Proceedings of the 2003 IEEE
Workshop on Information Assurance
United States Military Academy, West Point, NY June 2001

Detecting Insider Threats by Monitoring System Call Activity

Nam Nguyen and Peter Reiher
Computer Science Department
University of California, Los Angeles
{songuku, reiher}@cs.ucla.edu

Geoffrey H. Kuenning
Computer Science Department
Harvey Mudd College
geoff@cs.hmc.edu

Abstract -?One approach to detecting insider misbehavior is to monitor system call activity and watch for danger signs or unusual behavior. We describe an experimental system designed to test this approach. We tested the system's ability to detect common insider misbehavior by examining file system and process-related system calls. Our results show that this approach can detect many such activities. *

I. INTRODUCTION

While attacks on computers by outside intruders are more publicized, attacks perpetrated by insiders are very common and often more damaging. Insiders represent the greatest threat to computer security because they understand their organization's business and how their computer systems work. They have both the confidentiality and access to perform these attacks. An inside attacker will have a higher probability of successfully breaking into the system and extracting critical information. The insiders also represent the greatest challenge to securing the company network because they are authorized a level of access to the file system and granted a degree of trust.

In this paper we will present our analysis results on raw system call traces to see if it is possible to detect insider threats by monitoring file access and process activity. Many intrusion systems have already been developed by building profiles on the system call traces. However most of them only look at the system call level or session level. In this paper, we want to look at these raw data in a different manner: the relationships between users and files, users and processes, and processes and files.

By analyzing these models and relationships, we want to learn whether it is possible to build an effective insider threat detection system for each of these relationships. If any of our models do not work, we want to discover the reasons and all technical difficulties behind the problem. Furthermore, we want to discover any characteristics or promising approaches that can help to build good profiles for users and processes.

As a proof of concept, we implemented a small detection system that use one of these profiles to detect a large set of buffer-overflow attacks. The goal is not to build a perfect system, but to demonstrate how one can easily use a simple profile of process execution to effectively detect many buffer-overflow attacks.

II. RELATED WORK

Many intrusion detection systems have been developed. MIDAS [10] is an early intrusion detection system based on rules for discovering anomalous behavior. It uses events generated by the system, such as login time, number of bad logins, attempts to run special suid commands, etc. This approach is simple, but not effective against professional intruders.

NSM [9], Network System Monitor, was among the first systems using network traffic as the audit data. This tool looks at the data-path communication and the protocols to build a profile. Lee later suggested using a data-mining approach on raw audit data of network traffic [6, 7, 8]. He defined some attributes of network traffic, such as service types, timestamp, src_bytes, dst_bytes and built rules from these attributes. This approach requires "sufficient" training data that covers as much variation of the normal behavior as possible [7].

DPEM [3] is an intrusion detection tool that uses program execution traces to derive a policy for the correct behavior of some special privileged Unix processes. If a program execution varies from the policy, it will raise an alarm. This tool only protects a list of predetermined programs. If different programs are exploited, the tool does not notice. Moreover, the policy is too specific and specialized for each program.

System call traces and rule learners have also been used to detect possible intrusion [2, 5, 11]. The approach is to detect if the next system call is abnormal from the sequence of previous system calls. Such tools focus on the system call level and only work for certain attacks.

Because many attacks originate from buggy software, Wagner et al. proposed a solution to detect buffer overflow bugs by analyzing the source code of the software [12]. This approach requires software to be analyzed before coming to production and generates a large number of false alarms.

? * This research was supported by DARPA Grant F33615-00-C-1746

ISBN 555555555/$10.00 ? 2003 IEEE

The major difference between our approach and that of other tools (like DPEM) is to build the profile such that it is generic and easy to configure. By analyzing the system traces, we want to create a profile that is not specific to any particular program. Thus, tools that use our profile can detect a large set of attacks instead of only an individual attack. Moreover, the profile can be built automatically and does not require human intervention.

Another difference is that we look at this raw data differently, examining the relationships between users and files, users and processes, processes and files. Some existing tools [2, 3, 7, 11] look at these raw data at the system call level. For example, by using some expert rule learner, they want to guess the next system call from the sequence of previous system calls made. The problem of this approach is that the false alarm rates are often very high due to the large possibility of system calls and arguments. Moreover, most of these systems use sequences of system call traces, recorded for a specific version of the software, as the training data. As a result, the learning rules are too specific to that program, and the tool can only detect well-known or individual attacks.

Our goal is not to prove that our approach is better than these existing tools. Instead, we want to see if our models, based on the relationships between users and files, users and processes, and processes and files, can reveal good results and promising direction toward intrusion detection. From these results, we hope a sophisticated data-mining algorithm can be applied to build a good intrusion detection system.

III. Research Approach

In modern computers and operating systems, practically nothing of significance can be done without accessing files and running programs. The file system is a basic and fundamental mechanism for storing information; all users need to access file systems to do their work. A programmer creates and saves her program in files; a secretary saves and loads his office work in document files. File system access also happens when the user is not even aware of it; e.g., when the user surfs the web, cached information is stored in files. Many low-level system events, such as accessing networks and other devices, may require accessing files for determining the configuration of devices.

Process execution is also an important event. All executable software on a computer, including parts of the operating system, is organized into processes, each of which is responsible for a certain task. For example: when user A logs in, a process displays the login prompt. If the user logs in successfully, another process is executed to let the user begin working.

Because both file access and process execution are so crucial and unavoidable for the user, they can be excellent candidates for reflecting user behaviors. Thus, one particularly promising approach to detecting insider misbehavior is to trap and analyze file and process events. Based on the results, we can develop a model of the appropriate behavior of each individual user. By comparing ongoing activity against the predicted model, we can detect any deviation from the model and consequently signal an alarm. This type of detection is often referred to as anomaly detection.

To analyze file access and process execution, we needed a log of system activity. Fortunately, we already had a large database of system call traces, collected for the project using software developed for Seer [4]. The traces were collected from ten machines with twenty users over two years.

IV. Analysis Results

A. File Access

Our approach for analyzing file access was to develop patterns for two models: user-oriented and process-oriented.

In the user-oriented model, we tried to find access patterns that could be useful for building a profile for each user. We believe that each user normally does certain types of tasks, and thus has certain file access patterns that represent his normal behavior. For example, if user A is a programmer, he normally accesses files in his project directory. If he attempts to access files in another user's directory, that might be a signal of misbehavior.

In the process-oriented model, we looked at how processes access files. In many cases, the file access profile of a program is even more telling than the file access profile of a user. An insider may use the privileges of a program to access files in which he is interested. In other words, as he forces the program to behave in uncharacteristic ways, its file access profile will change. For example, a web server program normally accesses files in the *public_html* directory. As the attacker compromises it, the program begins to access files in other system directories such as */etc* or */var*. (e.g., buffer-overflow attacks on the IIS and Apache web servers).

1. User-Oriented Model

When analyzing patterns for each user, we decided to categorize the users into two sets: system users and normal users. Normal users are human, whereas system users are predefined users of the system such as *bin, daemon, xfs, nobody*, etc. Users from these two sets have different characteristics because of their nature. Human users are more interactive, and thus their behaviors are

ISBN 555555555/$10.00 ? 2003 IEEE

more dynamic and complex. System users are often dedicated to only a certain task, and thus they have a more static behavior with small working sets. Moreover, system users have certain privileges that are very specific to their job: the web server user has access to the *public_html* directory, whereas the *xfs* user can access X fonts and X-Window files. These system users have important system privileges, yet they also often interact with human users in normal operation. Misusing these interactions offers a misbehaving insider an opportunity to improperly expand his access privileges. Many attacks have been reported on the *xfont*, *nobody* and *web* users. Many administrators often only focus on human user activities and do not pay attention to the system user. This is dangerous because it opens a door for the attacker to gain access to the system.

a) Human Users

From our analysis, we observed that each user has a fairly static working set of files. For example: user A, a programmer, always accessed his project and mail directories; user B, an administrative user, always accessed his mail and document directories.

Therefore, we calculate each user's number and percentage of daily accesses to each file and directory, on both his personal laptop system and on the server. We expected that these numbers would be somewhat constant for each day; however, the results were unexpected. Figures 1 to 3 show the directory usage in percentage for each user on his personal laptop environment. For each day, we counted the percentage of times each directory was accessed. Due to the space limit, the chart only shows some of the top directories in the file system hierarchy. The irregular shape of the graph shows that the directory usage is very dynamic. This also applies to the file usage. One reason is that directory and file usage depend heavily on the programs run by the users. In other words, some programs access larger numbers of files than others, and thus when the user executes these programs, the distribution of file usage changes dramatically.

Figure 4 shows the total number of file accesses for the same three users. The graph is also very irregular. This is also due to the different execution of programs performed by users. Other users experienced similar patterns.

Besides looking at each user's personal machine, we also measured the activity of each user on a shared server. Figures 5 to 7 show the directory usage for each user on the same server. The directories shown here are chosen to be the same as Figures 1 to 3 for comparison. The graph is steadier in this case because the users often used the server for side-work such as checking mail, checking schedules, etc. Compared to the numbers in Figures 1 to 3, the numbers in Figures 5 to 7 reflect user activity less accurately because the users often did the majority of their work on personal laptops.

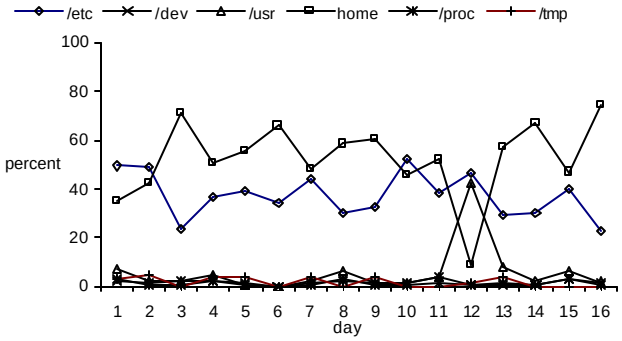

Figure 1. Directories accessed daily for user A (on his personal machine)

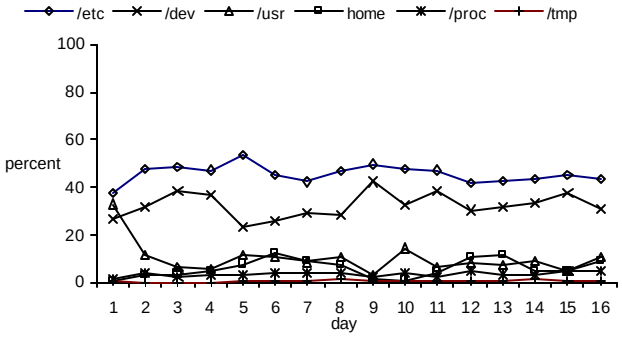

Figure 2. Directories accessed daily for user B (on his personal machine)

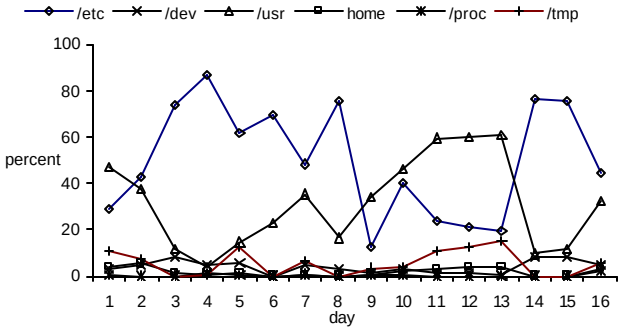

Figure 3. Directories accessed daily for user C (on his personal machine)

Finally, because the percentage of accesses is not steady enough, we cannot use this measurement to build profiles for each user. There are several reasons for this large variance on the file accesses. As we stated above, files accessed are affected heavily by the program execution. If the users execute some program that accesses files more often, the percentage will be changed. Second, some file accesses are not interesting or important. For example, when a program is executed, some ld_library files are almost always accessed. If we can differentiate

ISBN 555555555/$10.00 ? 2003 IEEE

between important and unimportant accesses, the statistical result on file accesses can give a more meaningful result. However, this kind of task faces some problems, which are discussed in section VI.

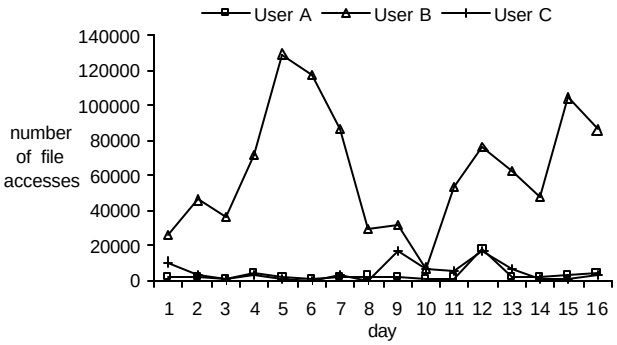

Figure 4. Files accessed daily for users (on their personal machine)

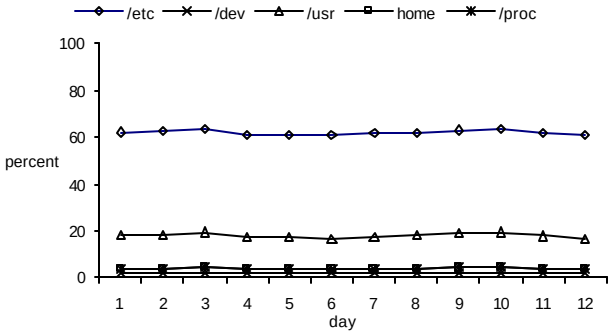

Figure 5. Directories accessed daily for user A (on the server)

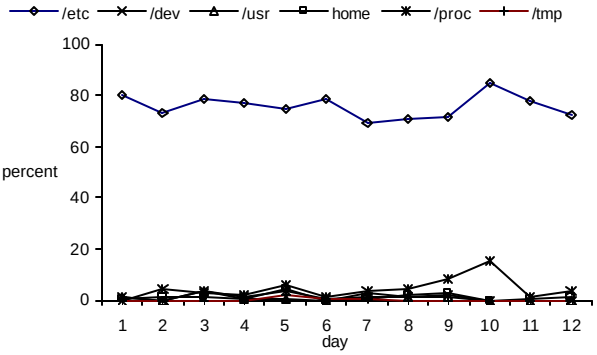

Figure 6. Direct ories accessed daily for user B (on the server)

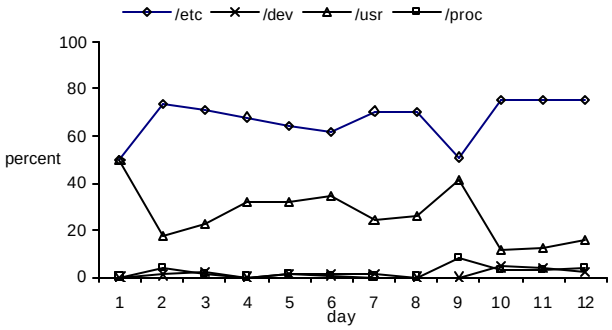

Figure 7. Directories accessed daily for user C (on the server)

b) System Users

In contrast to human users, system users have rigid working set behavior patterns. Figure 8 shows the number of files that each system user accesses during a three-month period. We can see that these users access a very small list of files, ranging from 2 to 10 files per user. Moreover, these files are very specific to the task of each user. These characteristics suggest a method of building profiles for system users by keeping track of the list of files that each user is allowed to access. Therefore, if the attacker gains access to these privileged accounts, we can detect the intrusion immediately because the attacker will be accessing files that are not on the authorized list for that specific account.

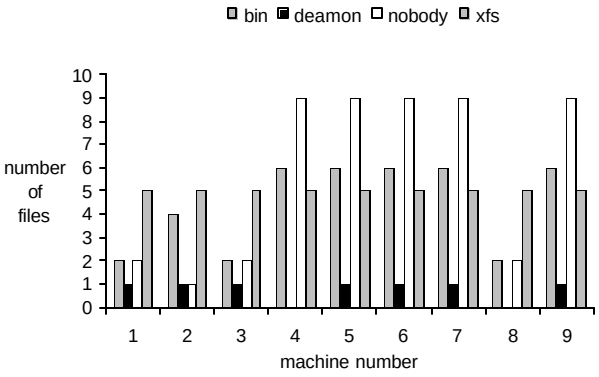

Figure 8. File set for special users

Not only do these users have a small list of files, the percentage of accesses to these files by system users is very steady. Figure 9 illustrated the percentage of file usage for system users. We can see that the percentage stays almost constant in some cases. This is reasonable because the number of files accessed by system users is very small. This also recommends another method for detecting malicious attacks on system users – that of checking the percentage of file accesses. If the percentage goes outside the expected range, an anomaly is signaled.

ISBN 555555555/$10.00 ? 2003 IEEE

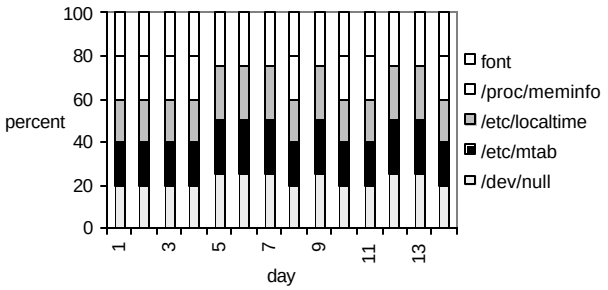

Figure 9. File usage daily for user "xfs"

Finally, some of the files are correlated to each other. In other words, when file A was accessed, file B was accessed as well. For example, user *nobody* often accesses two files, "/etc/hosts.deny" and "/etc/hosts.allow" together. These patterns could be used to detect attempts to misuse programs run by these users.

In summary, an attack on system users can be detected based on the following events: accessed files not in the normal list, frequency of access to files in the list, and changes in the correlation of files

2. Process-Oriented Model

According to our analysis, a process-oriented model gives better statistical results for user behavior. We found that 92% of the processes traced have a fixed list of files that they access. Moreover, in most cases, the percentage of accesses to different files on this list varies, but rarely by more than 20%. Thus, if an attacker attempts to compromise a program in this large class of processes, the detection system will be able to identify the anomaly behavior immediately as the attacker forces the process to access files outside of its fixed list, or access files on the list in improper proportions.

From this analysis, we also see the correlation between files accessed by each process. Many programs access files in particular patterns: first A, then B, then C, and so forth. For example, a shell typically opens a set of files in the sequence: "*profile*", "*/etc/localtime,*" "*csh.cshrc.*" Another example is that a programmer typically uses "vi" to opens file with the extension ".c" and ".h". If a user runs such a program multiple times, we also expect to see correlations between the numbers of user accesses to files. Thus, if a user typically spends much of his time on coding and compilation, we expect to see correlations that match those activities.

B. Process Execution

Because the traces provide detailed information whenever a process forks a child or executes a program, we are able to reconstruct the whole image of the process hierarchy of the system at any given time. By comparing different process trees over a long period of time, we expected to uncover interesting patterns related to process execution.

We began by looking at the list of all processes that are forked or executed by each program. The results are collected from a one-month trace period on a server with a total of 250 programs and 1 billion forks. According to our analysis results, the list of possible child processes for a given process is nearly as predictable and stable as the process's list of files accessed. 92% of all programs run in the traced environment have a fixed list of possible child processes (among these, 25% do not fork any children). Most programs, in other words, will only create a limited and highly predictable set of child processes. This information allows us to detect whenever a process has forked a child process when it usually does not do so. For programs in these classes, it is a sign of suspicious behavior if a child is created that is not in their normal set.

This statistic implies a method for detecting a very common class of vulnerability: buffer overflows. Buffer overflows account for more than 50% of today 's vulnerabilities, and this ratio seems to be increasing over time [12]. For this type of attack, the intruder attempts to stuff more data into a buffer than it can handle. As a result, data that goes beyond the size of the buffer will overwrite the stack and thus allow the attacker to cause the instruction pointer (IP) to point to his malicious code. By doing this to a privileged program, the attacker can force the program to execute other programs that allow him to change the system configuration or create damage. Many privileged programs are inherently capable of doing very limited things, but if the attacker can convince them to fork a general execution shell under their privileged identity, the attacker can gain general privileged access.

Thus, as the buffer-overflow attack occurs, we can immediately detect if the new child process of the exploited program is not on the authorized list. An authorized list is the list of child processes that the program normally forks. When a buffer overflow attack occurs and the exploited program starts to fork a new program (like shell), we can signal a possible intrusion alarm. Since more than 90% of processes have a deterministic set of authorized children, we can detect many buffer overflow attacks.

Observing process creation behavior allowed us to develop other models that are useful in detecting suspicious user behavior. For example, just as users have characteristic working sets of files used, they have favorite programs. Figures 10 and 11 depicted the favorite programs of two users on three continuous days (statistics are similar for the entire month). The average number of executions per day in the graph is 159 for user A and 9123 for user B.

ISBN 555555555/$10.00 ? 2003 IEEE

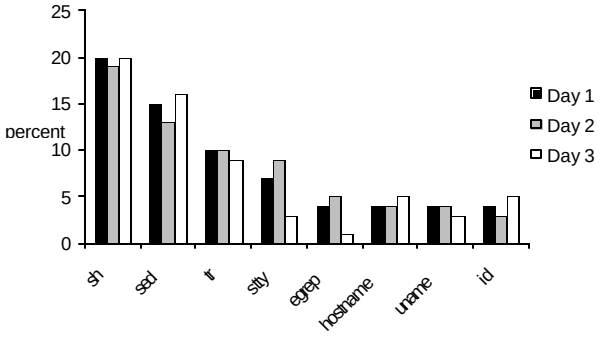

Figure 10. Favorite programs of user A

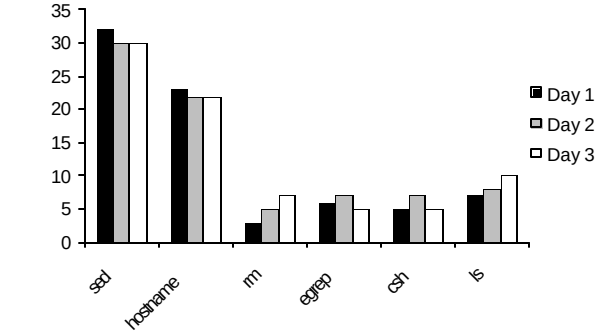

Figure 11. Favorite programs of user B

Figures 10 and 11 show that the frequency of use for these programs is quite stable. If a user decides to misbehave, in many cases he is likely to use different programs to do so. If he needs extra privileges, he will have to find vulnerabilities in privileged programs; this will generally require him to work with programs beyond the normal set that he uses. In many situations, the intruder often has to run attack toolkits that probe a large number of programs, most of which are not widely used by typical users. Moreover, he may have to repeat the attacks many times before he can successfully break the system. Such behavior would be far outside our models of proper behavior. Even if the user is working within his normal privileges, but in a dangerous manner, there is a good chance that the damaging acts he intends to perform will cause him to use programs he normally does not use, or at least cause him to use his favored set in different proportions.

The race-condition attack is an excellent example. It requires repetitively running programs that have probabilistic flaws until the attacker gains control. By counting the number of typical executions of particular programs over time for individual users, we can easily determine that a user is attempting to use this technique.

V. BUFFER-OVERFLOW DETECTION SYSTEM (BDS)

Based on the analysis described above, we have developed a viable system to detect insider threats. The purpose of this system is a proof-of-concept. Thus, in this version of the system, we only demonstrate that we can detect intrusion using the analysis results of Section IV. The current implementation of the system focuses on buffer-overflow and well-known vulnerabilities such as symlinks and race-conditions. We decided to concentrate on the buffer-overflow vulnerability since it is the most common type of attack (even though it has been around for many years). These attacks can be found in many crucial and popular applications, such as email servers, cron, web servers, ssh, etc.

One important point that BDS illustrates is that it is not designed for a specific exploit. Instead it tackles the whole class of buffer-overflow attacks, and thus can be used to detect new or unseen exploits.

A. Design of the System

BDS is built on top of the *FSOBSERVER* kernel, a modified version of the original Linux kernel. It was originally implemented as part of the Seer toolkit [4] and later extended. The code is patched at the entry and exit point of all system calls. All system calls made by users are recorded by the FSOBSERVER.

BDS is an agent placed in the kernel that can watch all system calls to determine if any are hazardous. If an intrusion is detected, BDS will raise an alarm to the system administrator. When the system is first started, it reads from its database (e.g., a file) the profile of a program's execution. This database of profiles is created from an analysis of the trace files. The database also gets updated when the agent detects a new type of profile.

This database contains a list of relationships. A relationship is defined as {X, y1, y2, .., yn}, where X is the program name and y_i is an authorized child. For example, a valid relation is {*netscape*, *netscape*}. This specifies that the program named "*netscape*" can only execute another "*netscape*" but nothing else. The system reads the input file and stores it in local hash tables. Whenever a process attempts to execute another child, the system will compare the child's name against its database. If the child is not a legitimate process, the system will raise an alarm.

There are two types of intrusion detection systems: passive and active [1]. In the active system, whenever an intrusion is detected, the malicious action is stopped and an alarm is raised. In contrast, the passive system only records actions in the system log and does not try to stop the action. Each system has different drawbacks. In the

ISBN 555555555/$10.00 ? 2003 IEEE

active system, if the action is a real intrusion, the system can stop the intruder immediately. However, if it is a false alarm, it can create deadlock or suspend legitimate users. On the other hand, the passive system cannot stop the intruder, but it reduces any damage and inconvenience caused by the false alarm. In our system, we decided to use the passive method because stopping a system call is such a serious action that it may cause the system to become nonfunctional. Since we decided to use the passive approach, we assumed that the log file was written into a non-rewritable media. In other words, the intruder cannot suppress the alarm by erasing the log files.

B. False Positive and False Negative Rate

As more than 90% of the programs have a fixed list of children, the system should have low false positive and false negative rates. We tested the system with different lengths of knowledge input files. The system was installed and ran in the background of user's personal machine for three weeks to measure the false alarm rates. The system was fed with the knowledge of different lengths of historical system call traces. As shown in Figure 12, when the system has three months of knowledge, the system performs with a zero false positive rate.

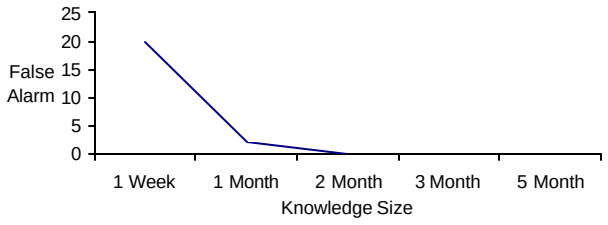

Figure 12. False alarm rate of the buffer overflow detection system

Since each version of the operating system experiences different kinds of exploits, we needed correct versions of the OS and library packages to test particular exploits. New versions of the OS often have been debugged and have less vulnerabilities to test. Moreover, it is difficult to retrieve the exploitation code for current versions of an OS because they are not posted for the public due to security concerns. Therefore, we decided to test our system using an old version of the OS (*RedHat 6*) in order to have enough vulnerabilities to test.

We downloaded and performed seven attacks for RedHat 6 successfully. There are many more attacks involving buffer overflows, but they require different versions of the library and environments. Out of these seven exploits, the system was able to detect six. The system was not designed specifically for any of these attacks. Our tool is designed to catch all buffer overflows as long as the attack occurs in the set of programs that have a fixed list of children. There was one exploit that we were unable to detect. This is not surprising, as it was a buffer-overflow for "*xterm*," and this program does not have a fixed list of children. Thus, this attack was not covered in the scope of our system. Due to the small number of attacks being tested, the success rate cannot be used to deduce a false negative rate.

C. Known Disadvantages of the System

As we mentioned above, if the buffer overflow occurs on a program that does not have a fixed list of children (such as shell, xterm, etc.), the system will not be able to detect the attack. These programs, by nature, can fork any kind of program at will, and thus it is hard to tell if the child process is legally forked. From our analysis in Section IV, these programs make up about 8% of the frequent-use programs in the system.

VI. TECHNICAL DIFFICULTIES

We first thought that the percentage of directories accessed by each user could be a good measure for detecting abnormal behavior. For example, if a user attempts an attack by searching different directories, the percentage of accesses per directory will change. However, the percentage of directory usage per directory for each user does not stay constant. In fact, it varies widely. One reason is that the file usage depends heavily on the process execution. For example, if a user runs process A only a little bit more than usual, the usage can change dramatically if the process opens many files. This is true when the process needs to open different configuration files (or open them repeatedly). Thus, if we build a system based on the percentage of files accessed, the false positive or negative rate will be high.

Another reason that the model is ineffective is that the number of files opened is on the scale of thousands per day. Thus, an attacker can easily fool the system by only opening a small number of files without affecting the accessed percentage rate.

Another model we considered involves focusing on the importance level of files in the system and removing unnecessary files from the analysis. From the trace data collected, we concluded that most of the files will open library files such as /etc/ld_cache.so. In fact, these files occupy more than 20% of accesses if the user executes a lot of programs. These files are normally unimportant and can therefore be removed. However, this approach is not desirable as it requires that we build a database of unimportant files. This is OS-specific and not portable as the system changes or upgrades. Also the database can grow quickly as the number of files in the system is immense. Moreover, some files are important in some cases and are not in other cases. For example, some processes often open */etc/passwd* to get the current working directory of the user. In this case, the file access

ISBN 555555555/$10.00 ? 2003 IEEE

can be considered as unimportant. However, how could the system distinguish between that case and the case where the process opens */etc/passwd* to break the password information?

Because it is undesirable to use the percentage of directories accessed per user, we used a different model – one where we look at the percentage of files accessed per process. This approach faces a challenging issue: many processes open a large number of files, and the system is required to understand the meaning of the files.

Many traces were collected from laptops, and the working style depicted by the traces is highly dynamic since it depends on user mobility. Many users do not have any fixed pattern of working time, and it is very difficult to define a good time window for analysis. Currently we use one day as our window time. However, this is not a perfect solution since some users work past midnight.

VII. CONCLUSION

The work described here indicates some promising directions for detecting misbehavior by insiders, as well as intruders whose initial penetration has gone unnoticed. The patterns of file accesses by many programs are sufficiently regular that attackers trying to misuse them will quickly be noticed. Similarly, process-calling behavior is sufficiently regular for large classes of programs to serve as a good indicator of misbehavior.

Other patterns of file systems access and process behavior do not appear to be good candidates for detecting attacks. Individual users have too much variability in their access patterns to allow simple statistical methods to detect suspicious changes in behavior, without also triggering alerts in many innocuous situations. Similarly, some processes, by their nature, tend to fork a wide variety of other processes.

Thus, a system that merely implemented the effective tools discussed in this paper would not catch all attacks. However, these techniques do appear to be useful candidates to include in a system that monitors computers for possible misbehavior. Our experience with the data gathering and monitoring steps shows that the necessary data can be gathered and threats checked without causing noticeable delays to the user. There are challenges to collecting and managing the large quantities of data necessary to build good models, but these challenges can be overcome.

VIII. REFERENCES

[1] S. Axelsson. Intrusion Detection Systems: A Survey and Taxonomy. *Technical Report.* Department of Computer Engineering, Chalmers University of Technology, Sweden, March 2000.

[2] S. A. Hofmeyr, S. Forrest, A. Somayaji. Intrusion Detection Using Sequences of System Calls. *Journal of Computer Security*, 6(3): 151-180, 1998.

[3] C. Ko, M. Ruschitzka, and K. Levitt. Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-Based Approach. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, Vol. ix, pp. 175-187, Oakland, CA, USA, May 1997.

[4] G. Kuenning, G. Popek. Automated Hoarding for Mobile Computers. In *16th ACM Symposium on Operating System Principles*, pp. 264-175, 1997.

[5] S. Kumar, and E. Spafford. A Pattern Matching Model for Misuse Intrusion Detection. In *Proceedings of the 17th National Computer Security Conference*, pp. 11-21, October 1994.

[6] W. Lee. A Data Mining Framework for Constructing Features and Models for Intrusion Detection Systems. PhD Thesis, Computer Science Department, Columbia University, 1999.

[7] W. Lee, S. J. Stolfo. Adaptive Intrusion Detection: A Data Mining Approach. In *Artificial Intelligence Review*, 14:533-567, 2001

[8] W. Lee, S. J. Stolfo, K. Mok. A Data Mining Framework for Building Intrusion Detection Models. In *IEEE Symposium on Security and Privacy*, Berkeley, California, May 1999.

[9] B. Mukherjee, L. Heberlein, and K. Levitt. Network Intrusion Detection. *IEEE Network*, June 1994.

[10] M. Sebring, E. Shellhouse, M. Hanna, and Whitehurst. Expert Systems in Intrusion Detection: A Case Study. In *Proceedings of the Summer USENIX Conference*, pp. 74-81, Baltimore, Maryland, 17-20 October 1988.

[11] R. Sekar, T. Bowen, M. Segal. On Preventing Intrusions by Process Behavior Monitoring. In *Proceedings of the 1st USENIX Workshop on Intrusion Detection and Network Monitoring*, April 1999.

[12] D. Wagner, J. Foster, E. Brewer, A. Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proceeding of the Year 2000 Network and Distributed Systems Security Symposium*, 2000.

ISBN 555555555/$10.00 ? 2003 IEEE