Host-based Anomaly Detection Using Wrapping File Systems [⋆]

Shlomo Hershkop, Linh H. Bui, Ryan Ferster, and Salvatore J. Stolfo

Columbia University, New York, NY 10027, USA
`<shlomo,lhb2001,rlf92, sal>@cs.columbia.edu`

Abstract. We describe an anomaly detector, called FWRAP, for a Host-based Intrusion Detection System that monitors file system calls to detect anomalous accesses. The system is intended to be used not as a standalone detector but one of a correlated set of host-based sensors. The detector has two parts, a sensor that audits file systems accesses, and an unsupervised machine learning system that computes normal models of those accesses.We report on the architecture of the file system sensor implemented on Linux using the FiST file wrapper technology and results of the anomaly detector applied to experimental data acquired from this sensor. FWRAP employs the Probabilistic Anomaly Detection (PAD) algorithm previously reported in our work on Windows Registry Anomaly Detection. The detector is first trained by operating the host computer for some amount of time and a model specific to the target machine is automatically computed by PAD, intended to be deployed to a real-time detector. In this paper we describe the feature set used to model file system accesses, and the performance results of a set of experiments using the sensor while attacking a Linux host with a variety of malware exploits. The PAD detector achieved impressive detection rates in some cases over 95% and about a 2% false positive rate when alarming on anomalous processes.

Keywords: Host-Based, Anomaly Detection, File System, Wrapping

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].

[⋆] This work has been supported in part by a contract from DARPA, Application-layer IDS, Contract No. F30602-00-1-0603.

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 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 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 architected to be as lightweight as possible. Second, the adaptability and extensibility of these systems complicates their use as updates or patches 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 to generate exploits that appear normal when executed.

We have taken an alternative view of host-based anomaly detection. Anomalous process executions (possibly those that are malicious) may be detected by monitoring the trace of events as they appear on attempts to alter or damage the machine's permanent store. Thus, a malicious attack that alters only run-time memory would not necessarily be detected by this monitor, while the vast majority of malicious attacks which do result in changes to the permanent store of the host might leave a trace of anomalous file system events. 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).

The Windows **R**egistry **A**nomaly **D**etector [1], (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 the case of Unix platforms there is no central registry to monitor so we focus the 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. A typical user or application will not behave in the same manner as a malicious exploit, and hence the behavior of a malicious exploit is likely able to be detected as an unusual or unlikely set of file system accesses.

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 auditing the Windows 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. Each record encodes a set of feature values that describe the processes, the file and directory, and the file characteristics. 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.

Our initial focus here is to regard the set of file system access records as a database, and to model the likely records in this database. Hence, any record analyzed during detection time is tested to determine whether it is consistent with the database of training records. This modeling is performed by the **P**robabilistic **A**nomaly **D**etection algorithm (PAD) introduced in our work on the Windows registry. The PAD algorithm inspects 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 conditioned on other feature values and the expected frequency of occurrence of each feature. One of the strengths of the PAD algorithm is that it also models the likelihood of seeing new feature values at run-time that it may not have encountered during training. 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 model file access data, merged with information about the running processes that invoke such accesses. We report on experiments using alternative threshold logic that governs whether the detector generates an alarm or not depending upon the scores computed by PAD,and as well the number of alerts generated by a distinct process. We compare the accuracy of the detector for the cases where one anomalous record is generated by a process, and an alternative strategy that raises an alarm when some percentage of records created by a process are deemed anomalous. We first provide the details of the wrapper technology employed in FWRAP and the data and features extracted during a file access.

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.

2 FWRAP System Architecture

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 audit sensor.

FWRAP represents one of several different types of sensors that one may deploy and correlate on a monitored host system. The correlation of *multiple host-based sensors* is beyond the scope of this paper. (See [22] for details of this approach, and a broader set of prior work dealing with network-based sensor correlation such as [23]).

Figure 1 illustrates the architecture that we developed as a standalone real time application on a single Linux host.

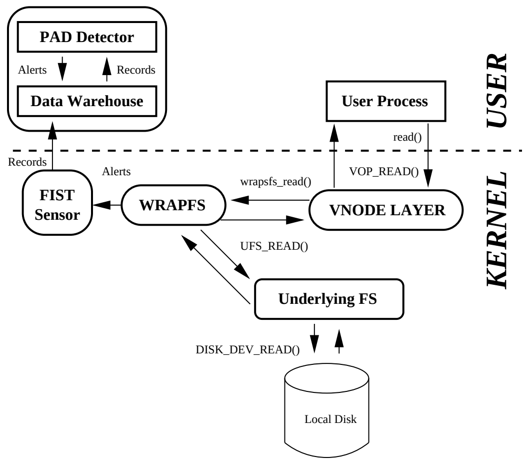

Fig. 1. The Architecture of FWRAP IDS

2.1 Requirements

Several requirements drove 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.

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, nor appreciated by end users.

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/her 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 by Erez Zadok in his thesis research 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.

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 to provide the means of reading from the log, formatting the data and sending it to the PAD module for analysis. A typical snippet of a line of text sent to the kernel logger by the Snoopfs file system is

```
Mar  9 19:03:14 zeno kernel:
snoopfs detected access by uid 0, pid 1010, to file cat
```

This record was generated when the root user accesses a file named ’cat’ on a machine named ’zeno’. We modified a C program to format this data for PAD exemplified by the following (partial) record.

```
<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>
```

2.5 PAD Detector

The data gathered by monitoring each file access is 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. PAD also models the likelihood of observing a new feature value at run-time that was not observed during training. Second order consistency checks score the likelihood of a particular feature value conditioned on a second feature. Thus, given n features in a training record, PAD generates n first order consistency checks, and $n * n - 1$ second order consistency checks. Although it is possible to use higher order consistency checks, the computational overhead and space constraints make it infeasible for the current implementation of PAD.

The feature vector available by auditing file accesses 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, 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. As an expediency such fields may be dropped from the model. Only 7 features are used in the experiments reported in this paper as detailed in the next section.

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, a record is encapsulated and provided to the PAD detector, and an alert is generated if the normal model deems the access is abnormal (or some number of accesses generated by a single process). Alerts are generated via threshold logic on the PAD computed scores.

As shown in Figure 1 the detector runs on the user 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. (Self-protection mechanisms for FWRAP are beyond the scope of this paper.)

2.6 FWRAP Features

The FWRAP data model consists of 7 features extracted or derived from the audit data provided by the FWRAP sensor.

Several of the features describe intrinsic values of the file access, for example, the name of the file, and the user id. We also encode information about the characteristics of the file involved in the access, specifically the frequency of touching the file. This information is discretized into a few categories rather than represented as a continuous valued feature. We generally follow a strategy suggested by the Windows OS. Within the add/change applications function of control panel in Windows, the frequency of use of an application is characterized as "frequently", "sometimes" and "rarely". Since it is our aim to port FWRAP to Windows (exploiting whatever native information Windows may provide and to correlate FWRAP with RAD), we decided in this experimental implementation of FWRAP on Linux to follow the same principle for the Unix file system but with a slightly finer level of granularity than suggested by Windows. Hence, we measured file accesses over a trace for several days, and discretized the frequency of use of a file into the four categories as described below.

The entire set of features used by in this study to model file system accesses are as follows:

UID This is the user ID running the process
WD the working directory of a user running the process
CMD This is the command line invoking the running process
DIR This is the parent directory of the touched file.
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 during an exploit.
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 such as .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:
1. NEVER (for processes that don't touch any file)
2. FEW (where a file had been accessed only once or twice)
3. SOME (where a file had been accessed about 3 to 10 times)
4. OFTEN (more than SOME).

Alternative discretization of course are possible. We computed the standard deviations from the average frequency of access files from all user processes in the training records to define the category ranges. An access frequency falls into the range of FEW or OFTEN categories often occurs for a file touched by the kernel or a background process.

Examples of typical records gathered from the sensors with these 7 features are:

```
500 /home/linhbui login /bin dc2xx10
725-705-cmdline Some    1205,Normal

500 /home/linhbui kmod /Linux_Attack kmod
1025-0.3544951178-0.8895221054 Never    1253,Malicious
```

The last items (eg., "1253,Malicious) are tab separated from the feature values and represent an optional comment, here used to encode ground truth used in evaluating performance of the detector. 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 /home/linhbui sh /bin 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/linhbui but the UID now has changed to 0. A record of this nature ought to be a low probability event.

3 Experiments

We deployed the FWRAP audit sensor on a "target host" machine in our lab environment, an Intel Celeron 800MHz PC with 256 RAM, running Linux 2.4 with an ext2 file-system. The data gathered by the sensor was logged and used for experimental evaluation on a separate machine. The latter test machine is far faster and provided the means of running multiple experiments to measure accuracy and performance of the PAD implementation. The target host was part of a Test Network Environment which allowed us to run controlled executions of malicious programs without worrying about noise from outside the network corrupting our tests, nor inadvertently allowing leakage of an attack to other systems. Data was not gathered from a simulator, but rather from runtime behavior of a set of users on the target machine.

We collected data from the target host for training over 5 days of normal usage from a group of 5 users. Each user used the machine for their work, logging in, editing some files on terminal, checking email, browsing some website, etc. The root user performed some system maintenance as well as routine sysadmin tasks.

The logged data resulted in a data set of 275,666 records of 23 megabytes which we used to build a PAD model on the other "test machine". This model will be referred to as the "clean model", although we note that PAD can tolerate some level of noise. The size of the model was 486 megabytes prior to any pruning and compression. We address the size of the models computed by PAD in the concluding section.

Once the model was computed, one of the users on the target machine volunteered to be the "Attacker", who then used the target machine for 3 experiments each lasting from 1 to 3 hours. The malicious user 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. The resultant monitoring produced records from the FWRAP sensors. These records were then tested and scored by the PAD model.

The PAD analysis was run on the test machine, a dual processor 1500 MHz with 2GB of ram. The total time to build the model of the 23 MB of training data was three minutes, with memory usage at 14%. Once, the model was created, we ran the model against the test data from the 3 experiments, while varying the thresholds to generate a ROC curve. Each detection process took 15 seconds with 40% of CPU usage and 14% of memory.

These performance statistics were measured on the test machine, not on the target host where the data was gathered. This experimental version was implemented to test the efficacy of the approach and has not been optimized for high efficiency and minimal resource consumption for deployment on the target machine. Even so, the analysis of the computational performance of the sensor measured during the experiments indicates that although training the model is resource intensive, run-time detection is far less expensive. A far more efficient professionally engineered version of FWRAP would reduce resource consumption considerably. That effort would make sense only if the sensor achieves the goal of detecting anomalous events indicative of a security breach.

It should be noted that the RAD sensor/detector using PAD running on Windows has been upgraded to run exceptionally fast, with a very small amount of memory and CPU footprint. For the current Windows implementation of PAD the 23 MB file can train the model in about 12 seconds at 95% CPU time on a Celeron 2 GHZ machine.

The initial implementation of the PAD algorithm on windows for the RAD sensor consumed 18 MB to store the PAD model trained on 23 MB of data. The re-engineered implementation now requires less than 1 MB and the run-time detector consumes at most 5% of CPU, barely discernible. This newer implementation of PAD is being ported to Linux so that FWRAP will also be far more efficient than the present prototype reported in this paper. The version reported here is the first proof of concept implementation without the performance enhancements implemented on the Windows platform.

3.1 Description of Attacks

We used three different exploits and three Trojan programs during the 3 experiments to measure how well the PAD algorithm could discern malicious attacks from normal file accesses.

Two of the 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. A bug in the kernel module loader code could allow a local user to gain root privileges. When a kernel feature is needed in a process, the kernel spawns a new modprobe 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. The exploits use this vulnerability to run a root shell (in somewhat different ways).

The third attack is a *Kanji* emulator for the console. It exploits the buffer overflow vulnerability in the command line parsing code portion of the kon program up to and including Linux version 0.3.9b. This vulnerability, once appropriately exploited, leads to local users being able to gain elevated root privileges. *kon2root* is the kon buffer overflow code to attack Red Hat Linux.

Once root access is attained, an attacker would use backdoor programs to remotely control the target computer without giving away the compromised state of the machine.

The first backdoor, *audpbackdoor*, is a client/server software program written in Perl. Once attackers gain root control, they execute the server with root access. It will then open port 520 by default for a remote client to connect to the target machine. The client will then have root access.

The second backdoor, *blackhole*, provides remote access with the access rights of the user that executed it. Therefore, once attackers execute it from root after compromising the system, they can telnet to the system on an open port set up by blackhole to have root privileges.

The third backdoor is a simple script to gain root privileges after the user with uid/gid 0 (root) logs in. All it does is make the file "touch" called /tmp/mcliZokhb, and either copy /bin/sh to /tmp/mclzaKmfa, or use Trojan to make a secure password protected shell. These files act like a password. The bash script will set /tmp/mclzaKmfa suid (+s) when /tmp/mcliZokhb exists.

Other than trojan and root exploits, to hide any record of their existence, as well as set up an easy way to remotely control the target 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 the experiments. The PAD algorithm evaluates each record output by the sensor by comparing the PAD score to a threshold value. We recorded the detection rate and false positive rate of the detector over the test records by varying the threshold for the consistency scores and plot ROC curves of the results.

We ran the test data against the trained PAD model, producing 49 scores for each consistency check for each record (7 first order + 7x6 second order). The minimum score over all consistency checks is then tested against the threshold. If the minimum score is below the threshold, an alert is generated.

An example of the PAD output with threshold = 0.1 is as follows:

0 /home/linhbui sh /bin su meminfo-debug-insmod Some : 8.873115 8.39732 7.69225 4.057663 0.485905 0.323076 6.527675 8.34453 7.464175 3.727299 0.0 0.0 5.971592 8.344 7.464175 3.727299 0.0 0.0 5.79384 7.45713 7.454335 4.060443 0.0 0.0 4.97851 3.732753 3.723643 4.039242 0.0 0.0 3.982627 0.458721 0.371057 0.439515 0.14842 0.0 0.221132 0.302689 0.20546 0.258604 0.090151 0.0 0.067373 5.978326 5.81323 5.015466 4.060443 0.0 0.0 : 1254,Malicious : **11,12,17,18,23,24,29,30,36,42,48,49**

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. We call these *critical features*. Most inconsistency scores (i.e scores below the threshold) are generated by the inconsistent value of critical features (i.e. first order consistency check) or from the conditional features (i.e. second order consistency checks) involving these features.

These three features are quite sensible. The user identity associated with a certain type of file generated anomaly alerts. (These are not the only alerts that were generated.) This suggests that FWRAP not only may detect anomalous file system accesses generated by malware execution, but it may also be useful as a detector of *masqueraders and insider abusers* [24–26]. Hence, FWRAP may provide evidence of these other nefarious activities.

We generated and display in the accompanying figures detection rates and false positive rates on both a per record and per process basis. The strategy here is that an otherwise non-malicious process may generate some abnormal file accesses, insufficient in number to alarm on the process. Hence, the per process scoring is based on measuring the number of records generated by a process at run-time until some critical number of such alerts are generated by PAD.

In real time operation of FWRAP the sensor would therefore track the number of alerts a process generates, and whenever this number exceeds some threshold, an alarm would be raised. It is this measure we report in the following figures.

4.1 Per record

In the case of per record scoring and alerting, we define "Detection Rate" as the percentage of all records (irrespective of the processes that generated those records) labeled "malicious" that produced PAD scores below the threshold. The "False Positive Rate" is the percentage of all test records labeled "normal" that likewise produced PAD scores that were below the threshold.

We illustrate the different detection rates and false positive rates over different threshold settings using ROC curves, with the data used to generate these plots in accompanying tables.

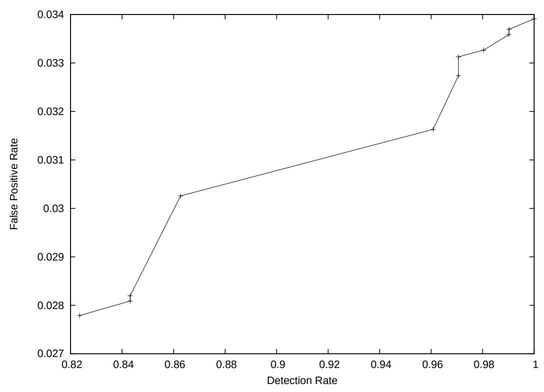

Fig. 2. Per Record ROC curve for Detection Rate versus false Positive Rate in experiment 1

Threshold	Detection Rate	False Positive
1.3	1.000000	0.033910
1.2	0.990196	0.033695
1.1	0.990196	0.033587
1.0	0.980392	0.033266
0.9	0.970588	0.033128
0.7	0.970588	0.032739
0.6	0.960784	0.031630
0.5	0.862745	0.030258
0.4	0.843137	0.028196
0.2	0.843137	0.028088
0.1	0.823529	0.027787

Table 1. Varying the threshold in Per Record detection and its effect on Detection and False Positive Rate. Experiment 1, number of records: 92,003, number of malicious records: 102

Threshold	Detection Rate	False Positive
2.1	0.995441	0.084994
1.5	0.995441	0.077038
1.4	0.986322	0.074712
1.3	0.986322	0.074058
1.2	0.984802	0.071630
1.1	0.965046	0.069370
1.0	0.965046	0.066468
0.9	0.879939	0.063561
0.8	0.797872	0.061900
0.7	0.797872	0.059658
0.6	0.589666	0.050107
0.5	0.530395	0.041816

Table 2. Experiment 2: per-record detection. Number of records: 17338, number of malicious records: 658

4.2 Per process

A process is identified as malicious if more than some minimum number of records it generates is scored as an anomaly. This number is a second threshold. We tested varying threshold levels applied to the PAD scores under different thresholds governing the number of anomalous records used to generate a final alert.

The decision process is evaluated by varying the percentage of anomalous records that are generated by a process in order to raise an alert. For example, a process might be considered malicious if it generates one anomalous record, or

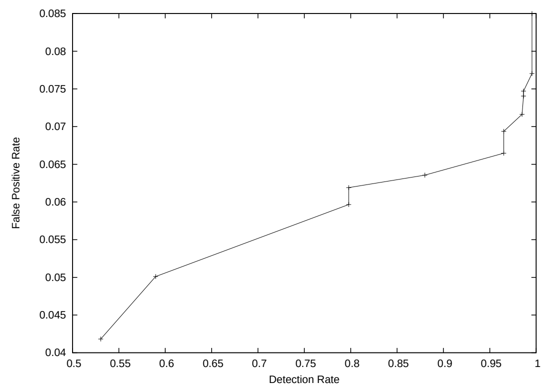

Fig. 3. Per Record ROC curve for Detection Rate versus false Positive Rate in experiment 2

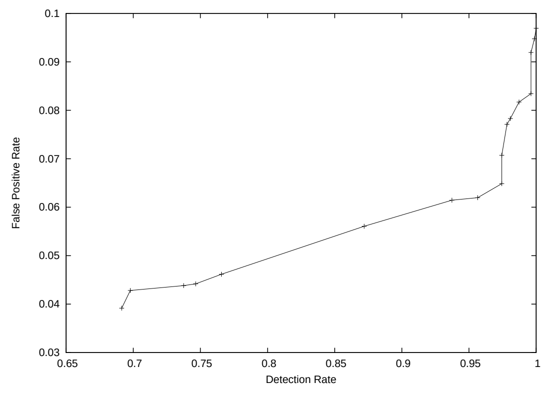

Fig. 4. Per Record ROC curve for Detection Rate versus false Positive Rate in experiment 3, total records: 14148, number of malicious records: 781

all of its records are anomalous, or some percentage of its records are deemed anomalous.

The latter strategy requires some discussion. At any point during run-time, we cannot know what records a process may yet generate in the future. Therefore, we model a decision process that keeps track of the records generated by a single process and whenever a specific percentage of the prior records scored as anomalies, the process would generate an alert. For example, at any moment in time, if 50% of the records of a an active process have generated an anomaly score below the PAD score threshold, FWRAP generates an alert. We vary this percentage in the following experiments from 10% to 80%.

The results indicate that the per-process detection logic provides better detection rate and lower false positive rates compared to the per-record case. This is not surprising as the same results were discovered in the RAD experiments [1]. This leads to the observation that a malicious process typically generates a considerable number of anomalous events, while many normal processes occasionally generate a few anomalous events.

Table 3, **Table 4**, and **Table 5** detail the results on a Per Process basis. The ROC curve are displayed in the accompanying figures. Note that the results from experiment 1 are relatively better than those from experiments 2 and 3. The primary reason concerns that amount of training performed during the different experiments. Experiment 1 had far more training data establishing the perhaps obvious point that as the sensor models more events its detection accuracy increases.

In the first experiment implemented on the target machine, there were 5,550 processes generated during the 3 hour period. 121 processes were generated during the attack period (i.e. the time between the initial launching of the attacking exploits and the Trojan software execution after he gained root access). However, only 22 processes generated during this time were spawned by the attack.

Threshold	**Detection Rate**	**False Positive**
1.1	1.0	0.027090
1.0	0.954545	0.02727
0.9	0.909091	0.026690
0.8	0.909091	0.026345
0.7	0.863636	0.025927
0.6	0.863636	0.025381
0.5	0.772727	0.023363
0.4	0.772727	0.021145
0.3	0.727273	0.020981
0.2	0.727273	0.020909
0.1	0.727273	0.020163

Table 3. Experiment 1, per-process detection. Number of processes: 5550, number of malicious processes: 22

Many of the false positives were from processes that were simply not run as a part of the training session but were otherwise normal file system programs.

False positives also occurred when processes were run under varying conditions. Command shell execution and file execution of a new application caused false positives to appear. Applications generate processes in different ways depending upon their underlying system call initiation. Furthermore, programs which require a network connection to run correctly caused a false alarm when executed without a network connection. These false alarms arise because the model has not seen behavior from all the different execution behaviors of a given program.

Threshold	Detection Rate	False Positive
1.3	1.0	0.072485
1.2	0.972973	0.071741
1.1	0.972973	0.071145
1.0	0.972973	0.070104
0.9	0.972973	0.068616
0.8	0.972973	0.0675
0.7	0.972973	0.066532
0.6	0.945946	0.062961
0.5	0.945946	0.057553
0.4	0.945946	0.057328
0.3	0.918919	0.057276
0.2	0.918919	0.057180
0.1	0.918919	0.046897

Table 4. Experiment 2, per-process detection. Number of processes: 1344, number of malicious processes: 37

Threshold	Detection Rate	False Positive
0.8	1.0	0.08889
1.7	0.98611	0.08850
0.6	0.97222	0.08647
0.5	0.86111	0.07732
0.4	0.80555	0.07544
0.3	0.79166	0.07498
0.2	0.79166	0.07357
0.1	0.77777	0.07107

Table 5. Experiment 3, per-process detection. Number of processes: 1279, number of malicious processes: 72

5 Conclusions

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

We observe that file system accesses are apparently quite regular and well modeled by PAD. Anomalous accesses are rather easy to detect. Furthermore, malicious process behavior generates a relatively significant number of anomalous events, while normal processes can indeed generate anomalous accesses as well.

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

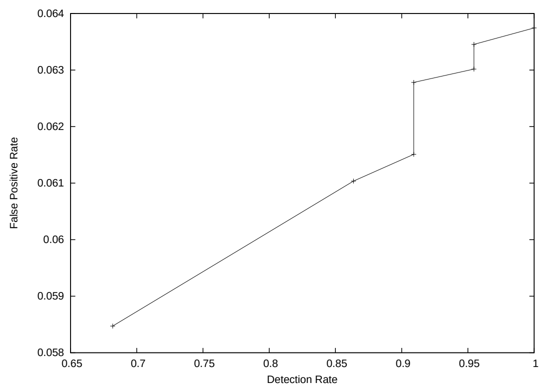

Fig. 5. Experiment 1, per-process ROC curve for Detection Rate versus false Positive Rate.

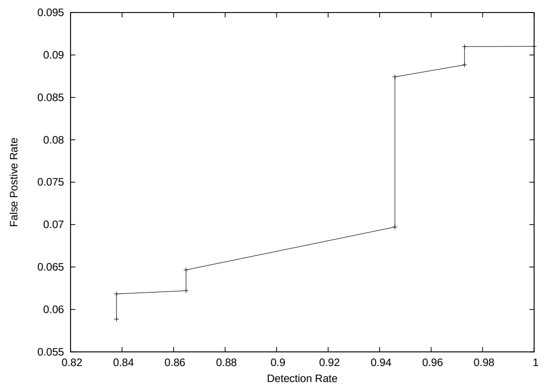

Fig. 6. Experiment 2, per process ROC curve for Detection Rate versus false Positive Rate.

involve calibration, pruning, feature selection, concept (or environment) drift, correlation and resiliency to attack.

Briefly, we seek automatic means of building anomaly detectors for arbitrary audit sources that are well behaved, and are easy to use. 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 anomaly detection models should be considered part of the evidence, and not be depended upon for the whole detection task. This means anomaly detector outputs should be correlated with other indicators or other anomaly detection models computed over different audit sources, different features or different modeling algorithms, in order to confirm or deny that an attack is truly occurring. Thus, it would be a mistake to entirely focus on a well calibrated threshold for a single anomaly detector simply to reduce false posi-

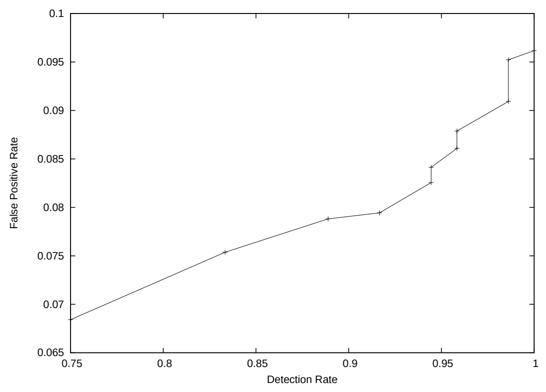

Fig. 7. Experiment 3, per process ROC curve for Detection Rate versus false Positive Rate

tives. It may in fact be a better strategy to generate more alerts, and possibly higher numbers of false positives, so that the correlation of these alerts with other confirmatory evidence reveals the true attacks that otherwise would go undetected (had the anomaly detector threshold been set too low).

In the experiments run to date PAD produces fine grained models that are expensive in memory. There are several enhancements that have been implemented in the Windows implementation of PAD for the RAD detector 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 for a feature at run-time. 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" drawn from the environment for a particular training epoch, 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. RAD's implementation on Windows provides the means of automatically retraining a model under a variety of user controlled schedules or performance measures. RAD includes 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. The same techniques are easily implemented for FWRAP as well.

We intend to continue this line of research using the various audit sources we have at our disposal, the FWRAP 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.

Finally, another aspect of the malicious exploit problem (whether it is worm, hacker,or malicious insider) not typically addressed concerns the forensic analysis and recovery/repair of detected host abuses.

FWRAP is situated on the file system where it may record the actions taken by a process detected as anomalous and possibly reverse the actions, if the security policy dictates a mitigation strategy that includes not only detection and reporting of events, but rollback and repair as autonomic operations.

There are of course at least two important issues, efficiency and resource consumption when storing state information about the actions of an anomalous file system access, and whether an action can be reversed safely.

FWRAP may generate alerts with a copy of the action (possibly also a copy of the file object changed by the malicious action) for only those accesses deemed anomalous. This will limit the resource consumption considerably, rather than logging all accesses and operations. The utility of this feature depends upon the accuracy of the anomaly detector however. Strategies that include perhaps a fixed size ring buffer in which objects and operations are stored may make this approach practical with almost no overhead.

5.1 Acknowledgment

We are grateful to Ke Wang and Erez Zadok for reading and suggesting changes to improve the paper.

References

1. F. Apap, A. Honig, S. Hershkop, E. Eskin and S. Stolfo. Detecting Malicious Software by Monitoring Anomalous Windows Registry Accesses. Fifth International Symposium on Recent Advances in Intrusion Detection, RAID-2002. Zurich, Switzerland, 2002.
2. R. Balzer. Mediating Connectors. 19th IEEE International Conference on Distributed Computing Systems Workshop, 1994.
3. B. Bauer and T. Bowden. The /proc Filesystem. http://www.linuxhq.com/kernel/v2.2/1/Documentation/proc.txt. January 1999.
4. B. Bloom. Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, Vol 13,Issue 7, 1970.

5. S. Forest, 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. A. Honig, A. Howard, E. Eskin, S. Stoflo. Adaptive Model Generation: An Architecture for Deployment of Data Minig-based Intrusion Detection Systems. Data Mining for Security Applications, (Jajodia, Barabara, Eds.). 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. W. Lee, S. Stolfo, and P.l 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. K. 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. E. Zadok and Ion Badulescu. A Stackable File System Interface For Linux. Linux-Expo 99. May 1999.
18. E. Zadok and Jason Nieh. FiST: A Language for Stackable File Systems. Usenix Technical Conference. June 2000
19. E. Zadok, Ion Badulescu, and Alex Shender. Extending File Systems Using Stackable Templates. Usenix Technical Conference. June 1999. page 7.
20. E. Zadok. Stackable File Systems as a Security Tool. Columbia U. CS TechReport CUCS-036-99. December 1999. page 5.
21. E. Zadok. Writing Stackable File Systems. Linux Journal , May 2003, pgs 22-25
22. O. Kreidl, and T. Frazier. Feedback Control Applied to Survivability: a Host-Based Autonomic Defense System. IEEE Transactions on Reliability, Vol. 53, No. 1, March 2004.
23. P. A. Porras, M. W. Fong, and A. Valdes. A Mission-Impact-Based Approach to INFOSEC Alarm Correlation. Proceedings Fifth International Symposium on Recent Advances in Intrusion Detection, RAID-2002, 2002.
24. M. Schonlau, W. DuMouchel, W. Ju, A. F. Karr, M. Theus, and Y. Vardi. Computer intrusion: Detecting masquerades. Statistical Science, 16(1):58-74, February 2001.
25. R. Maxion, and T. Townsend. Masquerade Detection Using Truncated Command Lines. International Conference on Dependable Systems and Networks (DSN-02), Washington, D.C., 2002.
26. K. Wang and S. Stolfo. One-Class Training for Masquerade Detection. 3rd IEEE International Conference on Data Mining, Workshop on Data Mining for Security Applications, Florida, Nov., 2003.