Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin

DFRWS 2017 USA — Proceedings of the Seventeenth Annual DFRWS USA

Carving database storage to detect and trace security breaches

James Wagner [a,*], Alexander Rasin [a], Boris Glavic [b], Karen Heart [a], Jacob Furst [a], Lucas Bressan [a], Jonathan Grier [c]

[a] DePaul University, Chicago, IL, USA
[b] Illinois Institute of Technology, Chicago, IL, USA
[c] Grier Forensics, USA

keywords:
Database forensics
File carving
Memory analysis

A B S T R A C T

Database Management Systems (DBMS) are routinely used to store and process sensitive enterprise data. However, it is not possible to secure data by relying on the access control and security mechanisms (e.g., audit logs) of such systems alone – users may abuse their privileges (no matter whether granted or gained illegally) or circumvent security mechanisms to maliciously alter and access data. Thus, in addition to taking preventive measures, the major goal of database security is to 1) detect breaches and 2) to gather evidence about attacks for devising counter measures. We present an approach that evaluates the integrity of a live database, identifying and reporting evidence for log tampering. Our approach is based on forensic analysis of database storage and detection of inconsistencies between database logs and physical storage state (disk and RAM). We apply our approach to multiple DBMS to demonstrate its effectiveness in discovering malicious operations and providing detailed information about the data that was illegally accessed/modified.

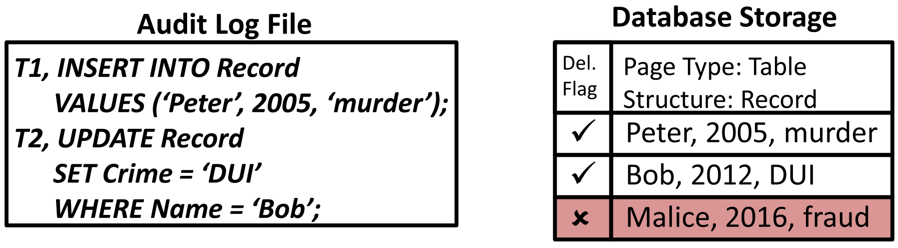
Introduction

Database Management Systems (DBMSes) are commonly used to store sensitive data and, accordingly, significant effort has been invested into securing DBMSes with access control policies. However, once a user has gained elevated privileges in the DBMS (either legitimately or through an attack), the security scheme put into effect can be bypassed, and therefore, can no longer assure that data is protected according to policy. A well-known fact from security research and practice is that it is virtually impossible to create security measures that are unbreakable. For example, access control restrictions 1) may be incomplete, allowing users to execute commands that they should not be able to execute and 2) users may illegally gain privileges by using security holes in DB or OS code or through other means, e.g., social engineering. Thus, in addition to deploying *preventive* measures such as access control, it is necessary to be able to 1) *detect security breaches* when they occur in a timely fashion and 2) in event of a detected attack *collect evidence* about the attack in order to devise counter-measures and assess the extent of the damage, e.g., what information was leaked or perturbed. This information can then be used to prepare for legal action or to learn how to prevent future attacks of the same sort.

When malicious operations occur, whether by an insider or by an outside attacker that breached security, an audit log containing a history of SQL queries may provide the most critical evidence for investigators (Mercuri, 2003). The audit log can be used to determine whether data has been compromised and what records may have been accessed. DBMSes offer built-in logging functionality but can not necessarily guarantee that these logs are accurate and have not been tampered with. Notably, federal regulations, such as the Sarbanes-Oxley Act (S.-O. Act) and the Health Insurance Portability and Accountability Act (A. Act, 1996), require maintaining an audit trail, yet the privileged user can skirt these regulations by manipulating the logs. In such cases, companies maintaining these systems are, technically, in violation of these regulations. Hence, assurance that security controls have been put into place properly cannot rest merely on the existence of logging capabilities or the representations of a trusted DBA. Internal controls are needed in order to assure log integrity.

* Corresponding author.
E-mail addresses: jwagne32@depaul.edu (J. Wagner), arasin@depaul.edu (A. Rasin), bglavic@iit.edu (B. Glavic), kheart@depaul.edu (K. Heart), jfurst@depaul.edu (J. Furst), lucasbressan3@gmail.com (L. Bressan), jdgrier@grierforensics.com (J. Grier).

http://dx.doi.org/10.1016/j.diin.2017.06.006



```
T1, INSERT INTO Record
    VALUES ('Peter', 2005, 'murder');
T2, UPDATE Record
    SET Crime = 'DUI'
    WHERE Name = 'Bob';
```

Del. Flag	Page Type: Table Structure: Record
✓	Peter, 2005, murder
✓	Bob, 2012, DUI
✗	Malice, 2016, fraud

Fig. 1. Illustrates that the active records for Peter and Bob can be explained by audit log events, whereas the deleted record Malice can not be explained by any audit log events.

Example 1. *Malice is the database administrator for a government agency that keeps criminal records for citizens (an example instance is shown in Fig. 1). Malice recently got convicted of fraud and decided to abuse her privileges and delete her criminal record by running* `DELETE FROM Record WHERE name = 'Malice'`. *However, she is aware that database operations are subjected to regular audits to detect tampering with the highly sensitive data stored by the agency. To cover her tracks, Malice deactivates the audit log before running the* `DELETE` *operation and afterwards activates the log again. Thus, there is no log trace of her illegal manipulation in the database. However, database storage on disk will still contain evidence of the deleted row (until several storage artifacts caused by the deleted are physically overwritten). Our approach detects traces of deleted and outdated record versions and matches them against the audit log to detect such attacks and provide evidence for how the database was manipulated. Using our approach, we would detect the deleted row and since it does not correspond to any operation in the audit log we would flag it as a potential evidence of tampering.*

In Section Reliability of Database Logs we showcase, for several databases, how an attacker like Malice can ensure that her operations are not being included in the audit log. Given that it is possible for a privileged attacker to erase log evidence and avoid detection, the challenge is to detect such tampering and collect additional evidence about the nature of the malicious operations (e.g., recover rows deleted by a malicious operation). It may not be immediately clear that this recovery of evidence is possible at all. However, any operation leaves footprints in database storage on disk (writes) or in RAM (both reads and writes). For instance, DBMSes mark a deleted row rather than overwrite it. Thus, if we recover such evidence directly from storage then, at least for some amount of time until the deleted value is overwritten by future inserts, we would be able to detect that there exists a discrepancy between the content of the audit log and database storage.

Given that evidence of operations exists in database storage, the next logical question to ask is whether Malice can remove this evidence by modifying database files directly. While a user with sufficient OS privileges may be able to modify database files, it is extremely challenging to tamper with database storage directly without causing failures (e.g., DBMS crashes). Direct manipulation of DBMS files will uncover the tampering attempt because: 1) in addition to the actual record data on a page, the database system maintains additional references to that record (e.g., in index structures and page headers). Deleting a record from a page without modifying auxiliary structures accordingly will leave the database in an inconsistent state and will lead to crashes; 2) databases have built-in mechanisms to detect errors in storage, e.g., checksums of disk pages. A tampering attempt has to correctly account for all of these mechanisms; 3) incorrect storage for a value can corrupt a database file. To directly modify a value, an attacker needs to know how the DBMS stores datatypes.

Because it is not only hard but, at times, next to impossible to spoof database storage, it follows that database storage can provide us with valuable evidence of attacks. We use an existing forensic tool called DICE (Wagner et al., 2017) to reconstruct database storage. However, we are still left with the problem of matching recovered artifacts to queries in audit log – doing so requires a thorough analysis of how database storage behaves. Our approach automatically detects potential attacks by matching extracted storage entries and reporting any artifacts that cannot be explained by logged operations (summarized in Fig. 2). Our method is designed to be both general (i.e., applicable to any relational database) and independent (i.e., entirely outside of DBMS control). Our system `DBDetective` inspects database storage and RAM snapshots and compares what it finds to the audit log; the analysis of this data is then done out of core without affecting database operations. `DBDetective` can operate on a single snapshot from disk or RAM (i.e., multiple snapshots are not required), but additional snapshots provide extra evidence and improve detection quality. Data that has changed between two snapshots need be matched only against audit log entries of commands that were executed during the time span between these snapshots. Thus, more frequent snapshots increase the detection accuracy because it is less likely to match a row against an incorrect operation and the probability that deleted rows are still present is higher. Moreover, frequency of snapshots increase the performance of detection because a smaller number of recovered rows have to be matched against a smaller number of operations. We can reduce storage requirements by only storing deltas between snapshots in the same fashion as incremental backups are used to avoid the storage overhead of full backups.

Our focus is on identifying the likelihood of database tampering, as well as pointing out specific inconsistencies found in database storage. Determining the identity of the party responsible for database tampering is beyond the scope of this paper. Due to the

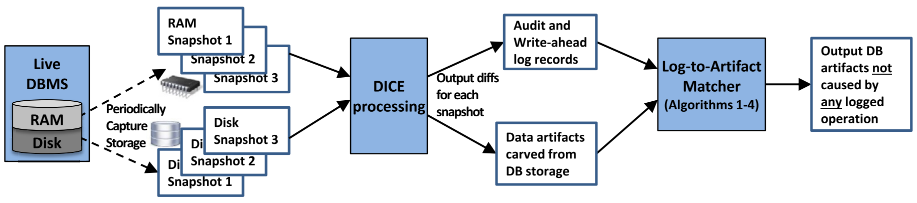

Live DBMS
RAM
Disk
Periodically Capture Storage
RAM Snapshot 1
Snapshot 2
Snapshot 3
Disk Snapshot 3
Snapshot 2
Snapshot 1
DICE processing
Output diffs for each snapshot
Audit and Write-ahead log records
Data artifacts carved from DB storage
Log-to-Artifact Matcher (Algorithms 1-4)
Output DB artifacts not caused by any logged operation

Fig. 2. Architecture of the DBDetective.