Policy-Based Enforcement of Database Security Configuration through Autonomic Capabilities

Ghassan 'Gus' Jabbour
The Volgenau School of IT & Engineering
George Mason University
Fairfax, VA 22030, USA

Daniel A. Menascé
Dept. of Computer Science, MS 4A5,
George Mason University
Fairfax, VA 22030, USA
menasce@gmu.edu

Abstract

Significant emphasis has been placed recently on the hardening of databases and on regular audits of such systems by independent auditors and certified Information Systems Security Officers (ISSO). Data centers hosting sensitive data and mission-critical systems, especially centers that belong to governmental agencies, have been under tremendous pressure to secure their databases in compliance with several security guidelines. Such requirements mandate that each system passes a strict security scan before it is deemed suitable to go into operational mode and that it be subjected to regular audits thereafter. This in turn has been putting tremendous pressure on database administrators who, in many cases, are already overwhelmed by the tasks of installing, properly maintaining, and configuring their systems in a way that provides optimal performance. However, it is becoming extremely challenging, time consuming, and resource intensive to address security demands under tight budgets and timelines. Therefore, it would be advantageous to implement autonomic features into database systems to address some aspects of this challenge. This paper presents a framework that embeds autonomic capabilities into database systems to provide self-protection features in case of unauthorized, inadvertent, or intentional change in security parameters. This is achieved by embedding into the database the capability to compare each security configuration parameter change attempt (or request) with an embedded predefined security policy before allowing or rejecting the change. The paper demonstrates how the proposed framework can be implemented in an Oracle 10g Release 2 database.

1. Introduction

Autonomic computing is a sub-discipline of computer science that deals with the design of self-managing systems, i.e., with systems that are self-optimizing, self-configuring, self-healing and self-protecting [1]. This paper deals with the self-protecting dimension of autonomic computing as it applies to database security. To that end, significant emphasis has been placed over the past few years on the hardening of databases according to security guidelines established by the system owner. This notion of increased awareness regarding securing computer systems, especially the ones that are considered mission-critical, has manifested itself more clearly in the wake of increased malicious attacks and the potential threat of terrorism against critical computer targets especially governmental ones. Therefore, with this growing concern of security threats and potential security breaches that may be related to terrorism or other types of attacks, system owners, especially government agencies, have been imposing strict security guidelines for all aspects of their systems as required by the United States Federal Information Security Management Act (FISMA) of 2002 [2]. System owners have been required to put in place very rigid requirements for keeping their systems fully compliant with strict security policies and to have their systems scanned on a regular basis to guarantee that no security configuration has been altered. This strict security requirement has been accentuated by several government laws, regulations, directives, and publications. According to U.S. Homeland Security Presidential Directive/Hspd-7, all agencies are required to identify and provide "information security protections commensurate with the risk and magnitude of the harm resulting from the unauthorized access, use, disclosure, disruption, modification, or destruction of information" and information resources consistent with FISMA of 2002 [3].

Recent advances in autonomic databases and autonomic

0-7695-3093-1/08 $25.00 © 2008 IEEE
DOI 10.1109/ICAS.2008.49

database management systems (ADBMS) have focused on automatic management of resources for achieving optimal performance [4], autonomic provisioning of databases [5], resource selection for database optimization and tuning [6, 7], policy-based decisions and management [8], self-sizing of clustered databases and buffer pools [9, 10], isolation of attacks on data and recovery from malicious transactions [11, 12], and role-based access control [13, 14]. But, since the security of database and information systems is a problem of confidentiality, integrity, and availability of the data stored in these systems [11, 12, 13], most security research efforts have focused on securing the data by either controlling and detecting any unauthorized access to it, or by recovering any lost data in case of a malicious attack. For many years, the main focus of attention has been on researching, developing, and implementing technologies such as firewalls, intrusion detection, and virtual private networks. Relatively little has been done to address the security of the applications and database management systems that access or host the data [15].

However, these challenging requirements have motivated database vendors as well as companies that build database scanning and monitoring tools to examine ways to improve the security of databases and to create software applications that detect suspicious activities that might result in unauthorized access to the database and possible compromise of its state. As a consequence, several software tools and technologies became available on the marketplace that address database security issues by providing mechanisms that investigate the configuration of a database and report potential risks and threats to its owners based on certain security guidelines. Such approaches emphasize the monitoring aspect by connecting to a database and constantly monitoring it for any changes mainly in the status of the services/processes that support its availability (e.g., listener, instance, firewall, etc.). Some tools also monitor changes in the configuration settings of the database. However, existing tools fail to address the challenge of really securing the database because monitoring tools can be disabled at any time by insiders such as systems and network engineers, network operating center (NOC) personnel, or database administrators. Whether the intention to disable the tools is good, and no matter how justifiable the reason to disable them is, the fact remains that during the period of no monitoring the database is rendered vulnerable to undetected unauthorized access and potential risk of compromising it or steeling its data.

Some of the existing tools monitor networks and database systems, gather information, and produce alerts. They are also capable of feeding the gathered information into statistical and analytical tools that make recommendations to the system owner or the database administrator on how to solve the problem. The rest however, depends on human intervention for implementing the appropriate resolution. In addition, the effectiveness of these tools is totally dependent on an absolute non-interruption of the communication with the database. An interruption of communication results in interruption in monitoring and protecting the database.

This paper presents a framework that addresses the shortcoming of these tools using the notion of embedding policies into the database itself and enabling these policies to block every attempt to compromise the state of the database, or to alter its configuration in a way that contradicts what has been established and fed into the policy by the system owner. These policies can be established at different granularity levels in such a way that the system owner can choose to invoke coarse-grained policies to monitor and control the behavior of the database as a whole through the use of global settings, or invoke fine-grained policies that affect specific aspects or configuration settings. But the absolute core principle of our framework is the notion that the security policies, as well as all the database objects and logic that enforces them, are made an integral and inseparable part of the database that they are meant to protect. They are embedded into the database and are protected from being accessed by any user or entity other than the system owners. Even then, the framework gives the system owners the ability to institute a composite password where each part of the password is owned by a different member of the system owners team for added security.

2. Policy-Based Framework for Securing the Database

Having a privileged access to a database implies having the power to impact or change foundational aspects of its configuration thus altering its intended behavior. Database administrators typically have such a privilege since they are usually tasked with managing the database configurations and can conceivably be a threat as has been shown by several database security breaches over the past few years [16, 17, 18, 19]. Whether intentional or not, database administrators or power users can alter security configurations in a way that could result in unauthorized access to and compromise of the database. An example would be that of granting privileged access to unprivileged users, or just simply misusing his/her privileged access. Another example is one that pertains to security scans or audits of the database. Independent auditors are usually hired to perform a security scan of the database and they work with the DBA to get the database to a point where it is hardened enough to pass the scan. However, a database administrator can temporarily (or permanently) set some or all of the configuration parameters back to their original settings in order to achieve certain goals that he/she thinks are justified. One

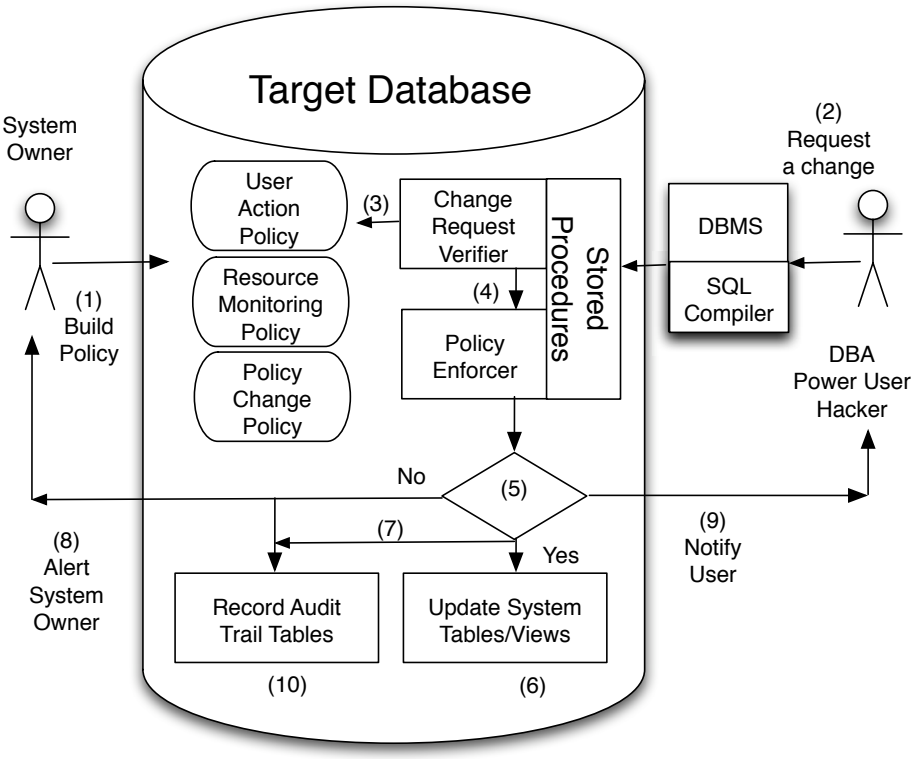

Figure 1. Architecture of a Self-protecting Database.

basic but dangerous scenario, for example, is that of altering the parameter that restricts using the same password after it expires. The DBA can easily set that parameter to unlimited, change the password to the same one, and then set that parameter back to what it is supposed to be. By doing so, the DBA would have violated the rule that applies to reusing the same password over and over again (see section 2.3) for a detailed example of this scenario). In this paper we describe a policy-based approach for enforcing database configurations even to those who have privileged access. We do not advocate minimizing the role of the DBA or restricting his/her access. However, we do advocate that each action gets verified and approved by system owner embedded, predefined configuration policies before it is applied to the database.

2.1. Architecture of the Framework

Unlike database security frameworks that exist today, which mostly detect imminent problems, generate an alert, and produce a report, our solution, which is an inseparable component of the database that it is meant to protect, mitigates any detected risk on its own without having to wait for human intervention. To illustrate this notion of uninterrupted self-protection we present an architectural overview of the proposed framework. Note that all the policies and the verification component reside within the database that is the subject of protection.

In this architecture (see Fig. 1), the system owner builds policies to support certain objectives (step 1). These policies are stored in the same database being protected (the target database) and are used to verify requests (or attempts) to change database configurations and to enforce the policy mandates. When a power user or a hacker initiates an attempt to change security configurations (step 2), the request goes through a process of verification before it can be processed. This step is carried out by database stored procedures that have built-in logic for checking the request against the policies (steps 3 & 4). If the request complies with the set policies that govern its scope of applicability (step 5), then the request is applied. Then, the database system tables/views are updated to reflect the change (step 6) and an audit trail is recorded (step 7). Otherwise, the request is rejected and the system owner is alerted (step 8), the user notified (step 9), and an audit trail is recorded (step 10).

Figure 2 shows the traditional way of managing database

configurations by the database administrator or any privileged user. The DBA logs onto the database with privileged access and executes SQL commands that alter the configurations.

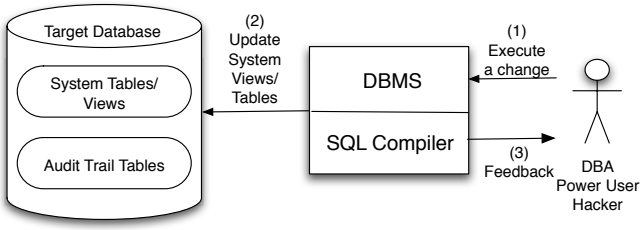

Figure 2. Traditional approach to managing database configurations.

Figure 3 shows a flowchart of the processes involved in providing uninterrupted self-protection through embedding autonomic capabilities into the database.

2.2. Policy Integration

As mentioned earlier, our framework is built on the notion that all policies must be an integral part of the database that they are meant to defend in a fashion that makes them directly associated with its very existence. Hosting them in an external application or database or even in another database instance on the same server as the target database, introduces the risk of isolating the policies from the target database and therefore, creating an opportunity for accessing and compromising the database without any lines of defense. In our framework, the policies are fed into physical database tables where they are stored and managed by the system owner. These tables are owned by the database itself and are not accessible to applications, application servers, or even power users. We normalize the presentation of policies into two primary tables, namely POLICY_HEADER and POLICY_MANDATE as shown in Tables 1 and 2 below.

2.3. Policy Representation

Policy-based computing is one of several techniques used to implement autonomic capabilities in computer systems. In this paper we apply the use of policies to implement autonomic capabilities into a database. We allow the system owner to create database configuration-specific policies that decide the actual run-time behavior of the database. These policies control and decide which changes are allowed and which ones are not. This is based on the database

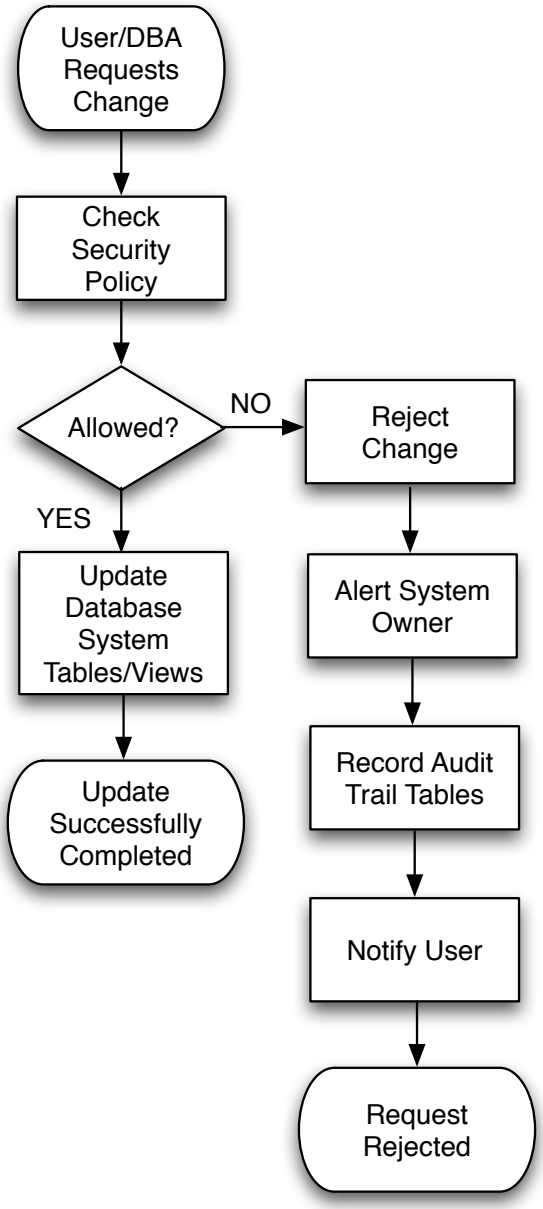

Figure 3. Flowchart of the uninterrupted self-protection approach.

being aware of its operational state being able to defend itself against input from various environmental sources such as users, malicious code, or external software systems. Policy-based mechanisms are also very useful in enforcing Service Level Agreements (SLA) especially those that are related to critical database availability and performance.

In general, policies provide the capability of controlling

Table 2. POLICY_MANDATE

POLICY_MANDATE	
MANDATE_ID	Unique identifier of the policy type - primary key
POLICY_ID	Unique identifier of the policy - foreign key
POLICY_DB_INSTANCE	The instance name of the DB to be monitored
POLICY_PARAMETER	A DB parameter that is a part of an overall security configuration
POLICY_PARAM_VALUE	A value assigned by the system owner to be enforced by the policy
POLICY_DB_COLUMN	The column in the DB table that refers to a configuration parameter
POLICY_DB_VALUE	The value of the DB column that refers to a configuration parameter
POLICY_PARAM_VERIFY	A DB command that selects the value of the parameter under investigation
POLICY_PARAM_VALID	A mechanism for ensuring that the request to modify a DB configuration parameter is valid
POLICY_QUERY	A SQL command that is used to monitor certain resources/behavior of the DB
POLICY_QUERY_CONDITION	A condition set by the system owner to enforce certain DB configuration as well as to maintain and forecast resource usage
POLICY_DRIVEN_ACTION	An action that the policy issues based on information it possesses, and information that it collects.
POLICY_ACTION_EXECUTE	Specifies the stored procedure that needs to be executed to fulfill the required action

Table 1. POLICY_HEADER

POLICY_HEADER	
POLICY_ID	Unique identifier of the policy
POLICY_NAME	Name of the policy
POLICY_OWNER	Policy owner
POLICY_TYPE	Objective of the policy as it relates to securing the DB
POLICY_CREATE_DATE	Date the policy was created
POLICY_MODIFY_DATE	Date the policy was modified
POLICY_EFFECT_DATE	Date the policy is to take effect
POLICY_ACTIVATE	Switch for turning policy ON or OFF without having to delete it
TARGET_DATABASE	Name of the DB the policy applies to

who can do what, when, where, and how. But the use of policies can also be exploited to predict the reason of performing a certain action, i.e., the why. Based on the specifics of the action, the system can gather and formulate intelligence that would be the basis for deducing the motive behind the action.

The purpose of using policies is mainly to enforce system owner requirements and constraints onto a system. A policy can be perceived as a declarative means for expressing directives to be carried-out by the system hosting these policies, which in this case is a database engine. In this framework, we implement policies into a database system for the purpose of making the database perform specific actions in response to attempts to alter its state or its configuration settings. Therefore, the creation and enforcement of policies requires the establishment of rules that invoke certain actions to be performed under certain conditions. To support our policy-based framework we enumerate four types of policies that we embed into an Oracle 10g database to demonstrate the effectiveness of the approach. Examples of these policies are shown below. While defining the policy, system owners may enter a wild card represented by * to indicate that a certain policy term can take as an input any value in its space. They could also use N/A represented by # to indicate that the term is not applicable to the scenario they are interested in.

Policies are usually created by the system owner for a specific domain or area of applicability that the policy intends to govern. That area of applicability is specified by the type of the policy. For example, policies of type one and two (defined below) are intended to govern the areas of enforcement of the password life time security configuration and database monitoring, respectively, while policies of type three and four govern changes to the security policy itself. The four types of policies are illustrated below. Other policy types can also be added depending on the need.

Type 1. Policy for Verifying & Controlling User Actions The role of this policy is to verify and control the actions of privileged users such as database administrators and power users. The validation process is performed as a response to the users input as shown below. The DBA calls a stored procedure specific for changing the life time of a password and chooses the values of the parameters he/she intends to change. These values are then verified according to the policy that governs their applicability. As shown in example 1.1 (see Table 3), the policy states that the life time of a password is set to 60 days and cannot be changed. It also stores the SQL command used to verify the change and the action that it takes depending on whether the change is approved or rejected.

Table 3. Example 1.1

[type]	enforcement of password life time security configuration
[name]	Password Life Time
[parameter]	password_life_time
[paramvalue]	60 (days)
[verification]	SELECT LIMIT FROM DBA_PROFILES WHERE RESOURCE_NAME = PASSWORD_LIFE_TIME AND PROFILE = DEFAULT;
[validation]	if input = UNLIMITED or input $\neq$ 60
[actionrejec]	alert system owner of failed attempt, notify user of disallowed parameter change, and record action in audit trail for further investigation
[actionexec]	execute password_life_proc

In example 1.2 (see Table 4), the policy enforces the archive log mode of the database. An attempt to change that mode is rejected because the policy only accepts the archive log mode to be either TRUE or ARCHIVELOG.

Table 4. Example 1.2

[type]	enforcement of the DB archive mode policy
[name]	Archive Log Mode
[parameter]	log_archive_start
[paramvalue]	TRUE
[dbcolumn]	log_mode
[dbvalue]	archivelog
[verification]	SELECT LOG_MODE FROM SYS.V$DATABASE;
[validation]	if LOG_MODE = NOARCHIVELOG
[actionrejec]	alert system owner of failed attempt, notify user of disallowed parameter change, and record action in audit trail for further investigation
[actionexec]	execute enf_archive_log_proc

The [actionexec] term of the policy determines the database stored procedure that is to be executed. In the case of example 1.2, the stored procedure would execute a set of commands and/or processes to send email to the system owner, to pop up a message window to inform the user of the outcome, and finally to record the action in an audit trail.

Type 2. Policy for Monitoring Database Resources The role of this policy is to proactively monitor database resources, such as active sessions, for the purpose of preempting situations that may deplete the database server of critical resources. The policy in example 2.1 (see Table 5) supports it designated stored procedures in monitoring and terminating all open but inactive sessions. The policy states that the logon time cannot be more than 24 hours.

Table 5. Example 2.1

[type]	database monitoring of inactive sessions
[name]	Monitor Inactive Sessions
[query]	SELECT USERNAME, SID, SERIAL#, to_char(LOGON_TIME,'Day HH24:MI') logon_time, to_char(sysdate,' Day HH24:MI') current_time FROM V$SESSION WHERE STATUS = 'INACTIVE';
[condition]	if logon_time = current_time + 24
[actionrejec]	kill the session
[actionexec]	execute kill_session_proc

The policy in example 2.2 (see Table 6) acts similarly to the previous one. However, it monitors CPU per session rather than inactive sessions. It stores the query to be used for monitoring the CPU and the conditions that must be met and the action to be taken in case they are not. The condition clause in example 2.2 can be relaxed a little by changing it to the following: if $\mathrm{CPU\_PER\_SESSION} > 70000$ or $\mathrm{CPU\_PER\_SESSION} < 50000$. This provides the DBA with some flexibility in changing configurations without compromising or harming the database.

Table 6. Example 2.2

[type]	database monitoring of CPU per session
[name]	Monitor CPU Per Sessions
[query]	SELECT RESOURCE_NAME, LIMIT FROM DBA_PROFILES WHERE PROFILE = APPUSER AND RESOURCE_NAME = CPU_PER_SESSION;
[condition]	if CPU_PER_SESSION $\neq$ 60000
[actionrejec]	alert system owner of failed attempt, notify user of disallowed parameter change, and record action in audit trail for further investigation
[actionexec]	execute monitor_CPU_proc

Type 3. Policy for Changing the Security Policy Conditions The role of the policy in example 3.1 is to allow system owners to make changes to the conditions that are set in the security policy itself (see Table 7). A comparison of this policy with that of example 2.1 indicates that the logon time has been set to 12 hours instead of 24.

Type 4. Policy for Changing the Security Policy Parameters The policy of example 4.1 (see Table 8) allows system owners to make changes to the security parameters value of the policy itself. Comparing this policy to the one example 1.2 we find that the archival mode of the database has

Table 7. Example 3.1

[type]	change the database resource monitoring policy
[name]	Change Resource Monitoring - Inactive Sessions
[parameter]	#
[limit]	#
[verification]	#
[query]	SELECT USERNAME, SID, SERIAL#, to_char(LOGON_TIME,'Day HH24:MI') logon_time, to_char(sysdate,'Day HH24:MI') current_time FROM V$SESSION WHERE STATUS = 'INACTIVE';
[condition]	if logon_time = current_time + 12
[actionrejec]	kill the session
[actionexec]	execute kill_session_proc
[effective]	immediately, i.e., current system date

changed to allow it to be in no archive log mode.

Table 8. Example 4.1

[type]	change the database archive mode policy
[name]	Archive Log Mode
[parameter]	log_archive_start
[paramvalue]	FALSE
[dbcolumn]	log_mode
[dbvalue]	noarchivelog
[effective]	immediately, i.e., current system date

There are four main features to our approach that make it effective, scalable, flexible, and extensible. First, the system owner can write as many policies as needed to govern a wide range of database security configurations having either global or specific applicability. Second, the structure for defining a policy is not limited but can be extended to fulfill any scope of applicability. Third, the fact that the policies are concise, targeted, and simple makes the task of creating them an easy one. Finally, the premise of this framework is scalable in that it can be applied to other domains where monitoring of system behavior or protecting system configurations is essential.

3. Implementation Scenarios

This section presents some implementation scenarios where the policy-based framework is utilized to autonomically enforce security configuration in databases thus enabling them to self-protect. To demonstrate the usability of the proposed framework we consider a policy for monitoring inactive sessions as presented in Table 5.

Ensuring that software application code closes all the sessions that it opens after they are no longer needed is vital for optimal database performance. However, this does not happen often especially in an environment where there is pressure to deliver on time, and where formal code reviews are not a common practice. This usually results in an increased number of inactive sessions to the point that it exhausts the database resources. Even though configuration parameters such as the connection timeout, and maximum and minimum connections allowed could be set properly, inactive sessions may still persist. However, this could be easily mitigated by creating and embedding a policy to monitor all active sessions. Our approach enriches the database with policy-based autonomic capabilities that enables it to overcome such a problem by autonomically enforcing policy mandates.

To implement and enforce the policy, a database stored procedure executes the SQL query provided by the policy once every few hours (or as specified by the DBA or system owner) in order to determine the duration of each inactive session. If the condition of the policy is met, then another stored procedure is executed to kill the offending sessions. This guarantees that the database resources are not depleted, without any human intervention.

Another implementation scenario that is extremely important to the security of any database is that of changing an existing database schema password to a different one every say sixty days. This rule is usually enforced by one or more database configuration parameters. In the case of an Oracle database, for example, the password parameters are set in the DBA_PROFILE system table.

Parameters such as PASSWORD_LIFE_TIME, PASSWORD_REUSE_TIME, and PASSWORD_GRACE_TIME control certain aspects of database passwords. The problem however, is that these parameters can be manipulated to allow the use of the same password over and over again. By implementing the enforcement of password reuse time policy shown in Table 9, system owners can rest assured that not even the DBA can reuse the old password after it expires.

We draw the reader's attention to the fact that the action clause simply initiated the notification to the user of the disallowed parameter change, alerted the system owner of the failed attempt, and initiated the recording of the audit trail. No other action was taken.

Finally, we tackle the scenario that deals with the issue of indirect privilege escalation such as getting DBA privileges from other privileges such as CREATE ANY TRIGGER, CREATE ANY VIEW, and EXECUTE ANY PROCEDURE. In Oracle 10g Release 2, a database account such as MDSYS, which is not a DBA, can be leveraged to gain DBA privileges. This could be accomplished through injecting SQL commands into triggers owned by MDSYS. In addition, by default, in Oracle 10g Release 2, the only user granted the CREATE ANY View privilege is SYS.

Table 9. Enforcement of password reuse time policy.

[type]	enforcement of password reuse time security configuration
[name]	Password Reuse Time
[parameter]	password_reuse_time
[paramvalue]	20 (times before a password can be reused)
[verification]	SELECT LIMIT FROM DBA_PROFILES WHERE RESOURCE_NAME = PASSWORD_REUSE_TIME AND PROFILE = DEFAULT;
[validation]	if input = UNLIMITED or input $\neq$ 60
[actionrejec]	alert system owner of failed attempt, notify user of forbidden parameter change, and record action in audit trail for further investigation
[actionexec]	execute password_life_proc

Table 10. Blocking indirect privilege escalation policy.

[type]	Create a DB schema and allowed privileges
[name]	Create New Schema and Privileges
[schema]	NewUser
[roles]	grant create session to NewUser, grant select on schema.table to NewUser;
[defaultTBS]	Users
[tempTBS]	Temp
[quotaType]	Unlimited
[quotaOn]	Users
[actionexec]	execute create_user_proc

But, if one can exploit a vulnerable procedure, he/she can gain DBA privileges. This topic has been well-illustrated in the literature [20]. A hacker would need to go through two or three simple steps before he could gain DBA privileges. The idea in its simplest form, in the case of CREATE ANY VIEW, is to find a SYS-owned vulnerable procedure and inject into it a SQL command to grant a DBA privilege to a certain user. The injection of SQL, which could be in the form of a function with a simple grant statement such as `grant dba to user;`, could be prevented by using our policy-based approach. In order to do that, the system owner preemptively creates a policy to prevent granting any privilege to any user unless that user's name along with the agreed upon privileges are specifically listed in the policy. In other words, the DBA has to discuss the creation of a new user and the set of privileges to be granted to it with the system owner. The system owner in turn would modify the policy to accept that new user account and the agreed upon set of privileges. Any attempt to grant privileges, other than the ones listed in the policy, through the use of SQL injection would not be allowed. See policy in Table 10.

As new policy terms are added (as is the case in the policy above), corresponding table columns get automatically added to the POLICY_MANDATE database table. It is also important to note that our stored procedures do not allow the user to input values into variables. Instead, the procedures present the user with a set of values to select from. This is important because it removes any possibility of the users injecting variations to the expected input.

4. Related Work

The use of policies has been applied to domains such as networking and network management [21, 22, 23, 24, 25, 26], quality of service architecture and management systems [27, 28], programming [29, 30], security management [31], agent based distributed systems [32], memory allocation [33], content delivery [34], routing [35], admission control [36, 37, 38], and autonomic computing [39, 40]. The concern of this paper however, is the application of policies to securing a database. Numerous software scanning and monitoring tools exit on the market today. Tools such as AuditPro for Databases [41], AppDetective [42], E-Trust Policy Compliance [43], NGSSQLCrack [44], Oscanner [45], Symantec Enterprise Security Manager for Databases [46], NeXpose [47], NGSSquirrel for Oracle [48], Appsentry for Oracle [49], ISS Database Scanner [50], SQLdict [51], and NGSSquirrel for SQL [48] have one common factor: they all automate and streamline the identification of vulnerabilities in addition to locating, reporting, and even helping to fix the discovered security vulnerabilities. They run independently of the database and quickly generate detailed reports with all the information needed to correctly configure and secure databases. While such tools have a lot to offer, their major deficiency is that they run totally independently of the database. This notion of residing and running on systems independent of the database make these tools vulnerable and inadequate for protecting the database.

A different approach to securing the database was provided by Sentrigo through the offering of a real-time auditing and monitoring platform called Hedgehog [52]. Hedgehog does not monitor the database via direct examination (interception) of system calls between the database and applications or log file analysis. Instead, it samples the database's shared memory in real-time. While this technology is the closest we found to our approach, we believe that our approach is more flexible because it allows the user to create any kind of policy he or she desires. It also gives the user the ability to set policy conditions and modify them as the need arises. In addition, while Hedgehog can be used to log events, issue alerts and terminate sessions, our approach

blocks unauthorized events without having to abruptly terminate sessions. In summary, the system owners control the building and customization of their policies to best fit their business model and business needs. The approach that we present is a full encapsulation and integration of the protection mechanism into the core being of the database that is to be protected.

Ou approach differs from the security reference monitor [53] concept because we provide autonomic computing capabilities. The topology of the Reference Model places the "reference monitor" component as a separate entity from the "resource"' component. In fact, the model allows for the implementation of one reference monitor for multiple resources and multiple reference monitors for multiple resources. This is directly opposite to our proposed approach, which makes the implementation of the security enforcement mechanism (similar to reference monitor) an integral and inseparable part of the system (or resource) that it is protecting.

5. Conclusion and Future Work

This paper presented an innovative approach to implementing autonomic capabilities into database systems in order to enable self-protection. The cornerstone of our approach is the full integration of security policies into the database that they are intended to protect. By doing so we embed into the database autonomic capabilities that provide it with a superior self-protection mechanism that surpasses, in its effectiveness, existing database security frameworks. This paper concentrated on the approach and policy aspects. A companion paper will show in detail the mechanism of enabling the database with self-protection capabilities through the use of database stored procedures that act as the enabling engine for request verification, command compilation, database protection, and system owner and user notification.

References

[1] Menascé, D.A. and J.O. Kephart, "Autonomic Computing," Guest-editors Introduction. IEEE Internet Computing, 2007. 11(1).

[2] FISMA,"Federal Information Security Management Act of 2002. FISMA," Title III of E-Government Act passed by the 107th Congress and singed into law by the persident, 44 U.S.C. 3541, et seq., 2002.

[3] US Executive Branch, Department of Homeland Security, Homeland Security Presidential Directive/Hspd-7. www.whitehouse.gov, 2003.

[4] Martin, P., S. Elnaffar, and T. Wasserman, "Workload Models for Autonomic Database Management Systems". IEEE, 2006.

[5] Chen, J., G. Soundararajan, and C. Amza, "Autonomic Provisioning of Backend Databases in Dynamic Content Web Servers". IEEE, 2006: p. 231-242.

[6] Bruno, N. and S. Chaudhuri, "Automatic Physical Database Tuning: A Relaxation-based Approach". SIGMOD, ACM, 2005.

[7] Oh, J.S. and S.H. Lee, "Resource Selection for Autonomic Database Tuning". Proc. 21st Intl. Conf. on Data Engineering (ICDE' 05), IEEE, 2005.

[8] Boughton, H., et al., "Workload Class Importance Policy in Autonomic Database Management Systems". Proc. 7^{th} IEEE Intl. Workshop on Policies for Distributed Systems and Networks (POLICY'06), 2006.

[9] Taton, C., et al., "Self-Sizing of Clustered Databases". Proc. Symp. World of Wireless, Mobile and Multimedia, IEEE, 2006.

[10] Tian, W., P. Martin, and W. Powley, "Techniques for Automatically Sizing Multiple Buffer Pools in DB2". Queens University, 2001.

[11] Ammann, P., S. Jajodia, and P. Liu, "Recovery from Malicious Transactions". IEEE Tr. Knowledge and Data Engineering, 2002. 14(5).

[12] Liu, P., "DAIS: A Real-time Attack Isolation System for Commercial Database". Proc. 17^{th} Annual Computer Security Applications Conf., 2001.

[13] Bertino, E. and R. Sandhu, "Database Security - Concepts, Approaches, and Challenges". IEEE Tr. Dependable and Secure Computing, 2005. 2(1).

[14] Jeong, M.-A., J.-J. Kim, and Y. Won, "A Flexible Database Security System using Multiple Access Control Policies". Proc. 4^{th} Intnl. Conf. Parallel and Distributed Computing, Applications and Technologies, 2003. PDCAT'2003, 2003.

[15] Kern, A., et al., "A Meta Model for Authorisations in Application Security Systems and their Integration into RBAC Administration". ACM, SACMAT, 2004.

[16] Conry-Murray, A., "The Threat From Within". www.networkcomputing.com, 2005.

[17] Stanley, N., DBA Dnaffles Data - the Inside Threat continues. www.it-analysis.com, 2007.

[18] Stiennon, R., Your DBA has his/her hand in the till. www.zdnet.com, 2007.

[19] CNNMoney, Bank security breach may be biggest yet. CNNMoney.com, 2005.

[20] Litchfield, D., *The Oracle Hacker's Handbook: Hacking and Defending Oracle*. 2007.

[21] Bao, J.Q., L. Guo, and W.C. Lee, "Ad hoc networks: Policy-based resource allocation in a wireless public safety network for incident scene management". Proc. 2006 Wkshp. Dependability Issues in Wireless Ad-hoc Networks and Sensor Networks DIWANS '06, 2006.

[22] Itani, W., A. Kayssi, and A. Chehab, "T1-B: computer and network security symposium: An enterprise policy-based security protocol for protecting relational database network objects". Proc. 2006 Intl. Conf. Wireless communications and Mobile Computing IWCMC '06, 2006.

[23] Wright, M.J., "Using policies for effective network management," *Intl. J. Network Management*, 1999.

[24] Kim, G., J. Kim, and J. Na, "Design and implementation of policy decision point in policy-based network". Proc. 4^{th} Annual ACIS International Conference on Computer and Information Science (ICIS05), 2005.

[25] Perez, G.M., et al., "Dynamic Policy-Based Network Management for a Secure Coalition Environment". IEEE Communications Magazine 2006.

[26] Olaussen, E. and A. Karlsen, "A policy-based priority and precedence framework for military IP networks". Military Communications Conference, MILCOM 2004. IEEE, 2004.

[27] Rudack, M., et al., "Policy-based quality of service mapping in distributed systems". Network Operations and Management Symposium, NOMS 2002, IEEE/IFIP, 2002.

[28] Flegkas, P., P. Trimintzios, and G. Pavlou, "A policy-based quality of service management system for IP DiffServ networks". IEEE Network, 2002.

[29] Montanari, R., G. Tonti, and C. Stefanelli, "Policy-based separation of concerns for dynamic code mobility management". Proc. 27^{th} Annual Intl. Computer Software and Applications Conf., COMPSAC 2003, 2003.

[30] McDaniel, P., "On Context in Authorization Policy". 8^{th} ACM Symp. Access Control Models and Technologies, 2003.

[31] Bhatti, R., K. Moidu, and A. Ghafoor, "Healthcare data integration and exchange: Policy-based security management for federated healthcare databases (or RHIOs)". Proc. Intl. Wkshp. Healthcare information and knowledge management (HIKM '06), 2006.

[32] Tripathi, A., D. Kulkarni, and T. Ahmed, "Policy-Driven Configuration and Management of Agent Based Distributed Systems". 4^{th} Intl. Wkshp. Software Engineering for Large-Scale Multi-Agent Systems (SELMAS'05), 2005.

[33] Alexandrescu, A. and E. Berger, "Policy-Based Memory Allocation: Fine-tuning your memory management," Doctor Dobb's C/C++ Journal, 2005.

[34] Kawarasaki, M. and R. Atarashi, "Policy Based Content Delivery Management Using Metadata". Proc. IEEE 2004 Intl. Symp. Applications and the Internet Workshops (SAINTW'04), 2004.

[35] Chau, C.-k., "Policy-based Routing with Non-strict Preferences". SIGCOMM 2006, 2006.

[36] Verdi, F.L., M. Magalhaes, and E.R.M. Madeira, "Policy-based admission control in GMPLS optical networks". Proc. First Intl. Conf. Broadband Networks (BROADNETS04), 2004.

[37] Yan, Y., et al., "A policy-based admission control algorithm for UMTS end to end QoS provision". 2^{nd} Intl. Conf. Mobile Technology, Applications and Systems, 2005.

[38] Ramirez, S.L., et al., "Performance Evaluation of Policy-Based Admission Control Algorithms for a Joint Radio Resource Management Environment". Electrotechnical Conference, MELECON 2006. IEEE Mediterranean, 2006.

[39] Badr, N., A. Taleb-Bendiab, and D. Reilly, "Policy-Based Autonomic Control Service". Proc. 5^{th} IEEE Intl. Wkshp. Policies for Distributed Systems and Networks (POLICY'04), 2004.

[40] Chan, H. and T. Kwok, "A Policy-based Management System with Automatic Policy Selection and Creation Capabilities by using a Singular Value Decomposition Technique". Proc. 7^{th} Intl. Wkshp. Policies for Distributed Systems and Networks (POLICY'06), 2006.

[41] AuditPro, Network Intelligence (I) Pvt. Ltd. www.niiconsulting.com.

[42] AppDetectivePro, Application Security, Inc. www.appsecinc.com/products/appdetective/.

[43] E-Trust Policy Compliance,. Service Strategies, Inc., www.ebusiness-security.com/eTrust_Policy_compliance.htm.

[44] NGSSQLCrack, Next Generation Security Software,. www.ngssoftware.com/products/database-security/ngs-sqlcrack.php.

[45] Karlsson, P., OScanner for Oracle, 2004.

[46] Symantec Enterprise Security Manager for Databases,. Symantec, www.symantec.com/region/can/eng/product/esm/databases.

[47] NeXposeł, Rapid7. www.rapid7.com.

[48] NGSSQuirreL, Next Generation Security Software. www.ngssoftware.com/products/database-security/ngs-squirrel-oracle.php.

[49] Appsentry, Integrigy Corporation. www.integrigy.com/products.

[50] ISS, ISS Database Scanner. Internet Security Systems, Inc., www.iss.net/about/index.html.

[51] SQLdict, ntsecurity. www.ntsecurity.nu/toolbox/sqldict/.

[52] Hedgehog, Sentrigo. www.sentrigo.com/.

[53] Anderson, J.P., Computer Security Technology Planning Study, ESD-TR-73, Vol. II, Oct. 1972.