Towards a NoSQL security map

Wilhelm Zugaj *wilhelm.zugaj@fh-joanneum.at*
FH JOANNEUM University of Applied Sciences
Kapfenberg, Austria

Anita Stefanie Beichler *anita.beichler@edu.fh-joanneum.at*
FH JOANNEUM University of Applied Sciences
Kapfenberg, Austria

Abstract

NoSQL solutions have recently been gaining significant attention because they address some of the inefficiencies of traditional database management systems. NoSQL databases offer features such as performant distributed architecture, flexibility and horizontal scaling. Despite these advantages, there is a vast quantity of NoSQL systems available, which differ greatly from each other. The resulting lack of standardization of security features leads to a questionable maturity in terms of security. What is therefore much needed is a systematic lab research of the availability and maturity of the implementation of the most common standard database security features in NoSQL systems, resulting in a NoSQL security map. This paper summarizes the first part of our research project trying to outline such a map. It documents the definition of the standard security features to be investigated as well as the security research and results for the most commonly used NoSQL systems.

Keywords: database security, NoSQL database systems, NoSQL security, database authentication, database authorization, database encryption

1. Introduction

Relational database management systems represent a mature technology for data management. They exist since the 1970s and are based on the solid scientific fundamentals of the relational data model developed by Edgar F. Codd. This maturity is expressed by the fact that apart from the core functionality, numerous complementary features are supported as part of the out of the box solutions of well-known commercial and open source database management systems.

The topic of security is an example for this situation. In fact, big commercial vendors provide some of their most sophisticated security solutions as add-ons to their standard products (e.g. Oracle Advanced Security). Yet it is possible to identify a range of out of the box security features, which might be available for any well-known relational database product.

NoSQL database management systems are designed to cope with the challenges of data management in the era of big data. They are specialized on horizontal scalability by providing partition tolerant distributed data processing. Thus, they have become the standard database technology for cloud applications [20]. According to the CAP theorem they have to trade these advantages for reduced consistency and ACID compliance [22]. NoSQL database management systems replace the traditional static relational database model for dynamic and flexible simple data models. This results in four major categories of such database systems: Document oriented stores, key value stores, wide column stores and graph databases [9].

NoSQL is a young technology compared to relational systems; its market is highly competitive and fragmented. There are over 225 different database implementations [8]. Thus,

it is obvious to expect that many vendors invest their resources into features they get significant attention for: performance and scalability. But what is the state of affair of out of the box security in the NoSQL world? It is our belief that an extensive map of the state of out of the box security of all major NoSQL systems is needed. This map must be created by practical lab research on availability and maturity of out of the box security features. Relying just on existing product documentation cannot replace profound security research.

For practical reasons, the concept of out of the box availability of security seems to us of special importance. We expect that developers are willing to make use of out of the box security when creating systems based on new database technology. But it is doubtful whether they are willing to invest extra time and money for additional security add-ons, especially if they have low security awareness. We apply the same argument also to users and organizations working with NoSQL systems.

Our research is organized as follows. Based on a literature review on the well-established field of standard database security we extract common security features of database security, whose availability and maturity we are going to investigate for NoSQL systems. We start our research with the most popular system from each of the four categories of NoSQL database systems as listed in [6]. Then we proceed by extending our security research to all additional systems listed in [6] as well as on further NoSQL technologies which are covered by relevant scientific publications. This paper discusses research and results of the first four systems we investigated, since research on further NoSQL database systems is still ongoing.

At the end of this introduction we provide details on the database security features investigated, and the four systems chosen. There exists a high number of valuable sources on database security. Exemplarily, we mention Ron Ben Natan who proclaims in [15] that database security must be implemented as part of a defense-in-depth strategy, to make sure that even if multiple layers are compromised, no significant damage will occur. He does not focus on a specific database brand, but rather provides a general view on the topic. Knox provides recommendations and best-practice solutions on Oracle security in [13]. He covers the entire security circle, from authentication and authorization to fine-grained access control and encryption. This publication is an excellent source from which universally applicable security ideas are derived from product-specific Oracle features. Hassan A. Afyouni, instructor at several universities, consultant, author, corporate trainer and database architect, describes database hardening and security in [1]. The importance of the features mentioned in these publications is underlined by the vulnerability list [27] released by OWASP (Open Web Application Security Project). From the pool of literature on database security we extracted user administration, authorization, authentication, password security, securing communication, encryption, auditing and log management to be common out of the box security features to research for in NoSQL database systems.

For the research conducted the following databases were selected: OrientDB 2.2.14 from the domain of graph databases, Redis 3.2 from the key-value databases, Cassandra 3.10. from the column-oriented databases and MongoDB 3.4.4. as an example of a document database. These systems represented the mostly used systems due to [6] at the time of our project start (this list is very volatile) except for OrientDB which we chose over Neo4j, because of its multi-model capabilities. Neo4J was selected to be covered in the next set of candidates for the next research step in our ongoing project.

2. Related work

In the search for scientific publications on NoSQL security for the four systems we have chosen to start our research with, we found a majority of publications to cover MongoDB and Cassandra. An IEEE-Explore search for “MongoDB” and “Security” revealed 837 hits (full text search)/ 24 hits (metadata search). The same search for “Cassandra” provided a similar amount of results, although not all papers addressed the NoSQL database system Cassandra, for example [4] introduces a role-based trust management system of the same name. Redis and OrientDB on the other hand provide less than 10 results each for the same query.

In the following, we provide an overview over those papers describing scientific work related to our research. [25] describes adding data encryption to MongoDB by introducing a transparent middleware as an add-on. However, our focus is on out of the box security. [23] provides remarks on out of the box MongoDB auditing and [11] briefly discusses missing security features after default installation. Concerning attacks, a NoSQL injection attack on MongoDB is studied in [12]. A detailed study on authentication, authorization, encryption and auditing of MongoDB can be found in [7], and a comparison of its security features with those of Oracle and MySQL is outlined by [24]. Both of these publications cover a good deal of standard out of the box security features, yet not all the features considered in our publication. Looking for research results for a fast-developing technology like NoSQL topicality has to be taken into consideration. Although [26] provides an interesting analysis of MongoDB security features, the version studied is long outdated.

Considering Cassandra, [2] describes attacks using malicious Cassandra nodes. [29] provides Redis security add-ons for authentication, encryption and data to persist, while [28] investigates attacks on data integrity of key value stores like Redis. An investigation of NoSQL security (in detail authentication, authorization, configuration, encryption and auditing) for Cassandra, Redis and MongoDB (and additionally CouchDB, HBASE and Couchbase) can be found in [30]. This investigation already dates back to 2014 and does not cover additional topics like server security. Additionally, a systematic list of complete results for all the features and database systems is missing. At least it is a valuable source for getting a quick impression of the security features of certain NoSQL systems.

To our best knowledge we have found no publication describing systematic lab research on a map of out of the box NoSQL security features. We also did not find any publication covering all of the four systems we started our research with combining them with the out of the box security feature set we based our research on.

3. Research and results

For each database implementation, a test setup was installed. Each implementation was examined hands-on for each of our selected security features. We investigated to which extent the features were available and if we could identify weaknesses. Then we compared our findings to the technologies documentation and informed the vendors in case of differences found. It is worth mentioning that all four vendors of the database systems examined give the security advice to use their solutions in trusted environments only.

As default configurations are a potential source of security issues, special attention was bestowed upon them. In this context, it is worth noting that we mention default configurations in the following only if they had proven to be a problem.

User administration

We tested the database systems against their capabilities to flexibly create and configure users, inherit from other users and manage them centrally. Redis turned out to offer no form of user management, its developers had even discontinued a project that had attempted to add this feature in 2014 [17].

Cassandra offers database roles that may represent a single user or a group of users for authentication and permission management. A client can identify using a role that has the login privilege upon connecting. Roles were introduced in Cassandra 2.2. Prior to that, authentication and authorization were based on the concept of a user. This means that creating a user is just another way of creating a role. The key difference is that roles can also be granted to each other. In this context we can think of them as groups, allowing related privileges to be bundled together by granting them to roles, which can in turn then be assigned to specific database users. The default super user should be disabled (revoking of the login permission), as it is a security risk. Figure 1 gives an example of roles in Cassandra with login option, superuser option and the salted password hash.

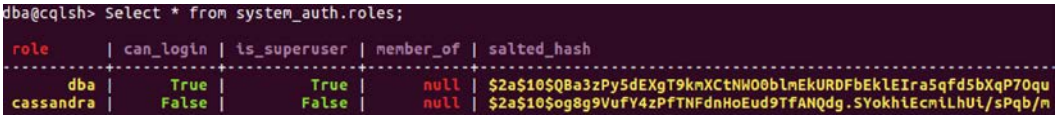

Fig. 1. Example of Cassandra roles (users) with option login and superuser and salted password hash

MongoDB offers a built-in database user and database administration roles are provided for each database. It can be distinguished between database user roles (read, readWrite), database administration roles (e.g. dbOwner), backup and restoration roles (e.g. readWriteAnyDatabase) and superuser roles. With superuser roles one needs to be careful as they provide direct or indirect system-wide superuser access (e.g dbOwner when scoped to the admin database). A user can be deactivated only by revoking permissions on resources.

During the setup of OrientDB, a set of default users is created in the configuration file. This is the only database which offers user management through the configuration file. It is strongly advised not to leave them in production because untrusted users could attempt to access the OrientDB server with the credentials of the default users. Roles can be assigned to users. Inheritance of permissions is possible. The default user types are: a server user, a database user and a system user. A server user can be configured in the configuration file and has permissions on server-related activities while a database user only has permissions and roles associated with that specific database. As mentioned above, three default users are created for each database: admin, reader and writer, with their default passwords being the admin, reader and writer. It is possible to activate and deactivate users. A system user is essentially a hybrid of a server user and a database user.

Authorization

Authorization determines whether an entity has the authority to access a resource. System and object privileges are coarse-grained security privileges while access to data tables is handled by fine-grained access control, also called row-level security. There is a commonly practiced security tenet that states that users can only access data which they have a “need to know”. Our research included testing the possibilities of assignment and inheritance of privileges as well as the handling of authorization in general. This contained using standard roles as available, creating custom roles, inheriting from roles, granting and revoking permissions and testing the limits of these permissions.

In Redis authorization is not implemented due to the drawback of the added complexity as stated in [19]. In a classical setup, this means that in addition to modifying all data, the client could also control the server configuration (e.g. changing the working directory or writing dump files at random paths) using the config command. To limit clients to a specific set of commands [18] suggest command-level security through obscurity by allowing an administrator to rename commands into unguessable names or disabling them by setting the name to a blank string. Nevertheless, this does not limit access to data.

Cassandra either allows any action to any user (default setting) or actions according to stored permissions. Permissions on database resources are granted to roles. Roles may be granted to other roles to create hierarchical permissions structures. In these hierarchies, permissions and superuser status are inherited. Contrary, the login privilege itself is not inheritable. Custom roles can also be created. It is recommended to disable the default superuser.

By default, authorization is not enabled in MongoDB but access to data and commands can be granted through role-based authorization. Built-in roles provide different levels of access to the database system. It is also possible to create user-defined roles. A role can include existing roles in its definition and inherits all the privileges of the included role. To add a user-defined role, the scope must be given, as inheritance is only possible from roles within this scope.

To restrict unauthorized users and possible attackers from giving themselves privileges on the OrientDB server or read configuration parameters, read and write access to the configuration file and the entire config directory should be disabled. Roles can inherit permissions from other roles, access rules for a role are predefined by OrientDB.

Authentication

Authentication is needed to securely identify a certain entity, which means it must provide proof to the server that it is who it is claiming to be. Security flaws resulting from default configurations were a topic already at the 2013 DEF CON, one of the world's largest hacker conventions. Ming Chow, a senior lecturer at Tufts University Department of Computer Science, described default values as easy prey for attackers. All they need to know is the database vendor, an IP address and an open port number [14].

The databases were tested according to their supported authentication methods. This is the only feature that is available in all reviewed solutions, yet it is disabled by default in all of them. Redis offers authentication via the auth command, yet we found that it sends the password unencrypted. Authentication must be enabled in the configuration file.

Authentication in Cassandra is configured within the configuration file using the authenticator setting. There are two options, the AllowAllAuthenticator (allows all connections, does not require authentication) and the PasswordAuthenticator (requires user credentials to allow a connection). To be able to use Cassandras permission system, authentication must be enabled.

Supported authentication mechanisms in MongoDB are SCRAM-SHA-1 (challenge-response authentication, per-user random salts, SHA-1 usage, client to server and server to client authentication), MONGODB-CR (verifies unencrypted user credentials against a user's name, password and the authentication database) and x.509 certificate authentication (requires secure TLS connection) [21].

In OrientDB one must be authenticated to a server instance to run certain commands like list databases or create database, while for other commands (e.g. create user) one must be authenticated to a specific database. Authentication is possible via username and password or certificates.

Password security

Passwords were tested according to where they are stored and in what form and if there are some built in functions to help creating secure passwords in the first place.

In Redis the password is set by the system administrator in clear text inside the unencrypted configuration file. Cassandra stores the password encrypted within a system table. The PasswordAuthenticator queries the table for the hashed password, a salt is not used.

In MongoDB passwords are created with a per-user random salt and a hash function (SHA-1). Alternatively, certificates can be used. Many applications still rely on SHA-1, even though it has been broken in practice [5]. Risk due to not changing the default superuser and password in the context of MongoDB were documented in [10]. In this particular incident, Niall Merrigan, a security researcher and Microsoft developer, used Shodan.io to pin down the number of MongoDB installations at risk and came up with a number close to 52,000 servers that are accessible from the internet without authentication [10].

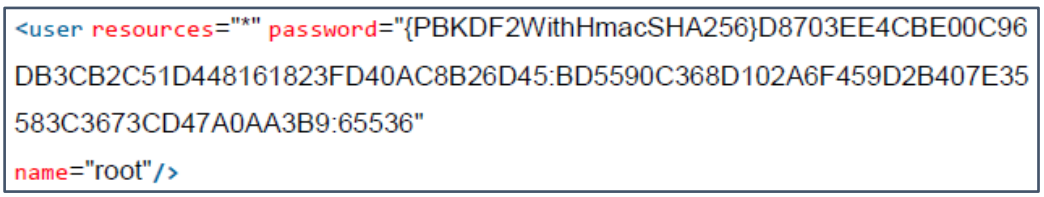

Fig. 2. Part of the OrientDB configuration file

In OrientDB the password for the server user is saved within the configuration file as a hash. The database user's password is saved as a hash within the OUser table. The used algorithm is PBKDF2, the number of iterations to generate the salt can be set as a parameter. PBKDF2 has a reported design flaw wherein the performance is lowered because in order to produce outputs of any size, PBKDF2 hashes each block of output all over [16].

Most RDBMSs offer more than one form of authentication, often including external authentication such as OS-based approaches (e.g. Windows authentication in Microsoft SQL

Server). Some other noteworthy options are Kerberos authentication, where information is exchanged over an open network by assigning a ticket (unique key) to a user; Lightweight Directory Access Protocol (LDAP), which uses a centralized directory database to store information about users in a hierarchical manner; Public Key Infrastructure (PKI), which uses a private and a public key for the authentication process; Transport Layer Security (TLS), which transmit the authentication information over the network in encrypted form; digital cards or smart cards, which require a card reader; and a device called a digital token, which provides a new pin as password for every authentication action [1]. Kerberos and LDAP functionalities are only offered in the enterprise editions of MongoDB and OrientDB.

Securing communication

The network poses a significant security issue, as it links together all clients and servers. Network security should provide data integrity and confidentiality, while also protecting data in transit from disruption. According to [13], network security can be segmented into encrypting data streams, providing integrity checks and limiting access to certain networks and servers to authorized persons. In terms of sever security authentication to the server instance, getting rid of default configurations, paying particular attention to configuration files and enabling logging was researched in our lab for the systems chosen.

Redis' default operating mode is the so called protected mode, which allows only connections from loopback, as it is supposed to be accessed by trusted clients in trusted environments only. Redis is optimized for performance and simplicity. But it is not for security, as Salvatore Sanfilippo - developer of Redis - states [19]. The Redis security model is: "... totally insecure to let untrusted clients access the system, please protect it from the outside world yourself. The reason is that, basically, 99.99% of the Redis use cases are inside a sandboxed environment. […] Adding security features adds complexity" [19]. At least authentication can be configured.

TLS can be enabled in all solutions, except Redis, to encrypt traffic between database servers and clients as well as between nodes within a cluster if one is using Cassandra. This will be discussed further for all solutions in the section "Encryption". Figure 3 shows the configuration of client encryption in Cassandra.

```
# enable or disable client/server encryption.
client_encryption_options:
    enabled: false
    # If enabled and optional is set to true encrypted and unencrypted connections
 are handled.
    optional: false
    keystore: conf/.keystore
    keystore_password: cassandra
    # require_client_auth: false
    # Set trustore and truststore_password if require_client_auth is true
    # truststore: conf/.truststore
    # truststore_password: cassandra
    # More advanced defaults below:
    # protocol: TLS
    # algorithm: SunX509
    # store_type: JKS
    # cipher_suites: [TLS_RSA_WITH_AES_128_CBC_SHA,TLS_RSA_WITH_AES_256_CBC_SHA,TL
S_DHE_RSA_WITH_AES_128_CBC_SHA,TLS_DHE_RSA_WITH_AES_256_CBC_SHA,TLS_ECDHE_RSA_WITH
_AES_128_CBC_SHA,TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA]
```

Fig. 3. Configuration of client encryption in Cassandra

OrientDB allows for security settings like disabling clickjacking, a malicious technique of tricking a web user into clicking on something different from what the user perceives they are clicking on, thus revealing confidential information or taking control of their computer. This can be done by setting the additional header X-FRAME-OPTIONS to DENY in all the HTTP responses. To enable it, one must set a couple of additional headers in orientdb-server-config.xml under the HTTP listener XML tag:

```
<listener protocol="http" ip-address="0.0.0.0" port-range="2480-2490"
socket="default">
          <parameters>
```

```
          <parameter    name="network.http.additionalResponseHeaders"
       value="X-FRAME-OPTIONS:DENY"/>
          </parameters>
</listener>
```

Encryption

Different types of data should be kept confidential and preferably encrypted. Even though an authorized person might access the data, sensitive data such as passwords should remain confidential. We did investigate if the databases offer any form of encryption at all. In doing so we focused on data-at-rest and data-in-transit.

Redis offers no data-in-transit encryption as well as no data-at-rest encryption. Redis Labs again puts a strong emphasis on the fact that Redis is supposed to be accessed by trusted clients inside trusted environments only.

Cassandra provides data-in-transit encryption between client machine and database cluster, as well as between nodes within a cluster. Supported protocols and cipher suites are configured in the configuration file, were the disabled encryption must be enabled first (figure 4). Cassandra additionally offers materialized views to secure individual rows and columns and to mask values and remove personally identifiable information. Data-at-rest encryption must be enabled and currently covers only two kind of files, “commitlog” (to avoid data loss and to keep recent changes in memory) and “hint” (if a node is down, writes missed are stored there for a period of time).

```
# enable or disable client/server encryption.
client_encryption_options:
    enabled: false
    # If enabled and optional is set to true encrypted and unencrypted connections
 are handled.
    optional: false
    keystore: conf/.keystore
    keystore_password: cassandra
    # require_client_auth: false
    # Set trustore and truststore_password if require_client_auth is true
    # truststore: conf/.truststore
    # truststore_password: cassandra
    # More advanced defaults below:
    # protocol: TLS
    # algorithm: SunX509
    # store_type: JKS
    # cipher_suites: [TLS_RSA_WITH_AES_128_CBC_SHA,TLS_RSA_WITH_AES_256_CBC_SHA,TL
S_DHE_RSA_WITH_AES_128_CBC_SHA,TLS_DHE_RSA_WITH_AES_256_CBC_SHA,TLS_ECDHE_RSA_WITH
_AES_128_CBC_SHA,TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA]
```

Fig. 4. Configuration of client encryption Cassandra

MongoDB encrypts data-in-transit via TLS, while data-at-rest encryption is not available in the community edition. It should not go unmentioned that MongoDB offers collection-level access control by creating a role that is specific to a collection in a particular database, meaning that a user’s privileges are limited to a specific collection. This is the finest-grained restriction that MongoDB offers, but there is also a way to further restrict access to documents. Field-level redaction restricts the contents of a document based on information stored in the document itself. Multiple access levels for the same data are enabled via an access field, best set to an array of arrays. Each array element contains a required set of tags that a user needs to have to be allowed to access the data.

OrientDB offers security of data-in-transit via TLS and security of data-at-rest through encryption with either AES or DES algorithm. The encryption key is not saved to the database but must be provided at run-time. It is possible to create an encrypted database through the Java API or the console. Using the encryption option together with the create database command is only possible after setting the encryption key through the config command, as shown in figure 5.

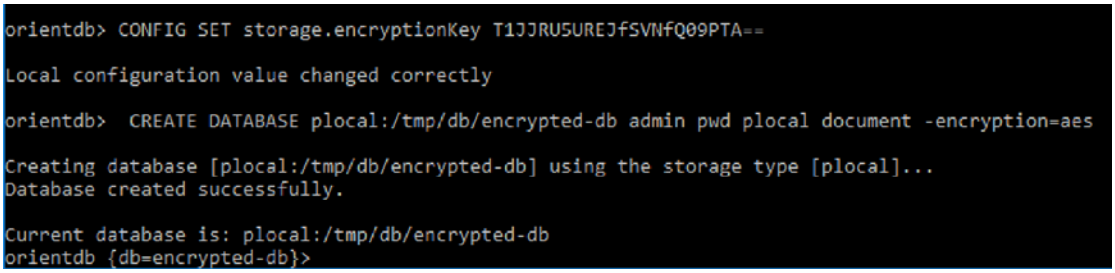

Fig. 5. Create encrypted database within OrientDB

There is also the possibility to manage security at record-level which allows the developer to apply a fine access control and security permissions to single records of a class, through which only authorized users are granted access to restricted records. To activate record-level security, classes must extend the ORestricted super-class. Information about authorization on each record is stored in special fields. Figure 6 shows the activated record-level security, while figure 7 shows how the records look to an authorized user.

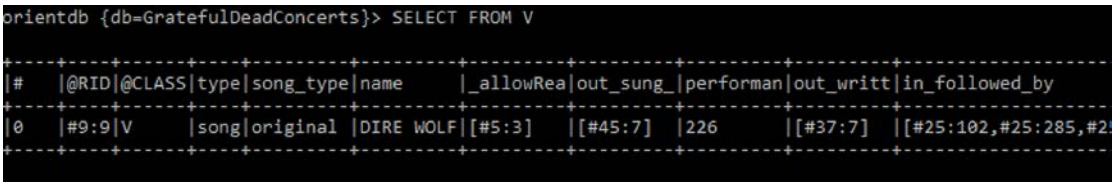

Fig. 6. User can only access one record

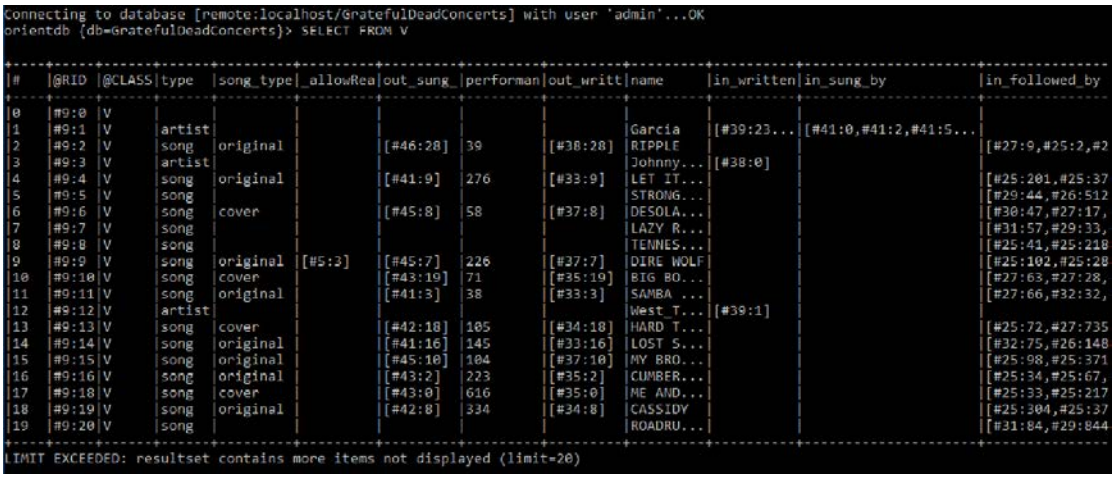

Fig. 7. All available records

Auditing

"There is no security without audit, and there is no need to audit without the need for security" [15]. Collecting information about security-relevant events such as operations on privileges, schemas, objects and statements, and analysing it can help to identify security gaps or issues with configurations in general.

Redis offers the monitor command which should show commands processed by the server, but it does not only reduce the throughput significantly, it also does not monitor important commands such as changes in the configuration, which makes it unusable for a complete auditing approach.

Cassandra, MongoDB and OrientDB offer no valuable out of the box auditing, although it is mentioned in Cassandra's documentation that audit log features could be added in a future version [3].

Log management

The purpose of log management is the central collection, transmission, storage, analysis and forwarding of log data. Our research focused on the question, to what level the provided logging commands are a usable and safe way of logging database actions.

Lab research showed that log management was either not supported at all or offered in a very basic form. Redis offers logging of basic information such as uptime in seconds, memory used and number of connected clients with the info command to the console as well as to an unencrypted file.

In Cassandra the monitoring possibilities mostly focus on performance which gives a good overview of the status of the cluster. In MongoDB the server activity as well as diagnostic information are logged but the server activity log is not encrypted. OrientDB offers logging to the console as well as to an unencrypted file.

4. Summary

In summary, the analysis at hand clearly shows that all databases provide password-based client-side authentication which is always disabled by default. Cassandra and OrientDB offer default super-users with default passwords which is a security problem. Server to server authentication is done through TLS and shared key-file approaches. Role-based authorization is implemented at a basic level in all solutions except for Redis. Support for custom defined roles is offered in all four database systems. The scope of role-based access rights varies.

The security of backups, data-at-rest and monitoring seem to be accountable to the database owner. Many security weaknesses arise from default configurations, including setting up a database with no password at all or default users with default passwords, no role and permission management, ports without protection and accepting connections from all clients. Secure access to a database requires a client to authenticate to the server and thus to verify identity. Server authentication, role-based security, role options, the scope of roles, database security and logging are all important factors that significantly simplify security administration and operations.

In the following tables, the reader can find a systematic listing of all our results.

Common abbreviations

BI	Built-in roles	CR	Custom roles	NS	Not Supported
DIS	Disabled by default	Dit	Data-in-transit	Dar	Data-at-rest

Table 1. User administration and authorization

DB Name	User administration	Authorization
Redis	NS	NS
Cassandra	default user	Role-based, BI, CR, inheritance
MongoDB	no default user	DIS, role-based, BI, CR, multiple roles per user, inheritance
OrientDB	set of default users, LDAP import of users and roles	Role-based, BI, CR, multiple roles per user, inheritance

Table 2. Authentication and password security

DB Name	Authentication	Password security
Redis	DIS, custom user/password	Clear text, sent unencrypted
Cassandra	DIS, custom user/password, super user	Default password for admin user, stored encrypted
MongoDB	DIS, custom user/password, X509, SCRAM-SHA-1,	Per-user random salt, strong hash function

	MongoDB-CR and TLS	
OrientDB	DIS, admin user per database, custom user/password and TLS	Default password for admin user, stored encrypted

Table 3. Server security and encryption

DB Name	Server security	Encryption
Redis	since v3.2 protected mode: only connections from loopback accepted	Dit: NS; Dar: NS
Cassandra	All settings disabled by default, TLS configurable	Dit: DIS, via TLS; Dar: DIS, limited support
MongoDB	All settings disabled by default, TLS configurable	Dit: DIS, via TLS; Dar: NS
OrientDB	Server-side scripting disabled, TLS configurable	Dit: DIS, via TLS; Dar: DIS, AES & DES

Table 4. Data security and Log management

DB Name	Data security	Audit	Log management
Redis	NS	NS	Basic functionalities through info and monitor command
Cassandra	Encryption of Dit and Dar; Materialized views	NS	Event logging, focus on performance metrics
MongoDB	Encryption of Dit, collection-level access control via specific role, field-level reduction based on information within document	NS	Server activity, monitoring utilities and reporting statistics, unencrypted file
OrientDB	Encryption of Dit and Dar; record-level via ORestricted super-class	NS	logging to the console or unencrypted file

5. Conclusion and future work

This work shows that for the four NoSQL database systems investigated, development is at the moment not primarily focused on the implementation of security features. The systems have reduced out of the box security functionality, some standard security concepts are completely missing. The obvious result is that for these systems database administrators and developers must be aware of the limitations in terms of security, as well as of the potential consequences of using these systems outside of their intended environment.

To mitigate the effects of the known security issues we advise to enable security features that are disabled by default, to evaluate default configurations like database password settings, protection of ports and accepted client connections; to deactivate default users with default passwords; to take care of data-at-rest encryption and to make use of user and role management if available.

Our findings also provide further motivation for our team to increase our current efforts to finish a systematic map of out of the box NoSQL security. Such a map will give users and organizations a hint on NoSQL systems having security advantages over other systems. Such a ranking will hopefully motivate vendors to invest into the security features of their systems. We have already defined two additional sets of NoSQL Systems to research for out of the box

security. We plan to publish a first version of our NoSQL security map when these research results are available. Final activities will be the completion of this map, and afterwards keeping it permanently up to date.

References

1. Afyouni, H. A.: Database security and auditing. Thomson/Course Technology, Boston 2006.
2. Aniello, L. et al.: Assessing data availability of Cassandra in the presence of non-accurate membership. In: Proceedings of the 2nd International Workshop on Dependability Issues in Cloud Computing, pp.1-6, September 30-30, 2013, Braga, Portugal
3. Apache Software Foundation: Apache License, Version 2.0. Wakefield 2004, http://www.apache.org/licenses/LICENSE-2-0.txt. Accessed December 9, 2017
4. Becker, M.Y., Sewell, P.: Cassandra: Distributed Access Control Policies with Tunable Expressiveness. In: Proceedings of the Fifth IEEE International Workshop on Policies for Distributed Systems and Networks, pp. 159-168. IEEE, POLICY (2004)
5. Cryptology Group at Centrum Wiskunde & Informatica (CWI), https://shattered.io. Accessed January 15, 2018
6. Database-Engines.COM, https://db-engines.com/de/ranking_definition. Accessed June 4th 2017 and December 7th 2017
7. Dissanayaka, A. M. et al.: A Review of MongoDB and Singularity Container Security in regards to HIPAA Regulations. In: Proceedings of the10th International Conference on Utility and Cloud Computing, pp. 91-97. Pages 91-97. ACM, UCC(2017)
8. Edlich, S.: NoSQL – List of NoSQL databases, http://nosql-database.org. Accessed June 4th 2017 and November 10 th 2017.
9. Gessert, F., Ritter, N.: Scalable data management: NoSQL data stores in research and practice. In: 2016 IEEE 32nd International Conference on Data Engineering (ICDE) 2016, pp 1420-1423. IEEE, ICDE (2016)
10. Heller, M.: Insecure MongoDB configuration leads to boom in ransom attacks, https://searchsecurity.techtarget.com/news/450410798/Insecure-MongoDB-configuration-leads-to-boom-in-ransom-attacks. Accessed April 21, 2017
11. Hasija, H., Kumar, D.: Compression & Security in MongoDB without affecting Efficiency. In: Proceedings of the Second International Conference on Information and Communication Technology for Competitive Strategies, Article No 96. ACM, ICTCS (2016)
12. Hou, B., et al.: Towards Analyzing MongoDB NoSQL Security and Designing Injection Defense Solution. In: 2017 ieee 3rd international conference on big data security on cloud (bigdatasecurity), ieee international conference on high performance and smart computing (hpsc), and ieee international conference on intelligent data and security, pp. 90-95. IEEE, IDS (2017)
13. Knox, D.: Effective Oracle Database 10g Security by Design. McGraw-Hill/Osborne, Emeryville, CA (2004).
14. Ming, C.: Abusing NoSQL Databases, https://github.com/mchow01/Security /blob/master/DEFCON21/DEFCON-21-Chow-Abusing-NoSQL-Databases.pdf. Accessed December 1, 2017
15. Natan, R., B.: Implementing Database Security and Auditing. Elsevier Digital Press, Burlington, MA (2005)
16. McLean, T.: The design flaw in PBKDF2, https://www.chosenplaintext.ca/2015/10/ 08/ pbkdf2-design-flaw.html. Accessed April 16, 2018
17. RCP 1 – Multi user AUTH and ACLs for Redis, https://github.com/redis/redis-rcp/blob/master/RCP1.md. Accessed December 10, 2017

18. Redmond, E., Wilson, J., R., Carter, J.: Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement. Pragmatic Bookshelf, Dallas (2012), p. 281
19. Sanfilippo, S.: A few things about Redis security, http://antirez.com/news/96. Accessed June 5, 2017
20. Schram, A., Anderson, K., M.: MySQL to NoSQL: data modelling challenges in supporting scalability. In Proc. of the 3rd annual conference on Systems, programming, and applications: software for humanity, 2012, pp. 191-202.
21. SCRAM-SHA-1, https://docs.mongodb.com/manual/core/security-scram-sha-1/#authentication-scram-sha-1., Accessed December 7, 2017.
22. Seth, G., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News, v. 33 issue 2, 2002, pp. 51–59.
23. Shetty, R., R., et al.: Secure NoSQL Based Medical Data Processing and Retrieval: The Exposome Project. In: Proceedings of the10th International Conference on Utility and Cloud Computing, pp. 99-105. UCC(2017)
24. Srinivas, S., Nair, A.: Security maturity in NoSQL databases - are they secure enough to haul the modern IT applications?. In: 2015 International Conference on Advances in Computing, Communications and Informatics, art. no. 7275699 , pp. 739-744. ICACCI (2015)
25. Tian, X., Huang, B., Wu, M.: A transparent middleware for encrypting data in MongoDB. In: 2014 IEEE Workshop on Electronics, Computer and Applications, pp. 906–909. IEEE (2014)
26. Okman, L., et al.: Security Issues in NoSQL Databases. In: Proceedings of the 2011 IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications, pp. 541-547. IEEE, Changsha (2011)
27. Open Web Application Security Project, https://www.owasp.org/index.php/Category: Vulnerability. Accessed January 11, 2018
28. Weintraub, G., Gudes, E.; Crowdsourced Data Integrity Verification for Key-Value Stores in the Cloud. In: Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pp. 498-503. IEEE, Madrid (2017)
29. Zaki, A., K., Indiramma, M.: A novel redis security extension for NoSQL database using authentication and encryption. In: Proceedings of the 2015 IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT) , pp. 1-6. IEEE, Coimbatore (2015)
30. Zahid, A., Masood, R., Shibli, M., A.: Security of sharded NoSQL databases: A comparative analysis. In: Proceedings of the 2014 Conference on Information Assurance and Cyber Security, pp. 1-8. IEEE, Rawalpindi(2014)