Detecting Malicious Software by Monitoring Anomalous Windows Registry Accesses

Frank Apap, Andrew Honig, Shlomo Hershkop, Eleazar Eskin, and Sal Stolfo
{fapap,arh,shlomo,eeskin,sal}@cs.columbia.edu

Department of Computer Science, Columbia University, New York NY 10027, USA

Abstract. We present a host-based intrusion detection system (IDS) for Microsoft Windows. The core of the system is an algorithm that detects attacks on a host machine by looking for anomalous accesses to the Windows Registry. The key idea is to first train a model of normal registry behavior on a windows host, and use this model to detect abnormal registry accesses at run-time. The normal model is trained using clean (attack-free) data. At run-time the model is used to check each access to the registry in real time to determine whether or not the behavior is abnormal and (possibly) corresponds to an attack. The system is effective in detecting the actions of malicious software while maintaining a low rate of false alarms.

1 Introduction

Microsoft Windows is one of the most popular operating systems today, and also one of the most often attacked. Malicious software running on the host is often used to perpetrate these attacks. There are two widely deployed first lines of defense against malicious software, virus scanners and security patches. Virus scanners attempt to detect malicious software on the host, and security patches are operating systems updates to fix the security holes that malicious software exploits. Both of these methods suffer from the same drawback. They are effective against known attacks but are unable to detect and prevent new types of attacks.

Most virus scanners are signature based meaning they use byte sequences or embedded strings in software to identify certain programs as malicious [10, 24]. If a virus scanner's signature database does not contain a signature for a specific malicious program, the virus scanner can not detect or protect against that program. In general, virus scanners require frequent updating of signature databases, otherwise the scanners become useless [29]. Similarly, security patches protect systems only when they have been written, distributed and applied to host systems. Until then, systems remain vulnerable and attacks can and do spread widely.

In many environments, frequent updates of virus signatures and security patches are unlikely to occur on a timely basis, causing many systems to remain vulnerable. This leads to the potential of widespread destructive attacks caused by malicious software. Even in environments where updates are more frequent,

the systems are vulnerable between the time new malicious software is created and the time that it takes for the software to be discovered, new signatures and patches created by experts, and the ultimate distribution to the vulnerable systems. Since malicious software may propagate through email, often the malicious software can reach the vulnerable systems long before the updates are available.

A second line of defense is through IDS systems. Host-based IDS systems monitor a host system and attempt to detect an intrusion. In the ideal case, an IDS can detect the effects or behavior of malicious software rather then distinct signatures of that software. Unfortunately, the commercial IDS systems that are widely in use are based on signature algorithms. These algorithms match host activity to a database of signatures which correspond to known attacks. This approach, like virus detection algorithms, require previous knowledge of an attack and is rarely effective on new attacks. Recently however, there has been growing interest in the use of data mining techniques such as anomaly detection, in IDS systems [23, 25]. Anomaly detection algorithms build models of normal behavior in order to detect behavior that deviates from normal behavior and which may correspond to an attack [9, 12]. The main advantage of anomaly detection is that it can detect new attacks and can be an effective defense against new malicious software. Anomaly detection algorithms have been applied to network intrusion detection [12, 20, 22] and also to the analysis of system calls for host based intrusion detection [13, 15, 17, 21, 28]. There are two problems to the system call approach to host based IDS which inhibits their use in actual deployment. The first is that the computational overhead of monitoring all system calls is very high, which degrades the performance of a system. The second is that system calls themselves are irregular by nature, which makes it difficult to differentiate between normal and malicious behaviors, which may cause a high false positive rate.

In this paper, we examine a new approach to host IDS that monitors a program's use of the Windows Registry. We present a system called RAD (Registry Anomaly Detection), which monitors the accesses to the registry in real time and detects the actions of malicious software.

The Windows Registry is an important part of the Windows operating system and is very heavily used, making it a good source of audit data. By building a sensor on the registry and applying the information gathered to an anomaly detector, we can detect activity that corresponds to malicious software. The main advantages of monitoring the Windows Registry is that the activity is regular by nature, can be monitored with low computational overhead, and almost all system activities interact with the registry.

Our anomaly detection algorithm is a registry-specific version of PHAD (Packet Header Anomaly Detection), an anomaly detection algorithm originally presented to detect anomalies in packet headers [25]. We show that the data generated by a registry sensor is useful in detecting malicious behavior. We shall describe how various malicious programs use the registry, and what data can be gathered from the registry to detect these malicious activities. We then apply an anomaly detection algorithm to this data to detect abnormal registry behavior

which corresponds to the actions of malicious software. By showing the results of an experiment and detailing how various malicious activities use the registry, we show that the registry is a good source of data for intrusion detection. The paper will also discuss the modifications of the PHAD algorithm as it is applied in the RAD system.

We present results of experiments evaluating the RAD system and demonstrate that it is effective in detecting attacks while maintaining a low rate of false alarms.

2 Modeling Registry Accesses

2.1 The Windows Registry

In Microsoft Windows, the registry file is a database of information about a computer's configuration. The registry contains information that is continually referenced by many different programs. Information stored in the registry includes the hardware installed on the system, which ports are being used, profiles for each user, configuration settings for programs, and many other parameters of the system. It is the main storage location for all configuration information for many Window programs. The Windows Registry is also the source for all security information: policies, user names, and passwords. The registry also stores much of the important run-time configuration information that programs need to run.

The registry is organized hierarchically as a tree. Each entry in the registry is called a key and has an associated value. One example of a registry key is

```
HKCU\Software\America Online\AOL Instant Messenger (TM)
\CurrentVersion\Users\aimuser\Login\Password
```

This is a key used by the AOL instant messenger program. This key stores an encrypted version of the password for the user name `aimuser`. Upon start up the AOL instant messenger program queries this key in the registry in order to retrieve the stored password for the local user. Information is accessed from the registry by individual registry accesses or queries. The information associated with a registry query is the key, the type of query, the result, the process that generated the query and whether the query was successful. One example of a query is a read for the key shown above. For example, the record of the query is:

Process: `aim.exe`
Query: `QueryValue`
Key: `HKCU\Software\America Online\AOL Instant Messenger (TM)\CurrentVersion\Users\aimuser\Login\Password`
Response: `SUCCESS`
ResultValue: `" BCOFHIHBBAHF"`

The Windows Registry is an effective data source to monitor attacks because many attacks show up as anomalous registry behavior. Many attacks take advantage of Windows' reliance on the registry. Indeed, many attacks themselves rely on the Windows Registry in order to function properly.

Many programs store important information in the Registry, notwithstanding the fact that other programs can arbitrarily access the information. Although some versions of Windows include security permissions and Registry logging, both features are rarely used (because of the computational overhead and the complexity of the configuration options).

2.2 Analysis of Malicious Registry Accesses

Most Windows programs access a certain set of Registry keys during normal execution. Furthermore, most users use a certain set of programs routinely while running their machines. This may be a set of all programs installed on the machine or more typically a small subset of these programs. Another important characteristic of Registry activity is that it tends to be regular over time. Most programs either only access the registry on start-up and shutdown, or access the registry at specific intervals. This regularity makes the registry an excellent place to look for irregular, anomalous activity, since a malicious program may substantially deviate from normal activity and can be detected.

Many attacks involve launching programs that have never been launched before and changing keys that have not been changed since the operating system had first been installed by the manufacturer. If a model of the normal registry behavior is computed over clean data, then these kinds of registry operations will not appear in the model. Furthermore malicious programs may need to query parts of the registry to get information about vulnerabilities. A malicious program can also introduce new keys that will help create vulnerabilities in the machine.

Some examples of malicious programs and how they produce anomalous registry activity are described below.

- **Setup Trojan:** This program when launched adds full read/write sharing access on the file system of the host machine. It makes use of the registry by creating a registry structure in the networking section of the Windows keys. The structure stems from `HKLM\Software\Microsoft\Windows \CurrentVersion \Network\LanMan`. It then creates typically eight new keys for its own use. It also accesses `HKLM \Security\Provider` in order to find information about the security of the machine to help determine vulnerabilities. This key is not accessed by any normal programs during training or testing in our experiments and its use is clearly suspicious in nature.
- **Back Orifice 2000:** This program opens a vulnerability on a host machine, which grants anyone with the back orifice client program complete control over the host machine. This program does make extensive use of the registry, however, it uses a key that is very rarely accessed on the Windows system. `HKLM\Software\Microsoft\VBA\Monitors` was not accessed by any normal

programs in either the training or test data, which allowed our algorithm to determine it as anomalous. This program also launches many other programs (`LoadWC.exe, Patch.exe, runonce.exe, bo2k_1_o_intl.e`) as part of the attack all of which made anomalous accesses to the Windows Registry.

- **Aimrecover:** This is a program that steals passwords from AOL users. It's actually a very simple program that simply reads the keys from the registry where the AOL Instant Messenger program stores the user names and passwords. The reason that these accesses are anomalous is because `Aimrecover` is accessing a key that usually is accessed by a different program that created that key.
- **Disable Norton:** This is a very simple exploitation of the registry that disables Norton Antivirus. This attack toggles one record in the registry, the key `HKLM\SOFTWARE\INTEL \LANDesk \VirusProtect6\CurrentVersion \Storages \Files\System \RealTimeScan \OnOff`. If this value is set to 0 then Norton Antivirus real time system monitoring is turned off. Again this is anomalous because of its access to a key that was created by a different program.
- **L0phtCrack:** This program is probably the most popular password cracking program for Windows machines. It retrieves the hashed SAM file containing the passwords for all users and then uses either a dictionary or brute force approach to find the passwords. This program also uses flaws in the Windows encryption scheme which allows the program to discover some of the characters in the password. This program uses the registry by creating its own section in the registry. This will consist of many create key and set value queries, all of which will be on keys that did not exist previously on the host machine and therefore have not been seen before.

Another important piece of information that can be used in detecting attacks, all programs observed in our data set, and presumably all programs in general, cause Windows Explorer to access a specific key. The key

```
HKLM\Software\Microsoft\Windows NT \CurrentVersion\Image File
Execution Options\processName
```

where processName is the name of the process being executed, is a key that is accessed by Explorer each time an application is run. Therefore we have a reference point for each specific application being launched to determine malicious activity. In addition many programs add themselves in the auto-run section of the Windows Registry under

```
HKLM\Software\Microsoft\Windows \CurrentVersion\Run .
```

While this is not malicious in nature, this is a rare event that can definitely be used as a hint that a system is being attacked. Trojan programs such as Back Orifice utilize this part of the registry to auto load themselves on each boot.

Anomaly detectors do not look for malicious activity directly. They look for deviations from normal activity. It is for this reason that any deviation from normal activity will be declared an attack by the system. The installation of

a new program on a system will be viewed as anomalous activity. Programs often create new sections of the registry and many new keys on installation. This will cause a false alarm, much like adding a new machine to a network may cause an alarm on an anomaly detector that analyzes network traffic. There are a few possible solutions to this problem. Malicious programs are often stealthy and install quietly so that the user does not know the program is being installed. This is not the case with most user initiated (legitimate) application installations that make themselves (loudly) known. The algorithm could be modified to ignore alarms while the install shield program was running because that would mean that the user is aware that a new program is being installed. Another option is to simply prompt the user when a detection occurs so that the user can let the anomaly detection system know that a legitimate installed program is under way and that therefore the anomaly detection model needs to be updated with a newly available training set gathered in real time. This is a typical user interaction in many application installations where user feedback is requested for configuration information.

3 Registry Anomaly Detection

The RAD system has three basic components: an audit sensor, a model generator, and an anomaly detector. The sensor logs each registry activity to either a database where it is stored for training, or to the detector to be used for analysis. The model generator reads data from the database and creates a model of normal behavior. The model is then used by the anomaly detector to decide whether each new registry access should be considered anomalous.

In order to detect anomalous registry accesses, RAD generates a model of normal registry activity. A set of five features are extracted from each registry access. Using these feature values over normal data, a model of normal registry 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 features of the current registry access are consistent with the normal data or not. If new activity is not consistent, the algorithm labels the access as anomalous.

3.1 RAD Data Model

The RAD data model consists of five features directly gathered from the registry sensor. The five raw features used by the RAD system are as follows.

- **Process:** This is the name of process accessing the registry. This is useful because it allows the tracking of new processes that did not appear in the training data.
- **Query:** This is the type of query being sent to the registry, for example, `QueryValue`, `CreateKey`, and `SetValue` are valid query types. This allows the identification of query types that have not been seen before. There are many query types but only a few are used under normal circumstances.

- **Key:** This is the actual key being accessed. This allows our algorithm to locate keys that are never accessed in the training data. Many keys are used only once for special situations like system installation. Some of these keys can be used to create vulnerabilities.
- **Response:** This describes the outcome of the query, for example `success`, `not found`, `no more`, `buffer overflow`, and `access denied`.
- **Result Value:** This is the value of the key being accessed. This will allow the algorithm to detect abnormal values being used to create abnormal behavior in the system.

Feature	**aim.exe**	**aimrecover.exe**
Process	aim.exe	aimrecover.exe
Query	QueryValue	QueryValue
Key	HKCU\Software\America Online \AOL Instant Messenger (TM) \CurrentVersion\Users \aimuser\Login\Password	HKCU\Software\America Online \AOL Instant Messenger (TM) \CurrentVersion\Users \aimuser\Login\Password
Response	SUCCESS	SUCCESS
Result Value	" BCOFHIHBBAHF"	" BCOFHIHBBAHF"

Table 1. Registry Access Records. Two registry accesses are shown. The first is a normal access by AOL Instance Messenger to the key where passwords are stored. The second is a malicious access by AIMrecover to the same key. The final column shows which fields register as anomalous. Note that the pairs of features must be used to detect the anomalous behavior of AIMrecover.exe. This is because under normal circumstances only AIM.exe accesses the key that stores the AIM password. Another process accessing this key generates an anomaly.

3.2 RAD Anomaly Detection Algorithm

Using the features that we monitor from each registry access, we train a model over features extracted from normal data. That model allows us to classify registry accesses as either normal or malicious.

Any anomaly detection algorithm can be used to perform this modeling. Since we aim to monitor a significant amount of data in real time, the algorithm must be very efficient. We apply a probabilistic algorithm described in Eskin, 2002 [14] and here we provide a short summary of the algorithm. The algorithm is similar to the heuristic algorithm that was proposed by Chan and Mahoney in the PHAD system [25], but is more robust.

In general, a principled probabilistic approach to anomaly detection can be reduced to density estimation. If we can estimate a density function $p(x)$ over

the normal data, we can define anomalies as data elements that occur with low probability. In practice, estimating densities is a very hard problem (see the discussion in Schölkopf et al., 1999 [26] and the references therein.) In our setting, part of the problem is that each of the features have many possible values. For example, the *Key* feature has over $30,000$ values in our training set. Since there are so many possible feature values relatively rarely does the same exact record occur in the data. Data sets with this characterization are referred to as sparse.

Since probability density estimation is a very hard problem over sparse data, we propose a different method for determining which records from a sparse data set are anomalous. We define a set of consistency checks over the normal data. Each consistency check is applied to an observed record. If the record fails any consistency check, we label the record as anomalous.

We apply two kinds of consistency checks. The first consistency check evaluates whether or not a feature value is consistent with observed values of that feature in the normal data set. We refer to this type of consistency check as a first order consistency check. More formally, each registry record can be viewed as the outcome of 5 random variables, one for each feature, X_1, X_2, X_3, X_4, X_5. Our consistency checks compute the likelihood of an observation of a given feature which we denote $P(X_i)$.

The second kind of consistency check handles pairs of features as motivated by the example in Table 1. For each pair of features, we consider the conditional probability of a feature value given another feature value. These consistency checks are referred to as second order consistency checks. We denote these likelihoods $P(X_i|X_j)$. Note that for each value of X_j, there is a different probability distribution over X_i.

In our case, since we have 5 feature values, for each record, we have 5 first order consistency checks and 20 second order consistency checks. If the likelihood of any of the consistency checks is below a threshold, we label the record as anomalous.

What remains to be shown is how we compute the likelihoods for the first order ($P(X_i)$) and second order ($P(X_i|X_j)$) consistency checks. Note that from the normal data, we have a set of observed counts from a discrete alphabet for each of the consistency checks. Computing these likelihoods reduces to simply estimating a multinomial. In principal we can use the maximum likelihood estimate which just computes the ratio of the counts of a particular element to the total counts. However, the maximum likelihood estimate is biased when there is relatively small amounts of data. When estimating sparse data, this is the case. We can smooth this distribution by adding a virtual count to each possible element. This is equivalent to using a Dirichlet estimator [11]. For anomaly detection, as pointed out in Mahoney and Chan, 2001 [25], it is critical to take into account how likely we are to observe an unobserved element. Intuitively, if we have seen many different elements, we are more likely to see unobserved elements as opposed to the case where we have seen very few elements.

To estimate our likelihoods we use the estimator presented in Friedman and Singer, 1999 [16] which explicitly estimates likelihood of observing a previously

unobserved element. The estimator gives the following prediction for element i

$$P(X = i) = \frac{\alpha + N_i}{k^0\alpha + N}C \tag{1}$$

if element i was observed and

$$P(X = i) = \frac{1}{L - k^0}(1 - C) \tag{2}$$

if element i was not previously observed. α is a prior count for each element, N_i is the number of times i was observed, N is the total number of observations, k^0 is the number of different elements observed, and L is the total number of possible elements or the alphabet size. The scaling factor C takes into account how likely it is to observe a previously observed element versus an unobserved element. C is computed by

$$C = \left(\sum_{k=k^0}^{L} \frac{k^0\alpha + N}{k\alpha + N} m_k\right)\left(\sum_{k \geq k^0} m_k\right)^{-1} \tag{3}$$

where $m_k = P(S = k)\frac{k!}{(k-k^0)!}\frac{\Gamma(k\alpha)}{\Gamma(k\alpha+N)}$ and $P(S = k)$ is a prior probability associated with the size of the subset of elements in the alphabet that have non-zero probability. Although the computation of C is expensive, it only needs to be done once for each consistency check at the end of training.

The prediction of the probability estimator is derived using a mixture of Dirichlet estimators each of which represent a different subset of elements that have non-zero probability. Details of the probability estimator and its derivation are given in [16] and complete details of the anomaly detection algorithm are given in [14].

Note that this algorithm labels every registry access as either normal or anomalous. Programs can have anywhere from just a few registry accesses to several thousand. This means that many attacks will be represented by large numbers of records where many of those records will be considered anomalous.

Some records are anomalous because they have a value for a feature that is inconsistent with the normal data. However, some records are anomalous because they have an inconsistent combination of features although each feature itself may be normal. Because of this, we examine pairs of features. For example, let us consider the registry access displayed in Table 1. The basic features for the normal program `aim.exe` versus the malicious program `aimrecover.exe` do not appear anomalous. However, the fact that the program `aimrecover.exe` is accessing a key that is usually associated with `aim.exe` is in fact an anomaly. Only by examining the combination of the two raw features can we detect this anomaly.

4 Architecture

The basic architecture of the RAD system consists of three components, the registry auditing module (RegBAM), the model generator, and the real-time anomaly detector. An overview of the RAD architecture is shown in Figure 1.

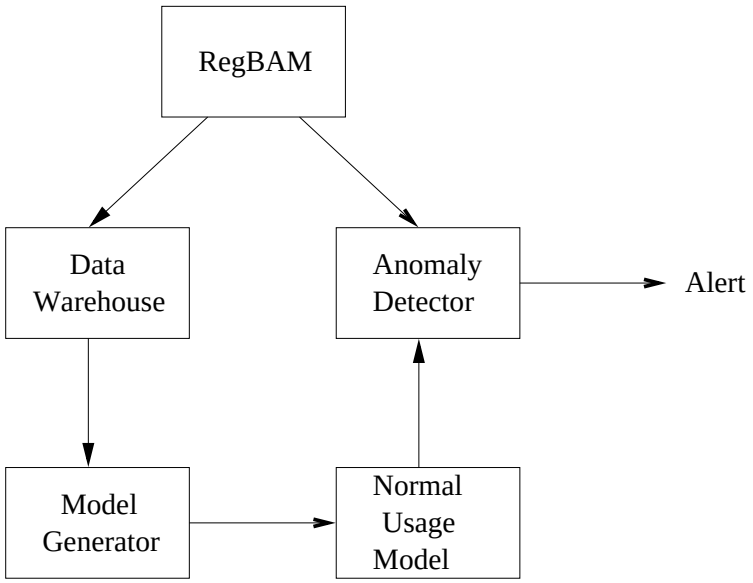

Fig. 1. The RAD System Architecture. RegBAM outputs to the data warehouse during training model and to the anomaly detector during detection mode.

4.1 Registry Basic Auditing Module

The RAD sensor is composed of a Basic Auditing Module (BAM) for the RAD system which monitors accesses to the registry. BAMs implement an architecture and interface for sensors across the system. They include a hook into the audit stream (in this case the registry) and various communication and data-buffering components. BAMs use an XML data representation similar to the IDMEF standard (of the IETF) for IDS systems [19]. BAMs are described in more detail in [18].

The Registry BAM (RegBAM) runs in the background on a Windows machine as it gathers information on registry reads and writes. RegBAM uses Win32 hooks to tap into the registry and log all reads and writes to the registry. RegBAM is akin to a wrapper and uses a similar architecture to that of SysInternal's Regmon [27]. After gathering the registry data, RegBAM can be configured for two distinct purposes. One use is as the audit data source for model generation. When RegBAM is used as the data source, the output data is sent to a database where it is stored and later used by the model generator described in Section 4.2 [18]. The second use of RegBAM, is as the data source for the real-time anomaly detector described in Section 4.3. While in this mode, the output of RegBAM is sent directly to the anomaly detector where it is processed in real time. An alternative method to collect the registry accesses is to use the Windows auditing mechanism. All registry accesses can be logged in the Windows Event Log. Each read or write can generate multiple records in the Event Log. However, this

method is problematic because the event logs are not designed to handle such a large amount of data. Simple tests demonstrated that by turning on all registry auditing the Windows Event Logger caused a major resource drain on the host machine, and in many cases caused the machine to crash. The RegBAM application provides an efficient method for monitoring all registry activity, with far less overhead than the native tools provided by the Windows operating system.

4.2 Model Generation Infrastructure

Similar to the Adaptive Model Generation (AMG) architecture [18], the system uses RegBAM to collect registry access records. Using this database of collected records from a training run, the model generator then creates a model of normal usage.

The model generator uses the algorithm discussed in Section 3 to build a model that represents normal usage. It utilizes the data stored in the database which was generated by RegBAM during training. The model itself is comprised and stored as serialized Java objects. This allows for a single model to be generated and to be easily distributed to additional machines. Having the model easily deployed to new machines is a desirable feature, since in a typical network, many Windows machines have similar usage patterns. This allows the same model to be used for multiple host machines.

4.3 Real-Time Anomaly Detector

For real time detection, RegBAM feeds live data for analysis by an anomaly detector. The anomaly detector will load the normal usage model created by the model generator define and begin reading each record from the output data stream of RegBAM. The algorithm discussed in Section 3 is then applied against each record of registry activity. The score generated by the anomaly detection algorithm is compared by a user configurable threshold to determine if the record should be considered anomalous. A list of anomalous registry accesses are stored and displayed as part of the detector. A user configured threshold allows the user to customize the alarm rate for the particular environment. Lowering the threshold, will result in more alarms being issued. Although this can raise the false positive rate, it can also increase the chance of detecting new attacks.

4.4 Efficiency Considerations

In order for a system to detect anomalies in a real time environment it can not consume excessive system resources. This is especially important in registry attack detection because of the heavy amount of traffic that generated by applications interacting with the registry. While the amount of traffic can vary greatly from system to system, in our experimental setting (described below) the traffic load was about 50,000 records per hour. Our distributed architecture is designed to minimize the resources used by the host machine. It is possible to spread the

work load on to several separate machines, so that the only application running on the host machine is the lightweight RegBAM. However this will increase network load due to the communication between components. These two concerns can be used to configure the system to create the proper proportion between host system load and network load. The RegBAM module is a far more efficient way of gathering data about registry activity than full auditing with the Windows Event Log.

5 Evaluation and Results

The system was evaluated by measuring the detection performance over a set of collected data which contains some attacks. Since there are no other existing publicly available detection systems that operate on Windows registry data we were unable to compare our performance to other systems directly.

5.1 Data Generation

In order to evaluate the RAD system, we gathered data by running a registry sensor on a host machine. Since there are no publicly available data sets containing registry accesses, we collected our own data. Beyond the normal execution of standard programs, such as Microsoft Word, Internet Explorer, and Winzip, the training also included performing housekeeping tasks such as emptying the Recycling Bin and using the Control Panel. All simulations were done by hand to simulate a real user. All data used for this experiment is publicly available online in text format at `http://www.cs.columbia.edu/ids/rad`. The data includes a time stamp and frequency of the launched programs in relation to each other.

The training data collected for our experiment was collected on Windows NT 4.0 over two days of normal usage (in our lab). We informally define "normal" usage to mean what we believe to be typical use of a Windows platform in a home setting. For example, we assume all users would log in, check some internet sites, read some mail, use word processing, then log off. This type of session is assumed to be relatively "typical" of many computer users. Normal programs are those which are bundled with the operating systems, or are in use by most Windows users. Creating realistic testing environments is a very hard task and testing the system under a variety of environments is a direction for future work.

The simulated home use of Windows generated a clean (attack-free) dataset of approximately 500,000 records. The system was then tested on a full day of test data with embedded attacks executed. This data was comprised of approximately 300,000 records most of which were normal program executions interspersed with attacks. The normal programs run between attacks were intended to simulate an ordinary Windows session. The programs used were Microsoft Word, Outlook Express, Internet Explorer, Netscape, AOL Instant Messenger, and others.

The attacks run include publicly available attacks such as aimrecover, browslist, bok2ss (back orifice), install.exe xtxp.exe both for backdoor.XTCP, l0phtcrack,

runattack, whackmole, and setuptrojan. Attacks were only run during the one day of testing throughout the day. Among the twelve attacks that were run, four instances were repetitions of the same attack. Since some attacks generated multiple processes there are a total of seventeen distinct processes for each attack. All of the processes (either attack or normal) as well as the number of registry access records in the test data is shown in Table 3.

The reason for running some of the attacks twice, was to test the effectiveness of our system. Many programs act differently when executed a second time within a windows session. In the experiments reported below our system was less likely to detect a previously successful attack on the second execution of that attack. The reason is that a successful attack creates permanent changes to the registry and hence on subsequent queries the attack no longer appears irregular. Thus the next time the same attack is launched it is more difficult to detect since it interacts less with the registry.

We observed that this is common for both malicious and regular applications since many applications will do a much larger amount of registry writing during installation or when first executed.

5.2 Experiments

The training and testing environments were set up to replicate a simple yet realistic model of usage of Windows systems. The system load and the applications that were run were meant to resemble what one may deem typical in normal private settings.

We trained the anomaly detection algorithm presented in Section 3 over the normal data and evaluated each record in the testing set. We evaluate our system by computing two statistics. We compute the *detection rate* and the *false positive rate*.

The natural way we may evaluate the performance of RAD is to measure detection performance over processes labeled as either normal or malicious. However, with only seventeen malicious processes at our disposal in our test set, it is difficult to obtain a robust evaluation for the system. We do discuss the performance of the system system in terms of correctly classified processes, but also measure the performance in terms of the numbers of records correctly and incorrectly classified. Future work on RAD will focus on testing over long periods of time to measure significantly more data and process classifications as well as alternative means of alarming on processes. (For example, a process may be declared an attack on the basis of one anomalous record it generates, or perhaps on some number of anomalous records.) There is also an interesting issue to be investigated regarding the decay of the anomaly models that may be exhibited over time, perhaps requiring regenerating a new model.

The detection rate reported below is the percentage of records generated by the malicious programs which are labeled correctly as anomalous by the model. The false positive rate is the percentage of normal records which are mislabeled anomalous. Each attack or normal process has many records associated with it. Therefore, it is possible that some records generated by a malicious program will

be mislabeled even when some of the records generated by the attack are accurately detected. This will occur in the event that some of the records associated with one attack are labeled normal. Each record is given an anomaly score, S, that is compared to a user defined threshold. If the score is greater than the threshold, then that particular record is considered malicious. Fig 2 shows how varying the threshold affects the output of detector. The actual recorded scores plotted in the figure are displayed in Table 2.

Threshold Score	False Positive Rate	Detection Rate
6.847393	0.001192	0.005870
6.165698	0.002826	0.027215
5.971925	0.003159	0.030416
5.432488	0.004294	0.064034
4.828566	0.005613	0.099253
4.565011	0.006506	0.177161
3.812506	0.009343	0.288687
3.774119	0.009738	0.314301
3.502904	0.011392	0.533084
3.231236	0.012790	0.535219
3.158004	0.014740	0.577908
2.915094	0.019998	0.578442
2.899837	0.020087	0.627001
2.753176	0.033658	0.629136
2.584921	0.034744	0.808431
2.531572	0.038042	0.869797
2.384402	0.050454	1.000000

Table 2. Varying the threshold score and its effect on False Positive Rate and Detection Rate.

Table 3 is sorted in order to show the results for classifying processes. From the table we can see if the threshold is set at 8.497072, we would label the processes `LOADWC.EXE` and `ipccrack.exe` as malicious and would detect the Back Orifice and IPCrack attacks. Since none of the normal processes have scores that high, we would have no false positives. If we lower the threshold to 6.444089, we would have detected several more processes from Back Orifice and the BrowseList,BackDoor.xtcp, SetupTrojan and AimRecover attacks. However, at this level of threshold, the following processes would be labeled as false positives: `systray.exe`, `CSRSS.EXE`, `SPOOLSS.EXE`, `ttssh.exe`, and `winmine.exe`. As we have mentioned, our future work on RAD will model and measure a Windows system for a far longer period of time over many more processes in order to generate a meaningful ROC curve in terms of processes. The measurements reported next are cast in terms of registry query records.

5.3 Detection

By varying the threshold for the inconsistency scores on records, we were able to demonstrate the variability of the the detection rate and false positive rate. We plot the false positive rate versus the detection rate in an ROC (Receiver Operator Characteristic) curve shown in Figure 2 and Table 2.

Many of the false positives were from processes that were simply not run as a part of the training data but were otherwise normal Windows programs. A thorough analysis of what kinds of processes generate false positives is a direction for future work.

Part of the reason why the system is successfully able to discriminate between malicious and normal records is that accesses to the Windows Registry are very regular which makes normal registry activity relatively easy to model.

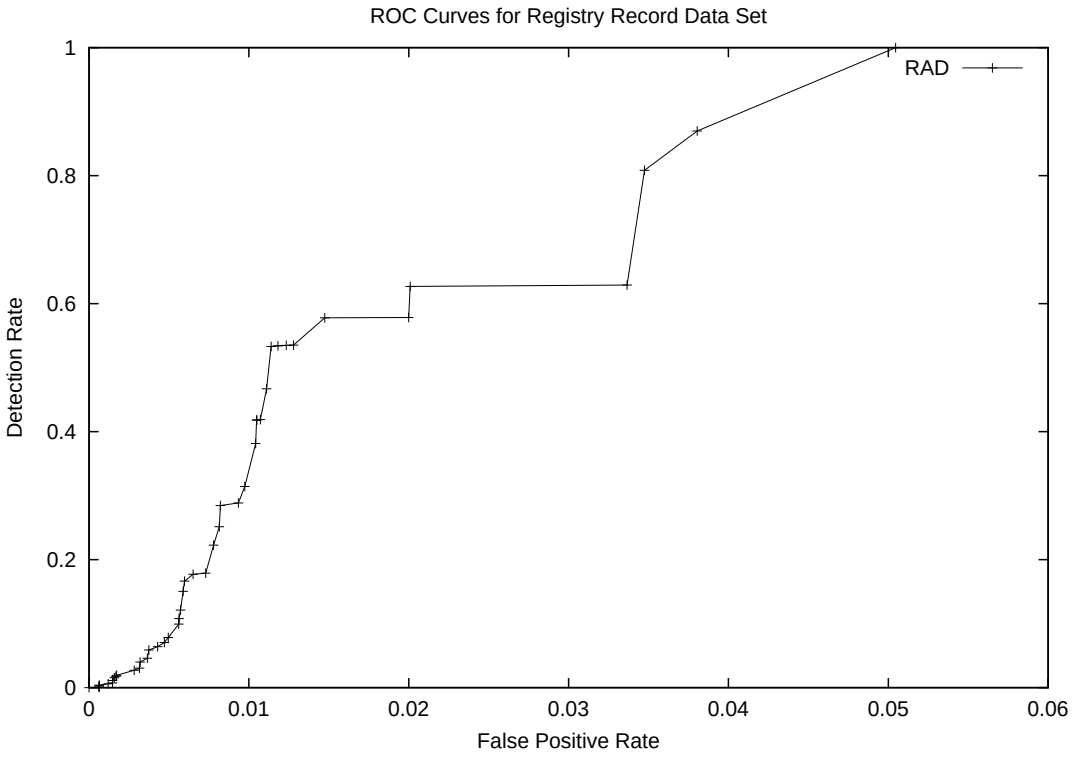

Fig. 2. Figure showing varying the threshold on the data set.

Program Name	Number of Records	Maximum Record Value	Minimum Record Value	Classification
LOADWC.EXE[2]	1	8.497072	8.497072	ATTACK
ipccrack.exe[6]	1	8.497072	8.497072	ATTACK
mstinit.exe[2]	11	7.253687	6.705313	ATTACK
bo2kss.exe[2]	12	7.253687	6.62527	ATTACK
runonce.exe[2]	8	7.253384	6.992995	ATTACK
browselist.exe[4]	32	6.807137	5.693712	ATTACK
install.exe[3]	18	6.519455	6.24578	ATTACK
SetupTrojan.exe[8]	30	6.444089	5.756232	ATTACK
AimRecover.exe[1]	61	6.444089	5.063085	ATTACK
happy99.exe[5]	29	5.918383	5.789022	ATTACK
bo2k_1_0_intl.e[2]	78	5.432488	4.820771	ATTACK
_INS0432._MP[2]	443	5.284697	3.094395	ATTACK
xtcp.exe[3]	240	5.265434	3.705422	ATTACK
bo2kcfg.exe[2]	289	4.879232	3.520338	ATTACK
l0phtcrack.exe[7]	100	4.688737	4.575099	ATTACK
Patch.exe[2]	174	4.661701	4.025433	ATTACK
bo2k.exe[2]	883	4.386504	2.405762	ATTACK
systray.exe	17	7.253687	6.299848	NORMAL
CSRSS.EXE	63	7.253687	5.031336	NORMAL
SPOOLSS.EXE	72	7.070537	5.133161	NORMAL
ttssh.exe	12	6.62527	6.62527	NORMAL
winmine.exe	21	6.56054	6.099177	NORMAL
em_exec.exe	29	6.337396	5.789022	NORMAL
winampa.exe	547	6.11399	2.883944	NORMAL
PINBALL.EXE	240	5.898464	3.705422	NORMAL
LSASS.EXE	2299	5.432488	1.449555	NORMAL
PING.EXE	50	5.345477	5.258394	NORMAL
EXCEL.EXE	1782	5.284697	1.704167	NORMAL
WINLOGON.EXE	399	5.191326	3.198755	NORMAL
rundll32.exe	142	5.057795	4.227375	NORMAL
explore.exe	108	4.960194	4.498871	NORMAL
netscape.exe	11252	4.828566	-0.138171	NORMAL
java.exe	42	4.828566	3.774119	NORMAL
aim.exe	1702	4.828566	1.750073	NORMAL
findfast.exe	176	4.679733	4.01407	NORMAL
TASKMGR.EXE	99	4.650997	4.585049	NORMAL
MSACCESS.EXE	2825	4.629494	1.243602	NORMAL
IEXPLORE.EXE	194274	4.628190	-3.419214	NORMAL
NTVDM.EXE	271	4.59155	3.584417	NORMAL
CMD.EXE	116	4.579538	4.428045	NORMAL
WINWORD.EXE	1541	4.457119	1.7081	NORMAL
EXPLORER.EXE	53894	4.31774	-1.704574	NORMAL
msmsgs.exe	7016	4.177509	0.334128	NORMAL
OSA9.EXE	705	4.163361	2.584921	NORMAL
MYCOME 1.EXE	1193	4.035649	2.105155	NORMAL
wscript.exe	527	3.883216	2.921123	NORMAL
WINZIP32.EXE	3043	3.883216	0.593845	NORMAL
notepad.exe	2673	3.883216	1.264339	NORMAL
POWERPNT.EXE	617	3.501078	-0.145078	NORMAL
AcroRd32.exe	1598	3.412895	0.393729	NORMAL
MDM.EXE	1825	3.231236	1.680336	NORMAL
ttermpro.exe	1639	2.899837	1.787768	NORMAL
SERVICES.EXE	1070	2.576196	2.213871	NORMAL
REGMON.EXE	259	2.556836	1.205416	NORMAL
RPCSS.EXE	4349	2.250997	0.812288	NORMAL

Table 3. Information about all processes in testing data including the number of registry accesses and the maximum and minimum score for each record as well as the classification. The top part of the table shows this information for all of the attack processes and the bottom part of the table shows this information for the normal processes. The reference number (by the attack processes) give the source for the attack. Processes that have the same reference number are part of the same attack. [1] AIMCrack. [2] Back Orifice. [3] Backdoor.xtcp. [4] Browse List. [5] Happy 99. [6] IPCrack. [7] L0pht Crack. [8] Setup Trojan.

6 Conclusions

By using registry activity on a Windows system, we were able to label all processes as either attacks or normal, with relatively high accuracy and low false positive rate, for the experiments performed in this study. We have shown that registry activity is regular, and described ways in which attacks would generate anomalies in the registry. Thus, an anomaly detector for registry data may be an effective intrusion detection system augmenting other host-based detection systems. It would also improve protection of systems in cases of new attacks that would otherwise pass by scanners that have not been updated on a timely basis.

We plan on testing the system under a variety of environments and conditions to better understand its performance. Future plans include combining the RAD system with another detector that evaluates Windows Event Log data. This will allow for various data correlation algorithms to be used to make more accurate system behavior models which we believe will provide a more accurate anomaly detection system with better coverage of attack detection. Part of our future plans for the RAD system include adding data clustering and aggregation capabilities. Aggregating alarms will allow for subsets of registry activity records to be considered malicious as a group initiated from one attack rather than individual attacks. We also plan to store the system registry behavior model as part of the registry itself. The motivation behind this, is to use the anomaly detector to protect the system behavior model from being maliciously altered, hence making the model itself secured against attack. These additions to the RAD system will make the system a more complete and effective tool for detecting malicious behavior on the Windows platform.

References

1. Aim Recovery. `http://www.dark-e.com/des/software/aim/index.shtml`.
2. Back Orifice. `http://www.cultdeadcow.com/tools/bo.html`.
3. BackDoor.XTCP. `http://www.ntsecurity.new/Panda/Index.cfm?FuseAction=Virus&VirusID=659`.
4. BrowseList. `http://e4gle.org/files/nttools/`,`http://binaries.faq.net.pl/security_tools`.
5. Happy 99. `http://www.symantex.com/qvcenter/venc/data/happy99.worm.html`.
6. IPCrack. `http://www.geocities.com/SiliconValley/Garage/3755/toolicq.html`, `http://home.swipenet.se/~w-65048/hacks.htm`.
7. L0pht Crack. `http://www.atstack.com/research/lc`.
8. Setup Trojan. `http://www.nwinternet.com/~pchelp/bo/setuptrojan.txt`.
9. V. Barnett and T. Lewis. *Outliers in Statistical Data.* John Wiley and Sons, 1994.
10. Fred Cohen. *A Short Course on Computer Viruses.* ASP Press, Pittsburgh, PA, 1990.
11. M. H. DeGroot. *Optimal Statistical Decisions.* McGraw-Hill, New York, 1970.
12. D. E. Denning. An intrusion detection model. *IEEE Transactions on Software Engineering*, SE-13:222–232, 1987.

13. Eleazar Eskin. Anomaly detection over noisy data using learned probability distributions. In *Proceedings of the Seventeenth International Conference on Machine Learning (ICML-2000)*, 2000.
14. Eleazar Eskin. Probabilistic anomaly detection over discrete records using inconsistency checks. Technical report, Columbia University Computer Science Technical Report, 2002.
15. Stephanie Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. pages 120–128. IEEE Computer Society, 1996.
16. N. Friedman and Y. Singer. Efficient bayesian parameter estimation in large discrete domains, 1999.
17. S. A. Hofmeyr, Stephanie Forrest, and A. Somayaji. Intrusion detect using sequences of system calls. *Journal of Computer Security*, 6:151–180, 1998.
18. Andrew Honig, Andrew Howard, Eleazar Eskin, and Salvatore Stolfo. Adaptive model generation: : An architecture for the deployment of data minig-based intrusion detection systems. In *Data Mining for Security Applications*. Kluwer, 2002.
19. Internet Engineering Task Force. Intrusion detection exchange format. In *http://www.ietf.org/html.charters/idwg-charter.html*, 2000.
20. H. S. Javitz and A. Valdes. The nides statistical component: Description and justification. Technical report, SRI International, 1993.
21. W. Lee, S. J. Stolfo, and P. K. Chan. Learning patterns from unix processes execution traces for intrusion detection. pages 50–56. AAAI Press, 1997.
22. W. Lee, S. J. Stolfo, and K. Mok. Data mining in work flow environments: Experiences in intrusion detection. In *Proceedings of the 1999 Conference on Knowledge Discovery and Data Mining (KDD-99)*, 1999.
23. Wenke Lee, Sal Stolfo, and Kui Mok. A data mining framework for building intrusion detection models. 1999.
24. MacAfee. Homepage - macafee.com. *Online publication*, 2000. http://www.mcafee.com.
25. M. Mahoney and P. Chan. Detecting novel attacks by identifying anomalous network packet headers. Technical Report CS-2001-2, Florida Institute of Technology, Melbourne, FL, 2001.
26. B. Schölkopf, J. Platt, J. Shawe-Taylor, A. J. Smola, and R. C. Williamson. Estimating the support of a high-dimensional distribution. Technical Report 99-87, Microsoft Research, 1999. To appear in *Neural Computation*, 2001.
27. SysInternals. Regmon for Windows NT/9x. *Online publication*, 2000. http://www.sysinternals.com/ntw2k/source/regmon.shtml.
28. Christina Warrender, Stephanie Forrest, and Barak Pearlmutter. Detecting intrusions using system calls: alternative data models. pages 133–145. IEEE Computer Society, 1999.
29. Steve R. White. Open problems in computer virus research. In *Virus Bulletin Conference*, 1998.

This article was processed using the LaTeX macro package with LLNCS style