Application Intrusion Detection using Language Library Calls

Anita K. Jones
jones@virginia.edu

Yu Lin
yl9e@cs.virginia.edu

University of Virginia

Abstract

Traditionally, intrusion detection systems detect intrusions at the operating system (OS) level. In this paper we explore the possibility of detecting intrusion at the application level by using rich application semantics. We use short sequences of language library calls as signatures. We consider library call signatures to be more application-oriented than system call signatures because they are a more direct reflection of application code. Most applications are written in a higher-level language with an associated support library, such as C or C++. We hypothesize that library call signatures can be used to detect attacks that cause perturbation in the application code. We are hopeful that this technique will be amenable to detecting attacks that are carried out by internal intruders, who are viewed as legitimate users by an operating system.

1. Introduction

An intrusion can be defined as any set of actions that attempt to compromise the integrity, confidentiality or availability of a resource [3]. There are two types of intruders, internal and external. External intruders do not have any authorized access to the system they attack. Internal intruders have some authority and therefore some legitimate access, but seek to gain additional ability to take action without legitimate authorization.

We can improve security through the use of tools such as Intrusion Detection Systems. An intrusion detection system, or IDS for short, detects either attempted intrusions into a system or activities of intruders after breaking in. Traditionally, intrusion detection systems detect intrusions at the OS level by comparing expected and observed system resource usage. Unfortunately, OS intrusion detection systems are typically insufficient to catch internal intruders because they are already legitimate users of the system. Their activities neither significantly deviate from expected behavior, nor exhibit the anticipated actions of first entry into the operating system from the outside.

One reason for the insufficiency of OS intrusion detection systems is that they depend only on resource usage as seen by the OS. Our approach is to detect intrusions at the application level using sequences of language library calls as signatures for program behavior. We regard an application as a black box that can emit some observable events (library call invocations) when executing. Thus our technique does not require analysis of the semantics of the application.

2. Related Work

Stephanie Forrest at the University of New Mexico proposed a method of intrusion detection using sequences of system calls [1][2][4]. This method uses short sequences of system call invocations as being descriptive of the execution of privileged processes in Unix systems. The method used to build a signature database is to trace system calls generated by a particular program, slide a window of size k across the trace, and record each unique sequence of length k that is encountered in the window. This method for enumerating sequences is called sequence time-delay embedding (stide). Intrusion detection experiments show that short sequences of system calls can be a remarkably good discriminator between normal and abnormal operating characteristics of common Unix programs. In other words, sequences of system calls are highly likely to be perturbed by intrusive activities.

The increasing trend towards distributed platforms, exemplified by CORBA, presents new challenges for intrusion detection. Researchers at Odyssey Research Associates have applied similar techniques to CORBA-based applications to detect intrusion in distributed object applications at the application level [8].

3. Library call signatures

Our library call signatures are defined to be sequences of library call invocations. For simplicity, only the identity of library calls and their sequence are preserved. All other aspects of a library call invocation, such as arguments, are ignored. Then we build a signature

database for an application by enumerating an "adequate" number of unique short sequences in a way that is similar to the technique used by the New Mexico researchers. Finally, we monitor the execution of the specific application for significant deviations. If "enough" of the observed sequences are different from those in the signature database, there may be intrusions.

3.1 Signature database definition

We use the term *robust* to describe a signature database that contains sufficient sequences to characterize the application in a substantial way. The algorithm used to build a robust signature database is to scan traces of library calls generated by the target application. We slide a window of size k across the trace and record each unique sequence of length k that is encountered. When monitoring multiple processes that execute the target application, we restrict sequences to library calls from one process, i.e. we don't mix library calls from multiple processes in one sequence.

Each application is characterized by its signature database. This means that, in practice, each application has a different signature database. That signature database is specific to application code, hardware architecture, software version and configuration, local administrative policies, and usage patterns of the user community. Once a robust signature database is constructed for a given application, it can be used to monitor the ongoing behavior of the processes that result from invoking that application.

The structure of the signature database is best illustrated with a concrete example. Suppose we have the following trace of library calls:

fopen, fread, strcmp, strcmp, fopen, fread, strcmp

We slide a window of size k across the trace and record each unique sequence of length k that is encountered. For example, if k = 3, then the result is the four unique sequences of length 3 depicted in Figure 1.

For efficiency, these sequences can be stored as trees in the database. Each tree is rooted at the first library call in its sequence. Two sequences with initial identical segments will share the same initial tree structure in the database. The advantage of trees is that the storage requirements are lower and that comparison of monitored sequences to sequences in the database is efficient.

3.2 Synthetic and real signature databases

There are two approaches to building a robust signature database. The *synthetic* signature database is built by exercising as many normal modes of usage of an application as possible. The *real* signature database derives from tracing the actual execution of the program in a live user environment. Usually it takes more time and facilities to build a real signature database than a synthetic signature database.

A real signature database may not be as robust as a synthetically built signature database. When building a signature database in a real, open environment, it is difficult to ensure that no intrusion occurred during database generation. Also, in a real execution environment, there is no mechanism to force execution of a majority of paths through the application code.

Two signature databases collected for the identical program and operating system in different real environments may differ significantly both in size, and in content. Unlike real signature databases, synthetic signature databases tend to be more similar because experimenters use a similar strategy to stimulate application execution. Therefore, synthetic databases are useful for replicating results, comparing performance in different settings, and performing different kinds of controlled experiments. For those reasons, we chose to experiment with synthetic signature databases.

3.3 Different Anomaly Measures

Once a signature database has been built, monitoring for deviant behavior uses essentially the same method that is used to build the signature database. Sequences of length k from the monitored behavior are compared to those in the signature database. If monitored sequences deviate significantly from those in the signature database, the application behavior is considered to be anomalistic.

We next define several different anomaly measures to express the strength of an anomalous signal. In this paper,

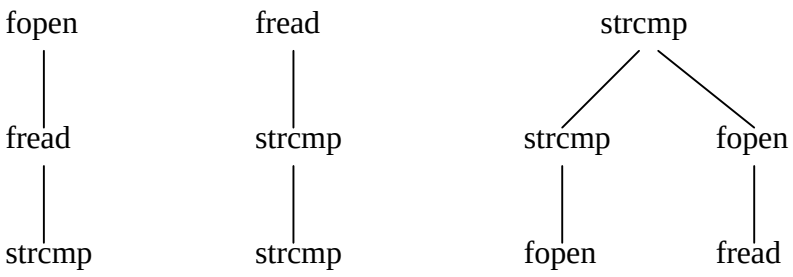

Figure 1. An example of library call sequence trees

we discuss three anomaly measures: the mismatch count, the locality frame count, and the normalized anomaly signal.

Monitored sequences that are not in the signature database are defined to be *mismatches*. The *mismatch count* records the number of mismatches. In experiments, we use not only the raw mismatch count, but also the percentage of mismatches (mismatch count divided by the total number of monitored sequences).

To determine that a new sequence of length k is a mismatch requires at most k comparisons, because the sequences in the signature database are stored as a forest of trees, where the root of each tree corresponds to a different library call. Similarly, it takes k comparisons to determine a match.

How many mismatches does it take to indicate truly anomalous behavior? Recall that the signature database is not guaranteed to contain all possible legitimate sequences. One answer is to count the number of mismatches occurring in the monitored behavior, and only consider that behavior to be anomalous when a mismatch threshold is reached. Even this solution is problematic because the mismatch count is dependent on trace length and some processes execute endlessly. Another answer is to compute the percentage of mismatches once "enough" sequences have been encountered to assure reasonable initialization.

The second measure is the *locality frame count.* It is based on the assumption that anomalous sequences due to intrusions will occur in local bursts. When a process is exploited, there may be a short period of time – a locality – when the percentage of anomalous sequences is much higher. For example, ten anomalies over the course of a long run may not be cause for concern. But ten anomalies within thirty overlapping sequences might be. Thus, it can be useful to observe how many anomalies occur during a limited interval. The number of sequences that are considered to be local to one another is called the size of the locality frame. In our experiments, we arbitrarily choose 20 as a (reasonable) size for the locality frame. We report the largest number of anomalies found within each locality frame. One advantage of the locality frame count is that it provides a real-time measure. Because the locality frame count is calculated locally, a system administrator can immediately be notified when an intrusion may be occurring.

The third measure, widely used to detect intrusions, is called *the normalized anomaly signal*, $\hat{S}_A$ [4]. It characterizes how much one sequence differs from existing sequences in the signature database. The difference between two sequences s1 and s2 is the Hamming distance d(s1, s2) between them, that is the number of calls by which they differ. For a monitored sequence s1, the minimal Hamming distance d_{min} (s1) is defined as the minimum of all Hamming distance measure between s1 and the sequences in the database, i.e.,

d_{min} (s1) = min { d(s1, s) for all sequences s in the signature database }.

Again, the d_{min} value indicates how much a monitored sequence, s1, differs from the signature database. When the execution of an application is monitored, the maximum d_{min} value that was encountered in a trace represents the strongest anomalous signal found in the monitored trace. So the anomaly signal, S_A, is defined as:

S_A = max { d_{min}(s) for all monitored sequences s }.

In order to compare S_A values when k varies, the anomaly signal $\hat{S}_A$ is normalized by the sequence length k, i.e.:

$\hat{S}_A = S_A / k$.

The normalized anomaly signal $\hat{S}_A$ is more compute intensive than other two measures. Like the mismatch count, it takes k comparisons to determine a match, and takes at most k comparisons to determine a mismatch. If a mismatch exists, we compute d_{min}. Because d_{min} (s) is the smallest Hamming distance between s and all sequences in the signature database, every sequence in the database must be checked to determine d_{min} (s). Assume that N is the number of sequences in the signature database, then the cost totals k*N comparisons. In reality mismatches are rare, so most of time, the algorithm confirms matches at a cost of k comparisons. If the rate of mismatches to matches is R_A, then the average complexity of computing d_{min} (s) per sequence is $(k*N*R_A) + k*(1-R_A)$, which is $O(k*R_A*N)$.

The normalized anomaly signal, $\hat{S}_A$, expresses how much a monitored sequence deviates from the signature database in a way that is independent of sequence length. The value of the normalized anomaly signal, $\hat{S}_A$, is between 0 and 1.

All three anomaly measures can be used to express thresholds, that when crossed, indicate the probability that an intrusion has occurred.

3.4 False positives vs. false negatives

An intrusion detection system can make intrusion detection decisions based on the observed values of the above measures. In the simplest case, these are binary decisions: Either a sequence is anomalous, or it is normal. There are two types of classification errors: false positives and false negatives [4]. A false positive, also known as a

false detection or false alarm, occurs when a sequence generated by legitimate behavior is classified as anomalous. A false negative occurs when none of the sequences generated by an intrusion are detected as anomalous, i.e., all sequences generated by the intrusion appear in the signature database.

We would like to minimize both types of classification errors. But system administrators are more willing to tolerate false negatives than false positives because false negatives can be reduced by adding layers of defense while layering will not reduce overall false positives. In some cases, layering even compounds false positives. The reason for false positives is that in reality it is difficult to collect signature sequences of *all* normal behavior for a complex application. Therefore, we set thresholds on the normalized anomaly signal values to limit false positives. We regard an intrusion to be in progress when the normalized anomaly signal, $\hat{S}_A$, is greater than C (0 <= C <= 1) where C is chosen based on experience. In our following experiments, we chose C as 0.5.

4. Experimental environment

We experimented with our library call approach using a variety of applications in Unix. These programs vary from small applications (e.g., *ls* and *ps*) to large applications (e.g., the *Apache httpd server*, and *Mini SQL*). The applications differ not only in size, but also in privilege. Privileged processes perform services that require access to system resources normally inaccessible to ordinary users. In our experiments, some applications run as privileged processes (e.g., *wu-ftpd*), while others (e.g., *finger*) execute without privilege. We observed no difference in the efficiency of our intrusion detection method based on this difference.

All signature data reported in this paper were derived from experiments performed under Linux, a Unix-based operating system. We selected Linux since its source code, which was developed under the GNU General Public License, is freely available. We used Red Hat 5.2 version of Linux with kernel version 2.0.36. We chose such an early kernel version because all experiment data about intrusion detection using sequences of system calls by the New Mexico researchers were conducted with this version of the kernel. We wanted to compare our library call approach with their system call approach in a similar environment.

5. Experimental results

Our experimentation required that we build signature databases for our selected applications. As discussed in the previous section, we chose to build synthetic signature databases. Next we performed experiments to determine if it is possible to detect intrusions that exploit security flaws in our selected applications. We needed to identify the vulnerabilities, i.e. weakness that could be exploited, in these applications. In a controlled experimental environment, intrusions were performed. Then we analyzed the intrusion detection results and compared them with the results of other approaches.

5.1 Build robust signature databases

This section explains the construction of experimental signature databases. A signature database is described as *robust* if the database records a sufficient portion of the legitimate sequences of library calls so that the false positive rate remains tolerable. Figure 2 shows the incremental generation of sequences to be added to the signature database for *ps*, a program to report process status. To build the signature database, the application was exercised extensively, i.e., in a carefully controlled way in order to exercise as much of code as possible. Unique sequences are added to the signature database as encountered. Figure 2 illustrates that the database size initially increases rapidly. Gradually, the number of new unique sequences to be added to the database drops off. We define the database to be sufficiently robust when the slope of growth "flattens", i.e., the number of unique sequences added to the database per the total sequences generated falls below a threshold set for the application.

Signature databases will differ greatly in size. Typically, larger applications (measured in source lines of code) are associated with larger databases. For example, our signature database the *Apache httpd server*, the most widely used web server, contains 1383 unique sequences of length 10. However, signature databases can be quite compact. For example, the signature database for *ps* contains only 571 unique sequences of length 10. To put this in context, *ps* contains no more than 60 different C library calls. So, for k=10 there are 60^{10} possible sequences. Thus our *ps* signature database contains only a very tiny percentage of the total possible number of sequences from the use of 60 different library functions.

5.2 Distinguish between programs

In this section we discuss comparison of the sequences generated by one application with the signature databases of other applications. We reasoned that if we could not distinguish between two applications in the absence of intrusion, then sequences of library calls do not characterize an application. As a result, sequences of library calls from an intrusion are unlikely to be detectable.

We performed this comparison for varying sequence lengths. We simply compared the sequences in one

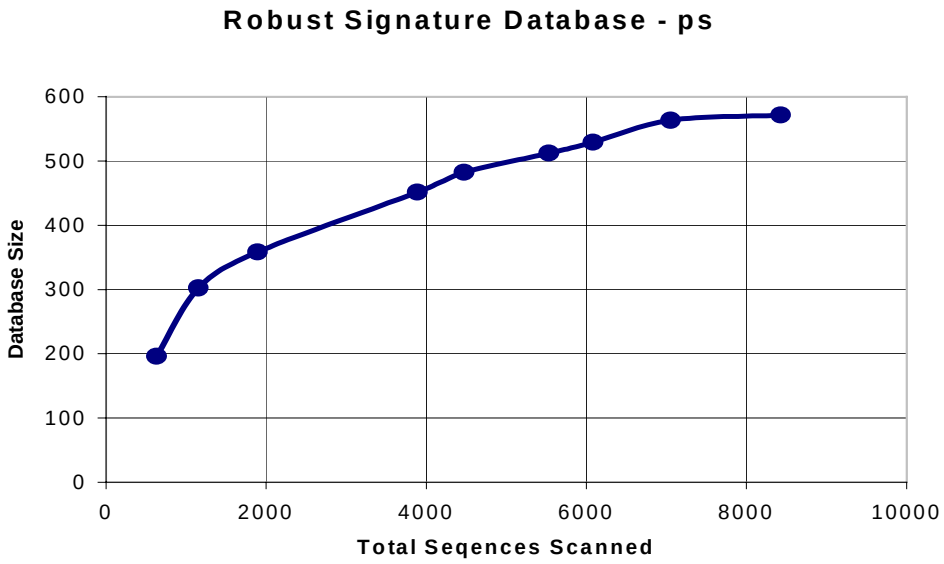

Figure 2. Derivation of a signature database for *ps* program. The x-axis indicates the total sequences generated, and the y-axis indicates the number of unique sequences. Sequence length is 10.

database against signature databases for other applications. When the sequence length is very low (close to one), there are very few mismatches. When the sequence length reaches k=30, mismatches are almost 100% against all applications. Results of comparisons of the *Apache httpd server* with other Unix utilities for k=10 are presented in Table 1. The results are based on the *Apache httpd server* signature database described in the previous section. We used New Mexico STIDE tool to compare the traces of library calls of these utilities against the *Apache httpd* database.

Program	Number Mismatches	%Mismatches
ls	209	76
ls –l	360	84
ls –a	356	69
finger	239	100

Table 1. Distinguishing *Apache* from other programs. Each column reports results for an anomaly measure: mismatches and percentage of mismatches. Sequence length is 10.

Each program showed a significant number of different sequences from the *Apache httpd server* signature database (at least 69%). The results demonstrated that the behavior of different programs is distinguishable using sequences of C library calls, i.e., library call signatures can be used to characterize applications.

5.3 Sequence length analysis

Next we preformed several experiments to explore the relationship between the library call sequence length and intrusion detection. We experimented with known attacks that exploit the vulnerabilities [6] in the *Apache httpd server*. We successfully detected intrusions using our library call signature.

The intrusion detection results for sequence length k = 10 are reported below. Although the percentage of mismatches is low, the normalized anomaly signal $\hat{S}_A$ indicates there may be intrusions because the most anomalous sequence among all of monitored sequences differs from the normal sequences in over 70 percent of its positions.

Attacks against *Apache*	Number Mismatches	Percent Mismatches	$\hat{S}_A$
Phf Remote Command Execution	251	1.5 %	0.7
nph-test-cgi Vulnerability	777	3.9 %	0.8

Table 2. Successful detection of two attacks against *Apache*. Each column reports results for an anomaly measure: mismatches, percentage of mismatches, and normalized anomaly signal. Sequence length is 10.

We varied sequence length, k, from 2 to 30. The minimum sequence length used is 2 because k=1 will yield $\hat{S}_A = 0$ or $\hat{S}_A = 1$, which is meaningless for sequence analysis. The maximum sequence length used here is 30. Note that the cost of computation increases significantly with sequence length. We observed that adequate detection resulted from much shorter sequences.

From Figure 3, we can conclude that varying the sequence length makes little difference in value of the normalized anomaly signal $\hat{S}_A$. In other words, varying sequence length has little effect on intrusion detection using C library call signatures. Also, we infer that the intrusion detection result is stable if sequence length is larger than 6. Considering that computation cost proportional to sequence length, we can see that sequence length 10 is sufficient for detecting *phf* vulnerability attack and that sequence length 9 is sufficient for detecting *nph-test-cgi* vulnerability attack. In our experiments, we chose sequence length 10 to allow for some margins of error while incurring acceptable computation cost.

5.4 Various intrusion detection experiments

To test the effectiveness of the library call approach in detecting intrusions, we experimented with a variety of applications and intrusions, including Buffer Overflow, Trojan programs, and Denial of Service. The purpose of this set of experiments is to prove that the library call approach is a general and effective way to detect different kinds of intrusions. The following sections discuss our results for each of the three attacks.

5.4.1 Buffer overflow. Buffer overflow experiments were conducted primarily on the Washington University ftp daemon (*wu-ftpd*), a very popular Unix ftp server that is shipped with many Unix distributions. Two vulnerabilities [6] in *wu-ftpd* are exploited to cause buffer-overflow attacks. Table 3 shows the results of detecting the two attacks. We can conclude that two attacks against *wu-ftpd* were successfully detected because $\hat{S}_A$ is at least 0.6. That indicates that the most anomalous sequence differs from all sequences in the signature database in over half of its positions.

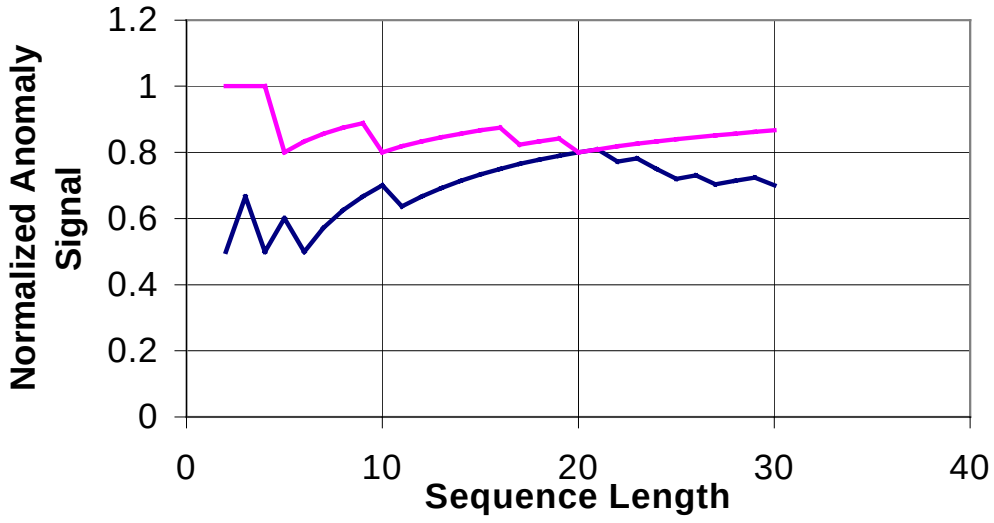

Figure 3. Detecting the same intrusion using different sequence lengths. The x-axis is sequence length and the y-axis is the normalized anomaly signal. The signal value remains above 0.5 for all lengths. Intrusion is detected in all cases.

Ftpd Buffer Overflow Attacks	Number Mismatches	Percent Mismatches	$\hat{S}_A$
Format String Stack Overwrite	467	3.5	0.7
Ftpd Realpath Vulnerability	569	2.7	0.6

Table 3. Successful detection of two buffer overflow attacks against *wu-ftpd*. Each column reports results for an anomaly measure: mismatches, percentage of mismatches, and normalized anomaly signal. Sequence length is 10.

The library call approach works well with buffer overflow attacks when, as is typical, the attacker code adds new sequences of library calls or even entirely new library calls. If attack code does call a library call routine not in application code or introduces new sequences of library calls, then it can be detected clearly. If attack code, however, makes no library calls at all, then the library call approach will not be effective.

5.4.2 Trojan Programs. The Linux root kit includes Trojan attack code for *ps*. It allows intruders to login through a backdoor and hide their activities from system administrators. Table 4 shows the results of detecting Trojan code. When Trojan code runs, new sequences that are not in the *ps* signature database are introduced. As a result, the Trojan code attack is detected.

Attack	Number Mismatches	Percent Mismatches	$\hat{S}_A$
ps Trojan Code	243	3.0	0.8

Table 4. Successful detection of Trojan code for *ps*. Each column reports results for an anomaly measure: mismatches, percentage of mismatches, and normalized anomaly signal. Sequence length is 10.

5.4.3 Denial-of-Service. We simulated a Denial-of-Service (DOS) attack that exhausts all available memory. Our experiment was conducted using the text editor *vi*. As the attack progresses, applications, like *vi*, make library calls/system calls requesting memory. These calls return failure (or anomaly). Applications then invoke exception handlers that introduce new call sequences. They are not likely to be in the signature database.

We conducted two tests to determine how effective our library call approach is for detecting DOS attacks. Table 5 states the results. The first run was normal execution of the *vi* program, i.e., without intrusion. There are no mismatches and $\hat{S}_A$ = 0.0 because all monitored sequences were in the *vi* signature database. The second run of the *vi* program is interrupted by the DOS attack. The value of $\hat{S}_A$ indicates this DOS attack is clearly detected.

The difference in these two runs of *vi* occurs because memory depletion causes invocation of a new library call, *fsync*. Note that *fsync* does not appear in either the database or application code. This new library call is used to synchronize a file's complete in-core state with that on disk. It copies all in-core parts of a file to disk. *fsync* is invoked indirectly by the application to deal with exception when all memory is not available. This experiment illustrates that new library calls may be introduced when intrusions occur.

Program – *vi*	Number Mismatches	Percent Mismatches	$\hat{S}_A$
Normal Run	0	0	0.0
DOS Attack	101	2.6	0.6

Table 5. Successful detection of DOS attack. Each column reports results for an anomaly measure: mismatches, percentage of mismatches, and normalized anomaly signal. Sequence length is 10.

5.5 Library call signature vs. system call signature

Our experiments demonstrated that the library call signature can be used to detect a variety of intrusions successfully. Similarly, New Mexico researchers used system call signatures to detect intrusions. We hypothesize that the library call approach is more effective than the system call approach for selected categories of applications. To test this hypothesis, we developed the following intrusion attack against *mSQL*, a lightweight relational database management system [5].

The attack against *mSQL* is a Trojan code attack that allows an intruder to illegally access the password file, group file, and host file. Table 6 shows the results of attempting to detect this Trojan code attack using both the system call and library call approaches. Two anomaly measures are cited. Recall that the locality frame count reports the largest number of anomalies found within a locality frame. Both measures indicate that the library call approach successfully detects the attack while the system call approach is less effective.

mSQL Trojan Code Attack	$\hat{S}_A$	Max Locality Frame Count
Library Call Approach	0.5	20
System Call Approach	0.4	9

Table 6. Successful detection of *mSQL* Trojan code attack by the library call approach. Each column reports results for an anomaly measure: normalized anomaly signal and maximum locality frame count. Sequence length is 10. Locality frame size is 20.

We speculate that the difference is related to the fact that system call sequences have too little variation. Because *mSQL* is a lightweight relational database management system, most operations for *mSQL* relate to disk I/O access. As a result, almost every combination of I/O system calls is likely recorded in the system call signature database. Our Trojan attack also relates to illegal access to disks. Unfortunately, the monitored sequences of system calls that reflect intrusion behavior also characterize normal behavior. Therefore, they are in the signature database or perhaps do not differ significantly from sequences in the signature database. Thus, the system call signature is unable to detect significant deviations. In this situation, we need to depend on library calls that don't generate system calls to detect anomalistic behavior.

6. Conclusions

We have presented an approach for intrusion detection at the application level based on the application's use of language libraries. We believe that library call signatures are more application-oriented than system call signatures since they permit one to tap the rich application semantics. From an application's view, library call signatures are not OS specific. We use three anomaly measures (mismatch count, locality frame count and normalized anomaly signal) to determine the strength of an anomalous behavior.

We performed experiments with a variety of applications based on C library calls. Our experiments performed using Linux demonstrated that sequences of C library calls can be used to characterize different applications. We build robust signature databases for our selected applications. Our experiments involved attacks such as buffer overflow, Trojan code and denial of service, which are among the most serious security problems on the Internet today. The library call approach appears to be very promising.

Also, we compared our library call approach with the system call approach in a specific situation and found that it performed better.

Acknowledgements

We thank the referees for this conference. They made some sage and very helpful suggestions for improvement.

References

[1] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A Sense of Self for Unix Processes. In *Proceedings of 1996 IEEE Symposium on Computer Security and Privacy*, 1996

[2] S. Forrest, S. Hofmeyr, and A. Somayaji. Computer Immunology. In *Communications of the ACM*, Vol. 40, No. 10, pp. 88-96, 1997

[3] R. Heady, G. Luger, A. Maccabe, and M. Servilla. The Architecture of a Network Level Intrusion Detection System. *Technical Report CS90-20*, Dept. of Computer Science, Univ. of New Mexico, August 1990.

[4] S. A. Hofmeyr, A. Somayaji, and S. Forrest. Intrusion Detection using Sequences of System Calls. In *Journal of Computer Security*, Vol. 6, pp. 151-180, 1998

[5] Hughes Technologies home page, *http://www. hughes.com.au/*

[6] Y. Lin and A. Jones. Application Intrusion Detection using Language Library Call. *Technical Report*, Department of Computer Science, Univ. of Virginia, June 2001

[7] Debian ltrace home page, *http://packages.debian.org/ stable/utils/ltrace.html*

[8] Matthew Stillerman, Carla Marceau and Mareen Stillman. Intrusion Detection for Distributed Applications. In *Communications of ACM*, 1999