Host-Based Intrusion Detection Using User Signatures

Seth Freeman
Computer Science Department
Rensselaer Polytechnic Institute
110 8th Street
Troy, NY 12180-3590
(518) 276-8326
freems@cs.rpi.edu

Alan Bivens
Computer Science Department
Rensselaer Polytechnic Institute
110 8th street
Troy, NY 12180-3590
(518) 276-8326
bivenj@cs.rpi.edu

Joel Branch
Computer Science Department
Rensselaer Polytechnic Institute
110 8th street
Troy, NY 12180-3590
(518) 276-8326
branchj@cs.rpi.edu

Boleslaw Szymanski
Computer Science Department
Rensselaer Polytechnic Institute
110 8th street
Troy, NY 12180-3590
(518) 276-8326
szymansk@cs.rpi.edu

ABSTRACT

An intrusion occurs when an attacker gains unauthorized access to a valid user's account and performs disruptive behavior while masquerading as that user. The attacker may harm the user's account directly and can use it to launch attacks on other accounts or machines. Developing "signatures" of users of a computer system is a useful method for detecting when this scenario happens. ***Our approach concentrates on developing precise user signatures characterizing multiple aspects of user activity****. Thus, anytime someone behaves in a manner inconsistent with their signature, our system will raise an alarm which strength corresponds to the unlikelihood of the current behavior to the signature.*

1. INTRODUCTION

Computer security has become an area of utmost importance as the number of people who use computers continues to increase. As the number of computer systems and networks grow, more and more bugs are discovered which attackers attempt to exploit. An intrusion detection system (IDS) is designed to monitor a computer or a network to detect invalid activity. IDSs can monitor users, applications, networks, or combinations of the three, in order to detect well-known and unknown attacks.

Walter Lincoln Hawkins
Graduate Research Conference

There are two main approaches to detecting intrusions, anomaly detection and misuse detection. Anomaly detection is the approach in which first a model of normal system or user behavior is created, and then any behavior that deviates from the model is called anomalous. The major advantage of this approach is that unknown or new attacks can be identified, because their pattern will be a deviation from the model of expected behavior. The misuse detection approach, by contrast, 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 reaction time can be reduced. The disadvantage however, is that no new attacks will be identified.

Intrusion detection systems are also classified based on the types of systems they monitor. The two main systems monitored for intrusions are host-based systems and network based systems. Host-based intrusion detection attempts to detect against attacks on a particular machine. This is typically done through analysis of a computers log files. Network-based intrusion detection attempts to detect against attacks on a network. This is typically done through analysis of network traffic.

2. MOTIVATION

The system 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.

Anomalous behavior is defined as any behavior that deviates from the model. Thus the anomalies this system detects may or may not correspond to an actual intrusion. In some instances, the user may simply be trying out a new set of commands, or their behavior is different due to fatigue or stress. Other times, there is in fact another person masquerading as the user. In either instance, the system described in this paper will detect that the behavior deviates from the model.

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 [4]. 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 [5].

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 [1]. 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 [3]. 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.

The work of Forrest, Hofmeyr, Somayaji and Longstaff details a system similar to ours that scans system call traces of normal user behavior and builds a database of characteristic user patterns [2]. The user patterns they store are essentially the way the system calls follow each other. They slide a window across the system call trace(s) of a user, and for each window of a given size k, they record the first system call and system calls that follows it at position 1 through position k After the database is generated, future sequences of system calls are compared against the database to determine if they are anomalous or not. **Our system could be viewed as an extension of this work. Instead of using a predefined window, we determine the command trace size by analyzing the user's activity. We further utilize the variability in command traces to account for users with extremely long command traces and large variety of commands.**

4. PROBABILISTIC STATE FINITE AUTOMATA

We represent each user signature as a probabilistic state finite automata (PSFA). A finite automata is a mathematical model consisting of a set of states, a set of transitions between states, an input alphabet, an initial state and a final state. A PSFA is an extended finite automata in which each state has an associated probability. The probability associated with each state corresponds to the probability that the state will be reached from the previous state. Each transition in the PSFA corresponds to a user command. The input alphabet for the PSFA is the set of commands the user entered. Figure 1 shows an example PSFA. The transitions are labeled with commands a through f, and the states each have an associated probability.

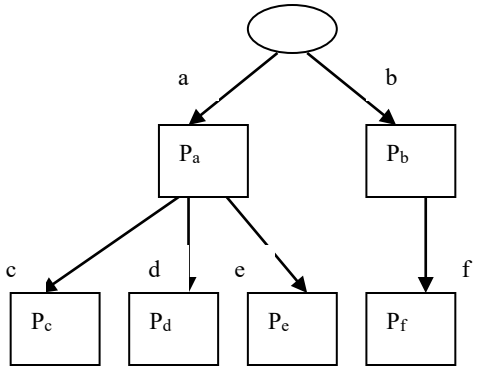


Figure 1. Example PSFA.

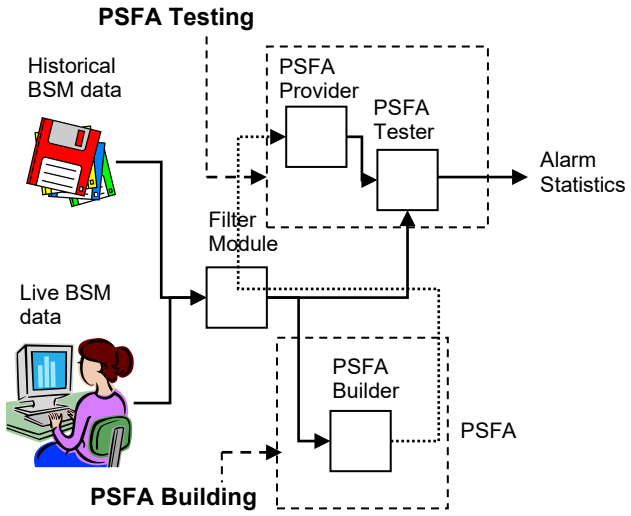


Figure 2. System Architecture.

5. SYSTEM OVERVIEW

The system described in this paper generates signatures for each user of a computer system and tests future command traces against signatures that are already created. The system consists of four basic modules. The first module, the Filter Module, is responsible for extracting user commands from log files, then converting the user commands with the associated timestamps into command traces. These command traces are then either

passed to the PSFA Builder, which constructs the PSFA, or to the PSFA Tester, which will test future command traces against the PSFA. The PSFA Provider is responsible for providing the PSFA used to test future command traces to the PSFA Tester. The diagram of the system is shown in Figure 2.

6. FILTER MODULE

6.1 Extracting User Commands from Log Files

The first step in filtering the user commands is to extract all the system calls generated by the user from the log files. The log file of a computer stores all the events that occurred on the computer. Each event that is recorded is called a system call. For every action a user makes, opening a file, typing a command, moving the mouse, etc. many system calls are generated. Each system call stored in the log file contains fields that identify the type of system call, the user who created it, the time, and other information. Extracting the system calls belonging to a given user is done through parsing the log file and extracting all system calls the given user generated.

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 use to construct our signatures are created from commands the actual user typed and not a masquerader, so that each user's signature is valid, thus invalid behavior will never match the signature.

6.2 Constructing Command Traces from User Commands

Once we have the listing of commands and timestamps we can convert them into a series of command traces. By modeling the command traces entered by the user rather than just the individual commands, we develop a more accurate representation of user behavior. The command traces that are generated are completely dependent on the time intervals between successive commands.

We define the time between two successive commands as ξ, and a time interval τ. We then specify that two successive commands will be considered in the same command trace, if $\xi < \tau$. A large τ will produce longer command traces, because there is a longer time window by which two successive commands can still be a part of the same command trace. A very large τ will in effect store an entire user session as once command trace. A small τ, on the other hand, will produce shorter command traces. The value of τ is very important in determining the structure of the PSFA. See Figure 3 and Figure 4 for an example of two sets of command traces derived from the same command listing, based on different time intervals, τ.

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

Figure 3. Example command and time listing.

$\tau = 1$

[login, cd, vi]

[ls, pico, mv]

[ls, mail, exit]

$\tau = 3$

[login, cd, vi, ls, pico, mv, ls, mail, exit]

Figure 4. Command traces derived from τ of 1 and 3.

In comparison to the approach taken by Forrest, Hofmeyr, Somayaji and Longstaff which uses a predefined window size to group together commands, our approach is more intelligent. While both our methods build models from command traces, our system does not arbitrarily group together commands into command traces, but instead groups them based on their respective timings. This provides a more accurate representation of user behavior.

7. PSFA BUILDER

7.1 Building the PSFA From Command Traces

The PSFA for each user is a collection of all the command traces for a given user. For each command trace we add, starting at the top of the PSFA, we step through the commands in the trace and check if there is a transition associated to each command. If there is no transition for the given command, then a new transition and resulting node will be added. We then follow the transition we added, or the old transition if there already was one. We then repeat the process for the next command in the command trace.

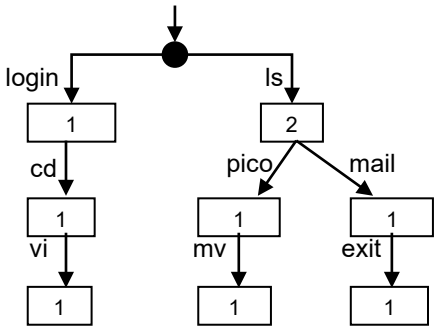


Figure 5. PSFA based on command traces derived from ☚t of 1 and 3.

Figure 5 shows the PSFA consisting of the command traces from the Figure 4 where ☚t = 1. The numbers in the nodes correspond to the number of times the node follows the parent node.

The next step is to assign probabilities P to each node in the automata. This is the simple computation of dividing the number of times the node was reached from the parent node by the total number of times all the children nodes were reached from the parent node.

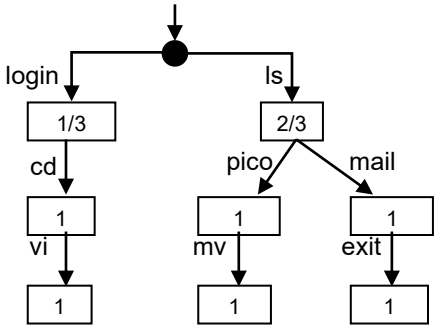


Figure 6. PSFA from Figure 5 with probabilities P assigned to each node.

Using the same example PSFA from Figure 5, we would compute the probability P of reaching the node following the 'ls' transition by dividing the number of times 'ls' is reached from its parent, 2, by the total number of nodes at the same level, 3. Thus the probability of reaching node following the 'ls' transition would be 2/3. Figure 6 shows the new automata, in which all the nodes have there associated probability.

At this point each node in our automata has a probability P associated to it, corresponding to the probability it will be reached from its parent. We can now trace down individual command traces to compute the probability of the command trace. For each command trace, we compute the multiplied probability P_m equal to the product of all the probabilities P of each node we encounter. We also compute the sum of all the probabilities P encountered and the number of all the nodes encountered in order to calculate the average probability P_a. From Figure 6, if we traced down the command trace ls → pico → mv, we would compute P_m and P_a for the given command trace as done in the equations of Equation 1 below.

$$P_m = 2/3 * 1/2 * 1 = 0.333$$
$$P_a = (2/3 + 1/2 + 1)/3 = 0.722 \quad (1)$$

With the ability to trace down the automata and compute P_m and P_a for each command trace stored in the automata, we now can distinguish frequently seen command traces from infrequent command traces. At this point we are able to test future command traces against the automata by tracing down the automata and computing P_m and P_a for the command trace being tested. If P_m and P_a are significantly low, the command trace being tested is determined to be anomalous. This will be described later.

7.2 Updating the Probabilities Using Standard Deviation Approach

When we began testing command traces against the PSFA's we ran into a problem. If when tracing down the PSFA we encounter a node with a very small probability P, the value will clearly lower the computed values P_m and P_a. In the case where all the sibling nodes of that node also have small probabilities, returning the small P unfairly lowers the P_m and P_a. We needed to implement a strategy that would only return low probabilities when the probability of reaching the node was significantly lower than the probabilities of reaching its sibling nodes. Thus we implemented an approach that assigned probabilities to each node based on how many standard deviations the probability of reaching the node is from the average probability of reaching all the sibling nodes.

To implement the standard deviation approach, each time the probability P of reaching a node is computed, all the probabilities P of the sibling nodes of the node and P are averaged. Then all of those same probabilities are compared to the average to determine the standard deviation. A second probability, P_2 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 P_2 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. Let P_{avg} be the average of all the nodes of all the siblings, and let X denote the number of standard deviations the probability P is from P_{avg}. The equation shown in Equation 2 is then used to compute the second probability P_2. Using these equations, Figure 7 shows the new automata in which each node has a second probability P_2 associated with it.

$$\text{If } P > P_{avg}$$

$$P_2 = 1 + (0.1 * X)$$

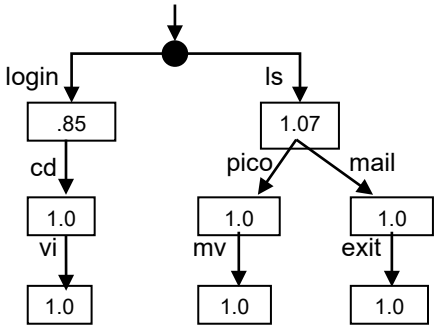
$$\text{If } P < P_{avg} \qquad (2)$$

$$P_2 = 1 - (0.2 * X)$$

$$\text{If } P = P_{avg}$$

$$P_2 = 1.0$$

Figure 7. PSFA from Figure 5 with probabilities P_2



assigned to each node.

There are two major reasons why this method was adopted. One reason is that by assigning a probability P_2 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 whose probability P is lower than the average will be assigned a lower probability P_2.

It is important to note that each time a command trace is added to the PSFA, the probability P that nodes will be reached, the average probability P_{avg} of all nodes on a given level, the standard deviation and the second assigned probability P_2 all have to be recalculated. This ensures that at all times the automata has the most recent signature of a user. Also, it was necessary to define one other data structure used alongside the PSFA that consists 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.

8. PSFA Tester

8.1 Testing User Signatures

Once a PSFA is created for a specific user, future command traces can be tested against the PSFA to determine if the behavior is anomalous or not. The command traces to be tested against the PSFA will be generated in the same manner as command traces used to build the PSFA and using the same time interval. During testing, the PSFA will return two probabilities for each command trace it tests, P_m and P_a described earlier. In both cases, keep in mind the probability being multiplied or averaged is P_2, the one based on the standard deviation from the average, not the probability of reaching the node P.

The process of traversing the PSFA for testing is done as follows. We get the first command from the test command trace and then look for a transition with the command in the first level of the automata. If there is an associated transition, we follow the transition and keep track of the probability P_2 at the node we encounter. If there is no transition for the command, we first check if the command has ever been seen, by checking the data structure 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 so an ultra low probability is assigned. Whenever there is no corresponding transition in the automata to take, the PSFA 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.. Figure 8 is a representation of part of an actual PSFA our system generated. The probabilities associated with each node in the figure correspond to P_2. **Figure 9 shows the results of testing the PSFA in Figure 8 against four different command traces. First the command trace being tested is listed, then below it are the computed values of P_m and P_a. Below the second command trace (lpr → lpr → lpr→ lpr) are the actual error messages our program generates since the commands were never seen anywhere in the PSFA.**

From Figures 8 and 9, it is clear that command traces that match the signature will clearly return higher probabilities than the command traces which do not 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.

9. PSFA PROVIDER

The PSFA Provider is responsible for providing the PSFA to the PSFA Tester. The PSFA it provides may come from the PFSA Builder after the completion of the building stage or it could be provided by an outside source like a distributed PFSA Manager. In this case, the manager would create an updated PFSA and send it to the PSFA provider module of the current detection application. When the PSFA Provider receives a new PSFA, it simply replaces the old PSFA in the PSFA Tester with the new PSFA.

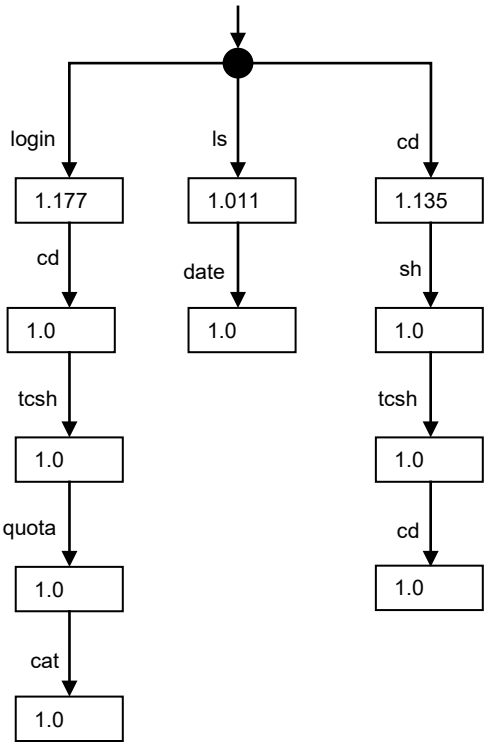


Figure 8. Graphical representation of PSFA.

login → cd → tcsh → quota → cat

P_m = 1.1777 P_a = 1.035

lpr → lpr → lpr→ lpr

flag.. lpr never seen before

flag.. lpr never seen before

flag.. lpr never seen before

flag.. lpr never seen before

P_m = 1.0E-16 P_a = 1.0E-4

ls

P_m = 1.011 P_a = 1.011

login → cd → quota → ls → cat → more

P_m = 2.2E-6 P_a = 0.391

Figure 9. Output of PSFA from testing command traces.

10. CONCLUSIONS

The system outlined in this project is very successful at identifying anomalous user behavior. The strength of the system 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 system 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.

Finally, a major strength of this system 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.

11. FUTURE WORK

One way to strengthen the signatures this system creates is to generate signatures of groups of user instead of individual users. This can be done through using clustering methods that would group together individual users that have similar signatures. By having PSFA for a group of users, future command traces can be tested against the automata to determine if the behavior matches the group signature. This approach has the advantage that the group signatures would be stronger since it would likely encompass more valid user activity.

12. REFERENCES

[1] L. Eschenauer. Inside the ImSafe IDS. ImSafe – Host Based Anomaly Detection Tool. *http://imsafe.sourceforge.net/*.

[2] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. A Sense of Self For Unix Processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 120-128.1996.

[3] 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.

[4] J. Ryan, L. Meng-Jane, and R. Miikkulainen. Intrusion Detection with Neural Networks. In *Advances in Neural Information Processing Systems 10*, May 1998.

[5] S. Stephen. Haystack: An Intrusion Detection System, 37-44. In *Proceedings of the Fourth Aerospace Computer Security Applications Conference*. Orlando, Florida, December 12-16, 1988. Washington, DC: IEEE Computer Society Press, 1989

.