On the Detection of Anomalous System Call Arguments

Christopher Kruegel, Darren Mutz, Fredrik Valeur, and Giovanni Vigna

Reliable Software Group[**]
Department of Computer Science
University of California, Santa Barbara
{chris, dhm, fredrik, vigna}@cs.ucsb.edu

Abstract

Learning-based anomaly detection systems build models of the expected behavior of applications by analyzing events that are generated during their normal operation. Once these models have been established, subsequent events are analyzed to identify deviations, given the assumption that anomalies usually represent evidence of an attack.

Host-based anomaly detection systems often rely on system call traces to build models and perform intrusion detection. Recently, these systems have been criticized, and it has been shown how detection can be evaded by executing an attack using a carefully crafted exploit. This weakness is caused by the fact that existing models do not take into account all available features of system calls. In particular, some attacks will go undetected because the models do not make use of system call arguments. To solve this problem, we have developed an anomaly detection technique that utilizes the information contained in these parameters. Based on our approach, we developed a host-based intrusion detection system that identifies attacks using a composition of various anomaly metrics.

This paper presents our detection techniques and the tool based on them. The experimental evaluation shows that it is possible to increase both the effectiveness and the precision of the detection process compared to previous approaches. Nevertheless, the system imposes only minimal overhead.

Keywords: Intrusion detection, anomaly models, system calls.

1 Introduction

Intrusion detection techniques have traditionally been classified as either *misuse-based* or *anomaly-based*.

Systems that use misuse-based techniques [7, 17, 18] contain a number of attack descriptions, or signatures, that are matched against a stream of audit data looking for evidence of modeled attacks. These systems are usually efficient and generate few erroneous detections, called false positives. The main disadvantage of misuse-based techniques is the fact that they can only detect those attacks

[**] This research was supported by the Army Research Office, under agreement DAAD19-01-1-0484.

that have been specified previously. That is, they cannot detect intrusions for which they do not have a signature.

Anomaly-based techniques [6, 9, 12] follow an approach that is complementary to misuse detection. In their case, detection is based on models of normal behavior of users and applications, called 'profiles'. Any deviations from such established profiles are interpreted as attacks. The main advantage of anomaly-based techniques is that they are able to identify previously unknown attacks. By defining an expected, normal state, any abnormal behavior can be detected, whether it is part of the threat model or not. The advantage of being able to detect previously unknown attacks is usually paid for with a high number of false positives.

In the past, a number of host-based anomaly detection approaches have been proposed that build profiles using system calls [8, 25]. More specifically, these systems rely on models of legitimate system call sequences issued by the application during normal operation. During the detection process, every monitored sequence that is not compliant with previously established profiles is considered part of an attack.

Recent research [23, 24, 26] has examined techniques to evade this type of detection by using mimicry attacks. Mimicry attacks operate by crafting the injected malicious code in a way that imitates (or mimics) a legitimate system call sequence. The results show that many models can be easily bypassed.

The weakness of existing systems is mostly due to the lack of comprehensive models that take advantage of the information contained in system call traces. In particular, satisfactory machine generated models for system call arguments have not been developed because the problem has been considered either too difficult or too expensive computationally.

This paper presents a novel anomaly detection technique that takes advantage of the information contained in system calls by performing an analysis of their arguments. We present several models to derive profiles for different types of arguments and a mechanism to evaluate a system call by composing the results delivered by different anomaly metrics for each of the system call parameters. In addition, we describe a host-based intrusion detection tool that implements our approach. A qualitative analysis shows that the system is capable of performing effective detection while a quantitative evaluation confirms that it is very efficient.

2 Related Work

Many different anomaly detection techniques have been developed that gather input from a variety of sources. Examples include data mining on network traffic [16], statistical analysis of audit records [11], and the analysis of operating system call sequences [8]. As our approach is based on system calls, work in this area is particularly relevant. Previously presented research falls into the areas of specification-based and learning-based approaches.

Specification-based techniques rely on application-specific models that are either written manually (e.g., [12], [3], [5]) or derived using static program analysis techniques (e.g., [25]). [10] and [19] describe systems that interactively create application-specific profiles with the help of the user. The profiles are then used as the input to a real-time intrusion detection system that monitors the corresponding application. When a non-conforming system call invocation is detected, an alarm is raised.

A major problem of specification-based systems is the fact that they exhibit only a very limited capability for generalizing from written or derived specifications. An additional disadvantage of hand-written specification-based models is the need for human interaction during the training phase. Although it is possible to include predefined models for popular applications, these might not be suitable for every user, especially when different application configurations are used. Systems that use automatically generated specifications often suffer from significant processing overhead. This is caused by the complexity of the underlying models. Another drawback is the fact that previously presented approaches do not take into account system call arguments, unless they are constants or can be easily determined by data flow analysis [25].

Learning-based techniques do not rely on any *a priori* assumptions about the applications. Instead, profiles are built by analyzing system call invocations during normal execution. An example of this approach is presented by Forrest [8]. During the training phase, the system collects all distinct system call sequences of a certain specified length. During detection, all actual system call sequences are compared to the set of legitimate ones, raising an alarm if no match is found. This approach has been further refined in [15] and [27], where the authors study similar models and compare their effectiveness to the original technique. However, these models suffer from the limitation that information about system call arguments is discarded.

To mitigate this weakness, we propose a learning-based technique that focuses on the analysis of system call arguments. By doing so, it is possible to considerably reduce the ability of an attacker to evade detection. We propose to use different models that examine different features of the arguments of a system call. This allows anyone to easily extend our system by introducing new models. To assess an entire system call, the results of the models are combined into a single anomaly score.

3 Design

Our anomaly detection mechanism is based on the application-specific analysis of individual system calls. The input to the detection process consists of an ordered stream $S = \{s_1, s_2, \dots\}$ of system call invocations recorded by the operating system. Every system call invocation $s \in S$ has a return value r^s and a list of argument values $< a_1^s, \dots, a_n^s >$. We do not take into account relationships between system calls or sequences of invocations.

For each system call used by an application, a distinct profile is created. For example, for the `sendmail` application the system builds a profile for each of the system calls invoked by `sendmail`, such as `read`, `write`, `exec`, etc. Each of these profiles captures the notion of a 'normal' system call invocation by characterizing 'normal' values for one or more of its arguments.

The expected 'normal' values for individual parameters are determined with the help of models. A model is a set of procedures used to evaluate a certain feature of a system call argument, such as the length of a string.

A model can operate in one of two modes, learning or detection. In learning mode, the model is trained and the notion of 'normality' is developed by inspecting examples. Examples are values which are considered part of a regular execution of a program and are either derived directly from a subset of the input set S (learning on-the-fly) or provided by previous program executions (learning from a training set). It is important that the input to the training phase is as exhaustive and free from anomalous system calls as possible, although some models exhibit a certain degree of robustness against polluted or incomplete training data. The gathering of quality training data is a difficult problem by itself and is not discussed in this paper. We assume that a set of system calls is available that was created during normal operation of the program under surveillance. Section 6 describes how we obtained the training data for our experiments.

In detection mode, the task of a model is to return the probability of occurrence of a system call argument value based on the model's prior training phase. This value reflects the likelihood that a certain feature value is observed, given the established profile. The assumption is that feature values with a sufficiently low probability (i.e., abnormal values) indicate a potential attack. To evaluate the overall anomaly score of an entire system call, the probability values of all models are aggregated.

Note that anomaly detection is performed separately for each program. This means that different profiles are created for the same system calls when they are performed by different applications. Although the same models are used to examine the parameters of identical system calls, they are instantiated multiple times and can differ significantly in their notion of 'normality'.

There are two main assumptions underlying our approach. The first is that attacks will appear in the arguments of system calls. If an attack can be carried out without performing system call invocations or without affecting the value of the parameters of such invocations, then our technique will not detect it. The second assumption is that the system call parameters used in the execution of an attack differ substantially from the values used during the normal execution of an application. If an attack can be carried out using system call parameter values that are indistinguishable from the values used during normal execution then the attack will not be detected. The ability to identify abnormal values depends on the effectiveness and sophistication of the models used to build profiles for the system call features. Good models should make it extremely difficult to perform an attack without being detected.

Given the two assumptions above, we developed a number of models to characterize the features of system calls. We used these models to analyze attack data that escaped detection in previous approaches, data that was used in one of the most well-known intrusion detection evaluations [13], as well as data collected on a real Internet server. In all cases, our assumptions proved to be reasonable and the approach delivered promising results.

4 Models

The following section introduces the models that are used to characterize system call arguments and to identify anomalous occurrences. For each model, we describe the creation process (i.e., the learning phase) and explain the mechanism to derive a probability for an argument value (i.e., the detection phase). This probability is then used to obtain an anomaly score for the corresponding argument.

4.1 String Length

Usually, system call string arguments represent canonical file names that point to an entry in the file system. These arguments are commonly used when files are accessed (`open`, `stat`) or executed (`execve`). Their length rarely exceeds a hundred characters and they mostly consist of human-readable characters.

When malicious input is passed to programs, it is often the case that this input also appears in arguments of system calls. For example, consider a format string vulnerability in the log function of an application. Assume further that a failed open call is logged together with the file name. To exploit this kind of flaw, an attacker has to carefully craft a file name that triggers the format string vulnerability when the application attempts and fails to open the corresponding file. In this case, the exploit code manifests itself as an argument to the `open` call that contains a string with a length of several hundred bytes.

Learning The goal of this model is to approximate the actual but unknown distribution of the lengths of a string argument and detect instances that significantly deviate from the observed normal behavior. Clearly, we cannot expect that the probability density function of the underlying real distribution would follow a smooth curve. We also have to assume that it has a large variance. Nevertheless, the model should be able to identify obvious deviations.

We approximate the mean $\dot{\mu}$ and the variance $\dot{\sigma}^2$ of the real string length distribution by calculating the sample mean μ and the sample variance σ^2 for the lengths $l_1, l_2, \ldots, l_n$ of the argument strings processed during the learning phase.

Detection Given the estimated string length distribution with parameters μ and σ^2, it is the task of the detection phase to assess the regularity of an argument string with length l. The probability of l is calculated using the Chebyshev inequality [4].

The Chebyshev inequality puts an upper bound on the probability that the difference between the value of a random variable x and μ exceeds a certain threshold t, for an arbitrary distribution with variance σ^2 and mean μ. Note that although this upper bound is symmetric around the mean, the underlying distribution is not restricted (and our data shows that the string length was not symmetric in the experiments). To obtain an upper bound on the probability that the length of a string deviates more from the mean than the current instance, the threshold t is substituted with the distance between the string length l of the current instance and the mean μ of the string length distribution.

Only strings with lengths that exceed μ are assumed to be malicious. This is reflected in our probability calculation as only the upper bound for strings that are longer than the mean is relevant. Note that an attacker cannot disguise malicious input by padding the string and thus increasing its length, because an increase in length can only reduce the probability value.

The probability value $p(l)$ for a string with length l, given that $l \geq \mu$, is calculated as shown below. For strings shorter than μ, $p(l) = 1$.

$$p(l) = p(|x - \mu| > |l - \mu|) = \frac{\sigma^2}{(l - \mu)^2} \tag{1}$$

We chose the Chebyshev inequality as a reasonable and efficient metric to model decreasing probabilities for strings with lengths that increasingly exceed the mean. In contrast to schemes that define a valid interval (e.g., by recording all strings encountered during the training phase), the Chebyshev inequality takes the variance of the data into account and provides the advantage of gradually changing probability values (instead of a simple 'yes/no' decision).

4.2 String Character Distribution

The string character distribution model captures the concept of a 'normal' or 'regular' string argument by looking at its character distribution. The approach is based on the observation that strings have a regular structure, are mostly human-readable, and almost always contain only printable characters.

A large percentage of characters in such strings are drawn from a small subset of the 256 possible 8-bit values (mainly from letters, numbers, and a few special characters). As in English text, the characters are not uniformly distributed, but occur with different frequencies. However, there are similarities between the character frequencies of parameters of legitimate system calls. This becomes apparent when the relative frequencies of all possible 256 characters are sorted in descending order.

Our algorithm is based only on the frequency values themselves and does not rely on the distributions of particular characters. That is, it does not matter whether the character with the most occurrences is an '`a`' or a '`/`'. In the following, the *sorted*, relative character frequencies of a string are called its *character distribution*. For example, consider the text string '`passwd`' with the corresponding ASCII values of '112 97 115 115 119 100'. The absolute frequency distribution is 2 for 115 and 1 for the four others. When these absolute counts are transformed into sorted, relative frequencies (i.e., the character distribution), the resulting values are 0.33, 0.17, 0.17, 0.17, 0.17 followed by 0 occurring 251 times.

The character distribution of an argument that is perfectly normal (i.e., non-anomalous) is called the argument's *idealized character distribution ($\mathcal{ICD}$)*. The idealized character distribution is a discrete distribution with:

$\mathcal{ICD} : \mathfrak{D} \mapsto \mathfrak{P}$ with $\mathfrak{D} = \{n \in \mathcal{N} | 0 \leq n \leq 255\}$, $\mathfrak{P} = \{p \in \mathfrak{R} | 0 \leq p \leq 1\}$ and $\sum_{i=0}^{255} \mathcal{ICD}(i) = 1.0$.

In contrast to signature-based approaches, the character distribution model has the advantage that it cannot be evaded by some well-known attempts to hide malicious code inside a string. In fact, signature-based systems often contain rules that raise an alarm when long sequences of 0x90 bytes (the `nop` operation in Intel x86-based architectures) are detected in a packet. An intruder may substitute these sequences with instructions that have a similar behavior (e.g., `add rA,rA,0`, which adds 0 to the value in register A and stores the result back to A). By doing this, it is possible to prevent signature-based systems from detecting the attack. Such sequences, nonetheless, cause a distortion of the string's character distribution, and, therefore, the character distribution analysis still yields a high anomaly score.

Learning The idealized character distribution is determined during the training phase. For each observed argument string, its character distribution is stored. The idealized character distribution is then approximated by calculating the average of all stored character distributions. This is done by setting $\mathcal{ICD}(n)$ to the mean of the n^{th} entry of the stored character distributions $\forall n : 0 \leq n \leq 255$. Because all individual character distributions sum up to unity, their average will do so as well. This ensures that the idealized character distribution is well-defined.

Detection Given an idealized character distribution $\mathcal{ICD}$, the task of the detection phase is to determine the probability that the character distribution of an argument is an actual sample drawn from its $\mathcal{ICD}$. This probability is calculated by a statistical test.

This test should yield a high confidence in the correctness of the hypothesis for normal (i.e., non-anomalous) arguments while it should reject anomalous ones. A number of statistical tests can be used to determine the agreement between the idealized character distribution and the actual sample. We use a variant of the Pearson χ^2-test as a 'goodness-of-fit' test [4]. It was chosen because it is simple and efficient to assess the 'normality' of the character distribution.

The χ^2-test requires that the function domain is divided into a small number of intervals, or bins. In addition, it is preferable that all bins contain at least 'some' elements; the literature considers five to be sufficient for most cases. As the exact division of the domain does not influence the outcome of the test significantly, we have chosen the six segments for the domain of $\mathcal{ICD}$ as follows:

{[0], [1,3], [4,6], [7,11], [12,15], [16,255]}. Although the choice of these six bins is somewhat arbitrary, it reflects the fact that the relative frequencies are sorted in descending order. Therefore, the values of $\mathcal{ICD}(x)$ are higher when x is small, and thus all bins contain some elements with a high probability.

When a new system call argument is analyzed, the number of occurrences of each character in the string is determined. Afterward, the values are sorted in descending order and combined by aggregating values that belong to the same segment. The χ^2-test is then used to calculate the probability that the given sample has been drawn from the idealized character distribution. The derived probability value p is used as the return value for this model. When the probability that the sample is drawn from the idealized character distribution increases, p increases as well.

4.3 Structural Inference

Often, the manifestation of an exploit is immediately visible in system call arguments as unusually long strings or strings that contain repetitions of non-printable characters. There are situations, however, when an attacker is able to craft her attack in a manner that makes its manifestation appear more regular. For example, non-printable characters can be replaced by groups of printable characters. In such situations, we need a more detailed model of the system call argument. This model can be acquired by analyzing the argument's structure. For our purposes, the structure of an argument is the regular grammar that describes all of its normal, legitimate values.

For example, consider the first parameter of the `open` system call. It is a null-terminated character string that specifies the canonical name of the file that should be opened. Assume that during normal operation, an application only opens files that are located in the application's home directory and its subdirectories. For this application, the structure of the first argument of the `open` system call should reflect the fact that file names always start with the absolute path name to the program's home directory followed by a (possibly empty) relative path and the file name. In addition, it can be inferred that the relative path is an alternation of slashes and strings. If the directory names consist of lowercase characters only, this additional constraint can be determined as well. When an attacker exploits a vulnerability in this application and attempts to open an 'anomalous' file such as '`/etc/passwd`', an alert should be raised, as this file access does not adhere to the inferred pattern.

Learning When structural inference is applied to a system call argument, the resulting grammar must be able to produce at least all training examples. Unfortunately, there is no unique grammar that can be derived from a set of input elements. When no negative examples are given (i.e., elements that should not be derivable from the grammar), it is always possible to create either a grammar that contains exactly the training data or a grammar that allows production of arbitrary strings. The first case is a form of over-simplification, as the resulting grammar is only able to derive the learned input without providing any level of abstraction. This means that no new information is deduced. The second case is a form of over-generalization because the grammar is capable of producing all possible strings, but there is no structural information left.

The basic approach used for our structural inference is to generalize the grammar as long as it seems to be 'reasonable' and stop before too much structural information is lost. The notion of 'reasonable generalization' is specified with the help of Markov models and Bayesian probability.

In a first step, we consider the set of training items as the output of a *probabilistic* grammar. A probabilistic grammar is a grammar that assigns probabilities to each of its productions. That means that some words are more likely to be produced than others. This fits well with the evidence gathered from system calls, as some parameter values appear more often, and is important information that should not be lost in the modeling step.

A probabilistic regular grammar can be transformed into a non-deterministic finite automaton (NFA). Each state S of the automaton has a set of n_S possible output symbols o which are emitted with a probability of $p_S(o)$. Each transition t is marked with a probability $p(t)$ that characterizes the likelihood that the transition is taken. An automaton that has probabilities associated with its symbol emissions and its transitions can also be considered a Markov model.

The output of the Markov model consists of all paths from its start state to its terminal state. A probability value can be assigned to each output word w (that is, a sequence of output symbols $o_1, o_2, \ldots, o_k$). This probability value (as shown in Equation 2) is calculated as the sum of the probabilities of all distinct paths through the automaton that produce w. The probability of a single path is the product of the probabilities of the emitted symbols $p_{S_i}(o_i)$ and the taken transitions $p(t_i)$. The probabilities of all possible output words w sum up to 1.

$$p(w) = p(o_1, o_2, \ldots, o_k) \;=\; \sum_{(paths\ p\ for\ w)} \; \prod_{(states\ \in\ p)} p_{S_i}(o_i) * p(t_i) \qquad (2)$$

The target of the structural inference process is to find a NFA that has the highest likelihood for the given training elements. An excellent technique to derive a Markov model from empirical data is explained in [21]. It uses the Bayesian theorem to state this goal as

$$p(Model|TrainingData) \;=\; \frac{p(TrainingData|Model) * p(Model)}{p(TrainingData)} \qquad (3)$$

The probability of the training data is considered a scaling factor in Equation 3 and it is subsequently ignored. As we are interested in maximizing the *a posteriori* probability (i.e., the left-hand side of the equation), we have to maximize the product shown in the enumerator on the right-hand side of the equation. The first term – the probability of the training data given the model – can be calculated for a certain automaton (i.e., for a certain model) by adding the probabilities calculated for each input training element as discussed above. The second term – the prior probability of the model – is not as straightforward. It has to reflect the fact that, in general, smaller models are preferred. The model probability is calculated heuristically and takes into account the total number of states as well as the number of transitions and emissions at each state. This is justified by the fact that smaller models can be described with less states as well as fewer emissions and transitions.

The model building process starts with an automaton that exactly reflects the input data and then gradually merges states. This state merging is continued until the *a posteriori* probability no longer increases. The interested reader is referred to [21] and [22] for details.

Detection Once the Markov model has been built, it can be used by the detection phase to evaluate string arguments. When the word is a valid output from the Markov model, the model returns 1. When the value cannot be derived from the given grammar, the model returns 0.

4.4 Token Finder

The purpose of the token finder model is to determine whether the values of a certain system call argument are drawn from a limited set of possible alternatives (i.e., they are tokens or elements of an enumeration). An application often passes identical values to certain system call parameters, such as flags or handles. When an attack changes the normal flow of execution and branches into maliciously injected code, such constraints are often violated. When no enumeration can be identified, it is assumed that the values are randomly drawn from the argument type's value domain (i.e., random identifiers for every system call).

Learning The classification of an argument as an enumeration or as a random identifier is based on the observation that the number of different occurrences of parameter values is bound by some unknown threshold t in the case of an enumeration, while it is unrestricted in the case of random identifiers.

When the number of different argument instances grows proportional to the total number of arguments, the use of random identifiers is indicated. If such an increase cannot be observed and the number of different identifiers follows a standard diminishing growth curve [14], we assume an enumeration. In this case, the complete set of identifiers is stored for the subsequent detection phase.

The decision between an enumeration and unique identifiers can be made utilizing a simple statistical test, such as the non-parametric Kolmogorov-Smirnov variant as suggested in [14]. This paper discusses a problem similar to our token finder for arguments of SQL queries and its solution can be applied to our model.

Detection When it was determined that the values of a system call argument are tokens drawn from an enumeration, any new value is expected to appear in the set of known identifiers. When it does, 1 is returned, 0 otherwise. When it is assumed that the parameter values are random identifiers, the model always returns 1.

5 Implementation

Based on the models presented in the previous section, we have implemented an intrusion detection system (IDS) that detects anomalies in system call arguments for Linux 2.4. The program retrieves events that represent system call invocations from an operating system auditing facility called Snare [20]. It computes an anomaly score for each system call and logs the event when the derived score exceeds a certain threshold.

Our intrusion detection tool monitors a selected number of security-critical applications. These are usually programs that require `root` privileges during execution such as server applications and `setuid` programs. For each program, the IDS maintains data structures that characterize the normal profile of every monitored system call. A system call profile consists of a set of models for each system call argument and a function that calculates the anomaly score for input events from the corresponding model outputs.

Before the IDS can start the actual detection process, it has to complete a training phase. This training phase is needed to allow the used models to determine the characteristics of normal events and to establish anomaly score thresholds to distinguish between regular and malicious system calls.

The training phase is split into two steps. During the first step, our system establishes profiles for the system calls of each monitored application. The received input events are directly utilized for the learning process of the models.

During the second step, suitable thresholds are established. This is done by evaluating input events using the profiles created during the previous step. For each profile, the highest anomaly score is stored and the threshold is set to a value that is a certain, adjustable percentage higher than this maximum. The default setting for this percentage (also used for our experiments) is 10%. By modifying the percentage, the user can adjust the sensitivity of the system and perform a trade-off between the number of false positives and the expected detection accuracy. As each profile uses its own threshold, the decision can be made independently for each system call for every monitored application. This fine-grained resolution allows one to precisely tune the IDS.

Once the profiles have been established – that is, the models have learned the characteristics of normal events and suitable thresholds have been derived – the system switches to detection mode. In this mode, each system call executed by an application is evaluated with respect to the corresponding profile. If the computed anomaly value is higher than the established threshold, an alarm is raised.

The anomaly score AS is equal to the sum of the negative logarithms of the probability values p_m returned by each model m that is part of the profile, that is, $AS = \sum_m -\log(p_m)$. To prevent $-\log(p_m)$ from getting too large when p_m

is close to 0, we set a lower bound of 10^{-6} for p_m. The equation is chosen such that low probability values have a pronounced effect on the final score.

All detection models used by our system are implemented as part of a general library. This library, called *libAnomaly*, provides a number of useful abstract entities for the creation of anomaly-based intrusion detection systems and makes frequently-used detection techniques available. The library has been created in response to the observation that almost all anomaly-based IDSs have been developed in an ad-hoc way. Much basic functionality is implemented from scratch for every new prototype and also the authors themselves have written several instances of the same detection technique for different projects.

6 Experimental Results

This section details the experiments undertaken to evaluate the classification effectiveness and performance characteristics of our intrusion detection system.

The goal of our tool is to provide reliable classification of system call events in a performance-critical server environment. This requires that the system performs accurate detection while keeping the number of false alerts extremely low.

6.1 Classification Effectiveness

To validate the claim that our detection technique is accurate, a number of experiments were conducted.

For the first experiment, we ran our system on the well-known 1999 MIT Lincoln Labs Intrusion Detection Evaluation Data [13]. We used data recorded during two attack free weeks (Week 1 and Week 3) to train our models and then performed detection on the test data that was recorded during two subsequent weeks (Week 4 and Week 5).

The truth file provided with the evaluation data lists all attacks carried out against the network installation during the test period. When analyzing the attacks, it turned out that many of them were reconnaissance attempts such as network scans or port sweeps, which are only visible in the network dumps and do not not leave any traces in the system calls. Therefore, we cannot detect them with our tool.

Another class of attacks are policy violations. These attacks do not allow an intruder to elevate privileges directly. Instead, they help to obtain information that is classified as secret by exploiting some system misconfiguration. This class of attacks contains intrusions that do not exploit a weakness of the system itself, but rather take advantage of a mistake that an administrator made in setting up the system's access control. Such incidents are not visible for our system either, as information is leaked by 'normal' but unintended use of the system.

The most interesting class of attacks are those that exploit a vulnerability in a remote or local service, allowing an intruder to elevate her privileges. The MIT Lincoln Labs data contains 25 instances of attacks that exploit buffer overflow vulnerabilities in four different programs. Table 1 summarizes the results found for the attacks against these four programs: `eject`, `ps`, `fdformat`, and `ffbconfig`. In addition, we present results for interesting daemon and `setuid` programs to assess the number of the false alerts. The *Total* column shows the sum of all system calls that are executed by the corresponding program and analyzed by our system. The *Attacks* column shows the number of attacks against the vulnerable programs in the data set. *Identified Attacks* states the number of attacks that have been successfully detected by our system and, in parentheses, the number of corresponding system calls that have been labeled as anomalous. It is very common that attacks result in a series of anomalous system calls. The *False Alarms* column shows the number of system calls that have been flagged anomalous although these invocations are not related to any attack.

Application	Total Syscalls	Attacks	Identified Attacks	False Alarms
`eject`	138	3	3 (14)	0
`fdformat`	139	6	6 (14)	0
`ffbconfig`	21	2	2 (2)	0
`ps`	4,949	14	14 (55)	0
`ftpd`	3,229	0	0 (0)	14
`sendmail`	71,743	0	0 (0)	8
`telnetd`	47,416	0	0 (0)	17
`Total`	127,635	25	25 (85)	39

Table 1. 1999 MIT Lincoln Labs Evaluation Results

In the second experiment, we evaluated the ability of our system to detect a number of recent attacks. Four daemon programs and one `setuid` tool were installed to simulate a typical Internet server. After the test environment was prepared, the intrusion detection system was installed and trained for about one hour. During the training period, we attempted to simulate normal usage of the system. Then, the intrusion detection system was switched to detection mode and more extensive tests were conducted for five more hours. No malicious activity took place. After that, we carried out three actual exploits against the system, one against `wuftpd`, one against `linuxconf` and one against `Apache`. All of them were reliably detected. As our system is currently not able to automatically determine when enough training data has been processed, the duration of the training period was chosen manually.

The left part of Table 2 shows, for each application, the number of analyzed system calls, the number of detected attacks with their corresponding anomalous system calls and the number of false alerts. An analysis of the reported false alerts confirms that all alarms were indications of anomalous behavior that was not encountered during the training phase. Although the anomalous situations were not caused by malicious activity, they still represent deviations from the 'normal' operation presented during the learning process. While many useful generalizations took place automatically and no alerts were raised when new files were accessed, the login of a completely new user or the unexpected termination of processes were still considered suspicious.

Application	Total Syscalls	Attacks	Identified Attacks	False Alarms
wuftpd	4,887	1	1 (86)	1
Apache	17,274	1	1 (2)	0
OpenSSH	9,562	0	0 (0)	6
sendmail	15,314	0	0 (0)	5
linuxconf	4,422	1	1 (16)	3
Total	51,459	3	3 (104)	15

Controlled Environment

Application	Total Syscalls	False Alarms
dhcpd	431	0
imapd	418,152	4
qmail	77,672	11
Total	496,255	15

Real-World Environment

Table 2. Detection Accuracy

The `7350wu` attack exploits an input validation error of `wuftpd` [1]. It was chosen because it was used by Wagner and Soto [26] as the basis for a mimicry attack to evade detection by current techniques based on system call sequences. Our tool labeled 86 system calls present in the trace of the `7350wu` attack as anomalous, all directly related to the intrusion. 84 of these anomalies were caused by arguments of the `execve` system call that contained binary data and were not structurally similar to argument values seen in the training data.

It should be noted that none of these anomalies would be missing were the exploit disguised using the mimicry technique suggested by Wagner and Soto [26]. Since each system call is examined independently, the insertion of intervening system calls to modify their sequence does not affect the classification of the others as anomalies. This shows that our technique is not affected by attempts to imitate normal system call sequences. Note that this does not imply that our IDS is immune to all possible mimicry attacks. However, by combining our system with a sequence-based approach the potential attack space is reduced significantly. This is due to the fact that the approaches are complimentary and an attacker would have to subvert both systems.

The attack against `linuxconf` exploits a recently discovered vulnerability [2] in the program's handling of environment variables. When the exploit was run, the intrusion detection system identified 16 anomalous `open` system calls with suspicious path arguments that caused the string length, the character distribution and the structural inference model to report an anomalous occurrence. Another example is the structural inference model alerting on `open` being invoked with the argument '`segfault.eng/segfault.eng`'. This is a path which is used directly by the exploit and never occurs during normal program execution.

The attack against `apache` exploits the `KEY_ARG` vulnerability in `OpenSSL v0.9.6d` for `Apache/mod_ssl`. When the attack is launched, our system detects two anomalous system calls. One of these calls, `execve`, is reported because `Apache` does not create a `bash` process during normal operation.

The third experiment was conducted to obtain a realistic picture of the number of false alerts that can be expected when the system is deployed on a real-world server. We installed our program on the group's e-mail server, trained the models for a period of two days and then performed detection on several important daemons (`qmail`, `imapd`, `dhcpd`) for the subsequent five days. The right part of Table 2 shows the number of analyzed system calls as well as the number of false alerts raised during the five days, listed for each of the monitored applications.

6.2 System Efficiency

To quantify the overhead of our intrusion detection system, we have measured its time and space performance characteristics.

The memory space required by each model is practically independent of the size of the training input. Although temporary memory usage during the learning phase can grow proportional to the size of the training data, eventually the models abstract this information and occupy a near constant amount of space. This is reflected in Table 3 that shows the memory used by our system for two different runs after it had been trained with data from normal executions of `wuftpd` and `linuxconf`, respectively. The results confirm that memory usage is very similar for both test runs, although the size of the input files is different by a factor of 2.5.

Application	Training Data Size	Memory Usage
`wuftpd`	37,152K	5,842K
`linuxconf`	14,663K	5,264K

Table 3. IDS Memory Usage

To obtain measures that can quantify the impact of our intrusion detection system on a heavily utilized server, we set up a small dedicated network consisting of three PCs (1.4 GHz Pentium IV, 512 MB RAM, Linux 2.4) connected via a 100 Mbps Ethernet. One server machine hosted the intrusion detection system and `wuftpd`. The remaining two PCs ran multiple FTP client applications which continuously downloaded files from the server. This experiment was run three times: once with `wuftpd` only, once with `wuftpd` and the auditing facility, and finally once with `wuftpd`, the auditing, and our intrusion detection system. In no cases were audit records dropped.

The server performance experienced by each client was virtually indistinguishable for all three cases. This indicates that the number of system calls that have to be analyzed every second by the intrusion detection system (210 on average in this case) is too low to be noticeable as performance degradation. Further analysis showed that the bottleneck in this experiment was the network.

To increase the system call rate to a point that would actually stress the system, we developed a synthetic benchmark that can execute a variable number of system calls per second at a rate that far exceeds the rate of system calls normally invoked by server applications. By measuring the resulting CPU load for different rates of system calls, we obtain a quantitative picture of the impact of our detection tool and its ability to operate under very high loads.

We ran the benchmark tool on an otherwise idle system for varying system call rates three times: once without any auditing, once with system call auditing

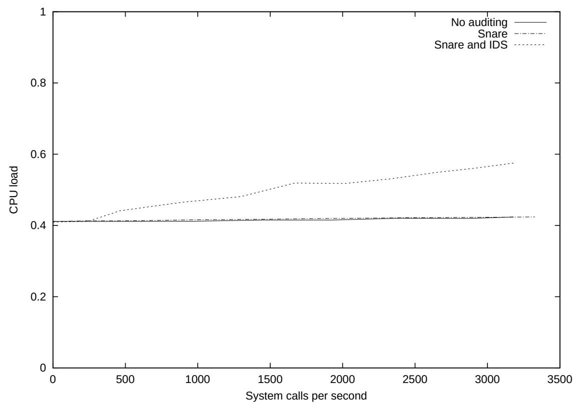

Fig. 1. CPU Load for different System Call Rates

(i.e., Snare), and finally once with system call auditing (i.e., Snare) and our intrusion detection system. Figure 1 shows the resulting CPU load observed on the system as an average of 10 runs.

The benchmark application used approximately 40% of the CPU on an idle system without auditing. As the number of system calls per second increased, a negligible impact on the CPU was observed, both with auditing turned completely off and with auditing in place. When our intrusion detection system was enabled, the CPU load increased up to 58%, when the benchmark performed about 3000 system calls per second. Note that this rise was caused by a nearly fifteen-fold increase of the number of system calls per second compared to the number that needed to be analyzed when `wuftp` was serving clients on a saturated fast Ethernet.

7 Conclusions

For a long time system calls and their arguments have been known to provide extensive and high quality audit data. Their analysis is used in security applications to perform signature-based intrusion detection or policy-based access control. However, learning-based anomaly intrusion detection has traditionally focused only on the sequence of system call invocations. The parameters have been neglected because their analysis has been considered either too difficult or too expensive computationally.

This paper presents a novel approach that overcomes this deficiency and takes into account the information contained in system call arguments. We introduce several models that learn the characteristics of legitimate parameter values and are capable of finding malicious instances. Based on the proposed detection techniques, we developed a host-based intrusion detection tool that monitors running applications to identify malicious behavior. Our experimental evaluation shows that the system is effective in its detection and efficient in its resource usage.

References

1. Advisory: Input validation problems in wuftpd. `http://www.cert.org/advisories/CA-2000-13.html`, 2000.
2. Advisory: Buffer overflow in linuxconf. `http://www.idefense.com/advisory/08.28.02.txt`, 2002.
3. M. Bernaschi, E. Gabrielli, and L. V. Mancini. REMUS: a Security-Enhanced Operating System. *ACM Transactions on Information and System Security*, 5(36), February 2002.
4. Patrick Billingsley. *Probability and Measure*. Wiley-Interscience, 3 edition, April 1995.
5. Suresh N. Chari and Pau-Chen Cheng. Bluebox: A policy-driven, host-based intrusion detection system. In *Proceedings of the 2002 ISOC Symposium on Network and Distributed System Security (NDSS'02)*, San Diego, CA, 2002.
6. D.E. Denning. An Intrusion Detection Model. *IEEE Transactions on Software Engineering*, 13(2):222–232, February 1987.
7. S.T. Eckmann, G. Vigna, and R.A. Kemmerer. STATL: An Attack Language for State-based Intrusion Detection. *Journal of Computer Security*, 10(1/2):71–104, 2002.
8. S. Forrest. A Sense of Self for UNIX Processes. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 120–128, Oakland, CA, May 1996.
9. A.K. Ghosh, J. Wanken, and F. Charron. Detecting Anomalous and Unknown Intrusions Against Programs. In *Proceedings of the Annual Computer Security Application Conference (ACSAC'98)*, pages 259–267, Scottsdale, AZ, December 1998.
10. Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the 6th Usenix Security Symposium*, San Jose, CA, USA, 1996.
11. H. S. Javitz and A. Valdes. The SRI IDES Statistical Anomaly Detector. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 1991.
12. C. Ko, M. Ruschitzka, and K. Levitt. Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-based Approach. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 175–187, May 1997.
13. MIT Lincoln Laboratory. DARPA Intrusion Detection Evaluation. http://www.ll.mit.edu/IST/ideval/, 1999.
14. Sin Yeung Lee, Wai Lup Low, and Pei Yuen Wong. Learning Fingerprints for a Database Intrusion Detection System. In *7th European Symposium on Research in Computer Security (ESORICS)*, 2002.
15. W. Lee, S. Stolfo, and P. Chan. Learning Patterns from Unix Process Execution Traces for Intrusion Detection. In *Proceedings of the AAAI Workshop: AI Approaches to Fraud Detection and Risk Management*, July 1997.
16. W. Lee, S. Stolfo, and K. Mok. Mining in a Data-flow Environment: Experience in Network Intrusion Detection. In *Proceedings of the* 5^{th} *ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD '99)*, San Diego, CA, August 1999.
17. U. Lindqvist and P.A. Porras. Detecting Computer and Network Misuse with the Production-Based Expert System Toolset (P-BEST). In *IEEE Symposium on Security and Privacy*, pages 146–161, Oakland, California, May 1999.
18. V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, January 1998.

19. Niels Provos. Improving host security with system call policies. In *Proceedings of the 12th Usenix Security Symposium*, Washington, DC, 2003.
20. SNARE - System iNtrusion Analysis and Reporting Environment. `http://www.intersectalliance.com/projects/Snare`.
21. Andreas Stolcke and Stephen Omohundro. HiddenMarkov Model Induction by Bayesian Model Merging. *Advances in Neural Information Processing Systems*, 1993.
22. Andreas Stolcke and Stephen Omohundro. Inducing probabilistic grammars by bayesian model merging. In *International Conference on Grammatical Inference*, 1994.
23. K. Tan and R. Maxion. "Why 6?" Defining the Operational Limits of Stide, an Anomaly-Based Intrusion Detector. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 188–202, Oakland, CA, May 2002.
24. K.M.C. Tan, K.S. Killourhy, and R.A. Maxion. Undermining an Anomaly-Based Intrusion Detection System Using Common Exploits. In *Proceedings of the* 5th *International Symposium on Recent Advances in Intrusion Detection*, pages 54–73, Zurich, Switzerland, October 2002.
25. D. Wagner and D. Dean. Intrusion Detection via Static Analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2001. IEEE Press.
26. D. Wagner and P. Soto. Mimicry Attacks on Host-Based Intrusion Detection Systems. In *Proceedings of the* 9th *ACM Conference on Computer and Communications Security*, pages 255–264, Washington DC, USA, November 2002.
27. C. Warrender, S. Forrest, and B.A. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *IEEE Symposium on Security and Privacy*, pages 133–145, 1999.