Computer Science Master's Project

Host-Based Intrusion Detection Using User Signatures

Author:	Seth Freeman
Degree:	Master of Computer Science
Submit Date:	May 2002

TABLE OF CONTENTS

ABSTRACT

Developing signatures of users of a computer system is a useful tool for protecting against potential intrusions. Many intrusions occur when an attacker gains unauthorized access to a valid user's account, then performs disruptive behavior while masquerading as a valid user. By developing signatures of each user of a system, we create a model which future user behavior can be compared against.

The signature of each user of a system is stored in a probabilistic finite automata (PFA). The PFA is essentially a representation of all the command traces the user has typed in. These command traces are extracted from the log files of a computer, and are determined by the timings between the commands. The probabilities in the automata correspond to the probability that a given command will follow another command in a given command trace.

Once we have a PFA for a given user, we can compare future command traces the user types against the automata. Each time we test a command trace against the PFA, a set of probabilities will be returned which corresponds to how well the trace matches the signature. Each and every time a low probability is returned, we detect anomalous behavior, which may or may not correspond to an actual intrusion.

1. Introduction

1.1 Host-Based Intrusion Detection

Computer security has become an area of utmost importance as the number of people who use computers continues to increase. As more and more computer systems and networks are setup, there are inevitably bugs in these systems which attackers attempt to exploit. Intrusion detection systems (IDS) are designed to monitor a computer or a network to detect and prevent against invalid activity, often using distributed data collection in the style of DOORS [1,2,3]. IDS can monitor users [4], applications, networks [5], or combinations of the three, in order to detect well-known and unknown attacks.

The two main methodologies to intrusion detection are anomaly detection and misuse detection. Anomaly detection is the methodology by which a model of normal system or user behavior is created, than any behavior deviating from the model is anomalous. The major advantage of this approach is that unseen or new attacks can be identified, because their pattern will be a deviation from the model of expected behavior.

Misuse detection on the other hand, attempts to model well-known attacks. Then any behavior that matches the model is recognized as an attack. The major advantage of misuse detection is that well known attacks can easily be identified, so the system can be protected in time. The disadvantage however, is that no new attacks will be identified.

The two main approaches to intrusion detection are host-based intrusion detection, and network based intrusion detection. Host-based intrusion detection attempts to detect against attacks on a particular machine. This is typically done through analysis of a computer’s log files. Network-based intrusion detection attempts to detect against attacks on a network. This is typically done through analysis of network traffic.

1.2 Methodology

The tool we created uses a host-based anomaly detection scheme to identify invalid user behavior. We first generate a signature of normal behavior for each user of a computer system. We make the assumption that each user has a sequence of commands that they frequently type in. They might search for a given directory, open a text editor, check their mail, compile a program, etc. By having a signature of all the frequent (and infrequent) command traces a user types, we can compare future command traces the user types against the signature. Since we are storing the actual command traces, we not only have a representation of the frequent commands the user types, we also have a representation of the orderings between commands.

We represent the signature as a probabilistic finite automata (PFA). We use a finite automata because this allows us to represent the signature such a way that each command is a state, and transitions between states depict how one command follows another. Each transition in the automata has a probability corresponding to the probability that the given state will be reached from the previous state. Testing future traces against the PFA is a simple process of stepping through the automata, and returning the multiplied probability, and average probability, of reaching each state along the automata.

Anomalous behavior is defined as any behavior that deviates from the model. Thus the anomalies this tool detects may or may not correspond to an actual intrusion. In many instances, the user may simply be trying out a new set of commands, or their behavior is different due to fatigue or stress. In any instance, the tool described will detect this behavior and raise an alarm.

1.3 Previous Work

There have been many previous intrusion detection systems that use the anomaly detection scheme. NNID (Neural Network Intrusion Detection) uses neural networks to predict the next command a user will enter based on previous commands [6]. Haystack, a combined anomaly detection/misuse detection IDS models individual users as well as groups of users. It assigns initial profiles to new users, and updates the profiles once a pattern of actual behavior is recognized [7]. GrIDS (Graph-Based Intrusion Detection System), uses a graphical representation to monitor the activity of not just a user, or a system, but an entire network [8].

ImSafe, a tool that has its roots in anomaly detection, monitors the system call traces produced by specific applications and tries to predict the next system call as accurately as possible [9]. First, ImSafe must go through a learning phase to construct a profile of the application to be monitored. Then, that profile is used during the detection process. The approach outlined in this project is similar to that of ImSafe except that user behavior is modeled instead of application behavior. Next, EMERALD eXpert-BSM, a real-time forward-reasoning expert system, uses a knowledge base to detect multiple forms of system misuse [10]. The forward-reasoning architecture helps eXpert-BSM detect intrusive behavior across multiple system event orderings while also accounting for specific pre- and post-conditions of those sequences.

2. Creating User Signatures

2.1 Extracting User Commands from Log Files

User signatures are created from the log files generated by the Basic Security Module (BSM) of the Solaris operating system [11]. The BSM is a kernel module that logs all events that occur on a given machine. Each of these events is actually a system call, and for each system call a record is generated, consisting of tokens corresponding to the type of event. In order to extract commands a given user typed, first all system calls generated by the user are extracted from the log file. This is done through the use of the Unix utility commands, auditreduce and praudit. The auditreduce command is used for audit management and record selection, while praudit is used to convert binary log files in to ascii format. At the command prompt, this is done as follows:

```
auditreduce -u  [username] [log file in binary form]  |  praudit – l
```

The auditreduce command takes the log file specified and extracts all system calls generated from the given username. The output of this is piped to the praudit command, which will then write each record (system call) to stdoutput one record per line. To save the output in a file, a script is started and ended before and after the auditreduce command. Below is an example of what one system call returned from praudit looks like:

```
header,86,2,login – telnet,,Tue 07 Mar 2000 10:14:08 AM EST, + 356353537 msec, subject,2051,2051,rjm,2051,rjm,2746,2746,24 0 172.16.113.105,text,successful login, return, success,0,trailer,88
```

Once all the system calls are extracted for a given user, the next step is to identify only the system calls that correspond to actual commands the user entered. To do this, a filter is used to identify execve system calls, because each execve system call corresponds to an actual command the user entered at a prompt. The filter also extracts a small set of system calls that are not execve, namely login and chdir. During the process of extracting the user commands, the filter also extracts the timestamp, and writes both the user command and timestamp to another file.

This process is repeated for each of the log files we wish to train on. It is essential that the log files we build our signatures on are attack-free, so that each user's signature is valid, thus invalid behavior will never match the signature. Given the listing of user commands and the timestamp of each command we are ready to build our probabilistic finite automata.

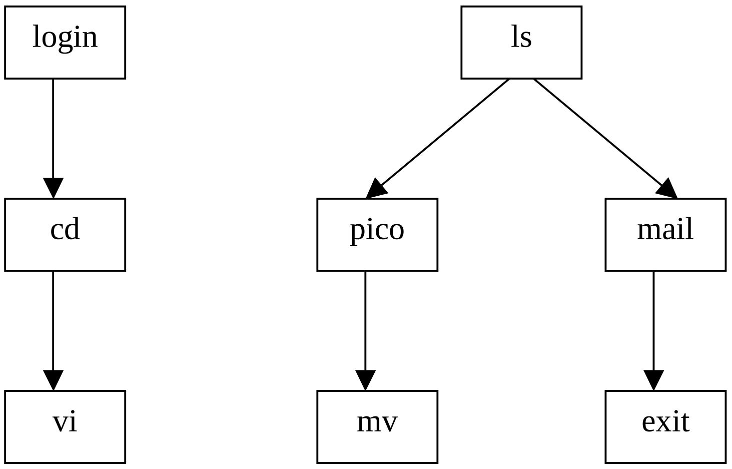
2.2 Building Probabilistic Finite Automata from User Commands

PFAs are constructed based on a listing of commands with associated timestamps. Since the PFA consists of frequently seen command traces, it is first necessary to filter the listing of commands and timestamps into a series of command traces. The command traces that are generated are completely dependent on the time intervals between successive commands. A small time interval will produce smaller command traces, while a long time interval will produce longer command traces. Below is an example of two sets of command traces derived from the same command listing, based on different time intervals, Δt.

10:00	login
10:01	cd
10:02	vi
10:04	ls
10:05	pico
10:06	mv
10:08	ls
10:09	mail
10:10	exit

$\Delta t = 1$	$\Delta t = 3$
[login, cd, vi]	[login, cd, vi, ls, pico, mv, ls, mail, exit]
[ls, pico, mv]	
[ls, mail, exit]	

The PFA is represented in a tree structure. Each node in the PFA represents a command, and the transitions between nodes are used to represent how one command follows another. Below is a simple diagram of a finite automata consisting of the command traces from the above example where $\Delta t = 1$.



Once we have the finite automata to represent the sequences of commands a given user has entered, the next step is to assign probabilities to the transitions.

Each node in the automata has a corresponding probability, which is the number of times the given command has been seen, divided by the total number of commands seen at the same level. From the above example, the probability of reaching the state login would be 1/3, and the probability of reaching the state ls would be 2/3. Once the probability of reaching each state is computed, the average probability at each level in the tree is computed as well as the standard deviation. A second probability is then assigned to each node, which corresponds to the number of standard deviations from the average probability the probability of reaching the given node is.

This second probability is computed by first obtaining the difference between the probability of reaching the node, and the average probability. By dividing the absolute value of this difference from the standard deviation, the number of deviations from the average is obtained. The following equations are then used to compute the second probability.

If prob of node A is X standard deviations greater than the average probability at the same level,

prob(A) = 1 + (0.1 * deviations)

If prob of node A is X standard deviations less than the average probability at the same level,

prob(A) = 1 - (0.2 * deviations)

If prob of node A is X equal to the average probability at the same level,

prob(A) = 1.0

There are two major reasons why this method was adopted. One reason is that by assigning a probability to a node which is greater than 1, when future command traces reach that given state, the high probability essentially rewards the behavior. Similarly nodes that have a low probability of being reached are assigned a lower probability. The other reason is this methodology works regardless of the time interval used to generate command traces. Since the time interval determines if long traces or short traces are created, this method assigns probabilities to each node in the automata relative to the surrounding nodes, regardless of the overall structure of the automata. If this model were not used, separate thresholds would have to be set up, based on the time interval.

It is important to note that each time a command trace is added to the PFA, the probabilities that nodes will be reached, the average probability of all nodes on a given level, the standard deviation and the second assigned probability all have to be recalculated. This ensures that at all times the automata has the most recent signature of a user. There is also one other structure used alongside the PFA when testing future command traces. A hashtable was constructed consisting of all the commands the user has entered, along with the number of times the command was entered. This data structure is important in the testing phase, when it is necessary to distinguish between commands not seen in a particular command trace, versus commands never seen before.

3. Testing User Signatures

Once a PFA is created for a specific user, future user activity can be tested against the PFA to determine if the behavior is anomalous or not. Since the PFA is essentially a representation of command traces, the PFA will detect anomalous command traces. There may be multiple command traces in a given session the PFA is tested against, in which case each trace will be tested individually.
Note that these command traces will be generated using the same time interval used to generate the command traces that the PFA consists of. During testing, the PFA will return two probabilities for each command trace it tests. The first probability is the multiplicative probability of each node traversed. The second probability is the average probability of the each node traversed. In both cases, keep in mind the probability used is the one based on the standard deviation from the average, not the probability of reaching the node.

The process of traversing the PFA for testing is done as follows. We get the first command from the test command trace and then look for the command in the first level of the automata. If there is an associated node, we move to the node, and keep track of the probability just encountered. If there is no node for the command, we first check if the command has ever been seen, by checking the hashtable of all the commands ever seen. If the command is in this table, then it has been seen, just not following the current state. A probability is assigned in this case which corresponds to the number of times the command has been seen in relation to all the commands that have been seen. If the command is not in this table, then the user has never typed this command before. An ultra low probability is assigned in this case, and the command is stored in a vector. Whenever there is no corresponding transition in the automata to take, the PFA remains in the same state, and the next command from the test command trace is evaluated. This process is repeated for all the commands in a given test trace. When all the commands have been evaluated, the multiplied and averaged probabilities are returned.

Below are some examples of the results of testing the PFA on frequently seen and infrequently seen command traces. Here is the PFA we are testing in graphical form:

```
vi  0.8558175541039827
cat  0.8558175541039827
pwd  0.8558175541039827
l[  0.8558175541039827
  l[  1.0
from  0.9389997344286081
  from  1.0
    from  1.0
telnet  0.8558175541039827
login  1.1774553180258673
  cd  1.0
    tcsh  1.0
      quota  1.0
        cat  1.0
          mail  1.0
ftp  0.8558175541039827
head  0.9389997344286081
uptime  0.8558175541039827
ls  1.0110909573766167
  date  1.0
cd  1.1358642278635547
  cd  1.0
  sh  1.0
    tcsh  1.0
      cd  1.0
  rm  1.0
    cd  1.0
more  0.9389997344286081
  col  1.0
lynx  1.0526820475389294
  lynx  1.0
    lynx  1.0
q  1.21904640818818
  q  1.0
    q  1.0
```

Here are sample outputs when testing the PFA against various command traces:

```
[login, cd, tcsh, quota, cat, mail]
prob is 1.1774553180258673 avg is 1.0295758863376445

[lpr, lpr, lpr, lpr]
flag.. lpr never seen before
flag.. lpr never seen before
flag.. lpr never seen before
flag.. lpr never seen before
prob is 1.0E-16 avg is 1.0E-4
```

```
[ls]
prob is 1.0110909573766167 avg is 1.0110909573766167

 [login, cd, quota, ls, cat, more]
prob is 2.2619987027301444E-6 avg is 0.39189472691735466
```

From the above examples, it is clear that command traces that match the signature will clearly return higher probabilities than the command traces which don't match any pattern. Furthermore, commands traces that do not match a given pattern still return higher probabilities than command traces consisting of commands never seen before.

4. Conclusions

The tool outlined in this project is very successful at identifying anomalous user behavior. The strength of the tool is that the signatures it creates stores information about the frequent commands as well as ordering between commands. The signatures created provide extremely accurate models of typical user behavior. Any user behavior that clearly detours from the signature will return low probabilities and raise an alarm.

The success of this tool is directly related to the strength of the signatures it creates. One problem that threatens the creation of strong signatures is the lack of sufficient training data to truly develop a strong representation of a users activity. This may somewhat be solved by using clustering methods, which attempt to group together patterns of similar valid behavior to further isolate invalid behavior.

Another limitation of this tool is the inability to differentiate attacks from variations in valid user behavior. This tool would produce way too many false alarms if used in isolation to detect attacks. This however, could be overcome if this tool was used in conjunction with other tools that monitor other aspects of a user or system.

Finally, a major strength of this tool is the fact that it could easily be reconfigured to monitor different patterns of behavior. The possibilities for this include network behavior or specific application behavior.

5. Implementation

The first step in building user signatures is to extract the user commands from the BSM log files. By combining the auditreduce and praudit unix utility commands as discussed earlier we can generate a list of all the system calls from a given user. The next step is to extract the actual user commands from the system calls, as well as the timestamp. This is done through the Java program Filter.java. The arguments of the program are as follows:

```
java Filter <filename of file with system calls> <output filename>
```

This program takes as input a file containing a listing of system calls, and outputs to another file the actual user commands and the timestamps. This output will then be used as input when creating, updating, or testing PFAs.

PFAs are implemented through two small Java programs PFANode.java and PFACollection.java. PFANode.java defines the class PFANode, which is the representation of each node in the PFA. Each PFANode has a label corresponding to the command it stores, an associated probability, and a hashtable of children nodes. PFACollection.java defines the class PFACollection, which defines how PFANodes are organized to generate a PFA. PPFACollection is responsible for creating, updating, and testing a PFA. The two major methods of PFACollection are addTrace and testTrace. The method addTrace takes a vector of commands (command trace) and traverses down the PFA, either adding new nodes or updating already existing nodes in the process. The method testTrace similarly traces down the PFA, but instead it calculates the multiplied and average probability of each PFANode it encounters.

There are six Java programs that are used to facilitate the processes of creating, updating, testing and printing PFA's. FilterCommandTraces.java defines the class FilterCommandTraces, which is responsible for generating command traces from a list of commands and timestamps. FilterCommandTraces is given the time interval used to separate traces of commands. This class is used when PFAs are created, updated, and tested. In each case, the commands to be added to the PFA, or tested against the PFA, must be filtered into command traces based on the time interval.

The program createPFACollection.java serves the purpose of creating a PFACollection object from a listing of user commands and timestamps. The arguments this program takes are as shown:

```
java createPFACollection –l <file which contains listing of filtered files> <interval>
java createPFACollection –f <filtered file> <interval>
```

The first command line argument it takes specifies whether the second command line argument is a file containing a list of commands, or a file containing a list of files, where each file in the list contains a list of commands. Above the term "filtered file" refers to a file containing a list of timestamps and commands, one per line. The third command line argument is the time interval that is given to FilterCommandTraces to generate the sequence of command traces. This program will create a PFACollection object, from the sequences of command traces, and write out a serialized object. The serialized object will be written to the name derived by appending the time interval and "_pfa" to whatever filename was specified as the second command line argument.

The program updatePFACollection serves the purpose of updating an already existing PFACollection object with further listing(s) of commands and timestamps. The arguments are below:

```
java updatePFACollection –l <serialized PFACollection>
                                        <file which contains listing of filtered files>
java updatePFACollection –f <serialized PFACollection> <filtered file>
```

The first command line argument is the same as described above. However it corresponds to the third command line argument, which can contain a listing of commands, or a listing of files. The program firsts converts the serialized PFACollection object specified in the second command line argument back to a PFACollection object. It then goes through the same steps as createPFACollection to generate command traces and add them to the PFA. It will write out a new serialized PFACollection object with the name of the old PFACollection appended by a "+". Note that the time interval passed to FilterCommandTraces is the same interval that was used to create the original PFACollection. This data is stored along with each PFACollection, and is written out with the rest of the PFACollection when serializing the object.

The program testPFACollection.java tests a listing of commands against the PFACollection. The arguments are as follows:

```
java testPFACollection –l <serialized PFACollection>
                          <file which contains listing of filtered files>
java testPFACollection –f <serialized PFACollection> <filtered file>
```

The parameters are exactly the same as in updatePFACollection, and again command traces will need to be generated based on the time interval saved in the serialized PFACollection object. However, once we have the command traces from the file(s), they will not be added to the PFACollection, but instead tested against the PFACollection.

There are two other programs SerializePFA.java and printPFACollection.java. SerializePFA.java contains the class SerializePFA, which is used to read and write PFACollection objects. This class is used by all the programs which must perform operations on a PFACollection. The program printPFACollection takes as an argument the serialized PFACollection object, and prints out a graphical representation of the PFA, with each command, and the associated probability to the terminal.

All of the files described above are located in the intrusion detection project workspace in the subfolder /seth_work/PFA/.

6. Contributions

I would like to acknowledge the following people for the contributions they made to this project. Alan Bivens wrote the filtering program that extracts user commands and timestamps from the BSM log files. Joel Branch contributed to the previous work section of this paper through his research on previous anomaly detection tools.

BIBLIOGRAPHY

[1] A. Bivens, L. Gao, M. F. Hulber and B.K. Szymanski, ``Agent-Based Network Monitoring," *Proc. Autonomous Agents99Conference*, Seattle, WA, May 1999, pp. 41-53.
[2] A. Bivens, P. Fry, L. Gao, M.F. Hulber, Q. Zhang and B.K. Szymanski, ``Distributed Object-Oriented Repositories for Network Management," *Proc. 13th Int. Conference*

on System Engineering, pp. CS169-174, Las Vegas, NV, August, 1999.

[3] A. Bivens, R. Gupta, I. McLead, B. Szymanski and J. White, ``Scalability and Performance of an Agent-based Network Management Middleware," *International Journal of Network Management*, submitted, 2002.

[4] J. Branch, A. Bivens, C-Y. Chan, T-K. Lee and B.K. Szymanski, 2002, ``Denial of Service Intrusion Detection Using Time Dependent Deterministic Finite Automata," *Proc. Graduate Research Conference,* Troy, NY, pp. 45-51.

[5] A. Bivens, M. Embrechts, C. Palagiri, R. Smith, and B.K. Szymanski, ``Network-based Intrusion Detection using Neural Networks," *Intelligent Engineering Systems through Artificial Neural Networks*, Vol. 12, Proc. ANNIE 2002 Conference, November 10-13, 2002, St. Louis, MI, ASME Press, New York, NY, 2002, pp. 579-584.

[6] Jake Ryan, Lin Meng-Jane, and Risto Miikkulainen. Intrusion detection with neural networks. In *Advances in Neural Information Processing Systems 10*, May 1998

[7] Stephen E. Smaha. Haystack: An intrusion detection system. In *Fourth AeroSpace Computer Security Applications Conference*, pages 37-44, Orlando, FL, December 1988, IEEE.

[8] S. Staniford-Chen, S. Cheung, R. Crawford, M. Dilger, J. Frank, K. Levitt, C. Wee, R. Yip, D. Zerkle, and J. Hoagland. GrIDS-a graph based intrusion detection system for large networks. http://seclab.cs.ucdavis.edu/arpa/grids/welcome.html, 1996.

[9] IMSafe. http://imsafe.sourceforge.net/

[10] P.A. Porras and P.G. Neumann. Emerald:Event monitoring enabling responses to anomalous live disturbances. In *Proceedings of the 20th National Information Systems Security Conference*, pages 353-365, October 1997.

[11] William Osser. Auditing in the Solaris 8 Operating Environment. http://www.sun.com/blueprints, February 2001.

[12] Seth E. Webster. The Development and Analysis of Intrusion Detection Algorithms. M.S. Thesis, MIT. May 1998.