

Host-based Anomaly Detection by Wrapping File System Accesses *

Shlomo Hershkop Ryan Ferster Linh H. Bui Ke Wang
Salvatore J. Stolfo

Columbia University, New York, NY 10027, USA
{shlomo,rlf92,lhb2001,kewang,sal}@cs.columbia.edu

Abstract

We describe a Host-based Intrusion Detection System that monitors file system calls to detect anomalous accesses. The approach we describe is a fast, automatic, and inexpensive monitoring system which can augment current IDS's without the overhead of OS system call tracing and wrappers. We report on a deployed system using the FiST file wrapper technology and results of an anomaly detector, we call FWRAP, implemented in our lab environment. FWRAP employs the Probabilistic Anomaly Detection (PAD) algorithm previously reported on work on Windows Registry Anomaly Detection, extended here to operate on Unix platforms. The detection system is not programmed, rather PAD is an unsupervised machine learning algorithm that is used to train a detector. The detector is first trained by operating the host computer and a model specific to the target machine is automatically computed by PAD.

1 Introduction

Some approaches to host-based anomaly detection have focused on monitoring the operating system's (OS) processes during program execution and alerting on anomalous sequences of system calls. For example, OS wrappers monitor each system call or DLL application and test a set of rules for "consistent" program execution [2, 8, 11]. This presumes that a program's legitimate system call execution can be

specified correctly by a set of predefined rules. Alternatively, some have implemented machine learning techniques that model sequences of "normal" execution traces and thus detect run time anomalies that exhibit abnormal execution traces [5, 14].

Anomaly Detection is an important alternative detection methodology that has the advantage of defending against new threats not detectable by signature based systems. In general, anomaly detectors build a (mathematical) description of **normal** activity, by training a model of a system under typical operation, and compare the normal model at run time to detect deviations of interest. Anomaly Detectors may be used over any "audit source" to both train and test for deviations from the norm.

There are several important intuitive advantages to auditing at the OS level. This approach may provide *broad coverage* and *generality*: for a given target platform it may have wide applicability to detect a variety of malicious applications that may run on that platform.

However, there are several disadvantages to anomaly detection at the OS monitoring level. *Performance* (tracing and analyzing system calls) is not cheap; there is a substantial overhead for running these systems, even if "lightweight". Second, the adaptability and extensibility of these systems question their practicality as updates to a platform may necessitate a complete retraining of the OS trace models.

Furthermore, OS system call tracing and anomaly detection may have another serious deficiency; they may suffer from *mimicry attack* [16], since the target platform is widely available for study by attackers.

*This work has been supported in part by a grant from DARPA, Contract No. F30602-00-1-0603.

We have taken an alternative view of host-based anomaly detection. Anomalous process executions (possibly those that are malicious) may not truly damage a system unless and until the malicious execution attempts to alter or damage the machine's permanent store. Thus, a malicious attack that alters run-time memory is perhaps less important than actions that attempt to damage permanent store of the host in question. In this case, the two very important host based systems to defend and protect are the Registry (in Window's case) and the file system (in both Window's and Unix cases).

Previous work reported in Raid 2002 [1], describes RAD, the Windows Registry Anomaly Detector. At its core, RAD monitors each Windows registry [12] query, and builds a model of **normal** registry use. This baseline model is then used at run-time to detect errant or abnormal registry accesses indicative of malicious program executions, either purposeful changes to that registry to harm a system, or to identify information about the target of the malicious exploit. In either case, the Registry is an important central source of information of interest to malicious program execution.

In subsequent work, the RAD system was designed to be "self protecting". The normal windows registry models computed by RAD are subject to attack either by alteration or mimicry attack by malicious code. Thus the RAD models are stored in the registry itself. Hence, any malicious executable attempting to attack the RAD sensor will necessarily require access to the registry to launch its attack, which will very likely be detected as an abnormal registry access by the sensor.

In the case of Unix platforms there is no central registry to monitor. In this case we focus our auditing on the underlying file system. The file system is the core permanent store of the host and any malicious execution intended to damage a host will ultimately set its sights upon the file system.

The File Wrapper Anomaly Detection System (FWRAP) is a host-based anomaly detector that utilizes file wrapper technology to monitor file system accesses. It is the counterpart of RegBam (the registry "wrapper") developed for RAD for the registry. The file wrappers implemented in FWRAP are based

upon work described in [17] and operate in much the same fashion as the wrapper technology described in [8, 2]. The wrappers are implemented to extract a set of information about each file access including, for example, date and time of access, host, UID, PID, and filename, etc. Each such file access thus generates a record describing that access. Intuitively, these records provide the same type of information associated with a Windows Registry access, and as such can be modeled in the same fashion.

RAD introduced the **Probabilistic Anomaly Detection** algorithm, which we refer to as PAD. The PAD algorithm is quite general and we use it here to model file system accesses. The PAD algorithm inspects historical feature values in its training data set, and estimates the probability of occurrence of each value using a Bayesian estimation technique. PAD estimates a full conditional probability mass function and thus estimates the relative probability of a feature value in comparison with other feature values and the expected frequency of occurrence of each feature. In other words, we assume that normal events will occur quite frequently, and abnormal events will occur with some very low probability.

In this work, we apply PAD to analyze and model file access data, merged with information about the running processes that invoke such accesses, to train an anomaly detector in much the same fashion as accomplished with RAD. In the same way RAD modeled the actions of running programs vis-a-vis the System Registry we are modeling running processes vis-a-vis the underlying file system.

Here we report the details about the wrapper technology employed in FWRAP, the data and features extracted during a file access, and the results of an experiment to measure the accuracy of the PAD-generated anomaly detector.

The rest of the paper is organized as follows. Section 2 discusses the architecture of the FWRAP system. We describe previous work and what we have added to the FiST system. Section 3 discusses the experimental setup while section 4 presents the results and discusses our findings. Section 5 describes the open problems in anomaly detection research and how this work can be extended, we then finish the paper with some closing thoughts.

2 FWRAPS

Previous work by Zadok [17, 18, 20] proposed a mountable file system for Unix and Windows which would allow additional extensions to the underlying operating system without having to modify kernel level functionality. The FiST technology developed in that work has been extended to provide a security mechanism via file system auditing modules. We implemented a FiST audit module that forms the basis of the FWRAP anomaly detector.

The architecture of the FWRAP system is based on the Adaptive Model Generation (AMG) [7] framework. FWRAP is a system which includes lightweight sensors and detectors on all computers that are audited. The sensors are lightweight and transparent; collecting data should be a background process, unknown to users that shows little or no perceptible decrease in performance of the monitored system.

In the AMG architecture [7] a central data storage is used to collect all of the audit data from each of the monitored systems where modeling is performed as an offline process. AMG was designed to perform experiments that correlate sensors and detectors over different types of data from different sources in real time. This is facilitated by a central store for all of the collected data. A central (or local) analyzer and detector would also be able to analyze audit data and detect intrusions at run time for each of the monitored systems attached to AMG. Additionally, once an intrusion is detected, we would like to be able to mitigate its effects on the target victim, while also taking steps to prevent its propagation on other machines being monitored. The centralized approach of AMG would facilitate this process.

FWRAP represents one of several different types of sensors that one may deploy on a network of monitored systems. AMG incorporates many different types of sensors, such as network sensors, registry sensors (for Windows machines), netstat sensors, and process sensors.

Figure 1 illustrates the architecture that we developed in our lab experiments reported in this report. The figure illustrates the parts of the system which run independently of each other. However it is straightforward to run the proposed FWRAP sys-

tem, or any of the other host-based sensors, entirely as a standalone real time application on a single host.

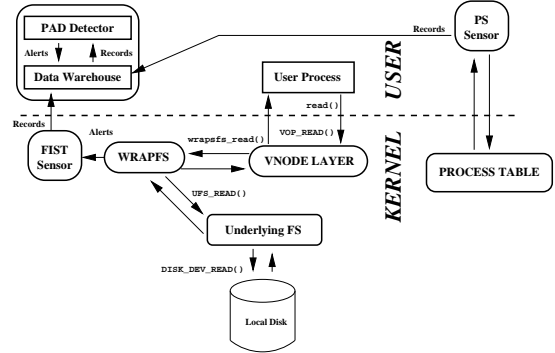


Figure 1: The Architecture of FWRAP IDS

2.1 Requirements

Several requirements were specified for the design of the FWRAP system. The file system sensor had to be lightweight, easily portable to different systems, and complete, in the sense that it is able to monitor all file-system accesses without loss of information. Perhaps the most important requirement is that the system must be transparent to the user.

2.1.1 Lightweight vs Portability

There are two types of file systems we are concerned with; native (kernel-level) and user-level. Requiring a lightweight file system sensor that is also part of the file system implies that it should be a native file system. Native files systems, such as ext2 and fat32, are relatively fast due to the fact that they are kernel-level. However, building or modifying native file systems usually requires recompiling parts of the kernel, which may not be readily available for some platforms.

User-level file systems, on the other hand, do not need a kernel compile, but they do suffer from slower speed, due mostly to context switches from kernel to user levels. A third type of file system, kernel-resident stackable file systems, the subject matter of the research in [19], attempts to combine the speed of native file systems with the ease of use of a user-level file system. This was accomplished through the Vnode Interface.

2.2 Vnode

A Vnode is a pointer to a file system entity and serves as a file system "wrapper", providing an interface to an underlying file system implementation. Calls to the Vnode are not file system specific. Hence, a process that uses a Vnode has no knowledge of the underlying file system implementation, only an interface to that file system. Additionally, it is possible to "stack" Vnodes on top of each other. This concept was first proposed by Rosenthal [13].

Each stacked Vnode thus accesses the Vnode beneath it as if it were accessing a single Vnode. This leaves the user free from worry about the specifics of the underlying file system and allows him to concentrate on the customizations on the file system accesses provided by each Vnode implementation. Since it is kernel-resident, the customized file system will run only slightly slower than a native file system and much faster than a user-level file-system [17].

2.3 FiST

FiST [18] is a high level language designed to aid the development of kernel-resident stackable file systems.

Fistgen is an executable included with FiST that generates C code from FiST code. The C code is then compiled and inserted as a module. At this point, the new file system is ready to be mounted. The advantage of using a high level language like FiST is that a user does not need to worry about the underlying details of the file system he is modifying; he only needs to describe it in FiST. Additionally, FiST code is very easily ported between different systems [18].

Finally, FiST can produce layering which allow fan-in and fan-out of mount points. Fan-out is useful for

load balancing, as well as replicating mount points. Fan-in is useful to directly access lower level mount points, without going through the intermediary file system [19]

In our work, we wrote FiST modules for auditing file accesses for use by FWRAP. This was accomplished by modifying an existing wrapper implemented in FiST called Snoopfs.

2.3.1 Snoopfs

Snoopfs is file system described by the FiST language and included in the FiST package. Snoopfs checks if any non-root user or file owner receives a "permission denied" or "file not found" error. If so, it sends a message to the kernel logger. In our implementation we removed all of the conditionals from Snoopfs and forced it to send all file accesses to the the kernel logger, which we redirect to a file similar to what is described in [20]. We did this by modifying the FiST file which describes the snoopfs file system, and then translating the FiST file to snoopfs.c. Once compiled, the kernel module snoopfs.o was loaded at runtime as a Linux kernel module (using insmod) and the directories were mounted.

One "feature" of stacking is that the underlying mount point is generally directly accessible. This feature could be thwarted by a malicious user or program by simply directly accessing the underlying file-system and avoiding the wrapper and file system logging. We addressed this security concern by limiting access to the underlying mount point by using an "overlay mount". This mount does not allow direct access [20] to the underlying file system. All file-system accesses are thus forced to go through the mount point, forcing the access logging.

2.4 Data Storage

Once all subsystem file accesses are logged in this fashion, its a straightforward matter in our architecture to provide the means of reading from the log, formatting the data and sending it to AMG. A typical line of text sent to the kernel logger by the Snoopfs file system is

```
Mar  9 19:03:14 zeno kernel:
```

snoops detected access
by uid 0, pid 1010, to file cat

This record was generated by having the root user, access a file named 'cat' on a machine named 'zeno'. We modified a C program to format this data and send it to AMG. The communication with AMG follows an XML style exemplified by the following:

```
<rec><Month str>Mar</Month><Day i>9</Day>  
<Time str>19:03:14</Time>  
<IP str>zeno</IP><UID i>0</UID>  
<PID i>1010</PID><File str>cat</File></rec>
```

Records are sent to the central AMG from multiple hosts. Once in the data warehouse, a PAD model of normal behavior is computed for each host. Each model may then be sent to each of the hosts, and all new file accesses are compared to this model locally. Alarms generated by each of the hosts are reported back to AMG.

Over very short training periods, most of the features collected by the FWRAP system such as time of day, date, and ip do not vary, and so are not relevant for building a model of normal system operation. What remains is data that describes only very little information about a file access such as which user, which process, and what file, and does not provide the context in which that access has occurred. Hence, another host based sensor was developed to provide this contextual information to augment information collected by the FWRAP detector.

2.5 The Process Sensor

All Unix hosts provide a /proc virtual file-system. This file-system makes available a simple interface to internal kernel data structures in user-space [3].

The /proc file system exposes dozens of pieces of information about each running process. Organized by the numeric process identifier, PID, this information allows one to monitor the execution of a process from creation to termination. The Basic Auditing Module (BAM) technology built as part of the AMG architecture was used to implement a process sensor. This PS BAM monitors 16 different parameters, including command lines, execution environments, user

ID's and open files. To reduce the data overhead, only processes that exhibit a change in key parameters are reported to the rest of the system. For example, the following PS BAM produced message is a single record showing the status of a web server operating on a particular host. Notice the XML tags that identify the meaning of each piece of data, as well as its data type.

```
<time l>1052016959</time>  
<cmdline s>/usr/local/sbin/bam/sensor_ps  
</cmdline><pid i>1111</pid>  
<ppid i>1</ppid><starttime l>14373</starttime>  
<state c>R</state> <priority i>0</priority>  
<uid i>0</uid> <gid i>0</gid>  
<tty i>0</tty> <wd s>/proc/1111</wd>  
<ctime l>0</ctime> <vsize u>1437696</vsize>  
<sigcatch i>16387</sigcatch>  
<fd s>/dev/console|pipe:[1494]|pipe:  
[1495]|socket:[1536]|/proc/1111/fd|</fd>  
<time l>1052016959</time>  
<cmdline s>syslogd</cmdline>
```

```
<pid i>702</pid> <ppid i>1</ppi d>  
<starttime l>13111</starttime>  
<state c>R</state> <priority i>0</priority>  
<uid i>0</uid> <gid i>0</gid>  
<tty i>0</tty> <wd s>/</wd>  
<ctime l>0</ctime> <vsize u>1507328</vsize>  
<sigcatch i>90113</sigcatch>
```

Data such as this allows for monitoring of activities such as the changing of user ID's, the excessive use of memory or processor time, or the accessing of files. This data enriches the information available from the FWRAP system.

2.6 PAD Detector

The data gathered by monitoring each file access is aggregated and merged with the data gathered by the process monitoring sensor. These two sources of information create a rich set of information that describes in great detail a single file access. Each piece of information may be regarded as a "feature" and

hence each record is treated as a feature vector used by PAD for training a normal model that describes normal file accesses.

PAD models each feature and pairs of features as a conditional probability. A single feature produces a "first order consistency check" that scores the likelihood of observing a feature value at run time. Second order consistency checks score the likelihood of a feature value conditioned on a second feature. Thus, given n features in a training record, PAD generates n first order consistency checks, and $n \times n - 1$ second order consistency checks.

The feature vector available by auditing file accesses and processes has 18 fields of information, some of which may not have any value in describing or predicting a normal file access. For example, one such feature may be the process identifier the PID, associated with the file access. PID's are "arbitrarily" assigned by the underlying OS and in and of themselves have no intrinsic value as a predictor of a file access. Such fields may be dropped from the model.

After training a model of normal file accesses using the PAD algorithm the resultant model is then used at runtime to detect abnormal file accesses. The PAD detector is shown in figure 1. Each file access is monitored by FiST and the PS sensor, a record is encapsulated and provided to the PAD detector, and an alert is generated if the normal model deems the access is abnormal. Alerts are generated via threshold logic on the PAD computed scores.

As shown in Figure 1 the detector exists on the user's level as a background process. Having it run on the user level can also provide additional protection of the system as the sensor can be hard-coded to detect when it is the subject of a process that aims to kill its execution, or to read or write its files.

2.7 FWRAP Features

In order to detect anomalous file accesses, FWRAP generates a model of normal file access activity. A set of 8 features are extracted from each file access. Using these feature values over normal data, a model of normal behavior is generated. This model of normalcy consists of a set of consistency checks applied to the features. When detecting anomalies, the model

of normalcy determines whether the values in the features of the current file access are consistent with the normal data or not. If new activity is not consistent, the algorithm labels the access as anomalous.

Consistency is determined by threshold logic. Each record is tested and scored. If the minimum score of all features is below the user defined threshold, an alert is generated indicating an anomalous record has been detected.

The FWRAP data model consists of 8 features gathered from the FWRAP sensor and process sensor (a subset of the 18 available features). The features used in this study are as follows:

UID This is the user ID running the process

PPID The parent PID of the running process. This feature may be helpful in linking multiple processes launched by one user masquerading as other users.

CMD This is the command line invoking the running process

WD the working directory of a user running the process

TTY The terminal ID number assigned to a user each time they login to the machine. This may be useful since this number is assigned by the OS differently for root and normal users, except when a user suddenly becomes root (after he/she successfully ran root attack).

FILE This is the name of the file being accessed. This allows our algorithm to locate files that are often or not often accessed in the training data. Many files are accessed only once for special situations like system or application installation. Some of these files can be changed to create vulnerabilities.

PRE-FILE This is the concatenation of the three previous accessed files. This feature codes information about the sequence of accessed files of normal activities such as log in, Netscape, statx, etc. For example, a login process typically follows a sequence of accessed files, for example, .inputrc, .tcshrc, .history, .login, .cshdirs, etc

FREQUENCY This feature encodes the access frequency of files in the training records. This value is estimated from the training data and discretized into four categories:

NEVER (for processes that don't touch any file),
FEW (where a file had been accessed only once or twice),
SOME (where a file had been accessed about 3 to 10 times) and
OFTEN (more than SOME).

Alternative discretization of course are possible. Note that we computed the standard deviations from the average frequency of access files from all user processes in the training records to get the cut off for the SOME category. And thus a access frequency falls in FEW or OFTEN categories are often from a file touched by kernel or background process.

Examples of typical records gathered from the sensors with these 8 features are:

```
500 1 /bin/login 1025 / dc2xx10
    725-705-cmdline Some
    1205,Normal

500 1244 ./kmod 1025 /home/ryan
    0.3544951178 0.8895221054
    Never 1253,Malicious
```

The last items are tab separated from the features and represent an optional comment, here used to encode ground truth. The first record with pid=1205 was generated from a normal user activity. The second was captured from an attack running the kmod program to gain root access. The distinction is represented by the labels "normal" and "malicious". These labels are not used by the PAD algorithm. They exist solely for testing performance of the computed models.

Another malicious record is

```
0 1252 sh 1025 /home/ryan
    su meminfo-debug-insmod
    Some 1254,Malicious
```

This record illustrates the results of an intruder who gained root access. The working directory (wd) is still at /home/ryan but the uid now changes to 0.

3 Experiments

We deployed the FWRAP system on a host machine in our lab environment, an Intel Celeron 800MHz PC with 256 RAM, running Linux 2.4 with an ext2 file-system. This host was part of a Test Network Environment which allowed us to run controlled tests of malicious program without worrying about noise from outside the network corrupting our tests. Data was not gathered from a simulator, but rather from runtime behavior of user's on the target machines.

The output of the sensor system described here result in both a FWRAP output and process sensor output. The merging of the two sources of information were implemented by scripts. All other information available in the process sensor such as CPU load, and memory load were ignored. They do not directly relate to file access and were not included in the study.

We collected data for training over 5 days of normal usage from a group of 5 users. Each user used the machine in typical way such as login, editing some files on terminal, checking email, browsing some website, etc and logout. The root user was also used to do some system maintenance as well as some of their routine work tasks.

This resulted in a data set of 313,634 records of 24 megabytes which we used to build a PAD model. This model will be referred to as the "clean model", although we note that it can tolerate some level of noise. The size of the model was 486 megabytes. We address the size of the models computed by PAD in the concluding section.

Once the model was computed, one of the users was chosen to be the "Attacker", who then used the machine for a period of 3 hours. Sometime between this 3 hours period, they ran the exploits from their home account. These exploits are publicly available on the Internet. The user was asked to act maliciously, and to gain root privileges using the attack exploits on hand. Once root control was acquired, the user further misused the host by executing programs which placed back-doors in the system.

The system was monitored while the attacks were run over this 3 hour period. The resultant monitoring produced 51,531 records of 4 megabytes from the

combined FWRAP and PS sensors. These records were then classified with the PAD algorithm using the PAD model.

The analysis of the models was run on a dual processor 1500 MHz with 2 gig of ram. Total time to build the model from the 24 meg of data was three minutes, with memory usage at 14%. Running the model against the 3 hour period to produce multiple thresholds took 2 minutes with 40% of cpu usage and 14% of memory. Once a threshold was selected, the 3 hour period was analyzed in 25 seconds using the PAD program.

3.1 Description of Attacks

We ran three different attacks to measure how well the PAD algorithm could discern malicious attacks from regular usage.

Two of our attacks, ptrace-kmod and kmod, exploit the same weakness in Linux 2.2 and 2.4 (see <http://www.securitybugware.org/Linux/6072.html>). The weakness is root-exploitable because of how the kernel handles module features. When a kernel feature is needed in a process, the kernel spawns a new process with euid and egid set to 0. Ptrace() can then immediately be used to attach to the new process and run arbitrary code with root access. Our exploits both use the weakness to run a root shell (in somewhat different ways).

Once root access is attained, an attacker would want to hide any record of their existence, as well as set up an easy way to remotely control the computer without giving away the compromised state of the machine. A rootkit is usually used. We used the 't0rn' rootkit, which is a pre-compiled rootkit widely available online. t0rn contains binary versions of the following tools:

```
/usr/bin/du
/usr/bin/find
/sbin/ifconfig
/bin/login
/bin/ls
/bin/netstat
/bin/ps
/usr/bin/sz
/usr/bin/top
```

These "tools" are actually hacked and hide any evidence that the machine is compromised. Since it is assumed the rootkit is installed on a compromised machine, it copies these files to their 'correct default' location, hence they will be in the PATH.

T0rn also copies some of its files into /usr/src/.puta, and starts a sshd to allow the attacker to reconnect to the machine.

4 Results

This section describes the results of our experimental tests. The PAD algorithm evaluates each record output by the sensors. By varying the threshold for the consistency scores we were able to vary the detection rate and false positive rate.

We ran the test data against the trained PAD model, producing 64 scores for each consistency check for each record (8 first order + 8x7 second order). The minimum score over all consistency checks is then tested against the threshold. If a record contain consistency scores below the threshold, we consider it is abnormal.

An example of the PAD output with threshold = -2.9 is as follows:

```
0 1252 sh 1025 /home/ryan su meminfo-debug-
insmod Some :
2.385726 -1.278873 -1.991977 -2.675945 1.993751 -0.175890
-2.944535 2.397863 2.342325 2.170739 2.133449 0.438297
0.931558 2.224713 -1.266756 4.864607 1.170862 1.119658
1.170979 1.094596 -0.845582 -2.169715 1.037885 4.609045
1.082764 0.930569 0.176130 -2.589351 -2.924091 -0.127774
0.433996 2.193895 0.628596 0.897942 0.133137 0.054751
2.297798 2.899492 4.706853 6.600091 0.587283 4.013465 -
0.595098 0.721933 0.003740 0.783238 0.419870 1.098572
0.998816 -0.168477 0.722015 0.001877 0.766187 0.415934
1.098588 0.960322 -3.103159 -2.414797 -3.515270 -0.134781 -
0.520264 0.693147 1.385568 : 1254,Malicious : 8,33,65,67
```

The sequence of numbers appearing at the end displays which consistency checks are below the threshold. We inspected these for each record marked as an anomaly. We learned that the features UID, FILE and FREQUENCY are the most important predictors. Most inconsistency scores (i.e scores below the

threshold) are generated by those features

Based on the results from PAD output. We generated detection rates and false positive rates on both a per record and per process basis.

4.1 Per record

In per record, we defined "Detection Rate" as the percentage of records labeled "malicious" that produced PAD scores below the threshold. The "False Positive Rate" is the percentage of records labeled "normal" that likewise produced PAD scores that were below the threshold.

We applied different thresholds, illustrated in **Table 1**, producing different detection rates and false positive rates. The rates are then plotted using an ROC curve displayed in **figure 2**.

Threshold	Detection Rate	False Positive
-6.9	0.000000	0.000504
-6.8	0.034482	0.000679
-6.4	0.068965	0.000795
-6.1	0.112068	0.000834
-5.9	0.155172	0.001285
-5.8	0.181034	0.001028
-5.5	0.198275	0.002833
-5.4	0.250000	0.002969
-5.3	0.267241	0.003609
-5.1	0.284482	0.004560
-5.0	0.293103	0.005375
-4.7	0.318965	0.006442
-4.6	0.405172	0.007044
-4.5	0.422413	0.007315
-4.4	0.431034	0.007665
-4.2	0.439655	0.008363
-4.1	0.448275	0.009392
-4.0	0.474137	0.010479
-3.6	0.517241	0.018319
-3.5	0.586206	0.018610
-3.1	0.603448	0.024373
-2.9	1.000000	0.027633

Table 1: Varying the threshold in Per Record detection and its effect on Detection and False Positive Rate

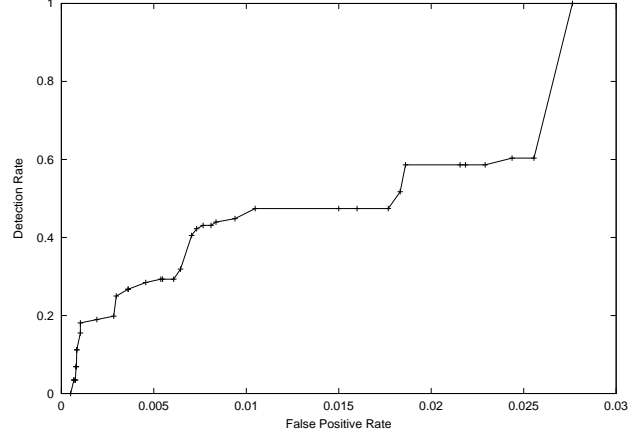


Figure 2: Per Record ROC curve for Detection Rate versus false Positive Rate

4.2 Per process

There were 521 processes generated from the 3 hours period experiment. 46 processes were generated during the attack period (i.e. time between the attacker launched the exploits and ran Trojan software after he gained root). However, some of the processes generated during this time were not purely from the attack. (For example, processes from the sensors or from system logs accessed during this time). We eliminated those processes and classified the list of 39 processes as emanating from attacks.

A process is identified as malicious if all of its records are labeled anomalous by PAD. An alternative decision process may apply. For example, a process might be considered malicious if it generates one anomalous record or some threshold number of anomalous records. The ROC curve presented is for the all or none strategy.

We define the "Detection Rate" in the per process experiment to be the percentage of processes with labeled malicious records that were detected. The False Positive Rate is the percentage of normal processes with normal records which were falsely detected as attacks.

Table 2 illustrates the results of detection rates vs. false positive rates on a Per Process basis. The

rates are then plotted using an ROC curve displayed in **Figure 3**.

Table 3, is sorted in order of consistency check score. This table shows some of the highest and lowest scores of 18 processes generated from one of the attacks and about 30 processes from normal usage before and after the attack.

In our tests, the false positive rates classified on the per process basis were higher than the false positive rates on a per record basis. This was found to be due to the fact that we were constrained by the number of processes (only 521 processes over 51,531 records) in our experiments. Moreover, the number of records in each process is not equally distributed, skewing the distribution somewhat. Consequently, normal processes that have only one or two records are more likely to cause false positive alerts than a process which has more records. As we can see from table 3, those processes higher on the list classified as "Normal" with only one record are among those that caused the false positive (i.e. when we applied the threshold = -2.9 resulting in 100% detection rate)

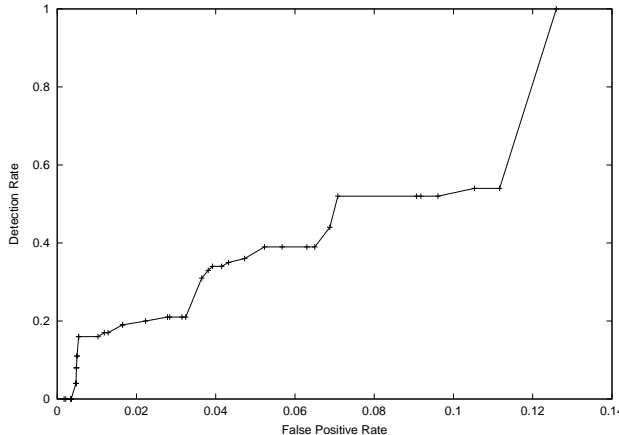


Figure 3: Per Process ROC curve for Detection Rate versus false Positive Rate

For the test case studied here, it appears that the good detection performance indicates that file accesses exhibit fairly consistent behavior. Future work includes a range of studies in different environments. We address some of the open research questions this

Threshold	Detection Rate	False Positive
-6.9	0.00	0.0035
-6.8	0.04	0.0047
-6.4	0.08	0.0048
-6.1	0.11	0.0050
-5.9	0.16	0.0054
-5.5	0.17	0.0118
-5.3	0.19	0.0164
-5.1	0.20	0.0223
-5.0	0.21	0.0278
-4.6	0.31	0.0364
-4.5	0.33	0.0381
-4.4	0.34	0.0391
-4.2	0.35	0.0432
-4.1	0.36	0.0473
-4.0	0.39	0.0523
-3.6	0.44	0.0687
-3.5	0.52	0.0708
-3.1	0.54	0.1052
-2.9	1.00	0.1259

Table 2: Varying the threshold in Per Process detection and its effect on Detection and False Positive Rate

line of work has revealed in our concluding remarks.

5 Conclusions

By using file system access on a Linux system, we are able to label all processes as either attacks or normal, with high accuracy and low false positive rate. For the experiments performed in this study, we have shown that the file system is a valuable auditing point on a IDS system.

The work reported in this paper is an extension of our research on anomaly detection. The PAD algorithm has been previously applied to network traffic, as well as the Windows Registry, as described earlier in this paper. There are a number of open research issues that we are actively pursuing. These issues involve calibration, pruning, feature selection, concept (or environment) drift, and correlation.

Briefly, we seek automatic means of building

anomaly detectors for arbitrary audit sources that are well behaved, and are easy to use. As it now stands, AD technology is a bit of a black art. To be sure, the use of anomaly detectors in security applications is still early and much science yet needs to be done to understand their fundamental value and limitations, and their practical implementation and deployment.

With respect to calibration, one would ideally like a system such as FWRAP, or RAD, to self-adjust its thresholding to minimize false positives while revealing sufficient evidence of a true anomaly indicative of an abuse or an attack. It is important to understand, however, that AD models should be considered part of the evidence, and not be depended upon for the whole detection task. This means AD outputs should be correlated with other indicators (even other AD models computed over different audit sources or different features or different AD algorithms) in order to confirm that an attack is truly occurring. Thus, it would be a mistake to entirely focus on a well calibrated threshold for an AD simply to reduce FP's. It may in fact be a better strategy to generate more alerts, higher numbers of FP so that correlating alerts with other evidence would reveal the true attacks that otherwise would go undetected (had the AD threshold been set too low).

In the experiments run to date PAD produces models that are expensive in memory. There are several enhancements that can be implemented to alleviate its memory consumption requirements. These include pruning of features after an analytical evaluation that would indicate no possible consistency check violation would be possible. Furthermore, most of the memory structures used by the current implementation of PAD can be reimplemented using Bloom Filters[4] to generate considerable compression advantages.

Finally, two questions come to most minds when they first study anomaly detectors of various kinds; how long should they be trained, and when should they be retrained. These issues are consistently revealed due to a common phenomenon, concept (or environment) drift. What is modeled at one point in time represents the "normal data" for that data drawn from the environment for that period of time,

but the environment may change (either slowly or rapidly) which necessitates a change in model.

The particular features being drawn from the environment have an intrinsic range of values; PAD is learning this range, and modeling the inherent "variability" of the particular feature values one may see for some period of time. Some features would not be expected to vary widely over time, others may be expected to vary widely. PAD learns this information (or an approximation) for the period of time it observes the data. But it is not known if it has observed enough.

What is needed is a decision procedure, and a feedback control loop, that provides the means to determine whether PAD has trained enough, and deems when it may be necessary to retrain a model if its performance should degrade.

We intend to continue this line of research using the various audit sources we have at our disposal, the FWRAP AD sensor, the focus of this paper, and the RAD and Network traffic sensors that employ the PAD algorithm.

Another interesting open question is how one may protect the FWRAP system from being tampered with. By storing the model on the kernel level, underneath the normal mount, the system would appear invisible to the overlying file system, allowing the model to be protected from malicious users. It remains to be seen how expensive an operation this may be.

References

- [1] Frank Apap, Andrew Honig, Shlomo HersHKop, Eleazar Eskin, Salvatore J. Stolfo. Detecting Malicious Software by Monitoring Anomalous Windows Registry Accesses. Fifth International Symposium on Recent Advances in Intrusion Detection (RAID-2002). Zurich, Switzerland: October 16-18, 2002.
- [2] Balzer, R. Mediating Connectors 19th IEEE International Conference on Distributed Computing Systems Workshop, 1994.

- [3] Bauer, Bodo and Terrehon Bowden. The /proc Filesystem. <http://www.linuxhq.com/kernel/v2.2/1/Documentation/proc.txt>. January 1999.
- [4] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, Vol 13, Issue 7 (July 1970)
- [5] Forest, St., A. Hofmeyr, A. Somayaji and T. A. Longstaff. A sense of self for unix processes, pages 120-128, IEEE Computer Society, 1996.
- [6] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. Journal of Computer Security, 6:151–180, 1998.
- [7] Andrew Honig, Andrew Howard, Eleazar Eskin, Sal Stolfo. Adaptive Model Generation: An Architecture for Deployment of Data Mining-based Intrusion Detection Systems. Data Mining for Security Applications. Kluwer 2002.
- [8] C. Ko, G. Fink, and K. Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. 10th Annual Computer Security Applications Conference, pages 134– 144, December 1994
- [9] K M.C. Tan and Roy A. Maxion. Why 6? Defining the Operational Limits of stide, an Anomaly-Based Intrusion Detector.
- [10] Wenke Lee, Sal Stolfo, and Phil Chan. Learning Patterns from Unix Process Execution Traces for Intrusion Detection. AAAI Workshop: AI Approaches to Fraud Detection and Risk Management, July 1997
- [11] Okena Incore Architecture, <http://www.okena.com/>
- [12] Description of the Microsoft Windows Registry. <http://support.microsoft.com/?kbid=256986>
- [13] Rosenthal. Evolving the Vnode Interface. Usenix Proceedings, pg 107-118, 1990.
- [14] Sana Security Profile Technology <http://www.sanasecurity.com>
- [15] Kevin Timm, Strategies to Reduce False Positives and False Negatives. <http://online.securityfocus.com/infocus/1463>
- [16] D. Wagner and P. Soto. Mimicry attacks on host based intrusion detection systems. Ninth ACM Conference on Computer and Communications Security, 2002.
- [17] Erez Zadok and Ion Badulescu. A Stackable File System Interface For Linux. LinuxExpo 99. May 1999.
- [18] Erez Zadok and Jason Nieh. FiST: A Language for Stackable File Systems. Usenix Technical Conference. June 2000
- [19] Erez Zadok, Ion Badulescu, and Alex Shender. Extending File Systems Using Stackable Templates. Usenix Technical Conference. June 1999. page 7.
- [20] Erez Zadok. Stackable File Systems as a Security Tool. Columbia U. CS TechReport CUCS-036-99. December 1999. page 5.

PID	Program Name	Total Records	Minimum Score	Maximum Score	Classification
1252	/tmp/x0x	1	-6.882326	6.600091	Malicious
1249	as	1	-6.882326	6.600091	Malicious
1306	w	1	-6.882326	7.275401	Malicious
1248	/usr/lib/gcc-lib/i386-redhat-linux/2.96/cc1	2	-6.476861	6.600091	Malicious
1246	gcc	2	-6.189179	6.600091	Malicious
1324	/usr/lib/gcc-lib/i386-redhat-linux/2.96/cpp0	3	-6.177048	8.527128	Malicious
1252	./kmod	1	-5.966035	6.600091	Malicious
1254	su	10	-5.819753	8.725036	Malicious
1253	/sbin/modprobe	2	-5.755573	7.538759	Malicious
1328	/bin/bash	1	-5.050946	7.538759	Malicious
1317	ftp	10	-4.685101	5.148742	Malicious
1319	/bin/sh	1	-4.503447	7.275401	Malicious
1319	/usr/lib/sa/sadc	1	-4.503447	4.065055	Malicious
1314	rm	2	-4.036982	7.275401	Malicious
1006	crond	5	-3.686345	6.20679	Malicious
1245	sh	1	-3.676769	6.600091	Malicious
1276	-bash	1	-2.924091	7.275401	Malicious
1126	/sbin/mingetty	1	-7.568797	7.538759	Normal
737	rpc.statd	1	-7.314065	6.20679	Normal
1093	/usr/local/sbin/bam/sensor_ps	191	-6.745411	8.940659	Normal
1618	sendmail:	3	-6.671864	4.531495	Normal
1350	whereis	2	-6.476861	7.275401	Normal
1299	/usr/bin/python	28	-6.266399	6.600091	Normal
1205	/bin/login	131	-5.570593	7.538759	Normal
1511	usleep	1	-5.177578	7.538759	Normal
1357	tput	1	-5.12645	7.275401	Normal
1315	/usr/bin/gnome-terminal	1	-4.647408	7.275401	Normal
1147	update	1	-4.631034	7.538759	Normal
1517	nautilus	4	-4.241943	7.275401	Normal
939	xinetd	1	-3.948424	7.538759	Normal
970	gpm	1	-3.605181	7.538759	Normal
1356	deskguide-applet	2	-3.56814	6.600091	Normal
1291	magicdev	5	-3.436208	7.275401	Normal
1354	tasklist_applet	3	-3.407676	6.600091	Normal
1196	-bash	1	-2.924091	7.275401	Normal
1229	/bin/bash	1	-2.924091	7.275401	Normal
1229	/bin/sh	3	-2.924091	7.275401	Normal
1199	cat	2	-2.924091	7.275401	Normal
1440	/usr/bin/gnome-session	6	-2.924091	7.275401	Normal
1215	grep	1	-2.899781	7.275401	Normal
1509	gconfd-1	7	-2.862862	7.538759	Normal
1210	ls	1	-2.671522	7.275401	Normal
1304	oafd	1	-2.532905	7.538759	Normal
1484	sawfish	45	-1.474298	7.275401	Normal
1132	/usr/local/sbin/bam/listen	131	-1.566555	7.538759	Normal
702	syslogd	817	-1.297392	7.538759	Normal
689	klogd	112	-1.297392	7.538759	Normal
1222	/etc/X11/X	1	-1.99501	7.275401	Normal

Table 3: Trace of both normal and attack data. In this attack, the attacker ran kmod program (pid 1252) to gain root control of the machine. After compromising the system the attacker executed some normal processes to cover the attack. Example: 'w' was run to see if anyone else currently logged into the system (pid 1306). 'su' to change to new user (pid 1254), 'ftp' to an outside network to get rootkits (pid 1317). Next the attacker compiled and ran the tool and script (pid 1248, 1324, 1246). Finally, the attacker removed the logs file (pid 1314)