SPDC: Secure Proxied Database Connectivity

Diogo Domingues Regateiro, Óscar Mortágua Pereira and Rui L. Aguiar
DETI, University of Aveiro, Instituto de Telecomunicações, 3810-193, Aveiro, Portugal

Keywords: Access Control, Software Architecture, Security and Privacy Protection, Network Communications, Database Connectivity.

Abstract: In the business world, database applications are a predominant tool where data is generally the most important asset of a company. Companies use database applications to access, explore and modify their data in order to provide a wide variety of services. When these applications run in semi-public locations and connect directly to the database, such as a reception area of a company or are connected to the internet, they can become the target of attacks by malicious users and have the hard-coded database credentials stolen. To prevent unauthorized access to a database, solutions such as virtual private networks (VPNs) are used. However, VPNs can be bypassed using internal attacks, and the stolen credentials used to gain access to the database. In this paper the Secure Proxied Database Connectivity (SPDC) is proposed, which is a new methodology to enhance the protection of the database access. It pushes the credentials to a proxy server and separates the information required to access the database between a proxy server and an authentication server. This solution is compared to a VPN using various attack scenarios and we show, with a proof-of-concept, that this proposal can also be completely transparent to the user.

1 INTRODUCTION

Security is an important aspect to consider when sensitive data is being served by some service. However, no computer system can be completely secure without making it unusable, so the key is to find the right balance. When it comes to databases, the usage of standard APIs to access and manipulate data, such as Java Database Connectivity (JDBC) (Oracle, 1997), Hibernate (Bauer and King, 2005) and other similar mechanisms, is very pervasive.

When client database applications connect directly to databases using such connectivity tools, some problems may arise in regards to access security. If an attacker gets access to the client application of a business that connects directly to a database, he can potentially obtain the database credentials and use them to connect to the database in an unsupervised manner. This is possible because the database credentials are normally written directly into the application code or configuration files, meaning that users of the application do not need to know them. This issue is normally addressed by securing the access to the database service using solutions such as virtual private networks (VPNs).

However, a global VPN solution to access many services in a company has limited security, since anyone with access to one service may attempt to connect to another. Similarly, a service specific VPN does not protect against internal attacks that can bypass the VPN altogether. By compromising the VPN server, the database becomes much more exposed, and many times just looking at the client application source code or configuration files can reveal the database credentials to an attacker.

Using an authentication service on top of the database service is another solution to this problem, e.g. web services, but the client application must use the interface of the service and discard database tools such as JDBC. Therefore, frameworks like Hibernate that rely on them do not work. Furthermore, simply removing the credentials from the application code is not easy since there may be many of them, so providing the user of the application with all the credentials is not an option in most cases.

The reason behind it is that, among others, the user would be prone to write the credentials down if a complex credential policy is in place, i.e. policies regarding password complexity and length (Shay et al., 2016), making it easier to be disclosed. Various results also show that developers are aware of the

Regateiro, D., Pereira, Ó. and Aguiar, R.
SPDC: Secure Proxied Database Connectivity.
DOI: 10.5220/0006424500560066
In *Proceedings of the 6th International Conference on Data Science, Technology and Applications (DATA 2017)*, pages 56-66
ISBN: 978-989-758-255-4

security issues regarding passwords (Yang et al., 2016). One might consider the usage of one-time passwords to secure the credentials, which also assumes the user has some sort of smartcard or similar device. However, database management systems (DBMS) do not usually support this type of authentication natively. Implementing them manually to verify if a valid one-time password was provided requires the exclusive use of stored procedures, which becomes hard to manage in complex scenarios, or triggers to be added to each query, which decays the database performance.

In this paper, we propose a solution called Secure Proxied Database Connectivity (SPDC) which aims: to provide a mechanism that separates the information needed to connect to the database between a proxy server and an authentication server running within the database server; allow standard database connectivity tools, such as JDBC or ADO.NET, to be used; and take the database credentials out of the client application, so that a compromised client application does not disclose them. This solution requires a malicious user to compromise both servers to be able to establish a connection to the database and access its data.

This proposal emerged from the work done in (Pereira et al., 2015; Regateiro et al., 2014; Pereira et al., 2014), where a distributed access control framework allows the clients to connect to a database through runtime generated access control mechanisms. There, client applications can use an interface based on JDBC, where the methods are only implemented by the access control mechanisms to access and manipulate data stored in a database if the user has permission to use them. A connection to a server side application is also established to configure the runtime generation of the access mechanisms, based on the security policy that applied to that client. While this eases the development effort to write a database application by giving developers interfaces tailored to the permissions given to the application, it lacks security in term of access to the database itself. SPDC was designed to enhance the access security to the database, while keeping support for the convenience of database connectivity tools, such as JDBC.

The paper is divided as follows: chapter 2 presents the related work, chapter 3 presents the core concepts of SPDC, chapter 4 presents a proof-of-concept implementing SPDC, on chapter 5 a performance assessment regarding network traffic is made and chapter 6 finalizes with a brief discussion of the presented contents.

2 RELATED WORK

To be best of our knowledge, there is not much work done to secure the JDBC or ADO.NET driver type protocols. A possible explanation is that most drivers used with these tools already support SSL/TLS (IETF, 2008) connections to DBMS using digital certificates. While this normally protects the credentials from being obtained from the network by eavesdropping it or through some other means, if the malicious users have access to the client application it may be possible to steal the credentials from there.

While there is not much work done to secure the driver protocols, the SSL/TLS protocol on the other hand has seen some work to try to improve its security. In (Oppliger et al., 2006) and (Oppliger et al., 2008), a session aware user authentication is introduced and expanded to thwart Man-In-The-Middle (MITM) attacks. Nevertheless, an internal attack on the database service to establish a direct connection is enough to allow a malicious user to access the data if the credentials are obtainable from the client application. In (Abramov et al., 2012) a methodology is proposed to assist developers and database designers to design secure databases that follow the organization's guidelines for access control. It is applied and verified at the organizational and application development levels to ensure the satisfaction of the security requirements. Once more, it does not protect the database from being read from or even from being modified directly if the credentials are stolen from the client application, depending on the client application permissions.

Another solution to this problem is to connect to the database through a web service instead of a direct connection to the database, such as a login service. While this has several advantages, such as only the server can establish connections to the database and the operations performed by the client applications can be more easily controlled, they must use the web service interface to access the database. The issue with this approach is that in most cases the interface provided differs from web service to web service. This means that the developers must master them, whereas tools such as JDBC and ADO.NET provide a well-known and well-established interface to access any supported database. The work presented in (Gessert et al., 2014) leverages the importance of this problem, stating the need to create a unified interface for cloud data stores that have emerged recently. Furthermore, compromising the login service may be enough to access the database, since the malicious user knows the database endpoint and the

Table 1: State of the art summary in relation to SPDC features.

Solution	Cred. on the Client	Nodes to Breach	API	Other Notes
Basic Driver	Yes	Client	Standard	N/A
Oppliger et al.	Yes	Client	Standard	MITM mitigation
Abramov et al.	Yes	Client	Standard	Security Guidelines
Webservice	No	Webservice	Custom	Access Proxy
HA-JDBC	Yes	Client	Standard	Access Proxy
Other Auth Methods	Yes	Client	Standard	May not be supported
VPN	Yes	Client	Standard	Internal access not prevented
SPDC	No	Auth. and Proxy Servers	Standard	Architecturally costly

database credentials are still configured in the client application.

Web services can almost be seen as proxy servers that users use to access to manipulate the data stored in a database according to their permissions, albeit with their own interface. Proxy servers are used throughout the literature in some way to provide better security to access resources, be it a database, the Internet, etc. The work presented in (Zarnett et al., 2010) and (Naylor et al., 2015) show how proxy servers can be beneficial in both providing security and additional services. We restate that use of proxy servers in this paper aims to provide increased security by removing the need for client applications to possess real credentials to the database, while allowing them to use database connectivity tools transparently. Furthermore, a malicious user should only get access to the data in the database by successfully compromising both the proxy server and the authentication server, where other solutions usually only require one service to be compromised.

HA-JDBC (Ferraro, n.d.) is an existing JDBC proxy project that has been implemented, is readily available and provides many features on top of what JDBC currently supports. However, it only focuses on being light-weight, transparent, and providing fault tolerant clustering capabilities to the underlying JDBC driver. Therefore, it still requires the client application to use the database credentials. In this light, it differs from the work done in this paper in which the client authenticates with the proxy itself using a server generated token and never uses the database credentials directly.

Other methods of authentication with databases have been proposed, such as using biometric data (Villager and Dittmann, 2008), digital certificates (Lavarene, 2010), third-party based authentication (Oracle, n.d.), etc. In the case of digital certificates, the application would have to store it within itself, meaning that it could be stolen just like we argue with username/password credential pairs. Additionally, SQL Server is able to use Microsoft Windows Integrated Security (Microsoft, n.d.) to use the operating system's user account as the credentials used for authentication. While it does prevent the client applications from having credentials hard-coded, it now matters which account the user is logged into when using the application. Furthermore, normally an application only has one set of credentials to access a database. User biometric data suffers from the same issue and is not necessarily a safer or a more convenient option over the alternatives (Zimmerman, 2003).

A summary of the information presented on this section can be seen in Table 1, showing if the database credentials are on the client application, the nodes required to breach to get the credentials and the API provided to the client to access the database.

3 SOLUTION CONCEPT

In this section the core concepts behind SPDC are presented. The main goal is to offer a new approach to secure the access to a database while allowing a standard tool, such as JDBC, to be used through a proxy server. Furthermore, we show that this method protects the database from being accessed by a malicious user even if information stored in the proxy server or the authentication server are disclosed individually.

To reiterate the issue SPDC is trying to solve, solutions like VPNs do protect access to services, such as databases, while still allowing client applications to use standard APIs such as JDBC to connect to them. However, VPN solutions can be used to access a global set of services within a company, a single service or something in-between. If a database service is provided within a VPN along with other services, then it becomes less secure, since users that can access one of the other services may attempt to connect to the database. If another login layer is used on top of the DBMS then other mechanisms must be implemented to support the usage of standard database connectivity tools, such

as JDBC, to connect to it. In the case the database service is served in its own VPN, and like the login layer solution just described, internal attacks can still leave the DBMS vulnerable to malicious users that have access to client applications with the credentials hard-coded. It only takes the VPN server to be compromised to leave a DBMS much more exposed, and if just by looking at a client application source code the database credentials can be obtained, then the data becomes accessible.

3.1 Conceptual Architecture

SPDC was designed to protect the database by using a proxy server to communicate with the database, as shown in Figure 1. The idea is that the client connects to a proxy server, authenticating with it, and then the proxy server connects to the database using credentials it has stored, proceeding to relay the communication between them. This proxy server was then devised to serve as a credentials provider, using the client application real credentials when establishing the connection to the database. These credentials are also encrypted to prevent them from being disclosed in plain text in the event the proxy server is compromised.

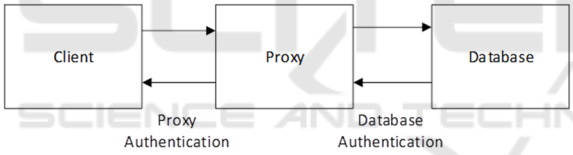

Figure 1: Conceptual Client to Database connection.

It is stated that the presented solution protects a database even when the information in the proxy server or the authentication server is compromised.

The only exception is if the DBMS itself is compromised in an internal attack and access is granted without proper authentication. To achieve this goal, both the proxy server and the authentication server were given an asymmetric key pair and the information required to access the DBMS was divided between them as follows:

- The **proxy server** possesses the user credentials, encrypted using the symmetric key *C* which is unique for each user, and requires them to establish the connection to the DBMS. It does not store the symmetric key *C*.
- The **authentication server** possesses the symmetric key *C* for each user and can open endpoints to the DBMS. It does not store the credentials to the database.
- The **client application** possesses no relevant information for authentication. This means that the user must know the credentials to connect to the authentication server and continue the process. It does possess dummy credentials to a database.

Table 2: Stored sensitive data per protocol participant.

Client Application	**Proxy Server**	**Authentication Server**
Dummy database credentials.	Encrypted user database credentials.	Open database connection endpoints.
		Symmetric keys *C*.

Table 2 shows a summary of this information. The dummy credentials stored in the client application must have connection permissions to the database. This comes from a shortcoming in the database connectivity tools used. In the case of JDBC, a connection object is only instantiated if a valid connection is made to a DBMS, and a connection object is required to be able to use JDBC on the client. Nevertheless, a separate DBMS with no data or an account stripped of permissions may be used for this purpose, making the credentials stored in the client application effectively useless to access the data.

In this scenario, the user must first authenticate with the authentication server to obtain the server generated token and present it to the proxy server. The proxy server uses this token to authenticate the user and establish a valid connection to the DBMS endpoint.

3.2 Deployment Assumptions

Let us consider what happens when each party in this exchange is compromised and what other security technologies are assumed to be deployed. During the SPDC design, there were a few security assumptions that were made. Thus, they are important to be considered when creating security attack scenarios.

First, it is assumed that the communication between all parties is protected to prevent the data from circulating in clear text, either using TLS or some other mechanism.

Second, access to the network where the database and authentication server are deployed is assumed to be restricted, for example by a firewall, allowing access to the database to be controlled and monitored.

Third, the credentials used to authenticate with

the authentication server to obtain the token *T* are assumed to not be stored anywhere in the client. If the user cannot be trusted with a password, a solution based on one-time passwords, for example, should be used.

Finally, the proxy server is assumed to be trusted by all the parties to not eavesdrop on data and to keep the database credentials protected. If this is not a possible scenario, the entity in charge of the database server should provide a proxy server on a distinct network with the same security requirements to access the database. It is also assumed that the asymmetric keys of the proxy and authentication servers are heavily secured and cannot be obtained easily by breaching the applications running on them.

3.3 Attack Scenarios

In this section, different attack scenarios that target distinct parts of the proposed solution are detailed and discussed.

Following the information shown in Table 2, if the **proxy server** is breached, then a malicious user would be in possession of the following data:

- The encrypted user credentials, which are a known function of the symmetric key *C* with the user username and password. Without the symmetric key *C* to decrypt them only a brute force attack is available.

However, the following data is not in the malicious user possession:

- An open database endpoint to connect to, which can be random for every access attempt. This prevents the malicious user from attempting online attacks.
- The user symmetric key *C*, which is different for every user and that only the authentication server stores and provides inside each token *T*. This prevents the malicious user from decrypting the credentials.

Attempting to impersonate the proxy server is impossible without stealing the proxy server asymmetric private key, since token *T* passed by the legitimate user was encrypted using the public key. Without it, the token *T* cannot be decrypted to retrieve the symmetric key *C* and the connection endpoint. Therefore, the connection to the database is impossible to be established, which also alerts the legitimate user that an attack could have occurred,

If the **authentication server** is compromised, then a malicious user would be in possession of the following data:

- The open database endpoints to connect to, which are only open when an access attempt is made. This could allow the malicious user to attempt online attacks on the database.
- The user symmetric keys *C*, which is different for every user. Without the encrypted credentials stored on the proxy server they are useless.

However, the following data is not in the malicious user possession:

- The encrypted user credentials, which are a known function of the symmetric key *C* with the user username and password. Therefore, the credentials are still safe.

Impersonating an authentication server is useless to the malicious user since the legitimate users do not provide the database credentials for authentication. Since the users do not have to know the database credentials, these should be randomly generated for each one of them.

If the **client application** is compromised, the malicious user is not capable of obtaining any relevant information to authenticate with the DBMS, since the credentials were pushed to the proxy server. Note again that the credentials used to obtain the token *T* are not meant to be stored on the client application, but to be known by the user of said application or using another authentication scheme such as one-time passwords.

3.4 Database Connection Proxying

In this section, a proposal to secure access to databases while allowing the usage of standard database connectivity tools are shown. This is achieved by separating the authentication material between the authentication and a proxy server, which relays the connection to the database. Unfortunately, most database connectivity tools do not allow the usage of proxy servers or custom sockets to communicate with the DBMS, requiring the usage of alternatives such as reflection mechanisms to achieve it.

To setup the proxy server on the client application, it must first create a connection object to communicate with the database, such as a JDBC connection object, and then modify it so that it connects to the proxy server instead. Furthermore, the proxy server is required to relay the communication between client application connection and the connection it established to the DBMS. This setup allows the client application to communicate with a database directly using a

database connectivity tool while keeping the real database credentials unknown to the user.

In short, setting an existing generic communication socket to the proxy server to be used by a database connectivity tool connection will generally require the following steps:

1. Create a generic communication channel to connect the client application and the proxy.
2. Obtain a token *T* from the authentication server and pass it on to the proxy server.
3. Have the proxy server connect to the DBMS using the information on the token *T* provided by the client and start relaying the data between them.
4. The client application creates a connection object from the database connectivity tool of choice, using a DBMS account without privileges if required.
5. The client application sets the socket to be used by the database connectivity tool to the pre-existing generic communication socket with the proxy server. The database connectivity tool can now be used as normal.

This is implemented in the proof-of-concept using reflection mechanisms, in which the SQL Server DBMS and JDBC are used to implement this solution. The authentication server must ensure that an endpoint is open so the proxy server can connect.

3.5 Deployment Considerations

Implementing the concept discussed so far allows for a functional system to be implemented, but it is not enough to make it secure given the assumptions made in section 3.2. It must be ensured that all the connections are made over secure connections, such as TLS, and that the database cannot be accessed using information stored on the client application, proxy server or authentication server individually.

This would force a malicious user to compromise both the proxy server and authentication server to be able to access the data, unless the database can be exploited using the unprivileged account. For this reason, a separate DBMS instance can be used for this purpose. SPDC also allows to set arbitrarily complex passwords on the database accounts, since the clients are not meant to know them.

Hence, SPDC has a few limitations that should be taken into consideration when deploying.

First, it does not protect the network communications, depending on existing protocols, such as TLS, to provide that protection. Its purpose is just to protect access to a database without disrupting existing database access APIs.

Second, if a malicious user were to be able to compromise the proxy server with an online attack to a point where the decrypted token is leaked from memory, then the user credentials could be obtained, as well as an endpoint to access the database. For this reason, the proxy server software must be robust to prevent memory from leaking.

Furthermore, the proxy server must be trusted or provided by the database server for the simple reason that, due to the proxying procedure, the proxy server would be able to eavesdrop on the communication. Finally, if a user chooses a poor password to authenticate with the authentication server, the client application was modified by an attacker, or the communication between the client application and the authentication server is not protected, then it would be possible to impersonate the user.

SPDC can be enhanced with other security mechanisms and protocols, such as one-time passwords, to address this last issue to some degree.

Finally, the solution has a whole has an increased cost in the architecture, since it requires a proxy and an authentication server, and should be considered against the security level required for a use case.

4 PROOF OF CONCEPT

In this chapter the SPDC implementation details and how the client can connect to the database with credentials that grant them no read/write permissions on the database is presented. For the proof-of-concept, Java was used as the programming language, SQLServer 2012 as the RDBMS with the Northwind sample database and JDBC to connect to it. The JDBC driver used was the SQL Server official release for JDBC version 4.

As mentioned, our previous work relied on data structures that communicated with the DBMS to execute operations and retrieve data. However, the client application's credentials were statically defined in the application's source code, allowing any user to retrieve those using mechanisms like reflection. This was a big security vulnerability, since the user had access to the credentials used to connect to the database with the sensitive data. This allowed him to bypass the security components completely and connect directly to the DBMS outside the application. Not only that, but the sensitive information was transmitted through the

network in clear text, so any eavesdropper could see the information.

To implement SPDC using JDBC, the following structures/mechanisms were implemented:

1. The token *T,* generated by the database side authentication service when the client application authenticates.
2. The JDBC connection object modification to use a custom communication socket so that the client application can use it while having the communication relayed through the proxy server.
3. The relaying mechanism between the database and the client application.
4. A mechanism to open endpoints to access the database on the database server side.

For the first point, a JSON (JavaScript Object Notation) object was used to encode the token. It is comprised of two main parts: the *token_data* and the *token_sig*. The *token_data* is another JSON object that contains the fields necessary for the proxy server to establish a database connection. In this proof-of-concept the *token_data* contains the following fields:

- **username:** The username of the user attempting to connect. This is an encrypted field.
- **database:** The name of the database being requested access. This is an encrypted field.
- **created:** The timestamp of the token creation.
- **expires:** A timestamp of the instant the token expires and can no longer be used.
- **nounce:** A random 32-bit integer.
- **endpoint:** The IP address and port to which the proxy must connect. This is an encrypted field.
- **C:** The symmetric key associated to the user, required to decrypt the credentials stored by the proxy server. This is an encrypted field.

All encrypted fields are encrypted using the proxy public key, previously shared with the authentication server. The symmetric key *C* is first generated when the user is given an account for the database. The associated credentials are then encrypted with it and sent to the proxy server securely. Since the proxy server must decrypt the credentials to establish a database connection, it may be possible to update the symmetric key periodically by providing both the old and the new symmetric keys on the token *T*.

The second half of the token, the field *token_sig*, contains a signature of the hash of the *token_data* field, signed using the authentication server private key so that the proxy server can validate it.

Considering the second point presented, some problems were found when trying to implement it. As discussed in section 3, the JDBC implementation does not allow an existing socket to be used to connect to the database. Using a connection string and successfully connecting to the database is the only option available to get the connection object instantiated, which is required to make requests to the database. The problem is that this connection object is connected to a DBMS directly and not the proxy server.

It so happens that to create a connection object, the real credentials are not required, just the credentials to an account that has no permissions to execute queries, i.e. an unprivileged public account, provides the client with the necessary connection object. To have the connection object communicate with the proxy server and not the DBMS, the client application can utilize reflection mechanisms. Using reflection, the internal socket and input/output streams can be set to those of the socket used to communicate with the proxy server using the generic communication channel. The usage of reflection mechanisms to achieve this has the downside of making the proof-of-concept work only for the Microsoft SQLServer driver. Other drivers may use other internal structures for the connection object, requiring a deep analysis of them to make them work with SPDC as it stands.

The connection class implementation of JDBC for SQL Server has several variables, but one is of particular interest: a variable named *tdsChannel*. This is the variable that holds the socket, named *channelSocket*, two identical input streams, named *inputStream* and *tcpInputStream*, and finally two identical output streams, named *outputStream* and *tcpOutputStream*. When it is said that they are identical, it is meant that they hold the same reference when the driver is used normally. The input/output streams are the streams used by the socket to read and write data.

With this information, it is easy to understand how the communication to the database can the transmitted through the socket that is connected to

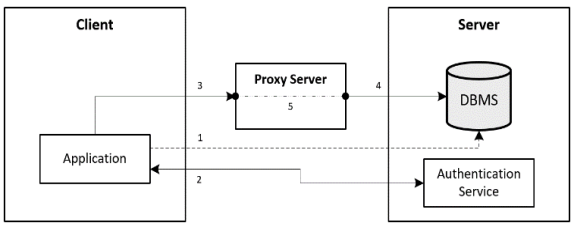

Figure 2: Sequence implementation diagram of the solution.

Table 3: Protocol messages between the client application and database server running the authentication server.

#	Client Application	Network	Authentication Server
1	*u, p*	→ (*u, p*)	
2	*u, p*	← (OK|NOK)	*u, p,* auth(*u, p*)
3	*u, p*	← (*T*)	*u, p,* *T* = genToken(*u*)
4	*u, p, T*		

the proxy instead of the database itself. All that is needed is to set the proxy socket and its input/output stream object references as the new values for the variables mentioned.

For the third point, the relaying between the client and the database performed by the proxy server is achieved using two worker threads, one for each communication direction, and the Apache Commons IO library for copying the data between the socket streams. With this setup, the client can continue to use the original connection object that was created to communicate with the database without ever knowing the real account credentials.

For the final point, simple firewall commands can be used to open and close ports to access the database by the authentication server.

To summarize, Figure 2 shows a diagram of the SPDC implementation, which is comprised of five main steps:

1. The client application instantiates the JDBC Connection object, using an unprivileged account on the database to achieve it.
2. The client application connects to the authentication server using the credentials given by the user. The server then generates and sends to the client application a security token *T* with the necessary information for the proxy server to be able to connect to the database. An access endpoint is also created.
3. The client application connects to the proxy server, presents the token *T,* waits for the token validation and modifies the JDBC Connection object as described. The proxy server decrypts the token *T* when it is received and validates the signature before replying to the application if the token is valid or not.
4. Using the symmetric key *C* within the token *T,* the proxy decrypts the credentials stored for that user and establishes a connection to the database using the endpoint indicated in the token.
5. The proxy server relays the data between the client application and the database.

Table 3 and Table 4 show the network protocol implemented in the proof-of-concept is shown.

Table 3 shows the network messages between the client application and the authentication server that runs on the database machine. It starts (1) with the client application presenting a username u and password p pair to the authentication server, which authenticates (2) using its own password database. The authentication result is sent back to the client application with a successful message ("OK") or not successful ("NOK"). If the authentication is not successful, the communication is terminated and additional security measures may be taken to prevent online password attacks. If the authentication is successful, then the authentication server generates a token T and (3) sends it back to the client application. This completes the communication between the client application and the authentication server (4).

Table 4 shows the network messages between the client application and the proxy server. It starts (1) with the client application presenting the token T it obtained from the authentication server beforehand and connecting to the database using the public account. The proxy server decodes and validates the received token *T* (2), replying to the user with a validation successful message ("OK") or not successful ("NOK"). Again, if the authentication is not successful, the communication is terminated and additional security measures may be taken to prevent online attacks to crack the authentication server private key. If the token *T* is valid, then the proxy server connects to the database using the information stored within the token *T* and the credentials stored in its own credentials database (3). This process also includes retrieving the stored encrypted credentials and decrypting them using the symmetric key *C* in the token *T.* At the same time, the client application sets the connection object it has with the database to

Table 4: Protocol messages between the client application and proxy server.

#	Client Application	Network	Proxy Server
1	*T, O* = connectToDatabase()	→ (*T*)	
2	*O*	← (OK|NOK)	*T,* validateToken(*T*)
3	*O,* useProxySocket(*O*)		*T,* connectToDatabase(*T*)
4	*O,* applicationCode(*O*)		relay()

use the network socket connected to the proxy server. Finally, the proxy server begins relaying the communication between the client application and the database while the client application executes the application code (4).

All unnecessary variables and attributes are removed from memory to minimize the possibility that they may be leaked. This is most evident with the token *T,* which the client application only stores until it is sent to the proxy server. Furthermore, the proxy server only keeps the token *T* in memory until the database connection is established.

Other more complete authentication protocols can be used to authenticate the user, such as Kerberos (Neuman and Ts'o, 1994) or RADIUS (IETF, 2000b) and even implementing the GSSAPI (IETF, 2000a) to support a wide variety of protocols, since SPDC is not concerned with the way users are authenticated. SPDC is only concerned with the proxy server obtaining the information required to establish a database connection and relay the communication to the client, which is why this proof-of-concept uses a basic authentication protocol.

5 PERFORMANCE ASSESSMENT

In this section the performance assessment made to test the solution is presented. This performance assessment includes a comparison between the network traffic generated by a VPN and the solution proposed in this paper. However, the performance in terms of delay is not included because everything is done during connection time.

The network traffic tests between the relay method herein described and using a direct connection through a VPN were made by establishing the initial connection and then issuing an aggregation query to the SQLServer Northwind sample database that calculates the total amount of money per order, sorting by the total.

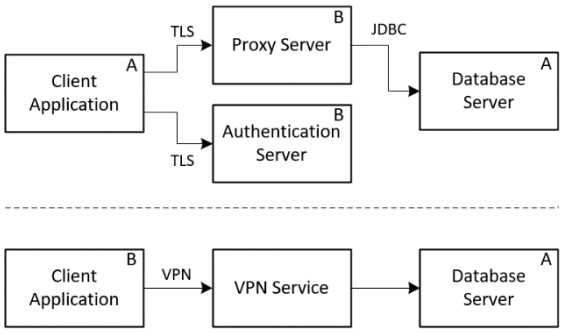

Figure 3: Test deployments, the top case using SPDC and the bottom using a VPN.

The machines used to carry these tests are specified on and the traffic was captured using Wireshark v2.2.2.

The direct approach using a VPN is trivial and was conducted by instantiating a JDBC connection object and then executing an aggregation query over Northwind's Orders table which contains 1000 rows. A third-party VPN server, supported by OpenVPN, was used to establish the VPN connection. The network traffic was captured by the client application to measure the amount of data sent through the VPN. The SPDC solution uses TLS to secure the connections described in this paper and was deployed on two machines: machine A and machine B. Both test cases, using a VPN and SPDC, are illustrated in Figure 3, which also shows which role each machine had.

To reiterate, after the client application authenticates with the authentication server, it waits for the authentication server to reply with the token *T* as described on section 3. The authentication server also opens an endpoint on the firewall, so the proxy server can connect using the details provided in the token. The client application (machine A) then connects to the proxy server (machine B), and sends the token it received from the authentication server. At the same time, it connects to a DBMS instance using the account that holds no privileges to

Table 5: Testing machines specification.

Machine	A	B
OS	Windows 10 Home	Windows 7 Ultimate
Architecture	x86_64	x86_64
Motherboard	LENOVO Lancer 5A2 (U3E1)	Gigabyte H87-HD3
CPU	Intel Core i7 4510U @2.00GHz	Intel Core i5 4670K @3.40GHz
Memory	8.00GB DDR3 @797MHz	8.00GB DDR3 @665MHz
Hard Drive	465GB SSHD-8GB	1863GB SATA
RDBMS	Microsoft SQL Server 2012	-
Other Programs	Netbeans IDE, Wireshark	Netbeans IDE, Wireshark

instantiate a JDBC connection object. When the proxy server acknowledges that the token is valid, the client application exchanges the socket on the JDBC connection object to the socket it used to connect to the proxy server. At the same time, the proxy server creates the actual JDBC connection object to the database, using the user credentials that it decrypted with the symmetric key *C* from the token, and begins relaying the communication between the client application and the database.

The query execution made by the client is the same as in the VPN approach, since it now possesses a JDBC connection object with permission to access the data on the database. From this point on, SPDC adds no further traffic overhead to the connection, meaning that it has no further impact after the connection is established.

On Table 6 a summary of the results obtained when capturing the traffic generated by each solution can be seen. On both solutions, the network traffic was captured by the client application and the packets were filtered to only the relevant ones. To ensure no unwanted traffic was captured on the VPN solution, the VPN server was configured to only allow connections made to the database server through.

Table 6. Traffic overhead results.

Solution	#Packets	Bytes Transmitted
Connection: **VPN**	177	31409
Connection: **SPDC**	48	19303
100 Queries: **VPN**	2356	1480972
100 Queries: **SPDC**	1649	1316834

It is possible to see that overall, SPDC using TLS to secure the connections is better than a VPN based solution in regards to network traffic. During connection, SPDC used around 73% less packets than the VPN counterpart, needing only 48 packets compared to the 177 used by the VPN. Furthermore, SPDC transmitted about 39% less data, needing only about 19KB compared to the 31KB used by the VPN. This is expected since VPN also must transmit information regarding routes and other network related information needed so the client can setup the VPN.

Considering the applicational data transmission with the database queries, the SPDC transmitted 30% less packets, using 1649 packets when the VPN solution used 2356. This can be explained due to the fact that SPDC, after the connection is established, has no other traffic overhead. Thus, the overhead in the SPDC solution is added just by the TLS protocol itself. The same reason can be applied to explain the data transmitted, since SPDC was able to serve 100 database queries while transmitting 11% less data than the VPN solution, needing about 1.26MB compared to the 1.41MB needed by the VPN.

It is important to note that it is possible to have the proxy server establish a VPN connection to the database server before connecting to the database. In this case, the network traffic overhead is expected to be the same as if only a VPN was being used, since SPDC adds no overhead once the connection is established.

6 CONCLUSION

This paper presented SPDC, a mechanism to secure access to databases using a proxy server while allowing client applications to access data in databases using standard database connectivity tools. Furthermore, it splits the information needed to access the database between the proxy server and the authentication server, as well as removing the hard-coded database credentials from client applications.

This way, a malicious user with access to the client application is not able to obtain database credentials just by looking at the source code. By having the proxy server connect to the database using the credentials associated with the client and relaying the communication between the client and the database, it is possible for the client to connect to the database using database connectivity tools without any useful credentials. Since neither the proxy server or the authentication server possess all the information needed to access the database, a malicious user must compromise both servers to be able to gain access.

However, several aspects of this solution must be taken into consideration. The cost of the architecture is greater than that of a VPN solution, for example, since it requires a proxy server and an authentication server, in contrast to only one additional server in the case of a VPN. Furthermore, the proxy server must be carefully monitored to avoid eavesdropping, given the nature of proxy servers the data must be taken from one secure channel to another. This data should not remain in memory or cached. Additionally, and while it is not a very hard adaptation to make, JDBC drivers other than SQL Server's must have a procedure created to modify the communication sockets.

Concerning performance, it was shown that connecting to the database using the proposed SPDC mechanism can be better than using a VPN solution in terms of network traffic overhead, both in terms

of connection and applicational data transmission.

One last possible concern with the use of a proxy server could be raised regarding scaling, since a server-side application must relay the communication between the client and the DBMS in the SPDC presented in this article. However, there is no issue in having several of these proxy servers to process the client's requests to connect to the database since they are inherently stateless. They establish a connection given a token, and once the connection terminates a new connection can only be made by presenting another token.

Finally, in terms of future work it would be interesting to address the eavesdropping issue on the proxy server. This could be achieved by adding a layer on top of the database that accepts connections and all data is encrypted using, for example, the authentication server password of the user. The implications of such alteration must be carefully studied and tested.

ACKNOWLEDGEMENTS

This work is funded by National Funds through FCT - Fundação para a Ciência e a Tecnologia under the project UID/EEA/50008/2013 and SFRH/BD/109911/2015.

REFERENCES

Abramov, J. et al., 2012. A methodology for integrating access control policies within database development. *Computers {&} Security*, 31(3), pp.299–314.

Bauer, C. & King, G., 2005. *Hibernate in Action*.

Ferraro, P., HA-JDBC: High-Availability JDBC. Available at: https://ha-jdbc.github.io [Accessed September 13, 2016].

Gessert, F. et al., 2014. Towards a scalable and unified REST API for cloud data stores. *Lecture Notes in Informatics (LNI), Proceedings - Series of the Gesellschaft fur Informatik (GI)*, P-232, pp.723–734.

IETF, 2000a. RFC 2743: Generic Security Service Application Program Interface Version 2, Update 1.

IETF, 2000b. RFC 2865: Remote Authentication Dial In User Service (RADIUS).

IETF, 2008. RFC 5246: The Transport Layer Security (TLS) Protocol - Version 1.2.

Lavarene, J. de, 2010. SSL With Oracle JDBC Thin Driver. Available at: http://www.oracle.com/technetwork/topics/wp-oracle-jdbc-thin-ssl-130128.pdf.

Microsoft, SQL Server Security Modes. Available at: https://msdn.microsoft.com/en-us/library/aa266913(v=vs.60).aspx [Accessed September 13, 2016].

Naylor, D. et al., 2015. Multi-Context TLS (mcTLS). *ACM SIGCOMM Computer Communication Review*, 45(5), pp.199–212.

Neuman, C.B. & Ts'o, T., 1994. Kerberos: An Authentication Service for Computer Networks. *IEEE Communications Magazine*, 32(9), pp.33–38.

Oppliger, R., Hauser, R. & Basin, D., 2006. SSL/TLS session-aware user authentication - Or how to effectively thwart the man-in-the-middle. *Computer Communications*, 29(12), pp.2238–2246.

Oppliger, R., Hauser, R. & Basin, D., 2008. SSL/TLS session-aware user authentication revisited. *Computers and Security*, 27, pp.64–70.

Oracle, Authentication Using Third-Party Services. Available at: https://docs.oracle.com/cd/B19306_01/network.102/b14266/authmeth.htm#i1009853 [Accessed August 13, 2016].

Oracle, 1997. JDBC Introduction. Available at: http://docs.oracle.com/javase/tutorial/jdbc/overview/index.html [Accessed March 3, 2014].

Pereira, O.M., Regateiro, D.D. & Aguiar, R.L., 2014. Role-Based Access control mechanisms. In *2014 IEEE Symposium on Computers and Communications (ISCC)*. Vancouver, BC, Canada: IEEE, pp. 1–7.

Pereira, Ó.M., Regateiro, D.D. & Aguiar, R.L., 2015. Secure, dynamic and distributed access control stack for database applications. *International Journal of Software Engineering and Knowledge Engineering*, 25(9–10), pp.1703–1708.

Regateiro, D.D., Pereira, Ó.M. & Aguiar, R.L., 2014. *A secure, distributed and dynamic RBAC for relational applications*. University of Aveiro.

Shay, R. et al., 2016. Designing Password Policies for Strength and Usability. *ACM Transactions on Information and System Security*, 18(4), pp.1–34.

Villager, C. & Dittmann, J., 2008. Biometrics for User Authentication. In *Encyclopedia of Multimedia*. Boston, MA: Springer US, pp. 48–55.

Yang, X.L. et al., 2016. What Security Questions Do Developers Ask? A Large-Scale Study of Stack Overflow Posts. *Journal of Computer Science and Technology*, 31(5), pp.910–924.

Zarnett, J., Tripunitara, M. & Lam, P., 2010. Role-based access control (RBAC) in Java via proxy objects using annotations. *Proceeding of the 15th ACM symposium on Access control models and technologies - SACMAT '10*, p.79.

Zimmerman, M., 2003. Biometrics and User Authentication. Available at: https://www.sans.org/reading-room/whitepapers/authentication/biometrics-user-authentication-122.