SADDLE : An Adaptive Auditing Architecture

by Ashish Gehani and Gershon Kedem
Department of Computer Science, Duke University

Abstract

Intrusion detection uses anomalous activity in a system to recognize an attack. Toward this end, events are audited and resources are monitored, with the logs thus generated serving as the input to an analysis engine. Increasing the fidelity of recognition implicitly requires refining the granularity of logging. Current systems can achieve this at the cost of increased computational load and storage space. We investigate an architecture that alleviates the apparent tension between the aforementioned goal and its associated cost.

Introduction

A prototypical intrusion detection system consists of several components. Sensors record events in the system or traffic on a network link, committing the data to log files. This is fed into an analysis engine, which uses a database to decide when an intrusion has occurred. The database can consist of a set of bounds on statistical measures. If current activity strays beyond the bounds, the analyzer concludes that anomalous behaviour is occurring. Alternatively, the database can contain rules that each describe a set of events that leads to an access control breach. If current activity matches a rule, the engine flags an intrusion. An online analyzer processes log entries at the rate they are generated, while an offline one does the analysis later.

[Denning87] framed the model used by current intrusion detectors. Numerous schemes have been tried for intrusion recognition [Axelsson00]. However, intrusion detectors are still not ubiquitously deployed. False positive rates, or how often an alarm is raised in the absence of a breach in security, are measured relative to the maximum number of possible false positives (which is proportional to the number of audit records). Intuitively, the utility of a detector diminishes if there is an increase in the proportion of the alarms that are raised but turn out to be false. [Axelsson99] uses the base-rate fallacy of Bayesian statistical estimation theory to argue that the criterion of interest should be the false positive rate relative to the number of expected intrusions. Current intrusion detection systems fare poorly using this metric. Thus there still exists a need to decrease false positive rates.

Refining Detection Signatures

There exists a general method to reduce the frequency with which normal system activity is incorrectly flagged as intrusive. This can be achieved by increasing the specificity of the signatures used for detection. For example, for anomaly detection using statistical signatures, sampling can occur at a greater frequency, derivatives of the frequency can be utilized, or cross-products of different metrics can be used. For rule-based signatures, in addition to the class of event, parameters can be matched, temporal constraints can be imposed, or filters based on sequence history can be used. A combinatorial analysis verifies that the expected value of the false positive rate can be drastically reduced by such an approach.

The disadvantage of using more detailed signatures is twofold. Systems based on detecting anomalous behaviour may experience an increase in false negatives. Since a signature that is less specific may match more subsequences of the audit record, the chances of it matching a real intrusion are higher. The increase in specificity makes the signature inherently more brittle in the face of variation in the attacker's statistical profile. On balance, the reduced false positive rate is likely to outweigh the increased false negative rate. Rule-based detectors, however, are not adversely affected. A more specific rule is constructed using *a priori* knowledge of a particular attack. Therefore, a real attack will continue to be detected, while a previous false negative may no longer match. Thus, the rate of false negatives is monotonically non-increasing with a rule's specificity.

The second concern is the increase in storage and processing overhead that is imposed on the auditing subsystem. To refine a signature used to detect anomalous behaviour, more detailed audit data is needed. For example, improving temporal fidelity involves increasing the frequency of sampling the variables of interest, such as the CPU, filesystem or network load. This results in a commensurate increase in the storage space   used as well as the processing done initially for logging, and subsequently for parsing and filtering, the audit data. In the case of rule-based signatures, refinement can be effected through the use of more contextual events. As more events are logged, the storage overhead increases. Refinement can also be achieved by increasing the details collected for each event, such as inclusion of parameters along with system calls. This increases the runtime cost of processing an event significantly. For example, the use of a tool that traces each system call as it is invoked imposes enough overhead that it renders any large application unusable in practice.

Design of the SADDLE Architecture

Using more detailed intrusion signatures does not inherently require a significant performance penalty. Rather, the impact described in the previous section is due to the architecture of current auditing subsystems. The first issue is the coarse granularity at which it can be manipulated. Logging involves all instances of a class of event. If the intrusion detection system requires a detailed version of one instance, detailed logging is turned on for all instances implicitly. The second issue is the static nature of determining whether a particular audit record should be generated. An intrusion detection system may only need to know about a small subset of all events at any given instant. Yet all events that may ever be of interest are logged. The final issue involves extensibility. If a system was designed to sample a variable, such as the processor load, at a fixed interval and later a change is required, no methods exist to implement this. The change may be quantitative such as the change in the sampling interval, or qualitative, such as the need to obtain the current rate of change of the variable instead of its current value.

The above limitations can be seen to derive from the unidirectional nature of the communication between the logging facility and the intrusion detector. It is possible to remedy the problems through the use of a reverse channel through which the detector can specify to the auditing subsystem exactly what information it currently needs. It should be possible to dynamically modify the description of what information should be audited. A programmable interface should be present to allow the filtration and processing of audit data before it is committed to the log record. This also facilitates automated changes instead of manual reconfiguration. [Bishop96] describes a formal procedure for deriving exactly which pieces of data must be logged to assure the security of a particular protocol. Event pattern recognition can occur in either the auditing subsystem being described or by the intrusion detector, depending on the event filters defined. Depending on the specifics of a rule, the right choice can be either one. The architecture must be flexible enough to handle this. A simple programming interface is needed which can be invoked by an intrusion detection application, and is amenable to the above constraints.

Initially, we work with an interface of the form:

registerCallback(Event event, Callback callback, Boolean logInFile);

When an `event' occurs, the handler `callback' is invoked with `event' as a parameter. This requires `callback' to have been defined to conform to this prototype. If `logInFile' is true, then the event is appended to the audit trail. This interface allows intrusion detectors without support for a SADDLE architecture to use an audit trail as usual through the use of wildcard events and null callback handlers. A SADDLE-compliant detector instead uses the callbacks to get exactly the data it needs, even turning off the logging to a file when fine grained signatures are used.

The next step is to allow the processing of individual events. This can conform to either an interface that preserves the type or one that transforms the type:

InputType preservingFilter(InputType input);
OutputType transformingFilter(InputType input);

In some cases the events may need to be combined, yielding an output event:

OuputType combiningFilter(FirstType first, SecondType second);

Each filter must conform to the Filter prototype. Finally any composition of the above filters will yield a new filter that can be registered

registerFilteredCallback(Filter filter, Callback callback, Boolean logInFile);

Implicit in the scheme is that the output of the filter is passed as a parameter to the callback handler.

Above, we analyzed the design requirements of the audit record consumer. Additionally, design choices must be made regarding the audit trail producers. The data can come from the network or the host. Typically data derived from the network is sampled from a raw stream. Stateful packet inspection can potentially allow more complex determinations to be made as well. The design of the architecture should not preclude the use of this data. However, in the absence of clear security semantics, data from such a source may be unreliable.

Audit data gathered on a host can come from trusted sources such as the kernel, system libraries in user space, or applications whose trustworthiness can be established by a mechanism external to the scope of the architecture. Checking the user or group identity, or verifying an authentication credential are typical ways to decide whether to trust a program. Audit data from the operating system has several advantages. Its format can be controlled regardless of the application that is executing. The semantics of the audit record are clear. It is trusted. A potential advantage of log data from applications is the richer semantics available. However, this comes with the tradeoff that this data is usually generated for the purposes of debugging rather than security, may not conform to an audit record specification, and may not be trustworthy since it is not feasible to verify the integrity of the source of a large application, leave alone that of precompiled binaries.

There may be multiple audit data producers and consumers, with potential duplication of requests from consumers. A means for these streams to interface is needed. We term the process that provides this service the Audit Registry. It exposes the programming interface outlined above that intrusion detection applications can invoke. It converts the collection of requests into canonical form and then initiates auditing as needed. An architecture that conforms to this description effects System Auditing via Demand Driven Logging of Events (SADDLE).

Implementation

We now describe the implementation of SADDLE on two different platforms. In both cases, the implementation is restricted to the use of mandatory logging in the operating environment and does not handle authenticated application generated audit trails. Additionally, only the simpler form of SADDLE without the use of filters is used.

The first implementation modifies the Java runtime environment (version 1.4). Auditing functionality is added to a subset of the methods in the classloader, access controller, filesystem and networking code. When a class is loaded, a privileged action is performed, a file is opened, a network connection is made or accepted, or when a network service is started, the Audit Registry is informed. (A native interface library was also written to sample continuous valued metrics such as CPU, filesystem and network load when the virtual machine is running on Linux. This is for use with a filter-enabled version of SADDLE.)

The second is done on Linux (with kernel 2.2.12). The kernel auditing facility is used to monitor a subset of the system calls that are analogous to those modified in the Java runtime. This is effected by inserting a kernel module that registers its own wrapper functions in place of the relevant system calls. The wrapper function audits the call, and then invokes the original call. The use of this module allows us to monitor when a privilege elevation occurs, when a program is executed, when a file is opned, when a network connection is made to or from a remote node, and when a program starts a network service.

Both the modified platforms now have auditing code that communicates with a common implementation of the Audit Registry described in the previous section. In addition, a SADDLE-compliant intrusion detector, JStat, modeled after the state-transition approach of [Ilgun95] is constructed with the input events being the set described above.

Evaluation

We now describe three experiments undertaken to evaluate the utility of SADDLE. First, we compare storage requirements generic auditing and auditing classes of events relevant to signatures, versus just the instances needed for correct operation of the intrusion detector. This enables us to characterize the benefits of using SADLLE-based selective auditing. Next, micro-benchmarks are performed on some of the calls that were modified to include auditing functionality. Finally, a macro-benchmark is run with a SADDLE-compliant state transitional analysis intrusion detector while varying the number of signatures. This allows the characterization of the scalability of a SADDLE-based system. The first and third are performed by driving a web server with a set of requests, since this is an application that exercises the subsystems that we have modified.

The first experiment measures the size of the audit trail generated in three cases: (i) when all the system calls in our audit set are logged, (ii) when only system calls that are present in intrusion detection signatures are logged, and (iii) when relevant system calls are logged only when they are actually being watched for by the intrusion detector. A set of 10 signatures, averaging 5 events in length, was used. The workload was generated by using [SPECweb99] with 10 clients and 1 Apache web server. The graph in Figure 1 shows the drastic reduction in audit trail size by filtering out classes of events that will never match any event in the intrusion detection signature database, and the further filtration by temporal relevance. The third type is effected done by a SADDLE-based JSat . We see the need if fine grain data such as system calls are used and the intrusion detector is to operate in real-time on a heavily used node such as a web server.

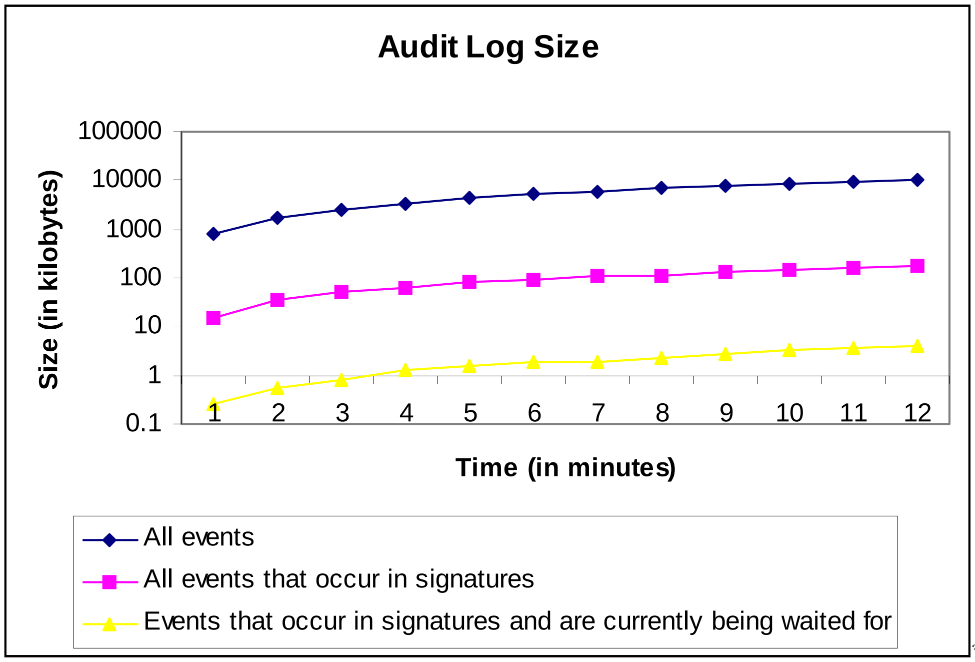

Figure 1 : Audit trail size reduction is effected via automated filtering of which system calls to log.

The modifications made in the runtime enviroment of Java and Linux introduce overhead. We wished to examine the extent of this. We picked the 3 most frequently invoked calls – opening a file, creating a network socket to a remote node, and executing a binary and measured their time with and without the SADDLE-based system in place. Network connections were made to local virtual hosts to avoid variation in network conditions, and Java classes and Linux binaries were generated with no other operations in them. The results are summarized below.

	Java	Java with Saddle	Linux	Linux with Saddle
File open	0.5 ms	4 ms	60 µs	120 µs
Network socket	150 ms	160 ms	10 ms	12 ms
Load class/binary	15 ms	20 ms	7.2 ms	7.8 ms

Initial implementation was done on the Java platform and synchronized calls by holding a lock in the Audit Registry to guarantee consistency in the Audit Registry's view of the sequence of events. This was improved in the Linux implementation through the use of FIFO buffer to which calls were reported from where the Audit Registry read them allowing the caller to proceed without blocking on an Audit Registry lock.

Another aspect of interest was how the SADDLE-enabled Jstat affected the performance of the workload as the number of intrusion detection signatures increased (and hence caused more instances of the modified system calls to be audited). To evaluate this we ran SPECweb99, varying the number of signatures in each run and noting how many requests were served per minute. From the graph in Figure 2 we see that the number of signatures did not affect the performance. Although the penalty paid on individual system calls can be high as seen above, the overall contribution of the set of audited calls is still not signiifcant – allowing performance to scale with the use of a large number of intrusion detection signatures.

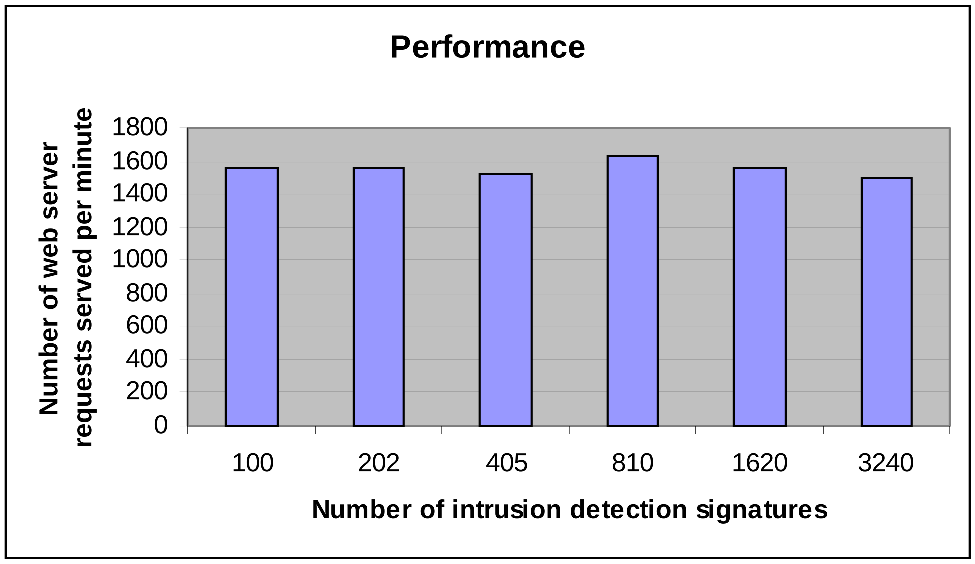

Figure 2 : Workload performance is not affected as the number of detection signatures increases.

Further evaluation is planned for the use of SADDLE with filters and continuous valued data, such as CPU load, that can only be sampled at a coarse granularity if the sampling is not done on demand as SADDLE allows. This will be reported on in the future.

Previous Work

[Anderson80] first used audit records to monitor for threats against Air Force computers. Most host-based intrusion detection systems still use logs[Axelsson00]. [Schneier99] tackles the problem of maintaining the integrity of log files on a host. [Silbert88] considered the problem of secure auditing in a distributed system. [Chapman00] focused on reconstructing formal semantics from logs. [Roger01] implemented an online algorithm that checks the model defined by declarative constraints stated in a temporal logic against log records. Previous efforts have focused on log *processing*, while our work hones log *generation* for intrusion detection.

Conclusion

The benefits of the use of SADDLE are threefold. First, by obviating the need to store all events of a class, it allows very fine grain auditing while maintaining reasonable storage requirements for the log files. Next, by migrating the decision whether to log closer to the point of event generation, there is a reduction in the propagation of unused data through the (typically slow) I/O subsystem, thereby potentially reducing the running time of applications. Finally, there is a reduction in the administrative burden of manual selection of event granularity and determination of which subset of events to audit.

Bibliography

[Anderson80] James P. Anderson, Computer Security Threat Monitoring and Surveillance, James P Anderson Co., Fort Washington, PA (1980).

[Axelsson99] Stefan Axelsson, The Base-Rate Fallacy and its Implications for the Difficulty of Intrusion Detection, 6th ACM Conference on Computer and Communications Security, 1999.

[Axelsson00] Stefan Axelsson, Intrusion Detection Systems: A Survey and Taxonomy, Technical Report, Chalmers University of Technology Department of Computer Engineering, Number 99-15, March 2000.

[Bishop96] M. Bishop, C. Wee and J. Frank, Goal Oriented Auditing and Logging, Submitted to IEEE Transactions on Computing Systems, 1996.

[Chapman00] Chapman Flack and Mikhail J. Atallah, Better Logging Through Formality: Applying Formal Specification Techniques to Improve Audit Logs and Log Consumers, Recent Advances in Intrusion Detection, Lecture Notes in Computer Science, Number 1907, pp. 1-16, Springer-Verlag, October 2000.

[Denning87] D. Denning, An Intrusion-Detection Model, IEEE Trans. on Software Eng., February 1987.

[Ilgun95] Koral Ilgun and Richard A. Kemmerer and Phillip A. Porras, State Transition Analysis: A Rule-Based Intrusion Detection Approach, IEEE Transactions on Software Engineering, 21(3), pp. 181-199, March 1995.

[Roger01] Muriel Roger and Jean Goubault-Larrecq, Log Auditing through Model Checking, Proc. 14th IEEE Computer Security Foundations Workshop, Cape Breton, Nova Scotia, Canada June 2001, pp. 220-236, IEEE Comp. Soc. Press.

[Schneier99] Bruce Schneier and John Kelsey, Secure Audit Logs to Support Computer Forensics, ACM Transactions on Information and System Security, 2(2), pp. 159-176, May 1999.

[Sibert88] W. Olin Sibert, Auditing in a Distributed System: Secure SunOS Audit Trails, 11th National Computer Security Conference pp. 81-91 (1988).

[SPECweb99] http://www.spec.org/osg/web99/