A Software Platform for Testing Intrusion Detection Systems

Nicholas Puketza, Mandy Chung, Ronald A. Olsson, and Biswanath Mukherjee

Correspondence Author: Ronald A. Olsson

Department of Computer Science

University of California, Davis, CA 95616

olsson@cs.ucdavis.edu

Tel: (916) 752-7017

Fax: (916) 752-4767

May 12, 1997

Abstract

Intrusion detection is a practical approach to enhancing the security of computer systems. Intrusion Detection Systems (IDSs) attempt to identify unauthorized use, misuse, and abuse of computer systems. The steady growth in the development and use of IDSs has created a demand for the development of tools and methods for testing IDSs. To address this demand, we have developed a software platform as well as a testing methodology to comprehensively test IDSs. Our software platform supports the simulation of computer users, including intruders as well as normal users. The platform, based on the Expect programming language, includes mechanisms that facilitate the simulation of both sequential and concurrent intrusions, as well as a record-and-replay feature. The platform supports our testing methodology to provide a systematic approach for testing IDSs, which should be particularly useful for the users and developers of IDSs. Our platform can also be employed in testing other applications that require the simulation of a user's interactive behavior.

Keywords: Computer Security, Intrusion Detection, Testing, Software Platform, Simulation, Concurrency, Synchronization.

Introduction

Over the last decade, several highly-publicized attacks on computers have forced the computer community to pay more attention to computer security. The infamous Internet Worm attack in 1988 "served as a major wake-up call for many UNIX users and vendors" [4]. Cliff Stoll's account of the activities of the so-called "Wily Hacker" [16] had a similar effect. Several more recent attacks, such as the flurry of attacks based on TCP sequence-number prediction (reported by the CERT Coordination Center in 1995 [2]), have provided further evidence that computer security cannot be ignored. One response to the need to improve security has been the development of a new approach to security called intrusion detection.

Intrusion detection is concerned with the identification of unauthorized use, misuse, or abuse of computers through monitoring of system activities. Tools implemented for intrusion detection are referred to as intrusion detection systems (IDSs). Sidebar 1 describes two approaches to intrusion detection — misuse detection and anomaly detection — and gives some examples of IDSs.

Intrusion detection is a practical approach to enhancing the security of a computer system. Traditional approaches to computer security, such as efforts to develop strong access-control mechanisms, have focused on completely *preventing* unauthorized computer activities. However, for many computer systems it is unlikely that all vulnerabilities can be eliminated, due to the complexity of computer system design, the possibility of configuration and administration errors, and the potential for authorized users to abuse their privileges [12]. When a system's vulnerabilities are exploited, intrusion detection can provide a second level of defense. Also, the intrusion detection approach is attractive because it does not require the expensive replacement of existing computer systems. (Similar arguments for intrusion detection appear in reference [3].) Finally, there is published evidence that an IDS can improve the process of identifying real intrusions. One report [11] indicates that, by using an IDS, TRW was able to speed up its investigations of intrusions. A second report [12] indicates that researchers using an IDS were able to identify many intrusions and intrusion attempts which were not detected by the standard methods of the system administrators at the site.

As the use of IDSs increases, the need for tools and methodologies for testing and evaluating IDSs is also growing. During the course of testing an IDS, computer user activities (both suspicious and normal) need to be simulated so that the IDS users can test the IDS's ability to detect suspicious activity. To address this problem, we have developed a software platform that facilitates the creation of intrusion simulations and testing of IDSs.

Intrusions can take one of two forms: sequential and 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 amongst themselves. Since window interfaces are ubiquitous, such concurrent intrusions are very likely to occur, so an IDS should be able to detect them.

Our software platform allows us to create scripts that simulate both sequential and concurrent intrusions. Normal user activity can also be simulated. Another part of our platform is a record-and-replay feature that helps us to create scripts in a quick and easy manner. We created our software platform to support a methodology that we developed for testing various aspects of an IDS [14]. The methodology includes techniques and approaches that we adapted from the general field of software testing. 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 approaches to testing. The software platform can also be employed in testing other applications that require simulation of computer users, especially multiple cooperating users.

In a previous paper [14], we focused on our methodology for testing IDSs. In this paper, we emphasize the software platform, especially the features for simulating concurrent intrusions. Below, we first provide a summary of our methodology for testing IDSs. Next, we describe our software platform and its basic functionality and then present some example scripts. We also describe how our software platform supports the simulation of concurrent intrusions and how our record-and-replay feature helps us in simulating one or more users' activities. We then present our experience with and results from using our software platform in testing a widely-used IDS. We conclude with potential future work.

In this article, we shall employ the following specialized terms. A *concurrent script set* is a set of scripts which are executed simultaneously in order to simulate a concurrent intrusion. A *process* refers to the execution of a single script that simulates a single user session.

Summary of Testing Methodology

To motivate the need for the software platform, we first present a summary of our methodology for testing IDSs (please see reference [14] for a complete discussion of the methodology). We will start with some background information about the structure of an IDS.

Misuse Detection Systems

Sidebar 1 describes the two major approaches to intrusion detection: *anomaly detection* and *misuse detection*. Our methodology is designed to test the latter approach. In general, a misuse detection system consists of two key components. The first is a database of intrusion *signatures*, which are encapsulations of the identifying characteristics of specific intrusion techniques. The second is a pattern-matching mechanism which searches for the intrusion signatures in the records of user activities (e.g., audit trails).

IDS Performance Objectives

As the basis for the testing methodology, we identified the following set of general IDS performance objectives:

- *Broad Detection Range:* for each intrusion in a broad range of known intrusions, the IDS should be able to distinguish the intrusion from normal behavior;
- *Economy in Resource Usage:* the IDS should function without monopolizing system resources such as main memory, CPU time, and disk space[1]; and

[1]Of course, different sites will tolerate different levels of IDS resource usage.

- *Resilience to Stress:* the IDS should still function correctly under stressful conditions in the system, such as a very high level of computing activity.

Test-Case Selection

The first step in the methodology is to select test cases for the testing experiments. In our approach, a test case is a simulated user session. While some of the tests require simulated "normal" sessions, most of the test cases are simulated intrusions. A key problem is to select *which* intrusions to simulate. The site that will use the IDS should select test cases based on its computer security policy, which defines what is and what is not an intrusion. Some of the test cases can be derived from published descriptions of well-known attacks, and other site-specific test cases can be developed based on the security policy and private information at the site. Ideally, the group that selects test cases should be different than the group that creates the IDS intrusion database. In previous work [14], we discuss strategies for test-case selection, based on various classifications of intrusions. After the test cases are selected, scripts can be developed to simulate the intrusive behavior in the test cases.

Testing Experiments

The rest of the methodology consists of using the simulation scripts in a variety of testing experiments. We have developed a set of detailed testing procedures, several of which are adaptations of the "higher-order" testing methods described by Myers [13]. Most of the procedures include the same basic steps. A set of test scripts is created or selected. The desired conditions (e.g., the level of "background" computer activity) in the computing environment are established. Then, the IDS is activated, and each of the test scripts is executed. Finally, the IDS's output is analyzed.

We have divided the test procedures into three categories, which correspond directly to the three performance objectives described earlier. *Intrusion identification* tests measure the ability of the IDS to distinguish intrusions from normal behavior. *Resource usage* tests measure how much system resources are consumed by the IDS. The results of these tests can be used, for example, to decide if it is practical to run a particular IDS in a particular computing environment. *Stress* tests check if the IDS can be affected by "stressful" conditions in the computing environment. An intrusion that the IDS would ordinarily detect might go undetected under such conditions. As an example, one of the stress tests evaluates the IDS's performance when there is a heavy load on the computer on which the IDS is running.

Potential Causes of Detection Failures

If the IDS fails to detect a simulated intrusion, the next step in the methodology is to identify the source of the failure. Recall that an IDS that performs misuse detection has two main components: an intrusion database and a pattern-matching mechanism. Also, the IDS depends on another computer system component: the component that logs system activity (e.g., the audit programs).

Thus, an IDS may fail to detect an intrusion during testing because:

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

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

The testing process should continue until all detection failures are eliminated or at least explained, so that other security mechanisms can be used to cover the weaknesses in the IDS.

Limitations of the Methodology

Even thorough testing may not expose all potential weaknesses in an IDS. The primary limitation of our IDS testing methodology is that it tests the IDS against *known* attacks. In many environments, detecting known attacks is valuable, even if the IDS does not detect *any* new attacks. At many sites, known attacks probably occur more frequently than new attacks, because the details of known attacks are often distributed widely (e.g., via scripts that appear in newsgroup postings), while the number of people who know the details of a truly new attack is probably small. However, sites that value security highly are also interested in detecting *new* attacks. If the test-case selection method is thorough, then the IDS should be able to detect new attacks well, because it has been tested against other attacks that are somehow similar to the new attacks. Unfortunately, evaluating the thoroughness of the test-case selection method is an open problem.

IDSs have other potential vulnerabilities. Many IDSs include programs that run on the computers that they monitor. If an intruder can take control of those computers, the intruder can manipulate the IDS programs themselves. Also, if the intruder has knowledge of the database of intrusion signatures in the IDS, then the intruder can attempt attacks that are not represented in the database. Thus, the privacy of the intrusion database should be protected.

Overview of Our Software Platform

Our software platform is based on the *Expect* package [9] and the Tcl Distributed Programming (*Tcl-DP*) package [15]. *Expect* is built on top of *Tcl* (Tool Command Language) and provides additional commands for controlling interactive programs. It is particularly useful for simulating a human computer user. *Tcl-DP* is an extension to *Tcl/Tk* and provides a suite of commands for creating client-server systems.

Our software platform extends the *Expect* language with mechanisms that facilitate intrusion simulation, including basic session commands for simulating common user commands (e.g., login, telnet), mechanisms for supporting concurrent intrusions (e.g., synchronization, communication), and a record-and-replay feature. Sidebar 2 shows some of the basic session commands and other mechanisms provided by our platform. Our platform allows you to create scripts that can include parameters with variable values to facilitate repetitive, sustained testing.

Basic Session Commands. Several commonly-used commands (especially those that require a user's interaction), such as *telnet*, *login*, and *ftp*, are provided by the platform as basic session commands. With these commands you can simulate many of the basic activities of an intruder.

Synchronization. The synchronization mechanism provides you a means to establish a fixed order of execution for important events even though these events may belong to separate scripts. You can specify the required synchronization in the concurrent scripts to enforce the dependency constraints among events, and hence their execution order, in different scripts.

Communication. To accurately simulate a group of users working together, concurrent processes must be able to communicate with one another. You can specify *send* and *receive* commands in concurrent scripts so that the scripts can exchange data with one another.

Record-and-Replay. Our platform provides a "record-and-replay" feature to facilitate the creation of 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.

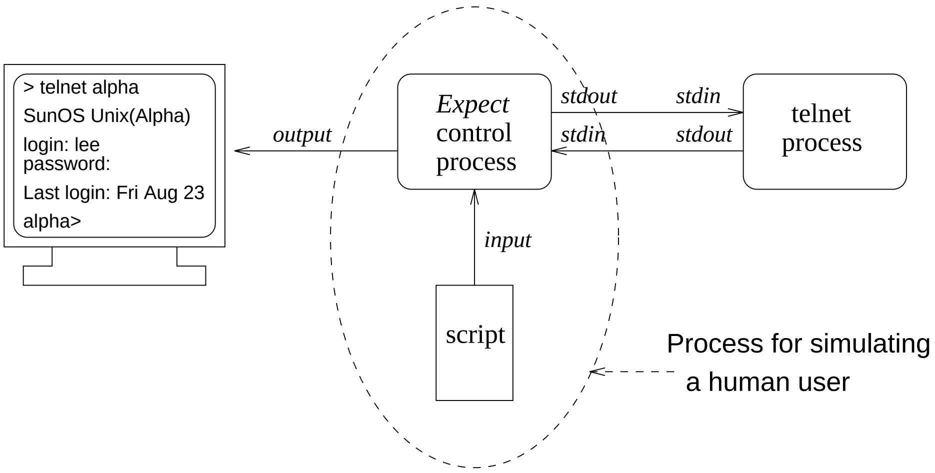

Figure 1: *Simulation of a human user's activity.*

Figure 1 depicts how our software platform simulates a human user's activity. *Expect* and *Tcl* provide much of the functionality needed for simulating a human user, and they form the core of our platform. Our software platform operates by creating an *Expect* control process as a substitute for the human user in an interactive session. The interactive session in Figure 1 is the telnet session. *Expect* connects the standard input (*stdin*) and the standard output (*stdout*) of the telnet process to the control process. The control process issues a sequence of commands from a script to the telnet process, just as if a human user were typing in the commands. The output of these commands is displayed on the terminal that invokes the execution of the script. Thus, the simulation produces the same output as a human user that issues the sequence of commands interactively.

To see how our platform can be used to easily simulate a user's basic activities, 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 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 (see Sidebar 2) to simulate a normal user who connects to a computer system via telnet, logs in, issues some shell commands, and then logs out.

Support for Concurrent Intrusions

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

Our synchronization mechanism allows you to define a fixed order for the important events in a concurrent script set. We will explain the mechanism by example. Consider 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 during the intrusion. One intruder is responsible for copying the cracker program from a remote machine and for cleaning up after the intrusion is completed. The second intruder is responsible for compiling and running the cracker program when the program is available on the target machine. The third intruder is responsible for checking the output generated by the cracker program and for logging in to the target system if a password is cracked.

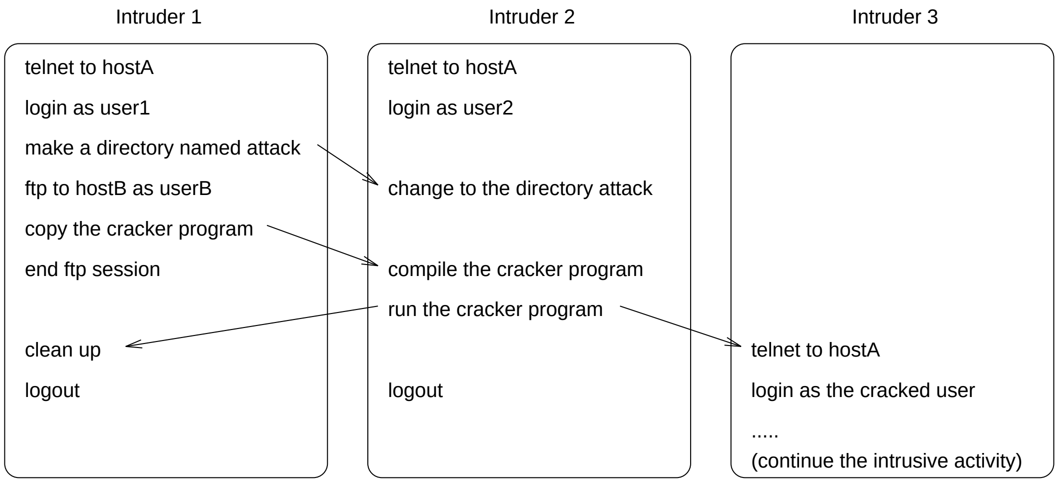

Figure 2: *Concurrent password-cracking intrusion.*

In this concurrent intrusion, several commands issued by the intruders have to 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 displays the simulation scripts for this concurrent 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 synchronization point has an "id tag" that is unique within a single script. The id tag is the argument to the sync command.

The *synchronization matrix* 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. 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 synchronization matrix in Figure 3 indicates that process 1 should stop at its synchronization point 3 until process 2 reaches its synchronization point c. The 0 in the last entry of the third row indicates that process 3 does not participate in this synchronization event. (Actually, since column 3 contains all 0's, process 3 does not participate in *any* synchronization events.) A useful feature of our synchronization mechanism is the ability to 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) is responsible for monitoring and controlling the synchronization and communication among the different concurrent processes. The server controls the synchronization among concurrent processes at runtime according to the synchronization matrix. The synchronization server also acts as a central storage agent for messages, and processes can exchange information by sending and receiving

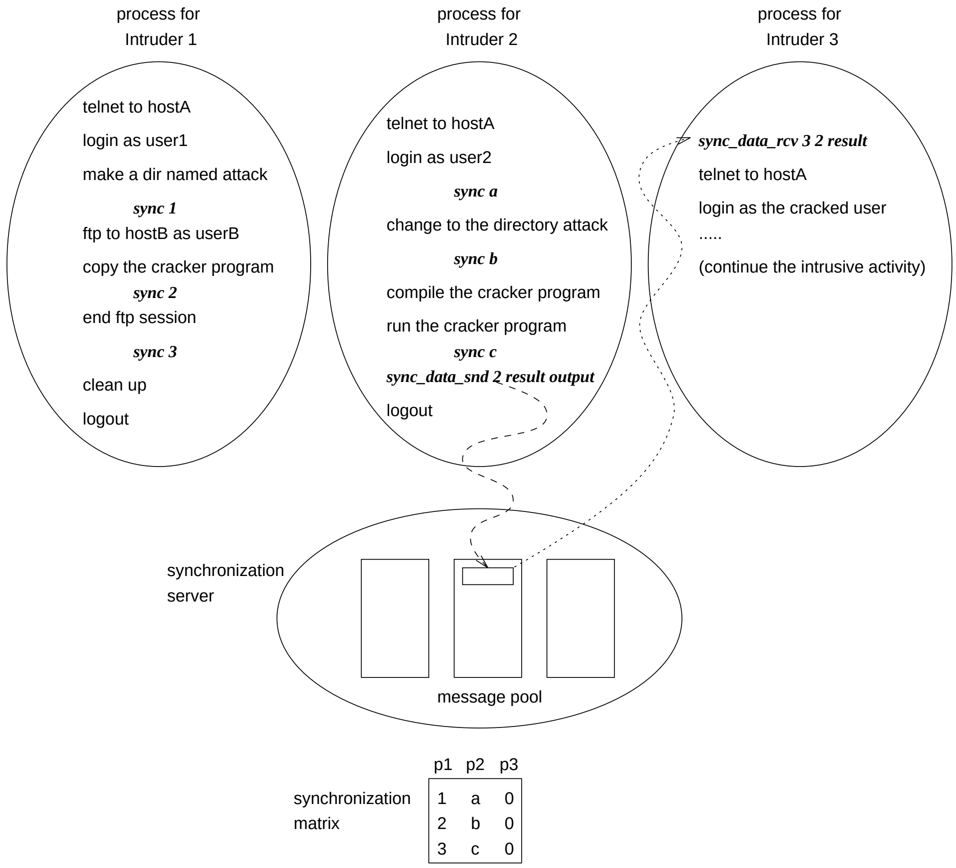

Figure 3: *Execution of the concurrent password-cracking intrusion.*

data through the server. Note that 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_send*" command (used in the second script in Figure 3).

Experience

We conducted several tests on a widely-used IDS called Network Security Monitor (NSM) [6]. NSM monitors all of the packets that are transmitted on the local area network (LAN) to which the computer running NSM is connected. NSM can associate each such packet with the corresponding computer-to-computer connection. The primary analysis that NSM performs is string-matching; it searches for instances of certain strings in the data streams of the connections. The set of strings is specified in a file by the administrator who runs NSM. The administrator should choose strings that indicate suspicious behavior. NSM assigns a warning value (between 0 and 10) to each connection based on which strings are matched, and on other considerations such as how often a similar connection has occurred in the previous several days. A higher warning value indicates that the activity associated with that connection is more 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 performed intrusion identification tests as described in the "Testing Experiments" section. The tests revealed that NSM's database of intrusion signatures was missing some of the signatures needed to detect the simulated attacks. For NSM, it is simple to solve this problem by 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 that it was monitoring. So the stress tests demonstrated that stressful conditions can affect the ability of an IDS to detect intrusions. Both the stress tests and the intrusion identification tests are described in more detail in reference [14].

We conducted a third set of tests on NSM, tests that involved simulated concurrent intrusions. We will describe these tests in detail now. To escape detection by an IDS, we hypothesize that intruders might try to distribute their intrusive 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 is designed to simulate a specific intrusive command sequence. Specifically, the scripts simulate the following behaviors:

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

Next, we created corresponding *concurrent* script sets for each of the same intrusive activities by distributing the intrusive activity over several scripts.

We activated NSM and for each intrusion simulation, we ran the sequential script and then ran the corresponding concurrent script set. We then compared NSM warning values for the sequential scripts with the warning values for the concurrent script sets. The results are displayed in Figure 4. NSM assigns a warning value to each network connection. Some of the sequential scripts and all of concurrent script sets initiate more than one network connection, but for clarity the figure shows only the maximum warning value for all network connections associated with each script or script set.

For the simulation of the transmission of the password file, Figure 4 shows that the warning value for the concurrent script set is the same as the warning value for the sequential script, and the warning values are high. A possible explanation for this is that the sequential script contains a very suspicious command or set of commands which cannot be divided when the concurrent script set is created. As a result, at least one of the processes related to the concurrent script set is just as suspicious as the execution of the original sequential script. The warning values for the sequential and concurrent versions of the password-cracking simulation are also equal, but in this case the values are lower. A likely explanation is that NSM was not configured to be sensitive to that particular intrusion. So, neither the sequential script nor the concurrent script set produced activity that appeared suspicious to NSM.

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

Taken together, our experiments indicate that intruders may be able to reduce their chance of detection by an IDS 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 the NSM evaluates each network connection independently, some 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.

Intrusion Description	Max Warning Value	
	Sequential	Concurrent
Transmitting passwd file	7.472	7.472
Password-cracking	3.160	3.160
Password-guessing	8.722	7.785
Exploiting loadmodule flaw	7.472	4.972

Figure 4: *NSM concurrent script sets experimental results.*

Simulating More Sophisticated Attacks

As our primary goal was to develop tools and a methodology for testing, we developed only a limited set of intruder simulation scripts for our testing experiments. However, our software platform can be used to simulate more sophisticated attacks.

For example, consider the well-known TCP sequence-number attack [1]. Briefly, the steps of the attack are as follows. The intruder sends a connection request packet to a server, and receives a reply. Then, the intruder sends a second connection request packet to the server, but this time the packet has a forged source address. The address is for a computer that is trusted by the server. The intruder does not receive the reply; the server sends the reply to the trusted computer. However, the intruder can predict the sequence number in the reply, based on the sequence number in the reply to the first connection request. Then, the intruder can send one more packet to establish the connection, and fool the server into believing that it is communicating with a trusted computer.

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. Then, an *Expect* script can be developed that runs these programs in the appropriate sequence. Additionally, 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 final packet and send it to the server to complete the connection.

Future Work

We could develop additional performance objectives and tests for IDSs based on the related work of other groups. For example, tests that measure the *processing speed* of an IDS are described in [7]. We could also extend our work by developing a comprehensive testing facility for IDSs. The most challenging task in that project would be the selection of test data (see the "Test-Case Selection" sub-section). 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, to run the simulation scripts, to analyze the output of the IDS, and to report test results. An independent group at Lincoln Laboratory has started to build such a testing facility [10]. We believe that our work can provide both a blueprint (the testing methodology) and a useful tool (the software platform) for such efforts.

Acknowledgements

Don Libes of NIST is the developer of *Expect*, which is the base of our software platform for simulating computer users. We thank Becky Bace, our NSA project monitor, for her guidance and enthusiastic support of this project. This work has been funded by the National Security Agency INFOSEC University Research Program under Contract No. DOD-MDA904-93-C-4084.

Sidebar 1 - Intrusion Detection Systems

An intrusion detection system (IDS) is a system that attempts to identify *intrusion* attempts by both unauthorized users (outsiders) as well as by authorized users who abuse their privileges (insiders) [12]. Generally, intrusions can cause loss or alteration of data, loss of money, denial of service to legitimate users, loss of trust in the computer/network system, and loss of public confidence in the organization that is the victim of an intrusion [5]. The following scenarios are some examples of intrusions:

- an employee browses through his/her boss' employee reviews;

- a system administrator modifies system files so as 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 router tables in a network to upset the delivery of messages; and
- an intruder installs a "snooping program" on a target computer to examine sensitive data (such as user passwords) contained in network traffic.

Interest in the research and development of IDSs has been growing over the last several years. Reference [12] describes several IDSs, including National Security Agency's MIDAS, AT&T's ComputerWatch, SRI International's Intrusion-Detection Expert System (IDES), Los Alamos National Laboratory's Wisdom and Sense, UC Davis' Network Security Monitor (NSM), and the Distributed Intrusion Detection System (DIDS) developed jointly by UC Davis, Lawrence Livermore National Laboratory, Haystack Laboratory, and the US Air Force.

The two approaches to intrusion detection used in most IDSs are *anomaly detection* and *misuse detection*. Anomaly detection is based on the premise that the behavior of an intruder will be noticeably different from that of a normal user [3]. An intruder to a system (possibly masquerading as a legitimate user) is very likely to exhibit a pattern of behavior different from the normal user. The objective of anomaly detection is to establish "profiles" of normal user activities such as typical user login time, CPU usage, favorite editor, disk usage, and login session length. The IDS can then use 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" (i.e., the deviation is 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 the misuse detection approach, the goal of the IDS is to recognize "specific, precisely-representable techniques of computer system abuse" [8]. The IDS detects intrusions by searching records of user activity for intrusion "signatures", which are encapsulations of the identifying characteristics of specific intrusion techniques.

The IDS detects intrusions by searching for any occurrence of the collected suspicious behavior in the record of user activities.

Anomaly detection provides a means to detect masqueraders or legitimate users abusing their privileges without requiring knowledge of security flaws in the target system [3]. Misuse detection, on the other hand, can identify intrusive behavior, even when the behavior appears to conform with established patterns of use (i.e., the established user profiles). For this reason, several IDSs employ both an anomaly detection component and a misuse detection component.

Sidebar 2 - Software Platform Commands

Our software platform provides four groups of commands, described in the section "Overview of Our Software Platform." Below, we describe some of the key commands in each of the four groups.

Basic Session Commands

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

Synchronization Commands

- **sync_make_server** — creates a synchronization server.
- **sync_connect_server** *host port#* — establishes a connection from the client to the server where *host* is the machine on which the server is running and *port#* is the port number for the connection to the server.
- **sync_add_client** — adds a process as a client of the server. This command must be executed after the execution of the process is initiated (after **exec** *script_name arguments*).
- **sync_server** — starts the synchronization server for monitoring and receiving signals from clients.
- **sync** *process_id sync_point* — establishes a synchronization point *sync_point* in a process.
- **thread_exit** *process_id* — terminates the execution of the process when execution of all concurrent processes are completed.

Communication Commands

- **sync_data_snd** *process_id tag data* — sends *data* from process *process_id* with *tag* as label.
- **sync_data_rcv** *process_id src_process tag* — receives data from *src_process* with label *tag*.

Record-and-Replay Commands

- **record** *filename* — starts recording a sequence of commands into the file *filename*.
- **replay** *script1, script2, ...* — replays the activity recorded in the specified files.

References

[1] S. M. Bellovin. Security Problems in the TCP/IP Protocol Suite. *ACM Computer Communication Review*, 19(2):32–48, April 1989.

[2] CERT Coordination Center. "IP Spoofing Attacks and Hijacked Terminal Connections". CERT Advisory CA-95:01, January 1995.

[3] D. E. Denning. "An Intrusion Detection Model". *IEEE Transactions on Software Engineering*, SE-13:222–232, February 1987.

[4] Simson Garfinkel and E. H. Spafford. *Practical UNIX and Internet Security, Second Edition.* O'Reilly & Associates, Inc., 1996.

[5] L. D. Gary. "Crime on the Internet". Presentation at *17th National Computer Security Conference*, Baltimore, MD, October 1994.

[6] L. T. Heberlein, G. Dias, K. Levitt, B. Mukherjee, J. Wood, and D. Wolber. "A Network Security Monitor". In *Proc. IEEE Symposium on Research in Security and Privacy*, pages 296–304, Oakland, CA, May 1990.

[7] K. Ilgun, R. A. Kemmerer, and P. A. Porras. "State Transition Analysis: A Rule-Based Intrusion Detection Approach". *IEEE Transactions on Software Engineering*, 21(3):181–199, March 1995.

[8] S. Kumar and E. H. Spafford. "A Software Architecture to Support Misuse Intrusion Detection". In *Proc. 18th National Information Systems Security Conference*, pages 194–204, Baltimore, MD, October 1995.

[9] D. Libes. *Exploring Expect: A Tcl-based Toolkit for Automating Interactive Programs.* O'Reilly & Associates, Inc., 1994.

[10] R. Lippman. Presentation at *4th Computer Misuse and Anomaly Detection Workshop*, Monterey, CA, November 1996.

[11] T. F. Lunt. "Automated Audit Trail Analysis and Intrusion Detection: A Survey". In *Proc. 11th National Computer Security Conference*, pages 65–73, October 1988.

[12] B. Mukherjee, L. T. Heberlein, and K. N. Levitt. "Network Intrusion Detection". *IEEE Network*, 8(3):26–41, May 1994.

[13] G. J. Myers. *The Art of Software Testing.* John Wiley & Sons, 1979.

[14] N. Puketza, K. Zhang, R.A. Olsson, B. Mukherjee, and M. Chung. "A Methodology for Testing Intrusion Detection Systems". *IEEE Transactions on Software Engineering*, 22(10):719–729, October 1996.

[15] B. C. Smith, L. A. Rowe, and S. C. Yen. *Tcl Distributed Programming.* Tcl-DP Distribution Package, Computer Science Division, University of California, Berkeley, 1994.

[16] C. Stoll. "Stalking the Wily Hacker". *Communications of the ACM*, 31(5):484—497, May 1988.