Intrusion Detection System at Operating System Level

Po-yuan Peng
Department of Computer Science, University of Auckland
ppen007@ec.auckland.ac.nz

Abstract

This paper examines intrusion detection systems that gather audit trails directly from operating system. It discusses the feasibility and problems such systems may encounter. Although each step itself in an intrusion scenario may not necessarily differ from non-intrusive activities, the objective of intrusion leads to distinctive abnormal change of system state in the underlying environment, or the unusual amount of usage/rate of particular privileged tools, which in term leads to high occurrence of certain sequences of system calls. Since both are observable at system level, tracing of system calls gives an intrusion detection system another discriminating approach to identify anomalies and intrusions. These intrusion detection systems are briefly described in this paper. Furthermore it discusses problems, advantages and limitations of this approach.

1. Introduction

Intrusion detection has always been an active research area in the field of software security. In the three stages of security – prevention, detection and response, it plays an important role since it also has a direct influence to response. As a result, an intrusion detection system has a much broader responsibility. In fact, intrusion detection systems are expected to incorporate the ability to carry out a response to prevent an intrusive scenario from succeeding any further. This type of system often possesses the knowledge of intrusion scenarios, and is categorized as misuse-based detection. Another type of detection focuses on deciding of acceptable user behaviours, which is known as anomaly-based detection.

The idea of intrusion detection exists long ago, and is widely adopted in our everyday life. From rules of a poker game, to the constitution of law, a set of acceptable behaviours are defined which as a honest game player or a lawful citizen, we should abide to. Everybody is under certain degree of surveillance. Whenever someone convicts a cheat in a game, or a crime, law enforcement authority, usually the police, collects evidence. Based on the evidence, a judge refers to the corresponding rules and clause breached and determines the appropriate punishment. Each case is recorded for future reference. Although the world of software security is nowhere like ours, the procedure does not differ by much. Activities are recorded and a detection unit looks into these records for a possible breach of security. A response unit determines the severity of the intrusion and takes the responding action. Just as our law system makes mistake sometime and does not catch all who breaks the law, an intrusion detection system is hardly perfect. False alarms, also referred as false positives can arise frequently and directly challenges the reliability of an intrusion detection system. False negatives can be causing damage to the system while an intrusion detection system is unaware of it.

The recording of activities itself can form a research topic. Considerations include what to be recorded, how much detail should be recorded, how the logged information will be used, defining a universal recording format and programming interface for portability across areas of system. Event Monitoring Enabling Responses to Anomalous Live Disturbances (EMERALD) is an example of monitoring at multiple levels. It collects data from tree different levels: service, domain and enterprise. In [1] the author John McHugh gives this approach a very high hope. Despite its ability to assimilate and correlate trails, and the potential to discover coordinated or distributed intrusions in a large loosely coupled enterprise network, the complexity involved to define and administrate the recorded data is relatively high. Perhaps we can try to look at the problem from a different perspective: instead of detecting intrusions from user behaviours, detect them from system behaviours. In 1995, Stat Transition Analysis Tool (STAT) has been proposed in [2]. In their UNIX prototype of STAT (USTAT), it collects audit trails from UNIX system calls. Later in 1998, Steven A. Hofmeyr el al in [3] proposed a way to apply anomaly-based intrusion detection approach using system calls as audit trails, too. Both have experimented the feasibility with their prototype system, and suggest the possibility of positioning an intrusion detection system at the operating system level.

In this paper, section 2 briefly talks about the general intrusion detection approaches used in various implementations. Section 3 describes into details how these approaches can be achieved at operating system level. The difficulties and problems are addressed in section 4. Section 5 summaries the paper and gives how this approach can help in intrusion detection.

2. Background

In order to surveillance user activities, their resultant events are necessarily recorded by applications such as apache web server, ftp daemon, secured shell login server… etc, and parts of the operating system for events like “file not found”, “insufficient permission”, “I/O error” and many other low level error messages. Most host-based intrusion detection systems obtain audit trails from a logging facility, like the syslogd which collects data from various sources including the ones mentioned above. However many question about logging remains unanswered. For example what kind of events should be recorded? How much detail will be enough? These two questions will determine the quantity and quality of the logged data, therefore have a direct influence to the discriminability of an intrusion detection system. How the logged information will be used suggests a better and more efficient format of logged data and logging interface. With a standard format it is possible to optimise for better efficiency and portability across different areas in the system. A standard interface encourages programmers to utilize the logging facility and hence enhances the sensitivity of the intrusion detection system. Although it may be easy to unify the interface to the logging facility, standardizing the format is non-trivial due to the wide varieties of purposes from applications and portions of system it covers.

Many intrusion detection systems have already existed for years. Based on the way they detect intrusions, we can generalize them into two main abstract categories: anomaly-based detection and misuse-based detection. From there we can further divide, depending on their distinctive approaches to the problem in practice. However none of the approaches alone is sufficient to detect all intrusions. Thus a typical intrusion detection system employs several different approaches to overcome the particular weakness in each one.

2.1 Anomaly-based detection

Based on the assumption that an intrusion must differ from normal activities, anomaly-based detection observes the difference between a given activity and a set of normal activities. Such detection tries to characterize an activity as normal through a set of profiles, which are characterizations of acceptable behaviours. The idea came with long history and was well adopted in the earlier intrusion detection systems.

Depending on the definition of acceptable behaviours, there are several distinctive detections. These include using summary statistics and rules about behaviours. Summary statistics defines a threshold and counts the occurrence of events. If an event observed has an abnormal rate of occurrence, it is believed to be abnormal and is causing concerns. Defining rules about acceptable behaviours is not only troublesome but difficult to conduct as well. It will require training data, which leads to another area of research. Such difficulties include comprehensiveness and correctness of data. Training data has to be comprehensive in order to cover all the normal behaviours. Failing to do so will cause some activities being identified as an intrusion, resulting in false alarms or false positives. The correctness of training data is essential because it is the only reference to determine an intrusion. If the correctness of training data is not guaranteed, then an intrusive activity may be classified as normal, creating false negatives.

Another assumption people usually make with anomaly-based detection is that the more different something is away from normal, the more likely an intrusion it is. However, except the case of summary statistics, it is often difficult to design a reasonable measure to determine the magnitude of anomaly and prove the reliability of the measure, thus difficult to implement in practice.

The advantage of anomaly-based detection lies on the ability to point out novel intrusions. However the necessity of training data pulls the performance of anomaly-based detection away from ideal. Anomaly-based detection usually has little information about an intrusion and can hardly counter-act as a response upon discovery of any intrusion.

Multics Intrusion Detection and Alerting System (MIDAS) and Network Anomaly Detection and Intrusion Report (NADIR) employ summary statistics in their anomaly-based detection. Intrusion Detection Expert System (IDES) and Wisdom and Sense (W&S) use rule-based anomaly detection approach as part of their detection.

2.2 Misuse-based detection

Instead of defining what normal is, another natural approach is to have an expert system and define what intrusions are. Misuse-based detection typically consists of a collection of patterns from all known intrusive activities. The exact method ranges from simple pattern matching to higher level of intrusion scenario models.

Pattern matching is commonly used in network-based detection. It identifies an intrusion by looking for particular signatures in audit trails. However derivative of intrusions can easily bypass this kind of detection with a slight modification. To address this type of issues, some intrusion detection systems introduce an abstract model to represent an intrusion scenario. This kind of intrusion detection is also known model-based detection. Since a model-based detection describes intrusion scenarios, it is considered under the generality of misuse-based detection approach. An intrusion scenario may consist of many steps. If the intrusion detection system is well aware of the steps required towards the success of an intrusion, it can prevent an intrusion from succeeding the next step upon detection and confirmation.

Misuse-based detection provides a strong indication of any possible intrusion. Another reason why many intrusion detection systems incorporate this as part of their detection scheme is the detailed information if has about the detected intrusion. Using misuse-based detection it is easy to provide a scenario with name and description, aiding the security officer or even the intrusion detection system itself to take the appropriate action against the attack. As the concept of misuse-based detection suggests, it can only reliably detect known intrusions. Although derivatives may be picked up by proper definition of intrusion, the general principle does not apply to novel intrusions.

NADIR, IDES and W&S all use misuse-based detection as part of their detection scheme. In fact these intrusion detection systems use a combination of anomaly-based and misuse-based approaches to complement the weakness in one another.

3. Approaches using system calls

A computer program is a set of instructions describing how a task is performed. Unlike humans, these programs are expected to be stable. Being stable we mean given the any input, the produced result will always be the same, due to the identical execution path in the process. In contrast, humans tend to do the same thing differently from time to time, which makes defining normal behaviours relatively difficult. Based on this idea, if an intrusion detection system examines the behaviour of a process (which is a running copy of a program), it can detect intrusions with better precision and distinguish between normal and anomaly with reduced fuzziness.

The thought of using system calls as audit records has been there for long. At the advantageous side it addresses most of problems of defining a good logging facility. For example, the linux kernel of version 2.4.20 uses 252 system calls. Although the exact number varies from one operating system to another, the number of system calls is finite. In fact, we will only be interested in a subset of system calls, which have an direct influence to security, such as create, read, write, execute, change permission, change ownership… etc. Thus it simplifies the choice of events to record. Detail level is limited to the parameters passed into the call. The format is consistent. In fact, all the system calls are accessed with the same interface deep inside the kernel.

In the modern design of operating system, the kernel is usually modular. The trend has evolved from earliest monopoly, where everything is compiled as a whole, to modular and micro kernel. The latter produces a much smaller consisting of essential functionalities and benefits from a reduced loading time less required storage, and dynamic extensibility of functionality. A module can be attached and detached from the kernel and thus can be later loaded from a separate storage. However it is the extensibility that benefits this approach of intrusion detection most. An auditing module can be written as an extension to the kernel and provide the intrusion detection system the data it needs. Since this module is part of the kernel, it can access essential information in the operating system easily, allowing an intrusion detection system to capture events in the system resulted from any process.

Once an intrusion detection system has identified an intrusion, it may decide to counter-act against that particular intrusion scenario. At this moment it may suggest that the next essential system call in the intrusion scenario shall fail. In this case the module will have to block the system calls and deny calls from particular process. Undoubtedly in the benefit of added functionality, this pays a price of increased complexity.

Not all detections mentioned in the previous section work well with this approach. Summary statistics anomaly-based detection for example, may fail miserably if one simply records the occurrence of a particular system call without considering the context. Recording the occurrence of sequences of system call leads to a large number of possible combinations, such as 40^{10} with sequences of length 10. To monitor different processes one must have multiple copies of these counters. Another problem arises when trying to define a threshold for a particular (sequence of) system call. How many times read() are called in one minute can be considered as abnormal? It could well be a database server trying to answer hundreds of queries simultaneously during the peak hour. There is little to claim when looking at the frequency that a particular system call occurs.

3.1 Profile-based Anomaly Detection

In 1998 an anomaly-based detection approach using sequences of system calls has been proposed in [3]. They define normal as short sequences of system calls. The parameters passed to system calls are omitted for simplicity. The detection consists of two main stages: profiling normal behaviours and detecting anomalies. In the profiling stage, training data is collected. This data is then used in the detecting stage.

To evaluate the feasibility, it tests the approach with both synthetic data and real data. The experiment with real data focuses with lpr, a popular printing service in UNIX and is conducted at both Massachusetts Institute of Technology's Artificial Intelligence laboratory and University of New Mexico's Computer Science Department. Hence there are different environments to be able to compare against. The expected rate of false alarms is very impressive, about 1%, calculated using the bootstrap technique. Although the experiment is not very comprehensive, the extremely low false alarm rate suggests this approach to be very accurate in detecting anomalies.

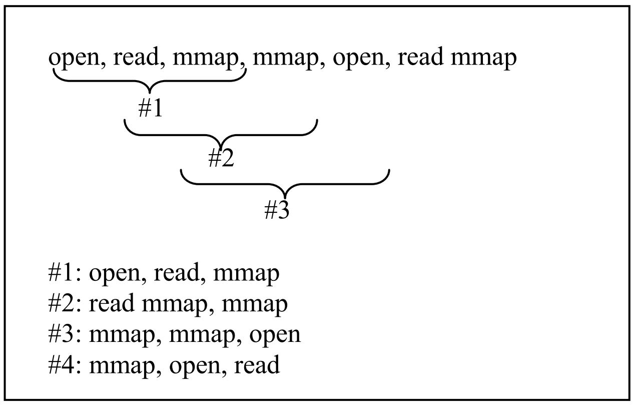
The following sub-sections describe the main ideas of the procedures. For more detailed information especially on the setting and condition of their experiments, readers are suggested to read [3].

3.1.1 Profiling normal behaviours

Since this approach uses short sequences of system calls to define normal behaviours, the purpose of establishing profiles is simply to gather all possible sequences of system calls that a process can generate. However in [3] they make a important claim:

> "We make a clear distinction here between *normal* and *legal* behavior … because our approach is based upon the assumption that normal behavior forms only a subset of the possible legal execution paths through a program, and unusual behavior that deviates from these normal paths signifies an intrusion or some other undesirable condition. We want to be able to detect not only intrusions, but also unusual conditions that are indicative of system problems."

It is natural to expect a small number of distinctive execution paths from a process in its normal operating conditions. These execution paths result in constant sequences of system calls, which can be used to determine normal. In here, they scan traces of system calls and obtain sequences from several executions. Each sequence is broken into several smaller sub-sequences with constant length. Figure 1 illustrates an example of sequences of length 3.



#1: open, read, mmap
#2: read mmap, mmap
#3: mmap, mmap, open
#4: mmap, open, read

Figure 1: sequences of 3 system calls

The idea is simply that in order to allow this long sequence, each sub-sequence must also be allowed. Notice the last sub-sequence is identical as sub-sequence #1 and hence not added. It is meaningless to allow duplications of the same sub-sequence in the database. These sub-sequences form the database of normal. They are used as the reference to determine unusual behaviours. In section 3.1, we refer k as the length of each sub-sequence. Although there is not much restriction to the choice of k, it should be neither too small nor too large. If k is too small, the sequence will fail to capture distinctive features of a process, causing a majority of activities to be covered, hence false negatives. If k is too large, each sequence becomes too specific such that an execution path may have different sequences under slightly different condition. It becomes difficult to comprehend the domain of normal, and introduces false positives or false alarms. The size of database is O(nk), where n is the number of distinctive sub-sequences. In Figure 1, n=4 and k=3. Because each system call will be looked up in the database, it is desirable to arrange in a tree and use as a dictionary. This will not only save the storage, but speed up the look up process as well.

After the structure of the profile is defined, here comes the inevitable – collection of training data. In [3] there is an in-depth discussion on the synthetic data. Since earlier in this section we have claimed that the purpose is to pick up abnormal rather than illegal behaviours, synthetic data only becomes interesting when experimenting with the feasibility and performance. Strictly speaking, real data is easy to obtain. Simply turn on the monitor and at the end of day you have a truckload of data. However it is the quality of data that is difficult to maintain. Imagine that during the training process, someone deliberately feeds the system with intrusions. Because it is only collecting data, the intrusion detection system is not operational at this stage to pick up these intrusions. Once the training is done, such intrusions will be matched as normal and accepted happily by the intrusion detection system, resulting in false negatives. The comprehensiveness of training data is another important consideration. If the training fails to include a scenario that is occasional but normal, such scenario will give rise of false alarms in the future. [3] suggests two possible solutions to obtain a high quality training data:

> "Collect normal in the real, open environment, whilst monitoring the environment very carefully to ensure that no intrusions have happened during our collection of normal."
>
> "Collect normal in an isolated environment where we are sure no intrusions can happen."

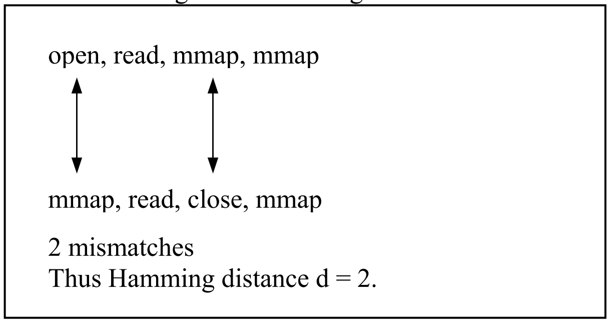
The problem with the second approach is obvious. An isolated environment may not represent the real environment faithfully. Although it is guaranteed to be intrusion free, the lack of completeness gives a higher false alarm rate. The first approach requires either experts to examine the log or a security officer to surveillance user activities during the training period. Still, it is both time consuming and subject to human-errors. In [3] they use this approach of monitoring user activities to gather the necessary training data for testing and evaluation.

3.1.2 Detecting anomalies

With a database of normal ready, an intrusion detection system can then compare a sequence of system calls captured against these profiles. Of course, the sequence captured will be broken into parts of length k before comparison. If a match is found, then this process is considered to be normal so far. When there is no match to the captured sequence, we would like to establish some method of telling the level of suspicion.

Defining a proper measure of difference is non-trivial. This indication should reflect to the logical difference from the most similar sequence and suggest the possibility of being an intrusion. The Hamming distance is chosen in [3] to be the measurement of difference between two sequences. Simply put, it is the number of mismatched locations. Since the sequence is of length k, this measurement produces a number between 0 and k inclusively. It is difficult to justify whether this is a proper measurement. In fact, there is no proof of the righteousness of this measurement. However it is meaningful to some extend and allows us to peek the difference in a measurable way. Until we find another finer measurement, the Hamming distance is used for now. Figure 2 shows a simple Hamming distance measurement.

Figure 2: Hamming distance



2 mismatches
Thus Hamming distance d = 2.

Computing whether or not a sequence is normal is relatively cheap. If a tree structure is used in the database, the worse case of finding a match (or mismatch) is number of system calls $N \times k$ comparisons. However since we describe the magnitude of anomaly by the difference from the most similar normal behaviour, for each captured sub-sequence we need to break into sub-sequences of length k, compare against each sequence in the database and obtain the smallest difference measured by the Hamming distance. This is equivalent as determining the minimum value across comparison against each normal sequence. The strongest indication in these sub-sequences represents the degree of anomaly. In [3], this value is referred as the signal of anomaly, S_A.

Let X be the set of sub-sequences of system calls captured by the system in a particular process.
Let Y be the set of sub-sequences of system calls defined in the database.
Let d(x,y) denote the Hamming distance between two sequences x and y, then:

$$S_A = \forall x \in X: \max(\forall y \in Y: \min(d(x,y)))$$

Up to this moment, all the measurements are dependent to k, the length of each sequence in the database. Undoubtedly the number of mismatches will increase, as k becomes larger. This restricts us to make comparisons within the same value of k. In order to have a measurement independent to the length of sequence k, the mismatch percentage is introduced. Let's call it S, which is simply S_A / k.

$$S = S_A / k$$

Now we have the signal of anomaly. We can define a threshold to discriminate anomalies from normal. Let's use C to denote the threshold in detecting anomaly, where $1 \leq C \leq k$. Notice we define C as an integer since it is mainly used in the detection, where the database consists of sequences of fixed length k. Therefore we can check S_A against C in a faster integer operations. We could use C as a mismatch percentage, but it will require a division to be performed for every $S = S_A / k$. By definition, a process is considered to be abnormal when $S_A \geq C$. C = 0 is meaningless, and hence C is bounded by 1 and k. Therefore we have the following two cases.

$$\text{Normal:} \quad 0 \leq S_A < C \leq k$$
$$\text{Abnormal:} \quad 1 \leq C \leq S_A \leq k$$

Is this discrimination powerful enough? Perhaps not for human behaviours which migrate and deviate all the time. But for a hard-coded program its behaviour is limited in general and can hardly change. The environment and runtime conditions may steer the exact execution path in a process, however these execution paths are usually finite and we are capable of enumerating each one of them under common conditions. On the bright side, this approach is able to pinpoint the source process of an intrusion accurately. One very effective use of this approach is the detection against buffer-overflow attacks. This type of intrusions introduces new code and hence new behaviours to be picked up. Unfortunately as all anomaly-based detections suffer from, this approach can hardly tell the type of intrusion upon any successful detection. The introduction of system calls complicates the reasoning of an intrusion. Imagine a security officer facing a bunch of system calls such as open, read, read, mmap. Security officers may have the source of intrusion, however they have to investigate before knowing what is going on. For example, a buffer-overflow attack may have taken place in an http server. The security officers are informed but have no idea whether this intrusion is a buffer-overflow attack or not. They have to investigate this http server before they can conclude that it is a buffer-overflow attack on the http server.

3.2 State Transition Analysis Tool

State Transition Analysis Tool uses a misuse-based detection approach to model an intrusion scenario. It breaks down an intrusion scenario into many states. The initial state is the system state prior to an intrusion and the compromise state is the state after a successful intrusion has taken place. Between two states there is an action which causes the change of system state from one to another. There must exist a sequence of states, from the initial to the compromised, where the system meets the initial state and the action in between is each successful in order for the intrusion scenario to succeed. Together these form a State Transition Diagram that is used to describe an intrusion scenario. In the paper [2] they use the term penetration as a successful intrusion scenario.

Based on the use of states to describe a scenario, STAT makes the assumption of following two features in all penetrations, quoted directly from [2].

"Penetrations require the attacker to posses some minimum prerequisite access to the target system."

"All penetrations lead to the acquisition of some previously unheld ability."

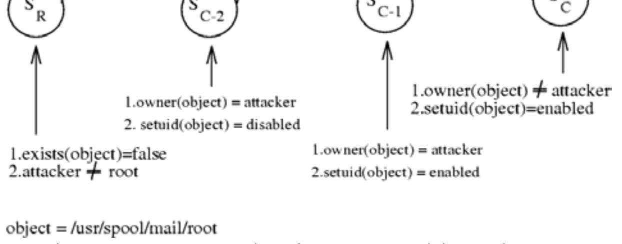
The first feature supports the idea of identifying the requirement, which is the initial state. The second feature suggests that there is a difference between the initial state and the compromised state, and hence detectable. After seeing a scenario as a whole, STAT begins to break it into critical steps. Essential states throughout the scenario are identified and the action responsible for state change is determined.

One of the main advantages of STAT is the use of State Transition Diagram to model an intrusion scenario. This presents the scenario in a visualizing manner, hence provides better understanding and analysis. Such a model makes STAT a misuse-based detection that deviates from traditional one-to-one direct and static relationship between rules and audit trails. The model may be static but the actual subjects, for example the particular file under attack, can be determined dynamically and applied to the general model. One model can thus be used to monitor multiple instances of an intrusion simultaneously, e.g. multiple attacks on several "SUIDROOT" files based on using the same trick of buffer-overflow. The updating of database in a rule-based detection often requires experienced technicians to plug in the new formula. Whereas with STAT, local security officers can study, draw the diagram and analyse new intrusion scenarios. Once the description boils down to a set of requirements and effects, essential actions can be determined and this information can be taken into STAT. Thus a methodology is available for creating new rules in STAT.

3.2.1 State Transition Diagrams

STAT uses State Transition Diagrams to describe an intrusion scenario in an abstract but precise and visualizing manner. These diagrams reflect to the essential concepts of STAT. Its visual representation provides better understanding, clear requirement and consequence of an intrusion scenario. Based on this information, STAT has the ability to prevent an attacker advancing towards a successful penetration.

A State Transition Diagram usually begins with only two or a few more states. The initial state corresponds to the requirements of the intrusion scenario. The compromised state represents the effect on the system once the intrusion has succeeded. Each state is considered as a node in a directed graph. An action is responsible for transition between states, i.e. a change of state. In a directed graph, actions are drawn as edges linking states. Refined with details, this directed graph becomes a State Transition Diagram. On page 6 of [2] there is an example of an attack taking advantage of the fact that this earlier *mail* program changes the ownership of the mailbox file (/usr/spool/mail/root in this example) without checking whether the file has set user id permission enabled. If an attacker can write in this directory (/usr/spool/mail), he/she can fabricate /usr/spool/mail/root by copying a shell program. When an e-mail is sent to the user root, mail realizes a mailbox already exists but with a wrong owner. It changes the owner to the recipient which is root and appends the new mail. The attacker will then have a shell that has root privilege upon execution. Figure 3 from [2] clearly illustrates this example in a State Transition Diagram.


attacker creates(object)
attacker mod_setuid(object)
attacker mod_ouid(object)
S_R
S_{C-2}
S_{C-1}
S_C
1.exists(object)=false
2.attacker ≠ root
1.owner(object) = attacker
2. setuid(object) = disabled
1.owner(object) = attacker
2.setuid(object) = enabled
1.owner(object) ≠ attacker
2.setuid(object)=enabled
object = /usr/spool/mail/root
Figure 3: An example of State Transition Diagram.

For an intrusion to succeed, the initial state (i.e. the requirements) must be met. Then, there must be a path from the initial state to the compromised state, where each edge corresponds to a feasible action. The length of the path gives some indication about the complexity of the intrusion. The more complex an intrusion is, the more chances it can fail. A short path would suggest an immediate threat since we have a very limited possibility of stopping it. In the previous example there are only 4 states and 3 actions. If the initial state is met, an intrusion can advance to the next state SC-2 leaving us only two more actions where we can prevent this intrusion from being successful. Sometimes a scenario may consist of several alternatives, resulting in multiple paths from the initial state to the compromised state. For example there may be two system states in the intrusion scenario A and B, where the order is irrelevant. As shown in Figure 4, we can make each branch a separate scenario. Thus we can now safely consider diagrams with only a single path.

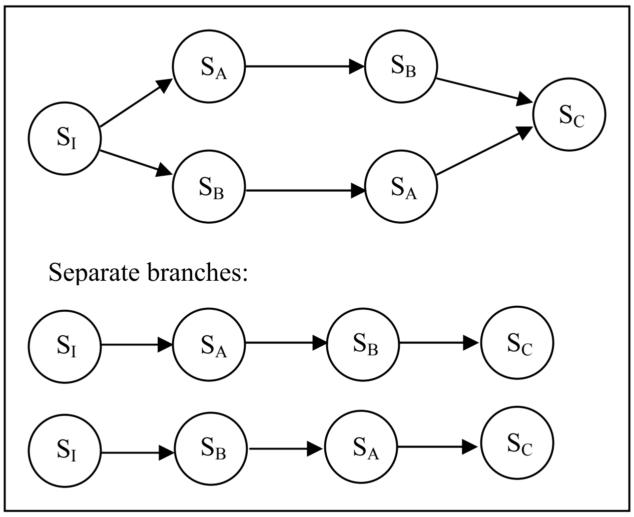


Figure 4: Separated paths for branches.

In order to optimise the use of STAT to catch mutants of a known intrusion, the State Transition Diagram should only describe the essential states. In this example how the attack creates the e-mail can vary. The process of creation of e-mail does not cause any threat to the system. Thus a proper diagram should not introduce additional states to model this process.

4. Discussion

Working with audit trails directly from system level has its pros and cons. We first look at the possible problems and then discuss about the benefits and limitation.

4.1 Problems with System Calls

With this approach we have to be very careful since it introduces many operating system or architecture specific problems, such as reduced system performance, compatibility, monitoring of states and race conditions. Each is discussed in more detail in the following.

4.1.1 Reduced System Performance

One obvious problem is the impact on system performance. Almost all processes make system calls in their lifecycle. Common program behaviours like reading a file or creating a process involve open, read, fork and other system calls. The module that collects trails introduces additional overheads in the handling of system calls, causing system calls to return slower and thus a reduced system performance.

For a better security it may be desirable to block the process, let the intrusion detection system inspect the current system call and decide whether it should be allowed to execute. The "Janus" system in Tal Garfinkel's paper [4] is an example of such. If we intend to block these system calls, then a process that uses system calls may have to halt before a decision from intrusion detection system is made. This phenomenon can quickly become very serious when frequent system calls are made, such as a database manager with frequent reading and seeking in large files.

As a result, the impact on system performance may indicate an exclusion of I/O intensive applications such as SQL database and http server from this approach.

4.1.2 Compatibility

Here the compatibility means whether an existing intrusion detection system will still work under different conditions. There are two aspects I focus on: across different operating systems, and across different versions.

We have seen so far many common system calls typically used in a UNIX operating system. Many of the UNIX clones share similar properties and interface, thus makes the porting of an implementation from one operating system to another a lot easier. However the kernel module involved severely reduces the compatibility of this approach. For each different kernel architecture, a specific module has to be written in order to collect the trails. Some operating system has quite a significant difference between major versions, such as Linux. Although this dependency on kernel architecture results a poor compatibility, the variety of server platforms is much less. It is possible to select few popular solutions to build the implementation based on.

Once a while software vendors release an update or a patch to fix known bugs in their product. Vendors will strongly encourage their users to update if there is a security fix against any flaw. Once a program is updated, new behaviours may be introduced and exhibited at runtime. At this point it may require training data to be collected again, since the old profile can no longer be used to describe the new behaviours. Or even worse, the difference may be initially unnoticeable when someone has experimented a little with the old profile. Based on that observation, system administrators and security officers may decide to not to go through the collection of training data again and reuse the old profile. When false positives occur due to the introduction of new code, they end up spending more frustrating time analysing. The collection of training data is always time-consuming and painful but unfortunately a necessary procedure. In order for this approach to be practical in general, the inevitable collection of training data must be able to be done in a cheaper, more efficient manner.

4.1.2 Monitoring of State

A very famous phrase in UNIX is "there is more than one way to do things." Indeed in UNIX, many distinct sequences of actions can be used to achieve the same goal. This holds particularly true for intrusion scenarios.

Recall that this approach mainly uses sequences of system calls in order to detect an intrusion. The sequences of system calls are generated as the attacker performs sequences of actions. Although we are seemingly capturing these actions, the ultimate goal is to reconstruct the system state, so that we know how far an attacker has progressed in the intrusion scenario. Actions provide a clue, not the solution. It is important to capture all actions that can influence the system state. Therefore the coverage of system calls to capture should be carefully considered.

A very clear example is given in [4]. In most operating system, files are handled through a table of descriptions. When a file is opened, its details such as path, size, permission and ownership are recorded in an entry of the table. The entry number is returned as a *file descriptor* and used as the key for later access by programs. dup2(a,b) is a function that duplicates entry a to entry b. If the intrusion detection system fails to capture the effects on this function, it is not monitoring the system state correctly. The example of intrusion in the *mail* program is modelled efficiently in STAT[2]. Suppose that it includes a verbose check of writing to file /usr/spool/mail/root. An attacker can go around that step of detection if dup2() is not captured.

5 = open("/usr/spool/mail/root", …)	The attacker opens the sensitive file, captured and monitored by IDS.
6 = open("/tmp/legalfile",..)	The attacker opens another file captured but not monitored by IDS.
dup2(5, 6)	IDS has no idea of the duplication of file descriptor so that both 5 and 6 are now for /usr/spool/mail/root.
write(6, ….)	IDS believes the attacker is writing to the file /tmp/legalfile, while /usr/spool/mail/root is actually written.

As a result, STAT will believe that the file /usr/spool/mail/root is never written and thus fail to detect this attack.

Similar problems arise when handling paths and links carelessly. In many operating systems, both absolute and relative paths are commonly used. It is crucial to determine the correct entity in order to conduct a correct detection. Links need to be taken care of too, since it provides an alternative mean to access files indirectly.

4.1.3 Race Condition

There are situations where the intrusion detection system needs to perform a counteraction based on a detection of vulnerability. Since the main body of an intrusion detection system resides in user space, a context-switch may occur and the execution is passed on to another process. At this moment, the executed process may be able to take advantage of the vulnerability while the intrusion detection system is in dormancy. For example, considering the apache web server. It allows administrator to specify a plain text file containing the login and password for accessing a specific directory. Suppose it is configured that if the login/password file "*passwd* "exists in a directory, the server allows only those with login name and password in this file to enter. If the administrator carelessly produces this file without setting the permission first, it may be set to default which is readable by everyone. Now the intrusion detection system detects an intrusion of "passwd" being world readable. It immediately fires a counteraction to correct the permission. However a context-switch may occur in this gap and cause this counteraction to be suspended. The file remains world readable, and the process that just gets executed can read everyone's login and password.

Race condition gets worse when multiple threads are involved. Since a thread is a lightweight copy of process, a lot of properties are shared between threads of the same group. For example two threads of the same program share the same working directory, thus if thread A changes the current directory to /tmp, thread B will be working relative to /tmp as well. This introduces another example of race condition as given in section 4.3.4 in [4]. However I question the validity of this example since like all intrusion detection system should be, Janus is supposed to add another layer of security, not replacing existing mechanism. The file /etc/shadow contains all encrypted passwords and is usually readable and writable only to root in most UNIX systems. Therefore /etc/shadow should remain unreadable since the call to open() will cause a security violation in the operating system. The approval of Janus should not override the underlying security mechanism in the operating system.

4.2 Strength

Operating intrusion detection system at system level has several benefits. First it is more reliable, since instead of observing human behaviours, process behaviours are being monitored. Second it is more applicable to low-end computer systems. STAT relies heavily on system calls to indicate a change of state and has many advantages compared to other misuse-based detection approaches.

Taking advantage of the stability in program behaviours, the anomaly-based intrusion detection shows a relatively low rate of false alarms and thus more reliable. It also enables us to define a rough measure to determine the extend of anomaly. The result in the experiment of [3] is very successful. The false positive they observed is estimated one out of a hundred. However this may still be insufficient to prove the effectiveness of this approach, especially in situation where the system is under extreme stress, causing exceptional conditions and behaviours.

This approach collects audit trails from operating system. It applies to all processes without having each process being specifically designed to give the required information. Therefore it is more practical for a host-based solution of intrusion detection system. This is favours many of low-end computer systems because buying software designed to feature a particular logging interface is then unnecessary. Imagine the need of spending $10,000 more so that your SQL database supports the logging facility of EMERALD.

STAT benefits a lot from using system calls. Not only it can detect multiple instances of the same intrusion scenario, using system calls enables it to detect intrusions through collaboration and multiple sessions. Collaboration can be picked up, since STAT concerns only about the system state. Because each process has the ability to alter the system state in some way, STAT does not distinguish between processes in particular. Thus its intrusion scenario is applicable to detect intrusions across multiple processes. It is applicable to detect across multiple sessions, too. Since the system state remains after one session ends, coordinating an attack across several sessions makes no difference in detecting. More ever, STAT contains the sequence of actions about an intrusion scenario. It can issue a response to an ongoing intrusion and effectively prevent the system from being compromised.

4.3 Limitation

While this approach opens its favourite area in intrusion detection, it is also limited by the nature of its audit trails – system calls. The limitation is mainly due to the lack of high-level information.

First it gives very little information about the attacker. For example if an anomaly-based detection found an attempt of buffer-overflow attack, it still has no idea where the source of the attack is based on the audit trails it collects. The information is simply not at the same level.

Many attacks visible in human behaviours cannot be picked up. For example it cannot detect a masquerader, who pretends to be another person. Nor can it detect the leakage by legitimate user, described in [1].

5. Conclusion

Intrusion detection system at operating system level is definite a feasible solution. As we have already seen, it has the benefit of being very reliable, effective and efficient in detecting its domain of intrusions. Intrusion detection systems such as STAT are developed based on this approach and exhibits several significant advantages. By adopting this approach the anomaly-based rule-based detection in [3] also shows an impressive low rate of false positives. The significance of this approach is unquestionable.

"No intrusion detection approach stands alone as a catch-all for computer penetrations."

This phrase is quoted from [2] and expresses the heart of this paper brilliantly. Once outside the domain of this approach, it is hardly useful at all. As discussed in section 4.3, it cannot detect possible intrusions in many human behaviours. Even if an intrusion is detected, there is little high-level information that can help us to understand about the situation.

Nevertheless each approach has its advantages and disadvantages. More importantly there are situations where one is more preferable than another. It is therefore approaches from different angles need to be studied in order to complement the weakness found in others. As John McHugh suggests in his writing [1], intrusion detection systems should collect audit trials from different hierarchical levels, providing ability to “collect, assimilate, correlate, and analyze information emanating from diverse sources in real-time” in order to cope with more sophisticated attacks.

Bibliographic

[1] John McHugh, “Intrusion and Intrusion Detection,” *International Journal of Information Security 1,* 2001, pp. 14-35.

[2] K. Ilgun, R.A. Kemmerer, and P.A. Porras, "State Transition Analysis: A Rule-Based Intrusion Detection Approach," *IEEE Transaction on Software Engineering*, 21(3), March 1995.

[3] Steven .A. Hofmeyr, S. Forrest, and A. Somayaji, “Intrusion Detection using Sequences of System Calls," *Journal of Computer Security*, August 1998

[4] Tal Garfinkel, “Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools,” In *proceedings of the ISOC Symposium on Networks and Distributed System Security*, Feb 2003.