Establishing Independent Audit Mechanisms for Database Management Systems

Alexander Rasin
School of Computing
DePaul University
Chicago, IL, USA
arasin@cdm.depaul.edu

James Wagner
School of Computing
DePaul University
Chicago, IL, USA
jwagne32@depaul.edu

Karen Heart
School of Computing
DePaul University
Chicago, IL, USA
kheart@depaul.edu

Jonathan Grier
Grier Forensics
Pikesville, MD, USA
jgrier@grierforensics.com

***Abstract*—The pervasive use of databases for the storage of critical and sensitive information in many organizations has led to an increase in the rate at which databases are exploited in computer crimes. While there are several techniques and tools available for database forensic analysis, such tools usually assume an apriori database preparation, such as relying on tamper-detection software to already be in place and the use of detailed logging. Further, such tools are built-in and thus can be compromised or corrupted along with the database itself. In practice, investigators need forensic and security audit tools that work on poorly-configured systems and make no assumptions about the extent of damage or malicious hacking in a database.**

In this paper, we present our database forensics methods, which are capable of examining database content from a storage (disk or RAM) image without using any log or file system metadata. We describe how these methods can be used to detect security breaches in an untrusted environment where the security threat arose from a privileged user (or someone who has obtained such privileges). Finally, we argue that a comprehensive and independent audit framework is necessary in order to detect and counteract threats in an environment where the security breach originates from an administrator (either at database or operating system level).

***Index Terms*—database forensics, security audit, evidence gathering**

I. Introduction

Cyber-crime (e.g., data exfiltration or computer fraud) is a significant concern in today's society. A well-known fact from security research and practice is that unbreakable security measures are virtually impossible to create. For example, 1) incomplete access control restrictions allows users to execute commands beyond their intended roles, and 2) users may illegally obtain privileges by exploiting security holes in a Database Management System (DBMS), OS code or through other means (e.g., social engineering). Thus, in addition to deploying *preventive* security measures (e.g., access control), it is necessary to 1) *detect security breaches* in a timely fashion, and 2) *collect evidence* about attacks to devise counter-measures and assess the extent of the damage (e.g., what data was leaked or perturbed). Such measures are most vital when the user is operating from a position of elevated privilege, which may enable them to suspend security components, alter audit logs or raw data while avoiding detection. The resulting forensic evidence can also provide preparation for legal action or help prevent future attacks.

DBMSes are targeted by criminals because they serve as repositories of data. Therefore, investigators must have the capacity to examine the contents of a DBMS. Currently, an audit log with SQL query history is a critical (and perhaps only) source of evidence for investigators [4] when a malicious operation is suspected. In field conditions, a DBMS may not provide the necessary logging granularity (unavailable or disabled). Moreover, the storage itself might be corrupt or contain multiple DBMSes.

The field of Digital Forensics strives to provide tools for independent analysis with minimal assumptions about the environment. A particularly important and well-recognized technique is file carving [9], which extracts files (e.g., PDF or DOC, but not DBMS files) from a disk image, including deleted or corrupted files. Traditional file carving techniques rely on presence of file headers to detect and interpret individual files. DBMS files, on the other hand, do not maintain a file header and are never independent (e.g., table contents are stored separate from table name and logical structure information). Even if DBMS files could be carved, they cannot be meaningfully imported into a different DBMS and must be parsed to retrieve their content. To accomplish that task, DBMSes need their own set of digital forensics rules and tools.

Even in an environment with ideal log settings, a DBMS cannot necessarily guarantee log accuracy or immunity from tampering. For example, log tampering is a concern when a breach originated from a privileged user such as an administrator (DBA or an attacker who obtained DBA privileges). Tamper-proof logging mechanisms were proposed in related work [7], [10], but these only prevent logs from atypical modifications and do not account for attacks that skirt logging (e.g., logging was disabled). Interestingly, even privileged users have little control of how the low level (internal) DBMS storage behaves – therefore, an analysis of forensic artifacts provides a unique approach to identify tampering in a compromised environment.

The rest of the paper is organized as follows: Section II further motivates the pronounced need for developing standalone audit tools that can independently verify DBMS behavior. Section III provides an overview of our prior work on database forensic and storage analysis. Finally, Section IV presents two attack vector categories and countermeasures we developed, and further argues for the need of a more organized and comprehensive approach to combat malicious behavior within a DBMS.

II. The Need for Independent Database Audit Tools

The effort to ensure cybersecurity has increased substantially over the years – and a great deal of attention has been directed towards network defense, intrusion detection systems, and malware counteraction. Surprisingly, far less attention has been paid to detecting and preventing security vulnerabilities and forensic analysis mechanisms in database systems that actually store data being guarded. One of the reasons for such discrepancy is due to DBMSes providing an extensive support for their own built-in access control and security audit tools. In order to deliver desirable features such as ability to recover from failure, data integrity validation, and a common language (SQL) DBMSes take over all aspects of data storage and management within the OS. However, for the many features they provide, DBMSes ultimately operate as a black-box and, by design, do not provide insight as to the current state of the data and potential breaches or current activity.

The important question is, therefore, what happens when the DBMS itself is compromised? When a built-in security component is disabled, or when data is accessed or altered without a trace in audit logs, the DBMS is incapable of detecting or reporting such an attack. In every other context, when dealing with significant amount of valuable or sensitive data, tools and well-defined systematic approaches for performing an external audit are already required. Nonetheless, a systematic and generalized (i.e., open source tools, even for closed-source databases) auditing and forensic DBMS tools are yet to be developed.

We have developed a generalized approach (currently supported across row-store relational DBMSes) to database forensic analysis [14]–[16], and several applications to security breach detection [12], [13] for both DBA and SysAdmin attack threats. Our current focus is on transitioning this research into tools that can be used by forensic and security analysts, and in developing a comprehensive audit framework for DBMSes by combining our prior work, detecting other types of security breaches and malicious access, and developing anti- and anti-anti-forensics techniques for DBMSes.

III. Database Forensics

Unlike traditional files (e.g., PDF), DBMS files do not contain headers that allow for file identification. Instead, DBMS data is both accessed and cached in page units. All row-store DBMSes use fixed-size pages to store user data, auxiliary data (e.g., indexes and materialized views), and the system catalog. Pages maintain a consistent structure, whereas individual record structure varies throughout DBMS storage, which is why we approach database forensics at the page level. In this section, we briefly describe page carving including our implementation (`DBCarver`), planned future work to answer forensic questions from `DBCarver` output, and anti-forensics techniques that can sanitize and hide data in DBMS storage.

A. *Page Carving*

Database page carving is a method we previously introduced for the reconstruction of relational DBMSes without relying on file system or the DBMS itself. Page carving is similar to traditional file carving [9] in that data, including deleted data, can be reconstructed from images or RAM snapshots without the use of a live system. Forensic tools, such as Sleuth Kit [1] and EnCASE Forensic [2], are commonly used by investigators to reconstruct file system data but are incapable of parsing DBMS files. None of the third party recovery tools (e.g., [5], [8]) are helpful for independent audit purposes because (at best) they only recover "active" data from current tables. A database forensic tool (just like a forensic file system tool) should also reconstruct unallocated pieces of data including deleted rows, internal auxiliary structures (indexes, materialized views), or buffer cache space.

While each DBMS uses its own page layout, a great deal of overlap between page layouts allowed us to generalize storage for most row-store DBMSes. In our prior work [14] we presented a comparative page structure study for IBM DB2, Oracle, MS SQL Server, PostgreSQL, MySQL, SQLite, Firebird, and Apache Derby. We also described a set of parameters to generally define page layout for the purpose of reconstruction.

Deleted Data. When data is deleted, the DBMS initially marks it as deleted, rather than explicitly overwriting it. This data becomes unallocated (free listed) storage – in [15] we described the expected lifetime of forensic evidence within database storage following deletion and defragmentation. We described three categories of deleted data: records, pages, and values. A deleted record can be attributed to a `DELETE`, an old version of an `UPDATE`, or aborted transactions. Deleted records are identified by delete marking during page reconstruction. Dropped or rebuilt objects create deleted pages, which are identified by carving system catalog tables. Values from deleted records are found in auxiliary objects – e.g., indexes; they are identified by mapping pointers back to records (only records but *not* index values are deleted). We presented generalized pointer deconstruction and pointer-record mapping algorithms in [13].

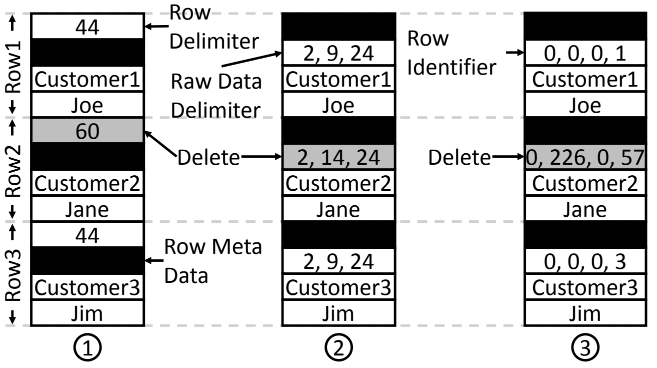

Fig. 1. Deleted row examples: 1-MySQL/Oracle, 2-PostgreSQL and 3-SQLite

Figure 1 visualizes an example of deleted records for several DBMSes. In all three pages, Row2-(*Customer2, Jane*) is deleted while Row1-(*Customer1, Joe*) and Row2-(*Customer3, Jim*) are active. Page#1 shows a case when the row delimiter is marked, such as in MySQL or Oracle. Page#2 shows when the raw data delimiter is marked in PostgreSQL. Page#3 shows when the row identifier is marked in SQLite. We omit DB2 and SQL Server as they only alter the row directory on deletion.

Column-Store and NoSQL DBMSes. Currently, our page carving approach only supports row-store DBMSes. In our future work, we intend to expand our database forensic methods to column-store (e.g., Vertica) and NoSQL (e.g., MongoDB) DBMSes.

B. *DBCarver*

We previously presented our implementation of page carving called `DBCarver` [16]. Figure 2 provides an overview of `DBCarver` architecture, which consists of two main components: the parameter collector (A) and the carver (F).

The parameter detector loads synthetic data into a DBMS (B), captures storage (C), finds pages in storage, and captures page layout parameters in a configuration file (E) – a text file describing page-level layout for that particular DBMS. Parameters include those described in [14], and have since been expanded to support other metadata. `DBCarver` automatically generates parameters values for new DBMSes, or new DBMS versions. While most DBMSes retain the same page layout across versions, we observed different parameter values between PostgreSQL versions 7.3 and 8.4.

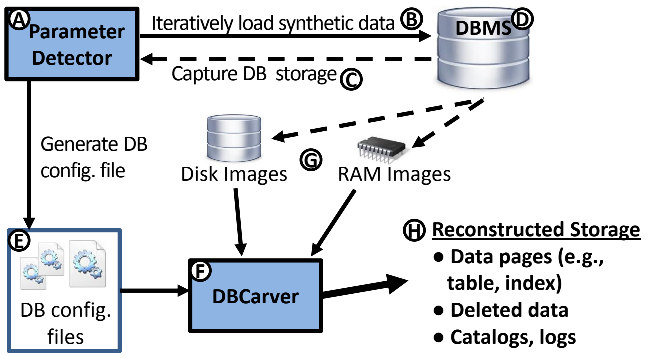

Fig. 2. DBCarver architecture.

The carver (F) uses the configuration files to reconstruct any database content from disk images, RAM snapshots, or any other input file (G). The carver returns storage artifacts (H), such as user records, metadata describing user data, deleted data, and system catalogs.

C. Meta-Querying

After storage artifacts are extracted by `DBCarver`, they must be analyzed to determine their evidential significance. By connecting reconstructed metadata and data, investigators can ask questions such as "Return all deleted records within a file." No such command is supported by DBMSes because deleted rows cease to exist from DBMS' perspective. More complex questions can be answered by combining disk and RAM data. We are interviewing forensic analysts to build a forensic querying system for real-world scenarios.

In [16] we offered a preliminary view of this system and presented several scenarios which an investigator may wish to explore. We consider forensic queries to be "meta-queries" because such queries are not executed on the original active DBMS but rather on reconstructed DBMS internals obtained through page carving, which include table data, deleted records, buffer cache contents, and internal data and metadata stored by the DBMS.

Scenario 1: Reconstruction of Deleted Data. An analyst wants to query the carved database storage for deleted values. Deleted row identification is of particular interest when the audit log is missing or altered. For example, the following logged query obfuscates what records were actually deleted:

```
DELETE FROM Customer
WHERE Name LIKE NameFunction()
```

Through database carving analysis, deleted records can be trivially identified with the following query:

```
SELECT * FROM Customer_Carved
WHERE RowStatus = 'DELETED'
```

Note that in order to determine whether extracted rows were deleted due to normal or malicious operations, we have later incorporated audit logs and other evidence sources into the query.

Scenario 2: Detecting Updated Data. An investigator wants to find the most recent updates. For example, consider the problem of searching for all recent product price changes in RAM. In order to form this query, we join disk and memory storage, returning the rows for which price is different:

```
SELECT * FROM RAM_Prod_Carv AS M,
              Prod_Carv AS D
WHERE M.PID=D.PID AND
      M.Price <> D.Price
```

D. Anti-Forensics

Anti-forensics (AF) is the field of interfering with forensic techniques [3]. We note that digital forensic tools can be used by either investigators and criminals, to both protect data and to interfere with a criminal investigation. In this section, we discuss future work that uses AF to protect data.

Two of the most representative AF techniques we considered are data wiping and steganography. A corporation can apply data wiping to erase already-deleted customer information to prevent potential data theft. Steganography a data hiding technique – e.g., a means to discretely blow a whistle on company's wrongdoing. Most prior work in database AF is highly DBMS-specific and limited in its ability to hide messages. Stahlberg erased deleted MySQL data by modifying the purge thread in source code [11]. We propose a more generalized sanitization method for all DBMSes (including closed-source DBMSes). We distinguish four categories of deleted DBMS data to wipe in order to prevent unintended data exposure: records, auxiliary data (e.g., indexes), system catalog, and unallocated pages. To effectively erase this data, the data itself must be overwritten and page metadata (e.g., checksums and pointers) must be updated accordingly. We further propose a steganography strategy that additively alters the database state through database file modification. This approach bypasses all constraints and logging mechanisms since the operation is performed without the DBMS.

a) Motivating Example: Alice is a spy who wants to send a message to Bob. If Alice is caught communicating with Bob, her identity as a spy will be compromised. If Alice uses cryptography to send her message to Bob, the message will be unreadable if intercepted. However, any evidence of communication with Bob is enough to prosecute Alice. Therefore, Alice uses steganography to hide the fact that she communicated with Bob.

b) Experimental Example: The details of how to alter DBMS storage are beyond the scope of this paper. We performed the changes in a chosen set of representative databases: PostgreSQL 9.6, MySQL 5.6, and Oracle 12c. Using a standard industry benchmark [6], we have manually added records to storage (bypassing all DBMS access controls, but requiring file system administrative privileges). By adding records that contained `NULL` as the primary key value (normally *not possible*), we have created "hidden" records which can be retrieved by targeted queries (e.g., `WHERE PrimaryKey IS NULL`) yet are naturally excluded from results of almost all regular user queries. Even an eventual rebuilding of the index is not going to interfere with our hidden records because `NULL` is always physically excluded from database indexes.

IV. Database Security

Privileged users (e.g., DBA), by definition, have the ability to control and modify access permissions. Therefore, audit logs alone are fundamentally unsuitable for the detection of malicious actions perpetrated by privileged users. DBMSes do not provide many tools to defend against insider threats. Interestingly, however, even DBAs have little to no control over how data is physically stored at the lowest level. Thus, malicious activity will still create inconsistencies within storage artifacts discovered by forensic analysis. In this section, we consider attack vectors that are detectable using database forensics methods from Section III. Some of these solutions (that rely on validating audit log integrity) assume that some level of logging was enabled and is available.

A. *DBDetective*

Audit logs are a critical piece of evidence for investigators – and existing research has explored tamper-proof logs. However, DBAs can disable logging for legitimate operations (e.g., bulk loads). Therefore, we consider an attack where logging was disabled, malicious activity was performed, and logging was re-enabled. We proposed `DBDetective` in our previous work [12] to detect activity missing from the logs.

To detect unlogged activity, `DBDetective` compares the disk images and/or RAM snapshots output from `DBCarver` against the audit logs. We classify two categories of hidden activity: record modifications and read-only queries (i.e., SQL `SELECT`). When a record is inserted or modified the record itself changes, page metadata may be updated (e.g., a delete mark is set) and index page(s) are likely to change. We flag any artifacts that cannot be explained by a log entry as suspicious, as shown in Figure 3.

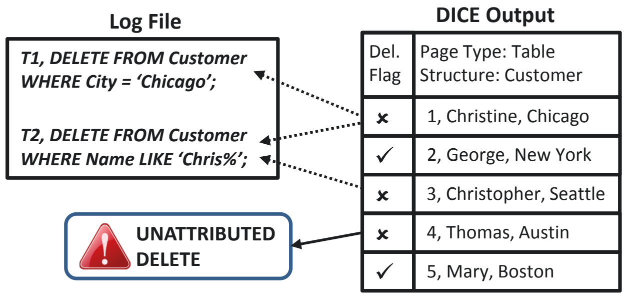

Fig. 3. Detecting unattributed deleted records

Figure 3 is an example of unaccounted, deleted row detection. `DBCarver` (aka DICE) reconstructed 3 deleted rows from *Customer*: *(1,Christine,Chicago)*, *(3,Christopher,Seattle)*, and *(4,Thomas,Austin)*. The log file contains two operations: `DELETE FROM Customer WHERE City = 'Chicago'` (*T1*) and `DELETE FROM Customer WHERE Name LIKE 'Chris%'` (*T2*). After comparing the deleted records to the log file operations, `DBDetective` returned *(4, Thomas, Austin)*, indicating a deleted record that could not be attributed to any of the logged deletes. Here, we cannot conclude whether *T1* or *T2* caused the deletion of *(1, Christine, Chicago)*, but that is not necessary to identify record #4 as an unattributed (i.e., unlogged) delete.

When a `SELECT` query reads a table from disk, it uses one of two fundamental access patterns: a full table scan or an index based access. Both of these query types produce a consistent, repeatable caching pattern. The same query-to-audit-log matching can be applied to monitor read-only query access by performing RAM scans. Using metadata from the pages in the buffer cache, we identify caching patterns and match them to the logged commands.

B. *DBStorageAuditor*

Privileged OS users commonly have access to database files. Consider a SysAdmin who, acting as the root, maliciously edits a DBMS file in a Hex editor or through Python. The DBMS is unaware of external file write activity taking place outside its own programmatic access and thus cannot log it. Such an attack is a 'black-hat' application of anti-forensics discussed in Section III-D. In our previous work [13], we proposed `DBStorageAuditor` to detect database file tampering.

To detect database file tampering, `DBStorageAuditor` [13] uses indexes to verify the integrity of table data. We first verify the integrity of the indexes by checking for tampering-based inconsistencies within the B-Tree structure. Once the index integrity is verified, we deconstruct the index pointers and match them to table records using the table page metadata; we generalized the deconstruction of index pointers for all major DBMSes. We organize the index pointers based on physical location to keep our matching approach scalable. Finally, any extraneous data or erased data found through index and table comparison is flagged as suspicious.

C. *Event Timeline Analysis*

Privileged users with access to the server OS have the capability to change server information, specifically the global clock. This can externally (i.e., outside of DBMS control) affect the veracity of audit logs. Consider a system administrator who changes the server global clock to an earlier date, performs a malicious activity, and resets the global clock. Such an attack backdates activity without altering the log files, and disguises the actual execution time of the malicious activity. As future work, we will detect such attempts to backdate log entries.

When a global system clock can not be assumed as reliable, we believe it is necessary to use storage metadata (which even a privileged user cannot modify) in order to create a timeline of events. The internal RowID pseudo-column is of particular interest to construct a timeline. RowID is used by indexes and reflects the physical location of a record including its PageID. Whenever a page is modified, we can store the PageID to know when data was modified. Thus, the order of the PageIDs must be consistent with the order of the log events. We are currently developing tamper-proof techniques to store and reconstruct the PageID.

D. *Quantitative Analysis and Reproducibility*

As future work, we will determine the detection accuracy for each attack described in this section. For each detection type, we will compute a confidence rating based on a variety of environment variables (e.g., buffer cache size, volume of operations, and DBMS storage engine). For example, given a low volume of `DELETE` operations in Oracle, `DBDetective` would detect attacks with higher accuracy because Oracle controls storage with a percent page utilization. This engine setting prevents deleted records from being overwritten until a page contains a significant quantity of deleted data.

To verify the presence of malicious operations, a repeatable analysis analysis must be guaranteed. We will develop algorithms to collect the minimal subset of storage artifacts needed to reproduce our results. These collected storage artifacts must be sufficient to verify the security breach independent of our analysis. For example, such functionality is needed to generate admissible evience that can be presented in court.

V. Conclusion

Despite existing internal controls and security mechanisms, the sheer volume of valuable data hosted within DBMSes demands access to forensic and investigative tools. In every other environment that deals with so much valuable or sensitive data, tools and systematic approaches for periodic external audits are mandated and available to investigators. Nonetheless, systematic auditing and forensic tools for databases have received surprisingly little attention in research literature thus far.

Acknowledgments

This work was partially funded by the United States National Science Foundation Grant CNF-1656268.

References

[1] B. Carrier. The sleuth kit. *TSK. http://www.sleuthkit.org/sleuthkit*, 2011.

[2] L. Garber. Encase: A case study in computer-forensic technology. *IEEE Computer Magazine January*, 2001.

[3] S. Garfinkel. Anti-forensics: Techniques, detection and countermeasures. In *2nd International Conference on i-Warfare and Security*, volume 20087, pages 77–84. Citeseer, 2007.

[4] R. T. Mercuri. On auditing audit trails. *Communications of the ACM*, 46(1):17–20, 2003.

[5] OfficeRecovery. Recovery for mysql. http://www.officerecovery.com/.

[6] P. O.Neil, E. O.Neil, X. Chen, and S. Revilak. The star schema benchmark and augmented fact table indexing. In *Performance evaluation and benchmarking*, pages 237–252. Springer, 2009.

[7] J. M. Peha. Electronic commerce with verifiable audit trails. In *Proceedings of ISOC*. Citeseer, 1999.

[8] Percona. Percona data recovery tool for innodb. https://launchpad.net/percona-data-recovery-tool-for-innodb.

[9] G. G. Richard III and V. Roussev. Scalpel: A frugal, high performance file carver. In *DFRWS*. Citeseer, 2005.

[10] R. T. Snodgrass et al. Tamper detection in audit logs. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 504–515. VLDB Endowment, 2004.

[11] P. Stahlberg, G. Miklau, and B. N. Levine. Threats to privacy in the forensic analysis of database systems. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 91–102. ACM, Citeseer, 2007.

[12] J. Wagner et al. Carving database storage to detect and trace security breaches. *Digital Investigation*, 22:S127–S136, 2017.

[13] J. Wagner et al. Detecting database file tampering through page carving. *21st International Conference on Extending Database Technology*, 2018.

[14] J. Wagner, A. Rasin, and J. Grier. Database forensic analysis through internal structure carving. *Digital Investigation*, 14:S106–S115, 2015.

[15] J. Wagner, A. Rasin, and J. Grier. Database image content explorer: Carving data that does not officially exist. *Digital Investigation*, 18:S97–S107, 2016.

[16] J. Wagner, A. Rasin, T. Malik, K. Hart, H. Jehle, and J. Grier. Database forensic analysis with dbcarver. In *CIDR*, 2017.