Event Processing to Verify Compliance with Security Policies

Sashikanth.Chandrasekaran@oracle.com

1. Abstract

Enterprises have several security policies for their backend databases and business applications. Security officers are faced with the challenge of verifying that these policies are being complied with across many databases and many applications. Commercial tools are available to verify simple policies – e.g., verify that the latest security patch has been installed. However, there is a lack of products that verify complex, dynamic, data dependent security policies that span multiple systems.

We describe a security event processing system that can be used to monitor enterprise databases and business applications. The user specifies the policies using an event specification language. The system allows complex correlations between security events, aggregations, pattern matches, etc. We continuously monitor security events generated in audit records, trace file entries etc. and flag events that violate the policies. Further, complex policies can be retroactively verified on past security events during a security audit.

2. Salient Features

2.1 Interface to Author Complex Security Policies

Complex policies can be composed based on the occurrence or non-occurrence of atomic security events. A user logging in to a database or the grant of a privilege to a user is an example of an atomic security event. Atomic security events can be combined using one or more of the following operations:

1. Correlation of attributes of atomic security events: Standard comparison operations can be used to chain atomic security events. As examples, we can use the equality condition on a *username* attribute to chain events that apply to the same user, apply ordering conditions on a *timestamp* attribute to order atomic events, use regular expression matches on an *IP address* attribute to test if events where generated within the same network, etc.
2. Aggregations: Aggregation functions such as *count, sum, min, max* and *avg* can be used to specify the number of occurrences of atomic security events within a time window, the earliest/latest occurrence of an event etc. As examples, we can use *count* over a time window of 60 seconds to determine if there were 3 or more login failures within 60 seconds, we can use *min* to determine the earliest occurrence of a security event, etc.
3. Test existence or non-existence of events: We provide *EXISTS* and *NOT EXISTS* clauses to test if a security event has occurred within a specified time window, between two other events, since the occurrence of an event and the current wall-clock time, etc.

Thus, atomic security events can be used as building blocks for complex pattern of events that succinctly capture the desired security policy. Compound security events can in turn be referenced in other event definitions, thereby allowing the user to author complex policies using a hierarchy of events.

2. 2 Enabling Audit Trails based on Security Policies

We need a source of security events feed into the policy evaluation engine, before we can check for compliance with said policies. The source of security events is usually some kind of an audit trail. It is infeasible to enable full auditing of the transactions executed by databases and applications. While most commercial systems have capabilities for fine-grained auditing, it is difficult to determine the appropriate auditing method so that the information needed for evaluating the policies is collected with minimal performance impact. Without a precise definition of security policies, most users either collect too much or too little audit or other security event information.

In our system, we compile the policy specifications and accordingly enable the audit trail settings in the source systems that generate the security events. Thus, only the events that are necessary for verifying compliance are collected. The user is freed from the burden of determining the settings needed in each source system to log security events with the smallest overhead.

2.3 Storage of Security Events in Append-Only Sequences

The security events are collected from multiple source systems and stored in a central repository as append-only sequences. The central repository allows us to easily evaluate policies that correlate security events generated in different sources. New security events are only appended to the repository; existing security events are never modified. This allows the user to perform a post-mortem audit or retroactively verify a security policy based on events that might have happened in the past. Further, the programmer is relieved of the burden of coding an explicit change-trail within the application logic. As an example, when a user's privileges are modified the append-only sequence of security events can be used to reconstruct the previous set of privileges granted to the user and when those privileges where modified. The application only needs to manage the current set of privileges.

3. Experimental Results

We translated security policies that are advocated as best practices for securing databases into our event specification language. For the purpose of comparison, we coded the same policies using vanilla SQL. The results demonstrate that it is not only easier to write complex policies using our system but it can also execute them efficiently. Briefly, the SQL queries are inefficient whenever the policy refers to a sequence of events in a single stream. These can be expressed in SQL only using numerous self-joins, which are inefficient to evaluate. It is possible to rewrite some of the SQL queries to either use more efficient advanced SQL analytic functions or decompose a complex query into smaller pieces and combine the results. Our expectation is that such query transformations are best performed automatically by a tool like the one we have described here rather than expect the user to hand-code efficient queries.

In the table below, the first column is a brief description of the policy. The security event data was generated from a simulation of the activity of 20,000 intranet users and 30 administrators in 7 different systems. Overall, 12.5 million security events were generated during the total period of the simulation. The policies were evaluated after every batch of 100K events was generated. The high-frequency security events such as logging in and accessing data were periodically purged after a million such events were processed. The low-frequency events were retained permanently. The policies were evaluated on a DELL x86 Linux desktop with 2GB RAM.

Policy	Our System	Vanilla SQL Queries
All user accounts must be managed via LDAP. For every login event, verify that there exists a prior event indicating that said user is *active* in LDAP.	5 secs	183 secs
Accounts must be locked if an incorrect password in entered 3 times within 1 minute. Verify that a sequence of 3 login failure events in 1 minute is not followed by a successful login, unless there is an intervening event that resets the password.	7 secs	52 secs
The root password must be changed within 24 hours after a user with root privileges has been terminated. Detect the following sequence of events: A user is granted root privilege, said user is terminated, 24 hours have passed and there is no event that would have been generated if the root password had been changed on that source system.	4 secs	1 sec
A user must not login from within the intranet and from outside the intranet within 5 minutes. It may indicate that the user has disclosed the password to an outsider.	7 secs	3 secs
Report logins into accounts that have not been used for a long time. Flag a login event if a long time has elapsed since the previous login event for the same user, but there have been many other users logging into the same system during this time interval.	12 secs	140 secs
An administrator must have only one active session into a database at any time. Detect the following sequence of events: A user is granted admin privileges, said user logs in, there exists another login event from said user before previous session is logged off.	17 secs	135 secs
Users must not share terminals that are within protected subnets. Test if an event has been generated that tags a subnet as *protected*, a user logs in from such a subnet, and another user logs in from same terminal in subnet before previous session is logged off.	22 secs	900 secs
Flag attempts to break-in by guessing usernames. Flag a sequence of 3 login failures in 1 minute, if there are no prior events that would have been generated if a valid account were created for supplied usernames.	4 secs	200 secs
Sensitive data must be accessed only from trusted applications. Test if any of the objects referenced in a logged statement has been marked by a prior event as sensitive. If such an event exists, flag if the client is not marked by a prior event as trusted.	8 secs	74 secs
Users must not be granted roles with a conflict of interest. Flag uses of a privilege P1, if another security event is generated for said user indicating privilege P2 was used, and privileges P1 and P2 have been marked as conflicting privileges.	26 secs	280 secs