A Sense of Self for Unix Processes†

Stephanie Forrest
Steven A. Hofmeyr
Anil Somayaji
Dept. of Computer Science
University of New Mexico
Albuquerque, NM 87131-1386
{forrest,steveah,soma}@cs.unm.edu

Thomas A. Longstaff

CERT Coordination Center
Software Engineering Institute
Carnegie-Mellon University
Pittsburgh, PA 15213
tal@cert.org

Abstract

A method for anomaly detection is introduced in which "normal" is defined by short-range correlations in a process' system calls. Initial experiments suggest that the definition is stable during normal behavior for standard UNIX programs. Further, it is able to detect several common intrusions involving `sendmail` *and* `lpr`*. This work is part of a research program aimed at building computer security systems that incorporate the mechanisms and algorithms used by natural immune systems.*

1 Introduction

We are interested in developing computer security methods that are based on the way natural immune systems distinguish self from other. Such "artificial immune systems" would have richer notions of identity and protection than those afforded by current operating systems, and they could provide a layer of general-purpose protection to augment current computer security systems. An important prerequisite of such a system is an appropriate definition of self, which is the subject of this paper. We view the use of immune system inspired methods in computer security as complementary to more traditional cryptographic and deterministic approaches. By analogy, the specific immune response is a secondary mechanism that sits behind passive barriers (e.g., the skin and mucus membranes) and other innate responses (e.g., generalized inflammatory mechanisms). In related work, we studied a number of immune system models based on these secondary mechanisms [10, 13, 11] which provide the inspiration for the project described here.

The natural immune system has several properties that we believe are important for robust computer security. These include the following: (1) detection is distributed and each copy of the detection system is unique, (2) detection is probabilistic and on-line, and (3) detectors are designed to recognize virtually any foreign particle, not just those that have been previously seen. These properties and their significance are discussed in [11].

Previously, we developed a computer virus detection method based on these principles [11]. The method was implemented at the file-authentication level, and self was defined statically in terms of files containing programs or other protected data. However, if we want to build a general-purpose protective capability we will need a more flexible sense of self. One problem with this is that what we mean by self in a computer system seems at first to be more dynamic than in the case of natural immune systems. For example, computer users routinely load updated software, edit files, run new programs, or change their personal work habits. New users and new machines are routinely added to computer networks. In each of these cases, the normal behavior of the system is changed, sometimes dramatically, and a successful definition of self will need to accommodate these legitimate activities. An additional requirement is to identify self in such a way that the definition is sensitive to dangerous foreign activities. Immunologically, this is known as the ability to distinguish between self and other. Too narrow a definition will result in many false positives, while too broad a definition of self will be tolerant of some unacceptable activities (false negatives).

This paper reports preliminary results aimed at establishing such a definition of self for Unix processes, one in which self is treated synonymously with normal behavior. Our experiments show that short sequences of system calls in running processes generate a stable signature for normal behavior. The signature has low variance over a wide range of normal operating conditions and is specific to each dif-

† In Proceedings of the 1996 IEEE Symposium on Security and Privacy, IEEE Computer Society Press, Los Alamitos, CA, pp. 120–128 (1996).

ferent kind of process, providing clear separation between different kinds of programs. Further, the signature has a high probability of being perturbed when abnormal activities, such as attacks or attack attempts, occur. These results are significant because most prior published work on intrusion detection has relied on either a much more complex definition of normal behavior or on prior knowledge about the specific form of intrusions. We suggest that a simpler approach, such as the one described in this paper, can be effective in providing partial protection from intrusions. One advantage of a simple definition for normal behavior is the potential for implementing an on-line monitoring system that runs in real-time.

2 Related Work

There are two basic approaches to intrusion detection [16, 15]: misuse intrusion detection and anomaly intrusion detection. In misuse intrusion detection, known patterns of intrusion (intrusion signatures) are used to try to identify intrusions when they happen. In anomaly intrusion detection, it is assumed that the nature of the intrusion is unknown, but that the intrusion will result in behavior different from that normally seen in the system. Many detection systems combine both approaches, a good example being IDES [18, 4, 8]. In this paper we are concerned only with anomaly intrusion detection.

Most previous work on anomaly intrusion detection has determined profiles for user behavior. Intrusions are detected when a user behaves out of character. These anomalies are detected by using statistical profiles, as in IDES [18, 4, 8], inductive pattern generation, as in TIM [19], or neural networks [12]. Generation of user profiles by such methods requires an audit trail of actions for each user. These are typically slowly adaptive, changing profiles gradually to accommodate changing user behavior. Abrupt changes in behavior are flagged as irregular and identified with intrusions.

An alternative approach is taken by Fink, Levitt and Ko [9, 14]. Instead of trying to build up normal user profiles, they focus on determining normal behavior for privileged processes, those that run as root. They define normal behavior using a program specification language, in which the allowed operations (system calls and their parameters) of a process are formally specified. Our approach is similar to theirs, in that we consider processes that run as root. However, it differs in that we use a much simpler representation of normal behavior. We rely on examples of normal runs rather than formal specification of a program's expected behavior, and we ignore parameter values. An advantage of our approach is that we do not have to determine a behavioral specification from the program code; we simply accumulate it by tracing normal runs of the program.

3 Defining Self

Program code stored on disk is unlikely to cause damage until it runs. System damage is caused by running programs that execute system calls. Thus, we restrict our attention to system calls in running processes. Further, we consider only privileged processes. Monitoring privileged processes has several advantages over monitoring user profiles[14]. Root processes are more dangerous than user processes because they have access to more parts of the computer system. They have a limited range of behavior, and their behavior is relatively stable over time. Also, root processes, especially those that listen to a particular port, constitute a natural boundary with respect to external probes and intrusions. However, there are some limitations. For example, it will be difficult to detect an intruder masquerading as another user (having previously obtained a legal password).

Every program implicitly specifies a set of system call sequences that it can produce. These sequences are determined by the ordering of system calls in the set of the possible execution paths through the program text. During normal execution, some subset of these sequences will be produced. For any nontrivial program, the theoretical sets of system call sequences will be huge, and it is likely that any given execution of a program will produce a complete sequence of calls that has not been observed. However, the local (short range) ordering of system calls appears to be remarkably consistent, and this suggests a simple definition of self, or normal behavior.

We define normal behavior in terms of short sequences of system calls in a running process, currently sequences of lengths 5, 6, and 11. The overall idea is to build up a separate database of normal behavior for each process of interest. The database will be specific to a particular architecture, software version and configuration, local administrative policies, and usage patterns. Given the large variability in how individual systems are currently configured, patched, and used, we conjecture that these individual databases will provide a unique definition of self for most systems. Once a stable database is constructed for a given process, the database can be used to monitor the process' ongoing behavior. The sequences of system calls form the set of normal patterns for the database, and abnormal sequences indicate anomalies in the running process.

This definition of normal behavior ignores many aspects of process behavior, such as the parameter values passed to system calls, timing information, and instruction sequences between system calls. Certain intrusions might only be detectable by examing other aspects of a process's behavior, and so we might need to consider them later. Our philosophy is to see how far we can go with the simple assumption.

3.1 Details

There are two stages to the proposed algorithm. In the first stage, we scan traces of normal behavior and build up a database of characteristic normal patterns (observed sequences of system calls). Forks are traced individually, and their traces are included as part of normal.[1] In the second stage, we scan new traces that might contain abnormal behavior, looking for patterns not present in the normal database. In our current implementation, analysis of traces is performed off-line.

To build up the database, we slide a window of size $k+1$ across the trace of system calls and record which calls follow which within the sliding window. Suppose we choose $k = 3$ and are given the following sequence of system calls to define normal behavior:

open, read, mmap, mmap, open, getrlimit, mmap, close

As we slide the window across the sequence, we record for each call the call that follows it at position 1, at position 2, and so forth, up to position k. For the first window, from index 1 in the sequence to index 4, the following database is produced:

call	position 1	position 2	position 3
open	read	mmap	mmap
read	mmap	mmap	
mmap	mmap		

Whenever a call occurs more than once, it can be followed by several different possible calls. These are all recorded. After sliding the window across the complete sequence, we produce this expanded database:

call	position 1	position 2	position 3
open	read, getrlimit	mmap	mmap, close
read	mmap	mmap	open
mmap	mmap, open, close	open, getrlimit	getrlimit, mmap
getrlimit	mmap	close	
close			

Once we have the database of normal patterns, we check new traces against it using the same method. We slide a window of size $k + 1$ across the new trace, determining if the sequence of system calls differs from that recorded in the normal database. In our work to date, we simply test for the presence or absence of legal sequences. As an example, suppose that we had constructed the above database and were given a new trace of calls, differing at one location from the normal sequence (open replaces mmap as the fourth call in the sequence):

[1]Due to a limitation of our tracing package, we are not currently following virtual forks.

open, read, mmap, open, open, getrlimit, mmap, close

This trace would generate 4 mismatches, because:

- open is not followed by open at position 3,
- read is not followed by open at position 2,
- open is not followed by open at position 1, and
- open is not followed by getrlimit at position 2.

We record the number of mismatches as a percentage of the total possible number of mismatches. The maximum number of pairwise mismatches for a sequence of length L with a lookahead of k is:

$$k(L-k)+(k-1)+(k-2)+\ldots+1 = k(L-(k+1)/2).$$

In our example trace, $L = 8$, $k = 3$, and we have 4 mismatches. From the above formula, we get a maximum database size of 18, giving a 22% miss rate. Mismatches are the only observable that we use to distinguish normal from abnormal.

This simple algorithm can be efficiently implemented to run in $O(N)$ time, where N is the length of the trace (in terms of system calls). For example, our current implementation analyzes traces at an approximate rate of 1250 system calls per second.

4 Experiments

In the previous section we introduced a definition for normal behavior, based on short sequences of system calls. The usefulness of the definition will largely be determined by the answers to the following questions:

- What size database do we need to capture normal behavior?
- What percentage of possible system call sequences is covered by the database of "normal" system call sequences?
- Does our definition of normal behavior distinguish between different kinds of programs?
- Does our definition of normal detect anomalous behavior?

This section reports our preliminary answers to these questions. In these experiments, we focus on `sendmail` although we report some data for `lpr`. The `sendmail` program is sufficiently varied and complex to provide a good initial test, and there are several documented attacks against `sendmail` that can be used for testing. If we are successful with `sendmail` we conjecture that we will be successful with many other privileged Unix processes. All of our data to date have been generated on Sun SPARCstations running unpatched versions of SunOS 4.1.1 and 4.1.4, using the included `sendmail`. The `strace` package, version 3.0, was used to gather information on system calls.

4.1 Building a normal database

Although the idea of collecting traces of normal behavior sounds simple, there are a number of decisions that must be made regarding how much and what kind of normal behavior is appropriate. Specifically, should we generate an artificial set of test messages that exercises all normal modes of `sendmail` or should we monitor real user mail and hope that it covers the full spectrum of normal (more in the spirit of our approach)? This question is especially relevant for `sendmail` because its behavior is so varied. If we fail to capture all the sources of legal variations, then it will be easier to detect intrusions and be an unfair test because of false positives. We elected to use a suite of 112 artificially constructed messages, which included as many normal variations as possible. These 112 messages produced a a combined trace length of over 1.5 million system calls. For a window size of 6, the 112 messages produced a database with ~1500 entries, where one entry corresponds to a single pair of system calls with a lookahead value (e.g., read is a legal successor to open at position 1).

Once the normal database is defined, the next decision is how to measure new behavior and determine if it is normal or abnormal. The easiest and most natural measure is simply to count the number of mismatches between a new trace and the database. We report these counts both as a raw number and as a percentage of the total number of matches performed in the trace, which reflects the length of the trace. Ideally, we would like these numbers to be zero for new examples of normal behavior, and for them to jump significantly when abnormalities occur. In a real system, a threshold value would need to be determined, below which a behavior is said to be normal, and above which it is deemed anomalous. In this study, we simply report the numbers, because we are not taking any action or making a binary decision based on them. Because our normal database covers most variations in normal, any mismatches are in principle significant.

Returning to our earlier questions, the size of the normal database is of interest for two reasons. First, if the database is small then it defines a compact signature for the running process that would be practical to check in real-time while the process is active. Conversely, if the database is large then our approach will be too expensive to use for on-line monitoring. Second, the size of the normal database gives an estimate of how much variability there is in the normal behavior of `sendmail`. This consideration is crucial because too much variability in normal would preclude

Type of Behavior	# of msgs.
message length	12
number of messages	70
message content	6
subject	2
sender/receiver	4
different mailers	4
forwarding	4
bounced mail	4
queuing	4
vacation	2
total	112

Table 1. Types and number of mail messages used to generate the normal database for sendmail.

detecting anomalies. In the worst case, if all possible sequences of length 6 show up as legal normal behavior, then no anomalies could ever be detected. A related question is how much normal behavior should be sampled to provide good coverage of the set of allowable sequences. We used the following procedure to build the normal database:[2]

1. Enumerate potential sources of variation for normal `sendmail` operation.
2. Generate example mail messages that cause `sendmail` to exhibit these variations.
3. Build a normal data base from the sequences produced by step 2.
4. Continue generating normal mail messages, recording all mismatches and adding them to the normal database as they occur.

We considered variations in message length, number of messages, message content (text, binary, encoded, encrypted), message subject line, sender/receiver and mailers. We also looked at the effects of forwarding, bounced mail and queuing. Lastly, we considered the effects of all these variations in the cases of remote and local delivery. For each test, we generated three databases, one for each different window size (5, 6 and 11). Each database incorporates all of the features described above, producing zero mismatches for mail with any of these features.

Table 1 shows how many messages of each type were used to generate the normal databases. We began with message length and tried 12 different message lengths, ranging from 1 line to 300,000 bytes. From this, we selected the

[2] We followed a similar procedure to generate the normal database for `lpr` and obtained a database of 534 normal patterns.

shortest length that produced the most varied pattern of system calls (50,000 bytes), and then used that as the standard message length for the remaining test messages. Similarly, with the number of messages in a `sendmail` run, we first sent 1 message and traced `sendmail` then we sent 5 messages, tracing `sendmail`, and so forth, up to 20 messages. This was intended to test `sendmail's` response to bursts of messages. We tested message content by sending messages containing ascii text, uuencoded data, gzipped data, and a `pgp` encrypted file. In each case, a number of variations was tested and a single default was selected before moving on to the next stage. These messages constituted our corpus of normal behavior. We reran this set of standard messages on each different OS and `sendmail.cf` variant that we tried, thus generating a normal database that was tailored to the exact operating conditions under which `sendmail` was running. Of the features considered, the following seemed to have little or no effect: the number of messages, message content, subject line, senders/receivers, mailers and queuing. Two more unusual features, forwarded mail and bounced mail, affected remote traces far less than local traces.

Figure 1 shows how new patterns are added to the database over time during a normal `sendmail` run. The data shown are for 10,000 system calls worth of behavior, but we have also performed runs out to 1.5 million system calls (data not shown), with essentially zero mismatches. Overall, the variability in the behavior of `sendmail` at the system call level is much smaller than we expected.

Finally, we ask what percentage of the total possible patterns (for sequences of length 6) is covered by the normal database. For example, if the database is completely full (all possible patterns have been recorded as normal) by 3000 system calls, then it would hardly be surprising that no new patterns are seen over time. However, as we discussed earlier, such variability would be useless for identifying anomalous behavior. Consequently, the goal is to find a database that is small with respect to the space of possible patterns. Our initial data here are encouraging. We estimate that the `sendmail` database described above covers about 5×10^{-5}% of the total possible patterns of system calls (that is, sequences built from all possible system calls, about 180 for Unix, not just those invoked by `sendmail`), an extremely small fraction. This figure is somewhat misleading, however, because it is unlikely that the `sendmail` program is capable of generating many of these sequences. The most accurate comparison would be against a database that contained all the patterns that `sendmail` could possibly produce. This would require a detailed analysis of the `sendmail` source code, an area of future investigation.

	5		6		11	
Process	%	#	%	#	%	#
sendmail	0.0	0	0.0	0	0.0	0
ls	6.9	23	8.9	34	13.9	93
ls -l	30.0	239	32.1	304	38.0	640
ls -a	6.7	23	8.3	34	13.4	93
ps	1.2	35	8.3	282	13.0	804
ps -ux	0.8	45	8.1	564	12.9	1641
finger	4.6	21	4.9	27	5.7	54
ping	13.5	56	14.2	70	15.5	131
ftp	28.8	450	31.5	587	35.1	1182
pine	25.4	1522	27.6	1984	30.0	3931
httpd	4.3	310	4.8	436	4.7	824

Table 2. Distinguishing sendmail from other processes. Each column lists two numbers: the percentage of abnormal sequences (labeled %) and the number of abnormal sequences (labeled #) in one typical trace of each process (when compared against the normal database for sendmail). The columns labeled 5, 6 and 11 refer to the sequence length (window size) used for analysis. The sendmail data show no misses, because sendmail is being compared against its own database.

4.2 Distinguishing Between Processes

To determine how the behavior of `sendmail` compares with that of other processes, we tested several common processes against the normal `sendmail` database with 1500 entries. Table 2 compares normal traces of several common processes with those of `sendmail`. These processes have a significant number of abnormal sequences, approximately, 5–32% for sequences of length 6, because the actions they perform are considerably different from those of `sendmail`. We also tested the normal database for `lpr` and achieved similar results (data not shown). `lpr` shows even more separation than that shown in Figure 2, presumably because it is a smaller program with more limited behavior. These results suggest that the behavior of different processes is easily distinguishable using sequence information alone.

4.3 Anomalous Behavior

We generated traces of three types of behavior that differ from that of normal `sendmail`: traces of successful `sendmail` intrusions, traces of `sendmail` intrusion

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | sales@docketalarm.com | 1-866-77-FASTCASE