an open problem.

Even thorough testing may not expose all potential weaknesses in an IDS. Many IDSs include programs that run on the computers that they monitor. If intruders can take control of those computers, they can manipulate the IDS programs themselves. Also, if intruders have knowledge of the database of intrusion signatures in the IDS, they can easily attempt attacks that are not represented in the



SOFTWARE PLATFORM COMMANDS

Our software platform has four groups of commands, each of which has different key commands.

BASIC SESSION COMMANDS

- **connect** *host*—spawns a telnet process to connect to a host
- **login** *user password*—logs in with a user name and password
- **logout**—logs out from the current user session
- **ftp** *host user password*—establishes an ftp connection to a host and logs in with a user name and password
- **send_cmd** *command*—sends a command to a spawned process and returns command output, if any

SYNCHRONIZATION COMMANDS

- **sync_make_server**—creates a synchronization server
- **sync_connect_server** *host port#*—establishes a connection from the client to a specific port on the server host
- **sync_add_client**—adds a process as a client of the server (must be executed after the process is initiated)
- **sync_server—**starts the synchronization server for monitoring and receiving signals from clients
- **sync** *process_id sync_id*—establishes a synchronization point with an ID tag (process_id identifies the client to the server)
- **thread_exit** *process_id*—informs the server that the client process is exiting

COMMUNICATION COMMANDS

- **sync_data_snd** *process_id tag data*—sends a message with an ID tag (process_id identifies the sender to the server)
- **sync_data_rcv** *process_id src_process tag*—waits to receive a message with a specific ID tag from a source process (process_id identifies the receiver to the server)

RECORD-AND-REPLAY COMMANDS

- **record** *filename*—starts recording a command sequence into a file
- **replay** *filename1, filename2,...*—replays the activity recorded in the specified files

database. The privacy of the intrusion database should be well protected.

OUR PLATFORM

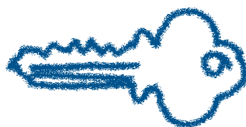
Our software platform is based on the Expect package[4] and the Tool Command Language Distributed Programming (Tcl-DP) package.[5] Expect is built on top of Tcl and provides additional commands for controlling interactive programs. It is particularly useful for simulating a human user. Tcl-DP is an extension to Tcl/Tk and provides a suite of commands for creating client-server systems.

Our platform extends Expect with mechanisms that facilitate intrusion simulation, including basic session commands for simulating common user commands (such as login and telnet) and a record-and-replay feature. You can also use the platform to develop a *concurrent script set*—a set of scripts that are executed simultaneously to simulate a concurrent intrusion. The platform includes synchronization and communication mechanisms for concurrent scripts.

The box “Software Platform Commands” on this page shows some of the basic session commands and other mechanisms provided by our platform.

- *Basic session commands.* The platform provides several commonly used commands—emphasizing ones that require user interaction, such as telnet, login, and ftp—as basic session commands. These basic session commands let you simulate many of an intruder’s basic activities.
- *Synchronization.* The synchronization mechanism lets you specify a fixed execution order for important events, even if they belong to separate scripts.
- *Communication.* To accurately simulate a group of users working together, concurrent processes must be able to communicate with one another (a *process* here refers to the execution of a single script that simulates a single user session). You can specify *send* and *receive* commands in concurrent scripts so that the corresponding processes can exchange data with one another.
- *Record-and-replay.* Our platform provides a “record-and-replay” feature to help you create scripts. You can record a sequence of commands during a recording session and then replay the recording at will. By using the record-and-replay feature, you can create an intrusion script quickly and easily without knowledge of Tcl and Expect. You can also create and replay concurrent scripts, complete with synchronization.

Platform in action. Expect and Tcl form the core of the platform, providing much



of the functionality needed to simulate user activity. The platform creates an Expect control process as a substitute for the user in an interactive session. Figure 1 shows an interactive telnet session. Expect connects telnet's standard input (stdin) and standard output (stdout) to the control process. The control process then issues a scripted command sequence to the telnet process, just as if a user were typing the commands. The output of these commands is displayed on the terminal that invokes the script's execution. Thus, the simulation produces the same output as a user.

To see how our platform simulates user activity, consider the following key part of a simple script:

```
# Spawn a telnet process to
# connect to machine alpha.
      connect alpha

# Login as user lee and check
# if login succeeds or not.
      if {![login lee my
      password]} {exit}

# Send normal user commands
# and then logout.
      send_cmd "who"
      send_cmd "ls"
      send_cmd "ps"
      logout
```

This script uses the basic session commands (explained in "Software Platform Commands") to simulate a user connecting to a computer system via telnet, logging in, issuing some shell commands, and then logging out.

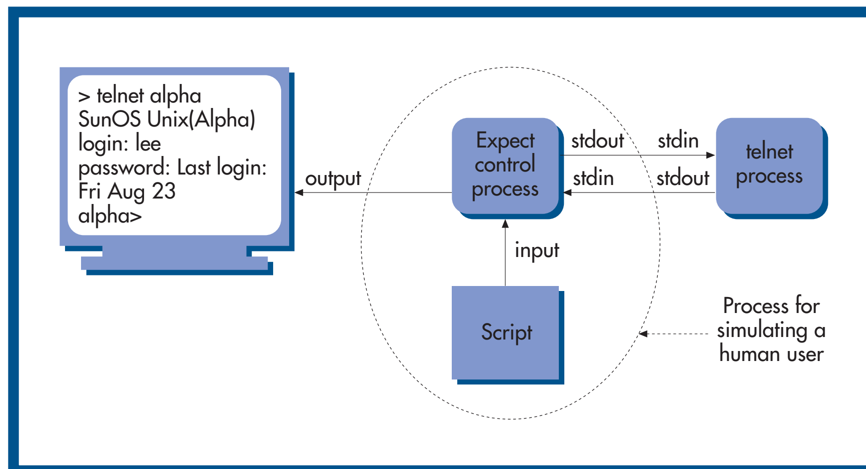


Figure 1. *Simulation of user activity during an interactive telnet session.*

SUPPORT FOR CONCURRENT INTRUSIONS

It is often necessary or desirable to repeat an IDS test. For example, you may repeat a test to determine why the IDS failed the test. In the repeated test, the event sequence in the execution of the test scripts should be the same as in the original test. To meet this condition, we use synchronization techniques for concurrent script sets. Otherwise, there is no guarantee that the overall event sequence will stay the same from test to test, even if the event order within each individual process is fixed. Randomness in system operations (such as variations in the duration of disk I/O events) can cause this nondeterminism in the execution sequence of a concurrent script set, resulting in a test you cannot reproduce.

Our synchronization mechanism allows you to define a fixed order for the important events in a concurrent script set. Consider, for example, a concurrent password-cracking intrusion, in which three intruders collaborate in an attempt to crack passwords on a target machine. Figure 2 lists the specific tasks for each intruder: one intruder copies the cracker program from a remote machine and cleans up after the intrusion; one compiles and runs the cracker program when the program is available on the target machine; one checks the cracker-program output and logs in if a password is cracked. In this concurrent intrusion, several of the intruders' commands must be synchronized (as indicated by the arrows in Figure 2). For example, the uppermost arrow indicates that Intruder 1 must create the "attack" directory before Intruder 2 can change to that directory.

Figure 3 shows the simulation scripts for this intrusion, along with the key elements of our synchronization mechanism. Each of the three simulated intruders has a separate script. The sync command is used to create synchronization points in the scripts. Each sync command takes an argument that specifies an "ID tag" for the synchronization point. Each ID tag is unique within a single script and should be a single character (number or letter) or a short string of characters. The synchronization matrix, shown in the lower portion of the figure, completes the specification of the synchronization. The matrix has a column for each script (process); the matrix entries are the ID tags of the synchronization points shown in the script bubbles. The rows of the matrix specify synchronization events. Each row lists a synchronization point for each process. Each process must wait at that synchronization point until each of the other processes has reached its corresponding synchronization point.

For example, the third row of the matrix indicates that process 1 should stop at its synchronization point with ID tag "3" until process 2 reaches its synchronization point with ID tag "c." The 0 in the

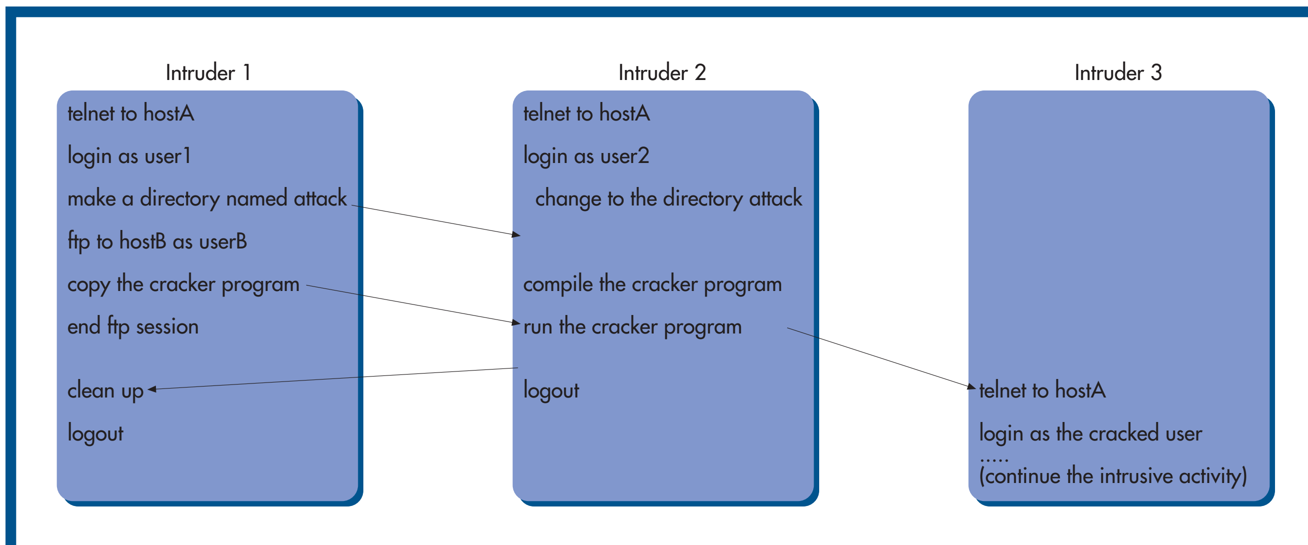


Figure 2. *A concurrent password-cracking intrusion. Arrows indicate the intruder commands that must be synchronized.*

last entry of the third row indicates that process 3 does not participate in this synchronization event (as the matrix shows, process 3 does not participate in any synchronization events). You can specify different synchronization events simply by changing the entries in the synchronization matrix.

During the execution of a concurrent script set, a synchronization server (also shown in Figure 3) monitors and controls

- synchronization of processes at runtime according to the synchronization matrix and
- communication among processes by storing messages and letting processes send and receive data through the server.

Message passing can also be used as a synchronization mechanism. The "sync_data_rcv" command (used in the third script in Figure 3) will cause a process to wait until data is received from a process that sends data with a "sync_data_snd" command (used in the second script in Figure 3).

EXPERIENCE

We conducted several tests on a widely used IDS called Network Security Monitor.[6] NSM monitors all packets transmitted on the host computer's local area network, associating each packet with the corresponding computer-to-computer connection. NSM primarily uses string matching for its analysis, searching for instances of certain strings in the connections' data streams. The strings set is specified by the NSM administrator, who chooses strings that indicate suspicious behavior. NSM assigns a warning value (between 0 and 10) to each connection, based upon the strings it matches and on other considerations such as how often a similar connection has occurred in the previous several days. A higher warning value indicates that a connection's activity is suspicious.

We ran NSM on a Sun workstation connected to the Computer Science LAN segment at UC Davis, and tested its performance with simulated traffic using our software platform. We first performed intrusion identification tests. The tests revealed that NSM's database of intrusion signatures was missing some of the signatures needed to detect the simulated attacks. In NSM, you can solve this problem by simply adding signatures to the database.

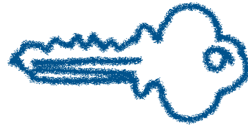
We also performed stress tests that measured the effect of increasing the CPU load on the NSM host computer. We found that as the load increased, NSM missed an increasing number of data bytes in the connections it was monitoring. The stress tests thus demonstrated that stressful conditions can affect an IDS's ability to detect intrusions. We have described both the stress tests and the intrusion identification tests in more detail elsewhere.[2]

Concurrent intrusions. To escape detection by an IDS, intruders might try to distribute their activity over several concurrent sessions. The premise behind this strategy is that the IDS will assign a higher warning value to one very intrusive session than it will to several less intrusive sessions. We conducted some experiments to test this premise. We first created a small set of test scripts using our software platform. Each script simulates an intrusive command sequence. Specifically, the scripts simulate

- transmitting a password file to a remote host,
- cracking a password (via examination of the password file),
- guessing a password using common passwords, and
- exploiting a vulnerability in a system program (loadmodule) to obtain superuser status.

We created corresponding concurrent script sets for each of these activities by distributing the activity over several scripts. Next, we activated NSM. For each simulated intrusion, we ran the sequential script and then the corresponding concurrent script set. We then compared NSM warning values for the sequential scripts with the warning values for the concurrent script sets.

Table 1 shows the results. NSM assigned a warning value to each network connection. Some of the sequential scripts and all of the concurrent script sets initiated more than one network connection, but for clarity's sake we show here only the maximum warning values. For password file transmission, the warning value for the concurrent script set was the same as the warning value for the sequential script. And the warning values were high. A possible explanation is that the sequential script had a very suspicious command or set of commands that could not be divided when the concurrent script set was created. As a result, at least one process



related to the concurrent script set was just as suspicious as the execution of the original sequential script. Password cracking also had the same warning values for sequential and concurrent script sets, but in this case the values were lower. This is likely because NSM was not configured to be sensitive to that particular intrusion.

For the simulations of password guessing and loadmodule attack, the warning value for the concurrent script set was lower than the warning value for the sequential script (dramatically so for the loadmodule attack simulation). In these cases, it was possible to divide the suspicious commands in the sequential script between two or more processes related to the concurrent script set.

Findings. Taken together, our experiments indicate that intruders may be able to reduce their chance of IDS detection by distributing their suspicious activities, although this strategy is not always successful. In future work, we plan to investigate the effects of this strategy on different IDSs. For example, while NSM evaluates each network connection independently, other IDSs evaluate a user's entire history of activity, including the current session and all previous sessions. Against such an IDS, dividing up intrusive activity over several sessions would not be effective, unless each session involved a different user name.

Script for intruder 1
telnet to hostA
login as user1
make a dir named attack
sync
ftp to hostB as userB
copy the cracker program
sync
end ftp session
sync
clean up logout

Script for intruder 2
telnet to hostA
login as user2
sync a
change to the directory attack
sync b
compile the cracker program
run the cracker program
sync c
sync_data_snd 2 result output
logout

Script for intruder 3
sync_data_rcv 3 2 result
telnet to hostA
login as the cracked user
.....
(continue the intrusive

Synchronization server
Message pool

Synchronization matrix

p1	p2	p3
1	a	0
2	b	0
3	c	0

Figure 3. *Execution of the concurrent password-cracking intrusion.*

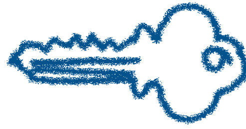


TABLE 1
NSM WARNING VALUES: NETWORK CONNECTIONS

Intrusion description	Maximum Warning Value Sequential	Concurrent
Transmitting password file	7.472	7.472
Cracking a password	3.160	3.160
Guessing a password	8.722	7.785
Exploiting loadmodule flaw	7.472	4.972

Increasing sophistication. Because our primary goal was to develop tools and a methodology for testing, we experimented with a limited set of intruder simulation scripts. However, our software platform can be used to simulate more sophisticated attacks.

For example, consider the well-known TCP sequence-number attack.[7] The abbreviated steps of the attack are as follows. The intruder sends a connection request packet to a server, and records the sequence number in the server's reply. The intruder then sends a second connection request packet to the server, but this time the packet has a forged source address of a computer the server trusts. The intruder does not receive the reply; the server sends the reply to the trusted computer. To complete the connection-opening protocol, the intruder sends another packet that includes the sequence number (plus one) of the server's reply to the trusted computer. The intruder predicts this sequence number based on the sequence number recorded in the first step of the attack. The attacker thereby fools the server into accepting the connection and thus has the power to harm the server data or software.

The first step in creating a simulation of this attack is to create programs (written in C, perhaps) that can send connection request packets and receive replies. You can then develop an Expect script that runs these programs in the appropriate sequence. Our extensions to Expect could be used to create a simulated concurrent attack: One process could make the first connection request and receive the first reply; a second process could make the second connection request and—based on the information received from the first process—construct the packet with the predicted sequence number and send it to the server to complete the connection.

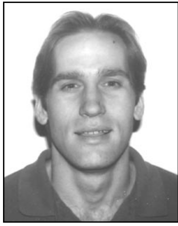
In the future, we may develop additional performance objectives and tests for IDSs based on the work of other groups, such as tests that measure IDS processing speed.[8] We may also extend our work by developing a comprehensive testing facility for IDSs. The most challenging task in that project would be selecting test data. We could then develop user simulation scripts from the test cases. To automate the testing process, we could develop driver scripts and programs to operate the IDS under test, run the simulation scripts, analyze IDS output, and report test results. An independent group at Lincoln Laboratory has started to build such a testing facility.[9] We believe our work can provide both a blueprint, in our testing methodology, and a useful tool, in our software platform, for such efforts. ◆

ACKNOWLEDGMENTS

Don Libes of the US National Institute of Standards and Technology is the developer of Expect, which is the base of our software platform for simulating computer users. We thank Becky Bace, our National Security Agency project monitor, for her guidance and enthusiastic support. This work was funded by the National Security Agency INFOSEC University Research Program under contract number DOD-MDA904-93-C-4084.

REFERENCES

1. B. Mukherjee, L.T. Heberlein, and K.N. Levitt. "Network Intrusion Detection," *IEEE Network*, May 1994, pp. 26-41.
2. N. Puketza, et al., "A Methodology for Testing Intrusion Detection Systems," *IEEE Trans. Software Eng.*, Oct. 1996, pp. 719-729.
3. G.J. Myers, *The Art of Software Testing*, John Wiley & Sons, New York, 1979.
4. D. Libes, *Exploring Expect: A Tcl-based Toolkit for Automating Interactive Programs*, O'Reilly & Assoc., Sebastapol, Calif., 1994.
5. B.C. Smith, L.A. Rowe, and S.C. Yen, *Tcl Distributed Programming*, Tcl-DP Distribution Package, Computer Science Division, University of California, Berkeley, Calif., 1994.
6. L.T. Heberlein et al., "A Network Security Monitor," *Proc. IEEE Symp. Research Security and Privacy*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1990, pp. 296-304.
7. S.M. Bellovin, "Security Problems in the TCP/IP Protocol Suite," *ACM Computer Comm. Review*, Apr. 1989, pp. 32-48.
8. K. Ilgun, R.A. Kemmerer, and P.A. Porras, "State Transition Analysis: A Rule-Based Intrusion Detection Approach," *IEEE Trans. Software Eng.*, Mar. 1995, pp. 181-199.
9. R. Lippmann and H.M. Heggestad, "Lincoln Laboratory Intrusion Detection Research," *Proc. 4th Computer Misuse and Anomaly Detection Workshop*, National Security Agency, Ft. Meade, Maryland, to appear.



Nicholas Puketza is working on his doctorate in the Computer Science Department at the University of California, Davis. From 1988 to 1992 he was a software engineer for Motion Control Engineering. His research interests include intrusion detection, software testing, and network security.

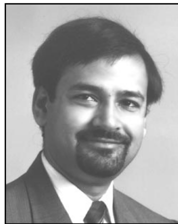
Puketza received a BS in electrical engineering from Stanford University in 1988.

Mandy Chung is a software development engineer at Hewlett-Packard in Cupertino, California. Her research interests include concurrent programming languages, parallel and distributed computing, and computer security.

Chung received a BS in computer science from the University of Hong Kong in 1990 and an MS in computer science from the University of California, Davis, in 1996.

Ronald A. Olsson is a professor and vice-chair of computer science at the University of California, Davis. His research interests include computer security, concurrent programming, programming languages, verification, systems software, and operating systems. He is the co-author (with Greg Andrews) of *The SR Programming Language: Concurrency in Practice* (Benjamin-Cummings, 1993).

Olsson received a BA in mathematics and computer science and an MA in mathematics from the State University of New York, Potsdam; an MS in computer science from Cornell University; and a PhD in computer science from the University of Arizona. He is a member of the ACM.



Biswanath Mukherjee is professor and chair of computer science at the University of California, Davis. His research interests include lightwave networks and network security. He is on the editorial boards of IEEE/ACM *Transactions on Networking* and the *Journal of High-Speed Networks*.

Mukherjee received a BTech from the Indian Institute of Technology, Kharagpur, in 1980 and a PhD from the University of Washington, Seattle, in 1987. He is a member of the IEEE.

Address questions about this article to Puketza at Department of Computer Science, University of California, One Shields Ave., Davis, CA 95616; puketza@cs.ucdavis.edu.

Classified Advertising

McMASTER UNIVERSITY
Faculty of Engineering
Faculty Positions in Software Design

McMaster University's Faculty of Engineering is forming a new department that will offer a new undergraduate programme in Software Engineering as well as the Faculty of Science's existing programme in Computer Science. Incorporating new faculty members, some members of the Faculty of Engineering, and the members of the current Department of Computer Science & Systems, the new Department's programmes will complement an established programme in Computer Engineering that is offered by the Department of Electrical and Computer Engineering. The new programme is designed with the philosophy that Software Systems Engineering is a new speciality within engineering and it is our intention to have it accredited by the Canadian Engineering Accreditation Board (CEAB).

We have openings for one senior tenured and several junior tenure-track faculty members interested in any aspect of Computer System Design, commencing in January or July, 1998. Applicants should have a Ph.D. in either computer engineering, electrical engineering, computer science or mathematics and have experience either in industry or teaching appropriate to the level of the appointment. The successful applicants must be able to teach both graduate and undergraduate courses in software design, real-time systems, information systems, performance prediction, and other aspects of computer system design. The successful candidate for the senior appointment will be expected to play a major leadership role in the development of the new department. All applicants must have a strong and demonstrated commitment to research in a university environment and will have normal faculty administrative and committee responsibilities.

Ability to be registered as a Professional Engineer in the Province of Ontario or become registered within three years of his/her appointment will be considered an advantage.

McMaster University has an employment equity programme that encourages applications from all qualified candidates, including women, aboriginal people, persons with disabilities and visible minorities. In accordance with Canadian Immigration requirements, priority will be given to Canadian citizens or permanent residents of Canada.

These positions are subject to final budgetary approval; salary is commensurate with experience and qualifications. Applications, including a curriculum vitae, a statement detailing research and teaching interests, and the names of five referees should be sent to: Dr. M.Shoukri, Dean, Faculty of Engineering, McMaster University, 1280 Main St. West, Hamilton, ON Canada L8S 4L7.

The University will begin evaluating applications on September 30, 1997, but will continue to accept publications until all positions are filled.