Security Issues in NoSQL Databases

Article · November 2011

DOI: 10.1109/TrustCom.2011.70

CITATIONS
61

READS
5,359

5 authors, including:

Nurit Gal-Oz
Ben-Gurion University of the Negev
25 PUBLICATIONS **246** CITATIONS
SEE PROFILE

Yaron Gonen
Ben-Gurion University of the Negev
8 PUBLICATIONS **81** CITATIONS
SEE PROFILE

2011 International Joint Conference of IEEE TrustCom-11/IEEE ICESS-11/FCST-11

Security Issues in NoSQL Databases

Lior Okman
Deutsche Telekom Laboratories
at Ben-Gurion University,
Beer-Sheva, Israel

Nurit Gal-Oz, Yaron Gonen, Ehud Gudes
Deutsche Telekom Laboratories
at Ben-Gurion University,
and Dept of Computer Science,
Ben-Gurion University,
Beer-Sheva, Israel

Jenny Abramov
Deutsche Telekom Laboratories
at Ben-Gurion University and
Dept of Information Systems Eng.
Ben-Gurion University,
Beer-Sheva, Israel

***Abstract*—The recent advance in cloud computing and distributed web applications has created the need to store large amount of data in distributed databases that provide high availability and scalability. In recent years, a growing number of companies have adopted various types of non-relational databases, commonly referred to as NoSQL databases, and as the applications they serve emerge, they gain extensive market interest. These new database systems are not relational by definition and therefore they do not support full SQL functionality. Moreover, as opposed to relational databases they trade consistency and security for performance and scalability. As increasingly sensitive data is being stored in NoSQL databases, security issues become growing concerns.**

This paper reviews two of the most popular NoSQL databases (Cassandra and MongoDB) and outlines their main security features and problems.

***Index Terms*—NoSQL; Security; Cassandra; MongoDB;**

I. Introduction

The recent advance in cloud computing and distributed web applications has created the need to store large amount of data in distributed databases that provide high availability and scalability. In recent years, a growing number of companies have adopted various types of non-relational databases, commonly referred to as NoSQL databases and as the applications they serve emerge, they gained extensive market interest. Different NoSQL databases take different approaches. Their primary advantage is that, unlike relational databases, they handle unstructured data such as documents, e-mail, multimedia and social media efficiently. The common features of NoSQL databases can be summarized as: high scalability and reliability, very simple data model, very simple (primitive) query language, lack of mechanism for handling and managing data consistency and integrity constraints maintenance(e.g., foreign keys), and almost no support for security at the database level.

The CAP theorem introduced by Eric Brewer [1], refers to the three properties of shared-data systems namely data consistency, system availability and tolerance to network partitions. The theorem [2] states that only two of these three properties can be simultaneously provided by the system. Traditional DBMS designers have prioritized the consistency and availability properties. The rise of large web applications and distributed data systems, makes the partition-tolerance property inevitable, thus imposing compromise on either consistency or availability.

The main promoters of NOSQL databases are Web 2.0 companies with huge, growing data and infrastructure needs such as Amazon and Google. The Dynamo technology developed at Amazon [3] and the Bigtable distributed storage system developed at Google [4], have inspired many of today's NoSQL applications.

In this paper we analyze the security problems of two of the most popular NoSQL databases, namely: Cassandra and MongoDB. Cassandra [5] is a distributed storage system for managing very large amounts of structured data spread out across many commodity servers, while providing highly available service with no single point of failure. Cassandra aims to run on top of an infrastructure of hundreds of nodes. At this scale, components fail often and Cassandra is designed to survive these failures. While in many ways Cassandra resembles a database and shares many design and implementation strategies therewith, Cassandra does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format. Cassandra was designed to support the Inbox search feature of Facebook [6]. As such it can support over 100 million users which use the system continuously.

MongoDB [7] is a document database developed by 10gen. It manages collections of JSON-like documents. Many applications can thus model data in a more natural way, as data can be nested in complex hierarchies and still be query-able and indexable. Documents are stored in collections, and collections are in turn stored in a database. A collection is similar to a table in relational DBMS, but a collection lacks any schema. MongoDB also provides high availability and scalability by using Shardings and Replica sets (see below).

The increasing popularity of NoSQL databases such as Cassandra and MongoDB and the large amounts of user-related sensitive information stored in these databases raise the concern for the confidentiality and privacy of the data and the security provided by these systems. In this paper we review the main security features and problems of these two database systems. We start with a brief overview of Cassandra and MongoDB functionality in section II. We then discuss security features of Cassandra and MongoDB in sections III and IV respectively. We conclude in section V. Since much of the discussion is based on open-source Internet documents, it naturally reflects the situation at the time this paper is written

978-0-7695-4600-1/11 $26.00 © 2011 IEEE
DOI 10.1109/TrustCom.2011.70

and may not include later updates to these systems.

II. Overview of Cassandra and MongoDB

A. Cassandra

Cassandra is a database management system designed to handle very large amounts of data spread out across many servers while providing a highly available service with no single point of failure. Cassandra provides a key-value store with tunable consistency. Keys map to values, which are grouped into column families. Columns are added only to specified keys, so different keys can have different numbers of columns in any given family. The values from a column family for each key are stored together. This makes Cassandra a hybrid data management system between a column-oriented DBMS (e.g., Bigtable [4]) and a row-oriented store.

Cassandra is shaped by two systems: Google's BigTable [4] and Amazon's Dynamo [3]. Both systems face the challenge of scaling, but they do it in different ways: BigTable uses the distributed file system Google already had, while Dynamo is based on a distributed hash table. Cassandra combines the data structure of BigTable, and the high availability of Dynamo.

The main features of Cassandra are summarized as follows.

1) *Symmetric* Cassandra is meant to run on a cluster of nodes, although it can run on a single machine as well. All the nodes in the cluster are functionally identical (meaning that the software resides in each node is identical but not the data), so there is no coordinator node, no manager node etc. The symmetry feature ensures no single point of failure, linear scalability and easy administration.
2) *Consistent Hashing (Distributed Hash Tables)* Consistent hashing [8] is a scheme that provides hash table functionality in a way that the addition or removal of one slot does not significantly change the mapping of keys to slots. (In contrast, in most traditional hash tables, a change in the number of array slots causes nearly all keys to be remapped.) Consistent hashing is utilized to address three requirements. The first is to find a primary, or a secondary server for a resource given the resource key. The second requirement is to assign work to servers in accordance to their capacity, and finally, to add capacity to the system smoothly, without downtime.
3) *Flexible Partitioning and High Availability* Both placement of the data and placement of replica is highly flexible. By placement, we mean the canonical location of where data goes. Growing a Cassandra cluster is easy since Cassandra's replica policy makes the data stored highly available. Cassandra presents two standard strategies for data replica placement: (1) in *Rack Unaware* Strategy: replicas are always placed on the next (in increasing Token order) nodes along the ring. (2) in *Rack Aware* Strategy: a replica is placed in the first node along the ring that belongs in another data center than the first; the remaining replicas, if any, are placed on the first nodes along the ring in the same rack as the first.
4) *The Client interface* Cassandra utilizes the Apache Thrift [9] framework to provide a cross language RPC based client interface. Thrift is not intended to be used directly by developers and applications, since the RPC interface it provides is not easy to use. Instead, developers are encouraged to utilize higher-level clients based on the Thrift interface that simplify the syntax required to access the database, and also add enterprise level functionality. For example, Hector [10], a high-level client library written in Java, encapsulates the entire raw Thrift API, and adds (among other features) connection pooling, automatic fail over, JMX (Java Management Extensions) support, type-safety and load-balancing requests across cluster members.
5) *Data Model* The Cassandra data model consists of the following main concepts:
   - Keyspace. A keyspace is the first dimension of the Cassandra hash, and is the container for column families. Keyspaces are of roughly the same granularity as a schema or database (i.e. a logical collection of tables) in the RDBMS world.
   - ColumnFamilies. A column family (CF) is a container for columns, analogous to the table in a relational system. A column family holds an ordered list of columns, that can be referenced by the column name.
   - Row. A row is a collection of columns or super-columns identified by a key. Each column family is stored in a separate file, and the file is sorted in row (i.e. key) major order. The row key determines which node the data is stored on. Related columns, that are accessed together, should be kept within the same column family.
   - Column. The column is the smallest increment of data. It's a triplet that contains a name, a value and a timestamp. In the reminder of this paper timestamp will be elided for readability.
   - SuperColumns. Super Column is a column whose values are columns, that is, a (sorted) associative array of columns.

   Following is an example of a JSON representation of a key $\rightarrow$ column families $\rightarrow$ column structure:

```
{
 "mccv":{
    "Users":{
       "email":{
        "name":"email",
          "value":"foo@bar.com"
       },
       "webSite":{
        "name":"webSite",
        "value":"http://bar.com"
       }
    },
    "Stats":{
```

```
        "visits":{
         "name":"visits",
         "value":"243"
        }
    }
 },
 "user2":{
    "Users":{
        "email":{
         "name":"email",
         "value":"user2@bar.com"
        },
        "twitter":{
         "name":"twitter",
         "value":"user2"
        }
    }
 }
}
```

In this example, the key "mccv" identifies data in two different column families, "Users" and "Stats". Notice that this does not imply that data from these column families is related. The semantics of having data for the same key in two different column families is entirely up to the application. Also note that within the "Users" column family, "mccv" and "user2" have different column names defined. This is perfectly valid in Cassandra. In fact there may be a virtually unlimited set of column names defined, which leads to fairly common use of the column name as a piece of runtime populated data.

6) *Partitioning and Placement* Partitioning is the policy of rows location, meaning which key resides on which node. The two major policies are *random partitioning* (RP) and *order-preserving partitioning* (OPP). OPP has one obvious advantage over RP: it provides the ability to perform range queries. However, using OPP may cause a load-balancing problem.
   With both RP and OPP, by default Cassandra will tend to evenly distribute individual keys and their corresponding rows over the nodes in the cluster. The default algorithm is that every time one adds a new node, it will assign a range of keys to that node such that it takes responsibility for half the keys stored on the node that currently stores most of the keys.
7) *Consistency* Cassandra is an *eventually consistent* data store, meaning that at any point in time each node might not be entirely up-to-date, but eventually all nodes are updated to the latest data values. The level of consistency can be chosen and indicates the number of replica blocked to execute an action. The most usable level is *quorum*. Quorum level guarantees that half of the records are updated before the action returns. Another option is the Write all Read one model in which case every read will be consistent.
8) *Actions: Write and Read* The client sends a write request to a single, random Cassandra node. This node acts as a proxy and writes the data to the cluster. The cluster of nodes is stored as a ring of nodes and writes are replicated to N nodes using a replication placement strategy. Each of those N nodes gets that write request and performs two actions for this message: (1) Append the data change to the commit log. (2) Update an in-memory Memtable structure with the change. These two actions are the only actions that are performed synchronously. This is the reason why Cassandra is so fast for write actions: the slowest part is appending to a file. Unlike a database, Cassandra does not update data in-place on disk, nor update indices, so there are no intensive *synchronous* disk operations to block the write. Reads are similar to writes in that the client makes a read request to a single random node in the Cassandra cluster (aka the Storage Proxy). The proxy determines the nodes in the ring (based on the replica placement strategy) that hold the copies of the data to be read and makes a read request to the appropriate nodes.

B. MongoDB

MongoDB (from "humongous") is a schema-free, document-oriented database written in the C++ programming language. The database is document-oriented in that it manages collections of schema-less JSON-like documents. This allows data to be nested in complex hierarchies and still be query-able and index-able. Following are the main MongoDB features:

1) *Data Model* A MongoDB database holds a set of *collections*. A collection is an equivalent term for a table, but unlike a table, a collection has no pre-defined schema. A collection holds a set of *documents*. A document is a set of fields. It can be thought of as a row in a collection. A document can contain complex structures such as lists, or even documents. Every document has an id.
2) *API* MongoDB has its own query language named *Mongo Query Language*. To retrieve certain documents from a db collection, a query document is created containing the fields that the desired documents should match. For example,
   `{name: {first: 'John', last: 'Doe'}}`
   MongoDB uses a RESTful API. REST (Representational State Transfer)is an architecture style for designing networked applications. It relies on a stateless, client-server, cacheable communications protocol (e.g., the HTTP protocol). RESTful applications use HTTP requests to post, read data and delete data.
3) *Architecture* A MongoDB cluster is different from a Cassandra cluster. The most obvious difference is the lack of symmetry: not every node in a MongoDB cluster is the same.
   A MongoDB cluster is built of one or more *shards*, where each shard holds a portion of the total data (managed automatically). Reads and writes actions are

automatically routed to the appropriate shard(s). Each shard is backed by a replica set - which just holds the data for that shard. A replica set is one or more servers, each holding copies of the same data. At any given time one server is primary and the rest are secondary servers. If the primary server goes down one of the secondary servers takes over automatically as primary. All writes and consistent reads go to the primary server, and all eventually consistent reads are distributed amongst all the secondary servers.

The cluster contains a group of servers called *configuration servers*. Each one holds a copy of the meta-data indicating which data lives on which shard.

Another group of servers is *routers*, each one acts as a server for one or more clients. Clients issue queries/updates to a router and the router routes them to the appropriate shard while consulting the configuration servers. Fig 1 copied from the official website of MongoDB [1], depicts the MongoDB architecture.

MongoDB supports two types of replication functionality: Master-Slave replication and Replica-set replication. In both types of replication, all write operations are performed against a single server (Master or Primary). Replica-Sets provide better flexibility, allowing automatic primary promotion (if enough of the secondary servers are available), automatic fail-over and better support for rolling upgrades. Both techniques provide data-redundancy and read-scaling (where data can be read from any of the servers in the cluster), however, in Master-Slave configuration, if a slave lags too far behind from the master, then the administrator needs to manually fix this, usually by rebuilding the slave instance.

The last piece of the MongoDB puzzle is its ability to automatically shard the data between multiple hosts. This effectively allows Mongo to scale horizontally to thousands of servers. When sharding is combined with replica-sets, the end-result is a highly scalable, redundant cluster, with no single point of failure.

4) *Sharding* MongoDB supports an automated sharding/partitioning architecture, enabling horizontal scaling across multiple nodes. For applications that outgrow the resources of a single database server, MongoDB can convert to a sharded cluster, automatically managing fail over and balancing of nodes, with few or no changes to the original application code. Sharding is the partitioning of data among multiple machines in an order-preserving manner. This makes it easier to support range queries and indexes which are inherent in MongoDB.

III. Cassandra Security features

As was stated in the introduction, security was not a primary concern of Cassandra's designers. As a result there are quite a few "holes" in its design. We review Cassandra security along several dimensions, and in each we outline briefly the main problems, and at the end we give some recommendations.

[1] http://www.mongodb.org/display/DOCS/Sharding+Introduction

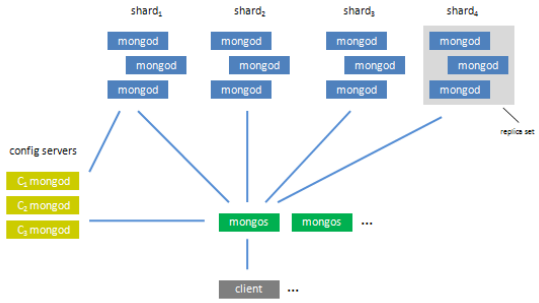

Fig. 1. MongoDB Architecture

1) *Cassandra Data Files* The data in Cassandra is kept unencrypted and Cassandra does not provide a mechanism to automatically encrypt the data in storage. This means that any attacker with access to the file-system can directly extract the information from the files. In order to mitigate this, the application must explicitly encrypt any sensitive information before writing it to the database. In addition, operating-system level mechanisms (file system permissions, file system level encryption, etc.) should be used to prevent access to the files by unauthorized users.
2) *Client interfaces* As mentioned above, Cassandra uses the Apache Thrift framework for client communications. The current stable branch of Cassandra (the 0.8.x branch) uses Thrift version 0.6. This version provides an SSL transport, however this transport is currently not being used. Consequently, all communication between the database and its clients is unencrypted. An attacker that is capable of monitoring the database traffic will be able to see all of the data as the clients see it. The client interface supports a `login()` operation, but both username and password are sent across the network as clear text.
3) *Inter-cluster communication* In general, nodes on a cluster can communicate freely and no encryption or authentication is used. As for inter-cluster communication, the current stable 0.8 branch can support requiring encryption on inter-cluster communication. This should be enabled, and client-certificates should be configured for all cluster members. An interface called *fat client* is used to load bulk data into Cassandra from different nodes in the cluster. No protection or authentication is provided for fat clients at all.
4) *Cassandra Query Language* In the latest stable version of Cassandra (0.8.0), there is a new interface to Cassandra called CQL (Cassandra Query Language). CQL is an SQL-like query language, compatible with the JDBC API. CQL syntax is similar enough to SQL that any developer that is familiar with SQL should be at home

with CQL. The CQL select statement has the following structure:

```
SELECT
  [FIRST N]
  [REVERSED]
  <SELECT EXPR>
FROM <COLUMN FAMILY>
  [USING <CONSISTENCY>]
[WHERE <CLAUSE>] [LIMIT N];
```

Since CQL is a parsed language, it is vulnerable to injection attacks, exactly like SQL. Although the JDBC driver provided with CQL implements the *PreparedStatement* interface, the parameters set via the various setters are actually escaped and substituted into the CQL query as it is being built, rather than being transmitted in a separate location in the underlying Thrift packet.

Since CQL is supposed to eventually replace the Thrift based clients, the CQL API should be carefully checked.

5) *Denial of Service problem* Cassandra uses a Thread-Per-Client model in its network code. Since setting up a connection requires the Cassandra server to start a new thread on each connection (in addition to the TCP overhead incurred by the network), the Cassandra project recommends utilizing some sort of connection pooling. An attacker can prevent the Cassandra server from accepting new client connections by causing the Cassandra server to allocate all its resources to fake connection attempts. The only pieces of information required by an attacker are the IP addresses of the cluster members, and this information can be obtained by passively sniffing the network. The current implementation doesn't timeout inactive connections, so any connection that is opened without actually passing data consumes a thread and a file-descriptor that are never released.
6) *Authentication* Cassandra provides an `IAuthenticate` interface, and two example implementations. The default implementation is one that turns off the requirement to authenticate to the database. The other provided implementation is `SimpleAuthenticator` class that allows setting up users and passwords via a flat Java properties file. The file is formatted as sets of password properties, and the password can be specified either in plain text, or as an unsalted MD5 hash [11]. It is important to note that the client interface as defined in Thrift will always transmit the password as plain-text, even if it is kept as an MD5 hash in the password file. This means that any attacker that is capable of sniffing the communication between a legitimate client and the database will be able to trivially discover the password. Additionally, the choice of an MD5 hash without utilizing any salt is unfortunate, since MD5 is a hashing algorithm that is not cryptographically secure, and it is easy to find appropriate plain-text to match a given MD5 hash using pre-calculated lists and rainbow tables found online.

TABLE I
SECURITY FEATURES IN CASSANDRA

Category	Status	Recommendation
Data at rest	Unencrypted	Protect with OS level mechanisms.
Authentication	The available solution isn't production ready.	Implement a custom `IAuthentication` provider.
Authorization	Done at the CF granularity level. The available solution isn't of production quality.	Implement a custom `IAuthority` provider.
Auditing	Not available OOTB	Implement as part of the authentication and authorization solutions.
Intercluster Network communication	Encryption is available	Enable this using a private CA.
Client communication	No encryption is available.	Add packet-filter rules to prevent unknown hosts from connection. Re-implement the Thrift server-side to use the SSL transport in Thrift 0.6. Add timeouts for silent connections in the Thrift server side, and cap the number of acceptable client connection.
Injection attacks	Possible in CQL.	If using the Java driver, prefer `PreparedStatements` to `Statements`. Always perform input validation in the application.

Another issue with the `SimpleAuthenticator` implementation is that the administrator must make sure that the flat password file found on each cluster member is synchronized with the rest of the cluster. Otherwise, the passwords used by each cluster member may not be the same for the same user.

7) *Authorization* Cassandra provides an `IAuthority` interface which is used in the Cassandra's codebase when a keyspace is being modified, and on each column family access (read or write). The current set of permissions is an enum containing `READ` and `WRITE` permissions. The `IAuthority` interface provides a single method returning a set of permissions for a provided authenticated user and hierarchical list of resource names. Similar to the authorization code, Cassandra provides two implementations of `IAuthority`. The first is a pass-through implementation that always allows full permissions, regardless of the user, and the second uses a flat Java properties file to allow matching permissions to usernames. The weakness in the `SimpleAuthority` implementation is that it depends on a flat file, and not on a consistent file that is maintained across the cluster. This means that the effective permissions granted to a user are the permissions listed in the file located on the cluster member to which the connection was established. Another issue with the provided SimpleAuthority implementation is that it does not reload the file on every access. This means that the effective

permissions `cannot` be changed without restarting the Cassandra process. Finally, it is important to note that Authorization in Cassandra is specified only on existing column families, and therefore no protection is available on newly added column families and columns, and also protection at the raw(object) level is not available.

8) *Auditing* Cassandra doesn't support inline auditing. However, if auditing is required, then a custom implementation of `IAuthority` can be written to provide full auditing of all operations that require authorization, and a custom implementation of `IAuthenticate` can be written to provide a full audit trail for all login success/failure occurrences.

Table I summarizes our finding on Cassandra security and gives some recommendations.

IV. MongoDB Security features

As was stated in the introduction, security was not a primary concern of MongoDB's designers. As a result there are quite a few "holes" in its design. We review MongoDB security along several dimensions, and in each we outline briefly the main problems, and at the end we give some recommendations.

1) *MongoDB Data Files* Mongo data-files are unencrypted, and Mongo doesn't provide a method to automatically encrypt these files. This means that any attacker with access to the file system can directly extract the information from the files. In order to mitigate this, the application must explicitly encrypt any sensitive information before writing it to the database. In addition, operating-system level mechanisms (file system permissions, file system level encryption, etc.) should be used to prevent access to the files by unauthorized users.
2) *Client Interfaces* Mongo supports a binary wire-level protocol, using TCP port 27017 by default. This protocol is used by all of the various drivers, and is the most efficient way to communicate with Mongo. In addition to the application drivers, the Mongo database uses this port and protocol in order to perform replication (both variants). Additionally, the port number that is 1000 more than the binary client port is used as a HTTP server (TCP port 28017 by default). This HTTP server provides some management level statistics, but can also be configured to provide a RESTful interface to the database, by adding rest=true to the database configuration file, or by using the command-line. The binary wire-level protocol is neither encrypted nor compressed, and the internal HTTP server doesn't support TLS or SSL in any way. The internal HTTP server however can be hidden behind a HTTP proxy server, like Apache HTTPD with the mod_proxy module (as a reverse proxy), and then the Apache HTTPD's robust authentication and authorization support can be used, in addition to robust SSL encryption for the connection.
3) *Potential for injection attacks* Mongo heavily utilizes JavaScript as an internal scripting language. Most of the internal commands available to the developer are actually short javascript scripts. It is even possible to store javascript functions in the database in the db.system.js collection that are made available to the database users. Because JavaScript is an interpreted language, there is a potential for injection attacks. For example, the following statements can all be used in order to perform the same equivalent query against the database with a where clause:

```
db.myCollection.find(
  { a : { $gt: 3 } }
);
db.myCollection.find(
  { $where: "this.a > 3" }
);
db.myCollection.find(
  "this.a > 3"
);
db.myCollection.find(
  { $where: function() {
    return this.a > 3;}
  }
);
```

In the second and third statements the where clause is passed as a string that might contain values that were concatenated directly with values passed from the user. In the fourth statement, the $where object is a JavaScript function that is evaluated per each record in the myCollection collection. This cannot be used to modify the database directly, since the $where function is executed with the database in a read-only context, however if an application uses this type of where clause without properly sanitizing the user input, then an injection attack should work against this form of where clause as well.

4) *Authentication* When running in Sharded mode, Mongo doesn't support authentication. However, when running in standalone or replica-set mode, authentication can be enabled in Mongo. The main difference between standalone and replica-set modes is that in replica-set mode, in addition to the clients authenticating to the database, each replica server must authenticate to the other servers before joining the cluster. In both modes, the authentication is based on a pre-shared secret. For replica-set mode server-to-server authentication, the pre-shared secret is provided using the keyfile parameter in the configuration file, and it is up to the administrator to verify that all of the servers in the replica-set cluster contain the same password. For client connections, the user must be configured by connecting to the relevant database (in a replica-set database, the MASTER server must be updated), and calling the db.addUser() method. A user that is added to a special database called admin is considered a DBA user with special administration privileges. The password itself is kept as a MD5 hash

of the string $< username >: mongo :< password >$, and is easily read from the admin data-files, which means that any attacker with access to the data files can easily recover the passwords for all the users defined in the database. When authenticating via the wire-level protocol, all passwords are passed as a nonce-based md5 hash, albeit in the clear. Since the plaintext being hashed contains a nonce, the password is not repeatable, however the MD5 algorithm is not considered a very secure algorithm, so it is conceivable that an attacker can still run a brute force attack via the wire-level protocol in a fairly efficient manner. If the application is using the RESTful API, then the Mongo internal HTTP server can be hidden behind a reverse proxy, and the reverse proxy can be used to provide both authentication and authorization in a very fine-grained manner.

5) *Authorization* When running in Sharded mode, Mongo doesn't support authentication, and therefore has no support for authorization. If authentication has been enabled, then the Mongo databases supports two types of users: read-only and read-write. Read-only users can query everything in the database on which they are defined, while read-write users have full access to all the data in the database on which they are defined. Any user defined on the admin database is considered a DBA user. Any user defined on the admin database has full read-write access to all of the databases defined in the cluster. While Mongo doesn't support this directly, if the RESTful API is being used behind a reverse proxy, then fine grained permissions can be defined on the proxy itself, in addition to stronger authentication.
   A reverse proxy could add distinct permissions for the test namespace and the data namespace.
6) *Auditing* MongoDB doesn't provide any facilities for auditing actions performed in the database. When a new namespace (database) is created, Mongo will write a line in the log about data file creation, but after the data files are allocated, nothing new appears in the log for any subsequent insertions, updates or queries.

Table II summarizes our finding on MongoDB security and gives some recommendations.

TABLE II
SECURITY FEATURES IN MONGODB

Category	Status	Recommendation
Data at rest	Unencrypted	Protect with OS level mechanisms.
Authentication for native connections	Available only in unsharded configurations.	Enable if possible.
Authorization for native connections	READ/READ-WRITE/Admin levels, only in unsharded configurations.	Enable if possible, requires enabled authentication.
Auditing	Not available in MongoDB	
AAA (authentication, authorization auditing) for RESTful connections	Users and permissions are maintained externally.	Available if configured on a reverse proxy
Database Communication	Encryption is not available	
Injection attacks	Possible, via JavaScript or string concatenation.	Verify that the application does reasonable input validation.

V. CONCLUSIONS

We reviewed the main functionality and security features of two of the most popular NoSQL databases: Cassandra and MongoDB. The main problems common to both systems include lack of encryption support for the data files, weak authentication both between the client and the servers and between server members, very simple authorization without support for RBAC or fine-grained authorization, and vulnerability to SQL injection and Denial of Service attacks.

Clearly the future generations of such DBMSs need considerable development and hardening in order to provide secure environment for sensitive data which is being stored by applications (such as social networks) using them.

ACKNOWLEDGMENT

The authors would like to thank the Duetsche Telekom team in Berlin especially Karl Thier and Tobias Martin for their important input.

REFERENCES

[1] E. Brewer. (2000, Jun.) Towards robust distributed systems. [Online]. Available: http://www.cs.berkeley.edu/ brewer/cs262b-2004/PODC-keynote.pdf
[2] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, pp. 51–59, June 2002. [Online]. Available: http://doi.acm.org/10.1145/564585.564601
[3] G. Decandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-Value Store," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, Stevenson, WA, Oct. 2007.
[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, pp. 4:1–4:26, June 2008. [Online]. Available: http://doi.acm.org/10.1145/1365815.1365816
[5] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, pp. 35–40, April 2010. [Online]. Available: http://doi.acm.org/10.1145/1773912.1773922
[6] Facebook. Facebook. [Online]. Available: http://www.facebook.com/
[7] MongoDB. Mongodb. [Online]. Available: http://www.mongodb.org/
[8] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web," in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, ser. STOC '97. New York, NY, USA: ACM, 1997, pp. 654–663. [Online]. Available: http://doi.acm.org/10.1145/258533.258660
[9] A. Agarwal, M. Slee, and M. Kwiatkowski, "Thrift: Scalable cross-language services implementation," Facebook, Tech. Rep., April 2007. [Online]. Available: http://incubator.apache.org/thrift/static/thrift-20070401.pdf
[10] R. Tavory. (2010, Feb.) Hector java source code. [Online]. Available: http://github.com/rantav/hector
[11] R. Rivest, "The md5 message-digest algorithm. RFC 1321," MIT Laboratory for Computer Science and RSA Data Security, Inc, Tech. Rep., April 1992. [Online]. Available: https://tools.ietf.org/html/rfc1321