Supporting Pre-shared Keys in Closed Implementations of TLS

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

Keywords: Software Architecture, Secure Communications, Information Security, Network Protocols.

Abstract: In the business world, data is generally the most important asset of a company that must be protected. However, it must be made available to provide a wide variety of services, and so it can become the target of attacks by malicious users. Such attacks can involve eavesdropping the network or gaining unauthorized access, allowing such an attacker to access sensitive information. Secure protocols, such as Transport Layer Security (TLS), are usually used to mitigate these attacks. Unfortunately, most implementations force applications to use digital certificates, which may not always be desirable due to trust or monetary issues. Furthermore, implementations are usually closed and cannot be extended to support other authentication methods. In this article a methodology is proposed to slightly modify closed implementations of the TLS protocol that only support digital certificates, so pre-shared keys are used to protect the communication between two entities instead. A performance assessment is carried out on a proof-of-concept to demonstrate its feasibility and performance.

1 INTRODUCTION

Security is an important aspect to consider when sensitive data is being served by some service, but as in many software solutions, it is often regarded at later phases of their development cycle. Therefore, many companies end up relying on the usage of secure communication channels to authenticate client applications and encrypt the data transmitted. To this end, digital certificates are many times used to secure the transmission of that data.

Digital certificates, which uses asymmetric cryptography, is a very good solution for situations where communication is required between endpoints that are unknown to each other, such like browsers and webservers. However, the usage of digital certificates places extra responsibility on the users of the client applications, among other security problems (Leavitt, 2011). The users have to maintain the public key certificate of the server secure if it is not signed by a certificate authority whose public key certificate is distributed along with the device being used, making the whole system rely on a Public Key Infrastructure (Weise, 2001).

There are several reasons as to why a company might not want to have their certificates signed by an external certificate authority, e.g. the fact that the company must trust a third-party company or the money required to sign the company certificate. This can lead to self-signed certificates being used, which requires a level of trust in the users of the client applications that may not be compatible with the business policies. A self-signed public key certificate can be easily replaced on the client application host device without the user noticing, meaning that an intruder can still impersonate the server to obtain sensitive information that can be used later for follow-up attacks. Even when the server certificate is signed by a certificate authority, it is still possible for an attacker to install a new root certificate authority on the client application host device and forge a new certificate claiming the attacker to be the server, since root certificate authorities' certificates are always self-signed. Incidentally, human users of systems that rely on these certificates do not always verify which root certificate authority signed the certificate in use.

This paper argues that TLS implementations based on pre-shared keys can address these concerns, because using pre-shared keys removes the need to trust certificate authorities, and by extension paying them to sign a digital certificate. Hence, it will focus on situations where asymmetric keys are not necessarily the best option, such as between a server and a custom client application that connects to it.

Furthermore, the client application does not have to validate the certification chain of the certificate

Regateiro, D., Pereira, Ó. and Aguiar, R.
Supporting Pre-shared Keys in Closed Implementations of TLS.
DOI: 10.5220/0006424701920199
In *Proceedings of the 6th International Conference on Data Science, Technology and Applications (DATA 2017)*, pages 192-199
ISBN: 978-989-758-255-4

presented by the server. If the server does not know the pre-shared key, then the connection simply cannot be made.

Additionally, using pre-shared keys is less prone than certificate authentication to certain types of configuration mistakes, such as expired certificates or mismatched common name fields. However, the system must apply restrictions on the accepted pre-shared keys to avoid low entropy.

Furthermore, such an implementation provides mutual authentication (i.e. the client and the server both authenticate each other), while TLS with server certificates only authenticates the server to the client, leaving the client authentication to the application logic. Client certificates can be used to authenticate the client, but it may be easier for a user to remember a pre-shared secret, such as a password, than to manage a certificate. Additionally, the business would have to trust the client to keep their private key safe.

However, using pre-shared keys can give rise to some new issues, such as the possibility of the users writing down the keys or how the endpoints agree on a shared-key securely. These issues can also be mitigated and will be discussed.

A previous work is presented (Pereira et al., 2014)(Regateiro et al., 2014) and (Pereira et al., 2015), called Dynamic Access Control Architecture (DACA), where a distributed access control framework allows the clients to connect to a database through runtime generated access control mechanisms. There, the client applications can use an interface based on Java Database Connectivity (JDBC), which is implemented by the access control mechanisms, to access and manipulate data stored in a server. A connection to a server side application is also made to configure the runtime generation of the access control mechanisms, based on the security policy that applied to that client. The method proposed in this paper can then be used to secure this framework without the issues described.

This paper contributes with a method to avoid the usage of digital certificates to authenticate the server and the client using TLS. It also analyzes the changes necessary to make this possible, the security considerations that must be taken into account and its impact on the performance of a system.

The paper is divided as follows: chapter 2 presents the related wok, chapter 3 presents the proposed concept solution, chapter 4 presents a proof of concept detailing the method of implementation and the related issues, on chapter 5 a performance assessment is carried out to measure the overhead of the solution and on chapter 6 a discussion of the proposed work is made.

2 RELATED WORK

The work described in this article attempts to add support for pre-shared keys in SSL/TLS (IETF, 2008), even though the standard does define cipher suits that make use of pre-shared keys instead of digital certificates. Therefore, this article is not aimed to modify the standard, but instead to make it possible to use pre-shared keys with closed implementations of TLS that do not support these cipher suits. One such case is the Java implementation (Oracle, n.d.), which is not extensible to support new cipher suits programmatically and does not implement the cipher suits using pre-shared keys. The method described in this paper will allow to use pre-shared keys in the Java implementation of TLS.

In terms of security, the SSL/TLS protocol has seen some work to try to improve it. In (Oppliger et al., 2006) and (Oppliger et al., 2008), a session aware user authentication is introduced and expanded in an attempt to thwart Man-In-The-Middle (MITM) attacks. Unfortunately, these approaches still use digital certificates. While the proposed solution is not completely invulnerable to these kind of attacks, it can detect when one occurs and preventive measures can be taken before sensitive data is disclosed.

There are SSL/TLS cipher suits that do not require digital certificates, and therefore certificate authorities and a public key infrastructure, that provide secure communication based on passwords such as TLS-SRP (Taylor et al., 2007). The Secure Remote Password (SRP) (Wu, 1998) is a password authenticated key exchange that allows to parties to agree on a common value derived from a password and a salt, known in advance. This value is then used to establish a TLS connection. The SRP protocol is also a form of augmented password authenticated key exchange, meaning that the server does not store any password equivalent data. The solution in this paper does not possess this feature, but is not as complex and requires less computing power to use since it does not utilize complex mathematics.

Other password authenticated key exchange protocols exist, such as the Simple Password Exponential Key Exchange (SPEKE) (Hao and Shahandashti, 2014) (MacKenzie and MacKenzie, 2001) or Password Authenticated Key Exchange by Juggling (J-PAKE) (Hao and Ryan, 2010) (Toorani, 2014) (MacKenzie and MacKenzie, 2001). However, no known actively supported implementation in TLS exist.

3 SOLUTION CONCEPT

The goal with this work is to adapt the SSL/TLS encryption protocol to use pre-shared keys instead of digital certificates on closed implementations that do not support the specific cipher suits that makes use of them.

To achieve this goal, the capabilities of the SSL/TLS encryption protocol are required to be maintained without the need for certificate authorities or a public key infrastructure. Instead, a pre-shared key will be used to establish the secure connection. While a pre-shared key can be obtained from a user of the client application for server impersonation, it is likely to be different between users. Provided that non-trivial pre-shared keys are used, any attempt at server impersonation will only affect that user/client application. This differs from a server certificate because, since it is the same for every client application, one forged certificate signed by a certificate authority installed on the device is all that is required to impersonate the server for every client, even easier if the certificate is self-signed.

In the event a pre-shared key is stolen from a user, many methods of determining that a non-legitimate client is trying to use the pre-shared key of another client can be put in place, such as authentication attempt notifications or client location based validation using the IP address. The protocol behavior of the proposed solution under certain attack scenarios will also be shown.

This chapter is divided as follows: section 3.1 introduces the adapted protocol which use pre-shared keys without the required cipher suit and section 3.2 will open some discussing regarding security considerations that are raised in the standard in the light of the adapted protocol.

3.1 TLS Protocol Adaptation Method

In this section the adapted protocol that is being proposed will be discussed. The basis of the solution for an SSL/TLS based protocol using a pre-shared key relies on setting up an initial SSL/TLS channel using the Diffie-Hellman key exchange protocol (Diffie and Hellman, 1976) in anonymous mode. This mode allows a connection to be made without the use of certificates, but it leaves the connection vulnerable to MITM attacks and neither the server or the client are authenticated. The connection encryption keys are then modified to provide these features.

The basic TLS handshake protocol message exchange pattern can be seen in Figure 1. since a certificate is used in this example, and for both parties

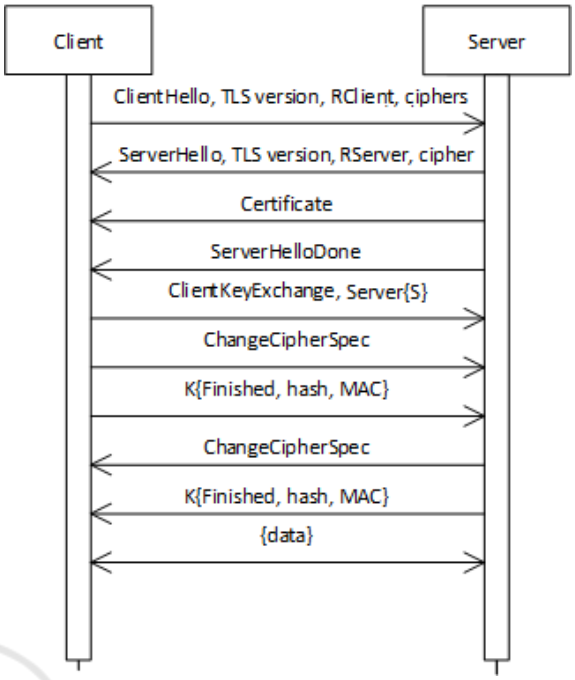

Figure 1: Basic TLS protocol using server certificate.

to agree on a common secret to encrypt the communication, the client must generate a secret and send it encrypted with the server public key. Since the data encrypted with the public key can only be decrypted with the private key and only the server should possess the private key, only the server can decode the shared secret that will be used to encrypt the communication with the client. Given that the client trusts the certificate authority that signed the server certificate, it has some degree of confidence that the secret can only be decrypted by the intended server. Further communication is then encrypted using the agreed secret.

Instead of using digital certificates due to the issues presented, the adapted protocol makes use of the Diffie-Hellman key exchange protocol. The basic Diffie-Hellman key-exchange protocol uses a mixture of public and private values so that two parties can agree on a secret, exchanging only public information that does not allow a third party to easily arrive at the agreed secret.

The TLS implementation using the Diffie-Hellman key exchange protocol in anonymous mode uses an agreed key K to derive the session key for encrypting the rest of the communication, but it does not authenticate the client or the server. Hence, it is vulnerable to MITM attacks since the endpoints are not verified before the key exchange takes place.

A small variation was introduced in the Diffie-Hellman protocol which aims to change the agreed

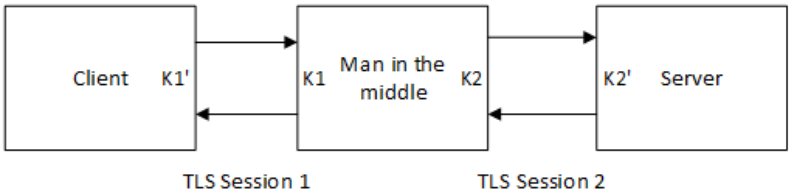

Figure 2: MITM attack scenario on the adapted TLS protocol.

key *K* the same way on both endpoints using a pre-shared key (PSK). This PSK should be agreed upon *a priori* using a separate communication channel.

If the key agreed initially using the Diffie-Hellman protocol is *K*, then the new agreed key *K'* used to calculate the session key becomes:

$$K' = H\left(K + H\left(F(PSK)\right)\right) \quad (1)$$

With PSK in Formula 1 being the pre-shared key, *F* a known function of the pre-shared key and *H* a cryptographic hash function. The function *F* can be a simple concatenation of a user's pre-shared secret with a salt value or something more complex, its main goal is to prevent the usage of rainbow tables (Marechal, 2008).

This transformation of the agreed key cannot be performed by an intruder in a MITM attack scenario, since they should not know the pre-shared key of the client. This way, when no MITM attack is in effect, both the client and the server can communicate using the SSL/TLS channel as normal, since they both perform the same transformation of the base key K. However, if an MITM attack is in effect, then the intruder cannot decrypt the data coming from either endpoint after the key K is transformed into K'.

This occurs because the modified keys K' do not match between the client and the server, as shown in Figure 2 (K1' and K2'), since the client and the server agreed on the initial key *K* with the MITM on two separate sessions and not between themselves. Thus, when the client and the server transform their agreed keys with the pre-shared secret, the attacker cannot decrypt the messages sent by either of them. In fact, even if the attacker tries to relay the data between them to hide the attack, neither can decrypt the data they receive since they agreed on different *K'* keys.

However, this method still contains one vulnerability. In a MITM attack scenario, after the agreed key has been hashed and the new session key calculated, the client will send a message to the server with a known structure. The intruder can then perform an offline dictionary attack to transform the known key *K* agreed with the client into *K'* and attempt to decrypt the message until the known structure is obtained. Fortunately, it is always possible to detect that a MITM attack has occurred. If a client does not receive a valid response from the server inside a time window or an invalid message is received instead, then it is best to assume a MITM attack has occurred and take precautionary measures as necessary, such as expire the user pre-shared key.

3.2 Security Considerations

In this section the security considerations made in the standard (Eronen and Tschofenig, 2005) that should be taken into account when dealing with security for communication protocols such as SSL/TLS are discussed.

The first consideration is regarding perfect forward secrecy (Günther, 1989) and it expresses the property of a communication protocol to not compromise past session keys if the long-term keys are compromised. Considering that the adapted protocol uses the Diffie-Hellman private key exchange, which generates a different key K for each handshake of the protocol, when the key K' is compromised it is only possible to decrypt that communication session, given that the attacker is also in possession of the client pre-shared key. Since the agreed keys K are independent from one another, all past communications remain uncompromised, providing perfect forward secrecy.

The second consideration regards to brute-force and dictionary attacks. The use of a fixed shared secret of limited entropy such as a pre-shared key chosen by a human (e.g. a password) may allow an attacker to perform a brute-force or dictionary attack to recover the shared secret. This may be either an off-line attack (against a captured TLS handshake message) or an on-line attack where the attacker attempts to connect to the server and tries different keys. In the case of a protocol that uses Diffie-Hellman, such as this adapted protocol proposal, an attacker can only obtain the message it requires by getting a valid client to connect to him, for example by using a MITM attack. While a weak pre-shared key can be obtained from such methods, only future communications between the client and the server are vulnerable, since the key *K* agreed via the Diffie-

```
32    SSLServerSocket serverSocket = (SSLServerSocket) ssf.createServerSocket(port);
33    serverSocket.setEnabledCipherSuites(new String[]{"TLS_DH_anon_WITH_AES_128_CBC_SHA"});
34    SSLSocket clientSocket = (SSLSocket) serverSocket.accept();
35    Object handshaker = ReflectionUtils.getFieldValue(clientSocket, "handshaker");
36    clientSocket.startHandshake();
37    changeSSLConnectionKeys(clientSocket, handshaker, secret);
38    return clientSocket;
```

Figure 3: Server socket creation process with the pre-shared key authenticated key exchange.

Hellman key exchange changes every time a new connection is established. Additionally, since a MITM attack is required to obtain the data needed for an offline attack, it is always detectable as explained in section 3.1. If the system triggers a forced pre-shared key reset, then future communications are not vulnerable. However, as with many other protocols, one attacker could carry out a denial of service attack by making MITM attacks on every communication attempt.

Finally, considering identity privacy, currently the adapted protocol does send the client identity in clear text. It is possible to enhance the protocol by sending the client entity after the Diffie-Hellman key exchange protocol has finished and the communication is encrypted by the agreed key. However, since the client identity is required for the server to know which pre-shared key to use to modify the agreed key *K*, this method would only prevent eavesdroppers from knowing the communicating parties. In a MITM attack scenario, the attacker would still be able to know the identity of the client.

4 PROOF OF CONCEPT

For the proof of concept, Java was used as the programming language for the client application and SQL Server 2012 as the relational database management system (RDBMS). In this section, the method for supporting pre-shared keys in the native Java implementation of TLS will be detailed.

The generic approach to implement the modifications needed to make the adapted protocol work can be seen in Figure 3, and it shows how an anonymous SSL/TLS connection is established using Diffie-Hellman and where the agreed key is altered. However, a problem surfaced while trying to implement this method. Since the SSL/TLS implementation in Java is closed and cannot be extended, the agreed key *K* mentioned on section 3.1 cannot be easily changed. Reflection features in the Java programming language were used to access the Java implementation of the SSL/TLS protocol and perform the necessary changes to make it work.

First, an SSL socket is created (line 32) and the desired cipher suit is specified (line 33). The cipher suit is a named combination of authentication, encryption, Message Authentication Code (MAC) algorithms and a pseudorandom function. The cipher suit "*TLS_DH_anon…*" states that the key exchange protocol to be used is Diffie-Hellman (DH) and no authentication will be made (anon), which removes the need to use certificates since they are used to provide authentication. The other fields are not relevant to this work.

The server then waits for a client to connect (line 34). When a client connects, the server saves the reference to the *handshaker*, which is an internal object of the *clientSocket* that handles the handshaking process of the SSL/TLS protocol (line 35). ReflectionUtils is a class that provides several functionalities based on reflection, where *getFieldValue(obj, fieldName)* retrieves variable with the name *fieldName* from the object obj. It is required to save this reference because after the handshaking process finishes, its reference is set to null. Then a normal handshaking process takes place (line 36), after which the agreed key is changed (line 37).

Figure 4 shows the same process from the client point of view. First an attempt to connect to the server is made (line 46). Since the cipher suit used is disabled by default it must be enabled in the client as well (line 47). Then a reference to the *handshaker* object is saved for the same reason as in the server (line 48) and then start the normal handshake process (line 49). Finally, the agreed key is changed using the same process as the one used on the server (line 50).

This handshake process uses Diffie-Hellman key exchange protocol to agree on a *preMasterKey* which is then used to derive a *masterKey* (our key *K*) from which the read and write ciphers are initialized. Since the *preMasterKey* is disposed of after the *masterKey* is created, it is not possible to change it. Therefore, it was decided that changing the *masterKey* instead was the better option, since it remained available throughout the whole process.

Only changing the *masterKey* into *K'* is not enough, however, because the read and write ciphers that read and write into the communication channel have already been initialized. The process necessary to change the *masterKey* into *K'* and update the read

```
46      SSLSocket socket = (SSLSocket) csf.createSocket(dest, port);
47      socket.setEnabledCipherSuites(new String[]{"TLS_DH_anon_WITH_AES_128_CBC_SHA"});
48      Object handshaker = ReflectionUtils.getFieldValue(socket, "handshaker");
49      socket.startHandshake();
50      changeSSLConnectionKeys(socket, handshaker, secret);
```

Figure 4: Client socket creation with the pre-shared key authenticated key exchange.

and write ciphers in Java is as follows.

1. Obtain the current private state of the socket and save it.
2. Obtain the *masterKey* (*K*) from the TLS session.
3. The *masterKey* bytes is hashed per Formula 1 and the output truncated to the original size of the *masterKey* (*K'*).
4. The *calculateConnectionKeys* private method from the *handshaker* object, which calculates the connections keys for the read and write ciphers, is invoked.
5. The *handshaker* reference in the socket is set, since it has been set to null after the handshake. This is required for step 7.
6. The socket state is set to *cs_HANDSHAKE*. The methods invoked in step 7 assert that the socket is still handshaking.
7. Call the private methods *changeReadCiphers* and *changeWriteCiphers* declared in the socket class.
8. Cleanup by setting the socket state back to the original value and the socket *handshaker* reference back to null.

The previous process requires the usage of Java Reflection in every step except step 3, because the variables set and methods invoked in the socket are private.

The reliability on the reflection functionality has the big problem of destroying the abstraction created by the public interface. Any changes made to the Java SSL/TLS internal structure could potentially break this particular solution since it is dependent on the implementation.

However, the overall methodology of changing the *masterKey* to implement the adapted protocol should remain the same at its core.

5 PERFORMANCE ASSESSMENT

In this section, several tests aiming to measure the performance of the proposal are presented. One aspect that is important to note is that the proposal had to be implemented using Java Reflection, which impacts performance considerably, but since it is only needed at the start of the communication its impact will be limited.

The performance was measured over three distinct variables: 1) the time it takes for of the adapted TLS protocol (PSK) and an anonymous TLS to connect; 2) the usage of TLS sessions; 3) and sending 50MBs of random data for communication overhead. The first variable aims to test the performance difference between establishing a normal TLS connection, albeit using anonymous Diffie-Hellman to be comparable with the adapted version, and the adapted version connection. The second variable aims to show how the automatic resumption of TLS sessions impacts performance on both cases. This is worth testing since when a TLS session is resumed, the adapted protocol procedure to change the master key is not executed, as the previous *master key* (*K'*) is reused. Finally, the third variable aims to show how the performance compares when data is sent over the connection.

This section is divided as follows: Section 5.1 defines the test environment and section 5.2 shows and discusses the results obtained.

5.1 Environment

This section details the test environment and the machine used to run the performance tests, shown in Table 1. Note that all unneeded programs and services were not running or disabled. Network connectivity was also disabled.

Furthermore, to maximize the overhead introduced by this proposal, simple client and server applications were run on the same machine to simulate an optimal environment with low network delay. The results shown were obtained from these client and server applications.

Table 1: Testing machine specification.

OS	Windows 10 Home
Architecture	x86_64
Motherboard	LENOVO Lancer 5A2 (U3E1)
CPU	Intel Core i7 4510U @ 2.00GHz
Memory	8.00GB Dual-Channel DDR3 @ 797MHz (11-11-11-28)
Hard Drive	465GB Seagate ST500LM000-SSHD-8GB (SATA)
Other Programs	Netbeans IDE

5.2 Performance Tests

In this section the performance tests conducted and their results will be presented. The tests show the time it took to establish a TLS connection, either using this approach with pre-shared keys (PSK) or not (TLS). The performance impact of session resumption is also tested by invalidating a session after each iteration of a test (NS) or not (S) and finally by the time it takes to send 50MBs of data (D). Each test had 10.000 running iterations except when the 50MBs of data were sent, in which case 100 iterations were used.

Table 2 shows the data that was obtained from the tests, including the average time, the standard deviation and the 99th, 95th and 90th percentile. The names of the tests indicate the test itself, i.e. PSK-NS indicates that the test used the approach using pre-shared keys and did not allow for session resumption. Note that to obtain these times, the *System.nanoTime()* service was used, which uses the current JVM's high-resolution time source and returns its value with a nanosecond precision and nanosecond resolution in the case of the machine used. It is not related to the current time and it is only usable to calculate elapsed time.

Table 2: Performance tests measurements in milliseconds.

	PSK-NS	PSK-S	TLS-NS	TLS-S	PSK-D	TLS-D
Avg. Time	7.40	7.19	1.44	1.32	2266	2256
Std. Dev.	2.92	2.25	4.42	4.57	39	43
99th Perc.	18.3	15.6	8.91	7.28	2332	2395
95th Perc.	11.3	10.3	2.56	2.35	2326	2325
90th Perc.	9.38	9.19	1.95	1.74	2324	2310

Overall, these results show that the unaltered TLS connection is established faster than the adapted PSK connection by about 6ms, from 7ms with PSK to 1ms with TLS. This is expected, since the master key of the TLS connection is modified in the PSK connection for each connection. Furthermore, it rarely takes more than 18ms as shown by the 99th percentile, with a 10ms difference to the TLS version. However, this increase in connection time does not impact the communication of data. This is discernable from the PSK-D and TLS-D test results, given that the difference of 10ms is well within the standard deviation of about 40ms. In many applications, the connection is made while they initialize, which can take some time (up to a few seconds). In these cases an increase in 6ms for the connection time can be considered negligible, since the connection, once made, can be reused for the entire duration of the session.

These results also show that resuming past sessions is beneficial, even though its benefit is decreased due to the minimal network delay in the performance setup. This is attributed to the fact that session resumption decreases the number of TLS protocol messages needed to establish a connection. Nevertheless, it allows a connection to be completed faster on average 0.12 ms (9%) for the PSK connection and 0.21 ms (3%) for the normal TLS connection.

These results confirm that the process of modifying the master key of a TLS connection does impact performance, given the different in connection time. However, the impact can be neglected in most use cases since it only occurs during the establishment of the connection and has no impact in the data communication process. Hence, the adapted TLS version to use pre-shared keys proposed in this article can be used in situations where digital certificates are not required and/or desirable, even when closed implementations of TLS are used, and with just a few milliseconds of overhead during connection.

6 CONCLUSION

This article presented an adapted version of the TLS communication protocol that was developed to encrypt the communication between the client applications and the server without the need to use digital certificates. This was meant to not only protect the database but also to allow companies that cannot trust certificate authorities or that cannot buy the certificates to not have to use self-signed certificates.

These self-signed certificates were argued to not be secure, since anyone can create a new self-signed certificate stating that they are the company they claim to be.

The proposed protocol in this paper works by using the anonymous Diffie-Hellman key exchange algorithm and by performing the same transformation of the agreed master key with a pre-shared key known *a-priori* by the server and the client. While this approach allows attackers under certain conditions to know the identity of the client, the attackers cannot

decrypt any communication made between them in the past.

If the pre-shared key is made vulnerable by a MITM attack, this fact is always known by the client and the server since they cannot communicate with one another. Forcing a pre-shared key reset will keep future communications secure, while past communications are always secure due to the ephemeral nature of the Diffie-Hellman keys.

Concerning performance, the creation of the communication channel using the adapted TLS protocol introduces just a few milliseconds when compared to the time a normal TLS connection in anonymous mode takes. Performance could be enhanced if the Java SSL/TLS API allowed to add new key exchange algorithms, avoiding the overhead resulting from Java Reflection, but the performance should be satisfactory in most use cases as is.

Further work into this problem can be carried out to fully evaluate and minimize the overhead caused by the usage of Reflection mechanisms, as well as evolving this solution to try to avoid the identity of the clients from being disclosed.

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

Diffie, W. & Hellman, M., 1976. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6), pp.29–40.

Eronen, P. & Tschofenig, H., 2005. [PSK] Pre-Shared Key Ciphersuites for Transport Layer Security (TLS) [RFC 4279]. *RFC 4279*, pp.1–15.

Günther, C., 1989. An identity-based key-exchange protocol. *Advances in Cryptology—Eurocrypt'89*, 434, pp.29–37. Available at: http://link.springer.com/10.1007%2F3-540-46885-4_5 [Accessed April 5, 2016].

Hao, F. & Ryan, P., 2010. J-PAKE: Authenticated Key Exchange without PKI. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6480(PART 2), pp.192–206.

Hao, F. & Shahandashti, S.F., 2014. The SPEKE Protocol Revisited. , (July).

IETF, 2008. RFC 5246: The Transport Layer Security (TLS) Protocol - Version 1.2. Available at: http://tools.ietf.org/html/rfc5246.

Leavitt, N., 2011. Internet Security under Attack: The Undermining of Digital Certificates. *Computer*, 44(12), pp.17–20.

MacKenzie, P. & MacKenzie, P., 2001. On the Security of the {SPEKE} Password-Authenticated Key Exchange Protocol., (2001/057), pp.1–17. Available at: http://eprint.iacr.org/2001/057.ps.gz.

Marechal, S., 2008. *Advances in password cracking*,

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, Java Cryptography Architecture Oracle Providers Documentation for Java Platform Standard Edition 7. Available at: https://docs.oracle.com/javase/7/docs/technotes/guides/security/SunProviders.html [Accessed October 10, 2016].

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

Pereira, Ó.M.Ó.M., Regateiro, D.D.D.D. & Aguiar, R.L.R.L., 2015. Secure, Dynamic and Distributed Access Control Stack for Database Applications. *International Journal of Software Engineering and Knowledge Engineering*, 25(09n10), 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. Available at: https://www.academia.edu/7913868.

Taylor, D., Wu, T. & Mavrogiannopoulos, N., 2007. *Using the Secure Remote Password (SRP) protocol for TLS authentication*, Available at: http://www.hjp.at/doc/rfc/rfc5054.html.

Toorani, M., 2014. Security analysis of J-PAKE. *2014 IEEE Symposium on Computers and Communications (ISCC)*, pp.1–6. Available at: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6912576.

Weise, J., 2001. Public key infrastructure overview. *Sun BluePrints OnLine, August*, pp.1–27.

Wu, T., 1998. The Secure Remote Password Protocol. In *Proceedings of the Symposium on Network and Distributed Systems Security NDSS 98*. pp. 97–111.