Authentication Protocols

Mike Castro Lundin

November 2016

1 Introduction

Authentication in client/server applications can be used to achieve several goals. It can be used to ensure confidentiality, integrity and availability, by limiting which users can access certain information/resources (confidentiality), determining which users can modify information/resources (integrity) and which users can delete/remove information/resources (availability, for example stopping the print server).

Additionally it can be used to ensure accountability, as actions will be linked to an authenticated user, so if users where charged per document printed, this information could be collected in the server.

The implementation discussed in this report assumes that all communication between client and server goes though a TLS channel, thus it is assumed that a secure pseudonymous channel is established, avoiding potential hijacking of the channel, or the use of replay or man-in-the-middle attacks. Implementing the TLS channel has not been done, since the server does not currently have a certificate, but could be done using the RMI SSL library.

This report will evaluate the problems in regards to

1. Password Storage: Usage of databases or file systems, hashing and salt usage.
2. Password transport: What TLS ensures, authenticated sessions compared to individual request authentication.
3. Password Verification: Where to validate the password, and how.
4. Issues in this implementation of the Kerberos protocol: Ticket lifetime, password transport, knowledge requirements, service key changing, clock synchronisation, replay attacks and ensuring freshness of authentication.

The same three factors are also relevant for keys used in symmetric encryption, though the implementation of safe key storing mechanisms is not considered in the implementation.

Two solutions are proposed, one for session based authentication based on the Kerberos protocol, and a simpler model for individual request authentication.

The software ensures secure communication between the client and the ticket granting service, as well as the client and the printing service. This is ensured by encryption of the messages and previous knowledge known by servers. It is assumed that all communication goes through TLS, in order to protect the transport of the password to the authentication server. The Kerberos protocol also avoids replay attacks by the use of timestamps. [1]

Thus the software together with TLS ensures confidentiality, integrity, availability and accountability, assuming the servers are not attacked by a denial of service attack.

2 Authentication

Authentication of a user is the process of a user proving his identity to the server. This can be done by something the user knows (password, PIN number, passphrase), something the user has (identity badge, physical key) or something the user is (biometrics). [1]

Password based authentication has several difficulties, since users can forget them (requiring to replace it without knowing it), re-use them in many services, disclose them to other people (either voluntarily or for example a Trojan horse) and a change in the password can lead to violating the availability goal if it was not done by the user. Password authentication also requires that the server somehow store the passwords for verification, leading to a possible vulnerability, and frequently used passwords can be used to guess the password using a dictionary attack. [1] Finally a brute force attack is possible, but an increasing delay when an incorrect password is input can make the attack much slower.

2.1 Password storage

As mentioned before, authentication using a password requires that the server has the previous knowledge connecting users to passwords in order to verify the authentication. This is the password storage problem, and this report will consider saving the passwords in a system file using the operating system mechanisms protection mechanisms, in a public file and using a database.

We assume all options store the passwords in a one-way encrypted form (not plaintext), and use salt (not reusing the same salt) in order to avoid dictionary attacks (E(pw+salt)). While a system file might sound safe, but there are vulnerabilities caused by the fact that the whole operating system will have access to the file anyway, and if the operating system is not partitioned between users, then all users will have access to all privileged information. It will also mean that the solution will be OS dependent.

The second option (public file) is vulnerable to being leaked as memory. Even if the encryption is strong and salt is used, it can still be vulnerable to the attacker trying popular passwords using the salt, and it does not take into consideration the fact that it will still leak potentially sensitive information (if a user has an account). Options one and two have the issues with scalability, since if there is more than one server, each must have an updated copy of the file. Also backups will contain the passwords, and thus are a vulnerability. Since they are file systems, then they are also scaling badly with many users, since iterating though a file to find a user might take long time, and multiple requests accessing the same file can be an issue in particular when it needs to be modified.

The third option is perhaps the best option. A DBMS will provide great scalability, since the amount of servers can be increased and concurrency is ensured. Fault tolerance is ensured and multiple access is not an issue. Security is also taken care of by access control. This also helps isolate the storage form the service.

Thus the third option has been selected, since it is considered the most scalable both in terms of user amount and request amount and is secure assuming implemented correctly. The implementation simply uses a hash table to simulate the database for simplicity and so the code is executable in any device.

2.2 Password transport

Authenticated sessions is a solution to avoid forcing a user to authenticate multiple times, since this process can be tedious and time consuming to some users. A session implies that on successful authentication, the user is granted temporary access to whatever resource he was authenticating into. This raises issues in regards to representing a session in terms of a ticket which can only be used by

this user. There is scepticism around using sessions in particular when dealing with very sensitive information (for example credit cards) [1], but in applications that require less strict security it can be an acceptable compromise of security for usability.

The implementation uses the Kerberos protocol in order to authenticate sessions and adding the ticket parameter to all methods in the print interface, while individual request authentication is done in a simpler model, where each method in the print server requires two additional parameters (user and password). Both implementations assume that TLS is used to secure the communication, and the ticket is destroyed when the client is done.

The Kerberos implementation currently checks the time stamp down to the minute, but in reality it should be much more strict and consider the second too. Ticket lifespan is one hour, since usability has a large importance and security is not a big concern compared to if there was a payment involved in printing.

In regards to the possibility of hashing the password before sending it to the server, this is not a better solution because it means the salt must be sent from the server and stored in the user workstation. Also logically then the hash becomes the password, and means that an attacker who has the hash of a user could use it to log-in without knowing the user password. [2] The safety of the password transport relies entirely on the TLS protocol safety, and thus can be sent safely relying on the TLS encryption.

2.3 Password verification

Once the password arrives in the server, it must be verified that it is the same as the one in the database. This process is done in the server. The server fist gets the hash and salt from the database corresponding to the user, and uses the salt and the password to generate a hash using the same algorithm. If the results matches the expected hash, then authentication succeeds.

2.4 Assumptions

Since it is assumed TLS ensures secure communication, then it is assumed that there is confidentiality integrity and availability during the initial message from the client to server in Kerberos. TLS also ensures that replay attacks are not viable for the print commands after the confirmation of the server ({|t+1|}K(Client, PrintServer)), though this assumption could be removed if each command had the additional parameter {|t|}K(Client, PrintServer), to use the timestamp in every command to ensure no replay attack happened. This however does not ensure the ticket can be hijacked, and thus the implementation just relies on TLS which by ensuring confidentiality prevents an external party from getting the ticket from traffic.

Correct database implementation and user enrolment is also assumed, so there are no vulnerabilities or leaks there. Shared key storage is also not considered (they are just saved in variables in the implementation), since in practice the servers will use certificates and asymmetric cryptography, and the symmetric key only needs to be stored by the user safely.

3 Design and Implementation

Two implementations have been designed to represent the session based authentication model, and the individual request authentication model.

3.1 Session Authentication

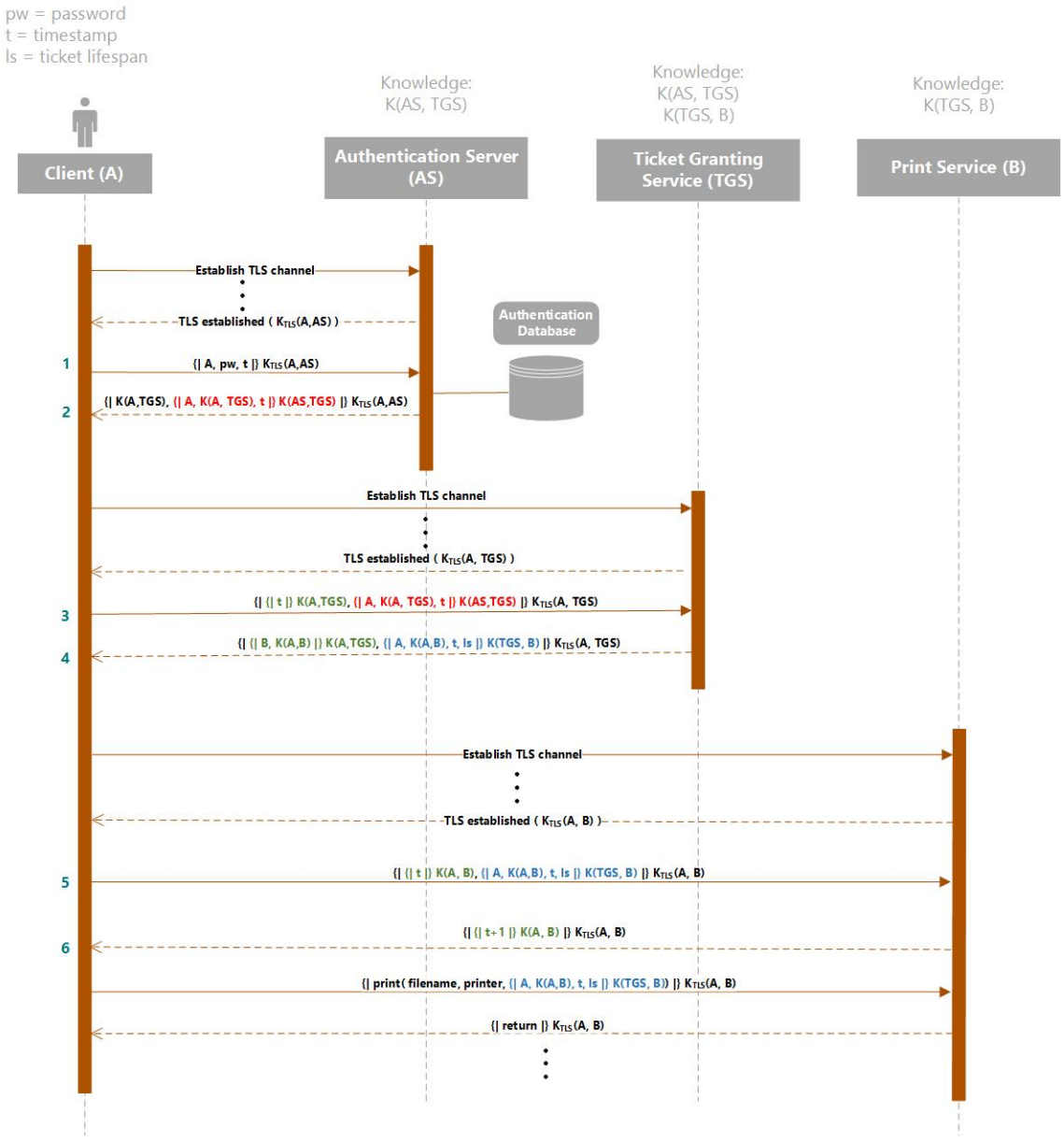

Figure 1: Kerberos Session Authentication

This authentication model is based on the Kerboros protocol as taught in lecture 7. As mentioned in the lecture and in "Security in Computing" [1], the protocol has the following issues:

1. Re-authentication every time a new service is contacted: Since there is currently one service this will be no issue.
2. Password sent across network in the clear: Not anymore since this is after TLS handshake.

3. Single point of failure: Everything relies on Authentication Service and Ticket Granting Service being available. Can be solved by having many authentication servers.
4. Clock synchronisation: Clocks must be kept synchronised, but since it is localhost this is not an issue.
5. Sniffer to capture or reusing workstation: Ticket is deleted when client terminates, so it is not in memory, and TLS prevents an outside party getting the ticket.
6. Ensure freshness of Authentication: Done by adding a timestamp to the authentication result (nr 2 in figure 1 in the red term).
7. Ticket reusing makes it vulnerable: The longer the validity of the ticket, the more vulnerable it is theoretically.
8. Subverting server time allows using expired tickets.
9. Password guessing: Since the implementation relies on TLS instead of using the password to encrypt communication between the client and the authentication service, this is not a vulnerability.
10. Single authentication and ticket granting servers limit scalability: Servers can be duplicated at the cost of duplicating keys (increase of risk exposure) or second set of keys (all processes must check for both keys, duplicating the work).

On the other side the advantages of using the Kerberos protocol are:

1. Single-sign in: Users do not need to authenticate in every request, making the system more usable.
2. Authentication is separated from the service, allowing for other services to be added easily.
3. Cryptographic protection against spoofing: Keys ensure only the user can read the messages.
4. Limited validity: Brute-forcing tickets is made unfeasible, since they should expire before the attack will succeed
5. Timestamps prevent replay attacks: Since messages are valid for so short time, most replay attacks are invalidated.
6. Mutual authentication: Users authenticate services (message 6 proves server knows K(TGS,B)). This can be used to create a unique channel in which theoretically encryption is unnecessary. [1]

This particular implementation uses the TLS key in message 1 and 2 instead of the password hash, since concerns about using passwords as keys were raised in lecture 9.

3.2 Individual Request Authentication

Individual request authentication establishes a TLS channel, and then sends command encrypted in this channel, containing the log-in credentials (username and password) every command. This can be annoying for the client, but results on a much simpler model. While it could be thought that this model was more vulnerable because credentials are sent more times, TLS will ensure that communication is different since K(A,B) will be different every time the channel is created. The overhead will depend on retrieval of credentials in the database and hashing (which should be slow) instead of decrypting a ticket and verifying timestamps. I would argue that this makes the overhead larger overall, though the session based solution will have a longer process to initialise (getting ticket from TGS). In terms of scalability this solution could scale easily by simply connecting another server to the database. Authenticity of the server however is not verified internally, but rather by TLS relying on the print server having a certificate to authenticate. The protocol is shown in figure 2.

This solution also removes the issues related to clock synchronisation, and is arguably just as secure thanks to TLS.

3.3 Code implementation

Salts are generated using safe random, and are not reused. The password and the salt are used to generate the key using 10000 iterations to get a 256 bit key. Then PBKDF2WithHmacSHA1 is used (though this can be replaced with PBKDF2WithHmacSHA512 for 512 bits hash instead of 160) to

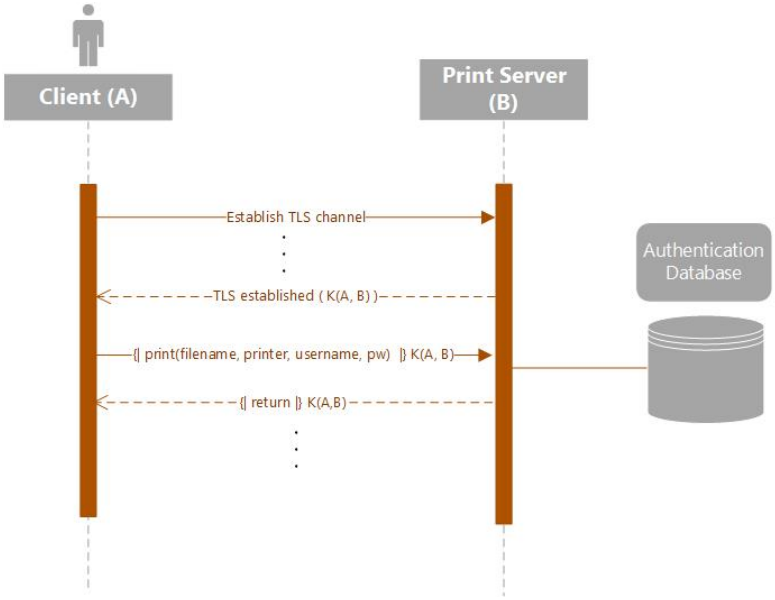

Figure 2: Individual Request Authentication

get the hash, which is stored with the salt in the database.

New keys are generated when needed and not stored after they are used. The keys are generated using secure random and are 256 bytes long. The key is used to encrypt/decrypt messages using AES.

To execute the session authentication solution servers are started in class /ALSession/ServerStart.java and any number of clients can be run using the class /ALSession/ClientSession.java. Before terminating the client will destroy the ticket, so it cannot be reused.
To execute the individual request authentication solution, the server is started using /ALIndividualRequest/ApplicationServer.java, and then clients are run using the /ALIndividualRequest/Client.java class.

Both are setup with manually populated simulated databases with credentials User1, Password1; User2, Password2... up to User6. These users can be modified in the classes /ALIndividualRequest/PrintServant.java (line 36) and /ALSession/AuthenticationServantSession.java (line 33). RMI is implemented like described in the given tutorial.

4 Evaluation

Both implementations ensure that the user is authenticated before the service is invoked. In the session implementation, the user proves he is authenticated by owning a ticket which contains his username among other information, thus at the start of the method invoked the ticket is verified and if successful, the username and the method can be logged, and the command executed.

Similarly, in individual request authentication the username and password are verified to authenticate the user at every method invocation, and the action is only done if the authentication succeeds. Logging the user and method would simply require adding it to the method after authentication. Authentication happens inside the method invocation, but the design could be changed to have an authentication server that sends the authenticated method calls to the service only when the authentication succeeds, thus isolating the service from the authentication process. This design would look like figure 3

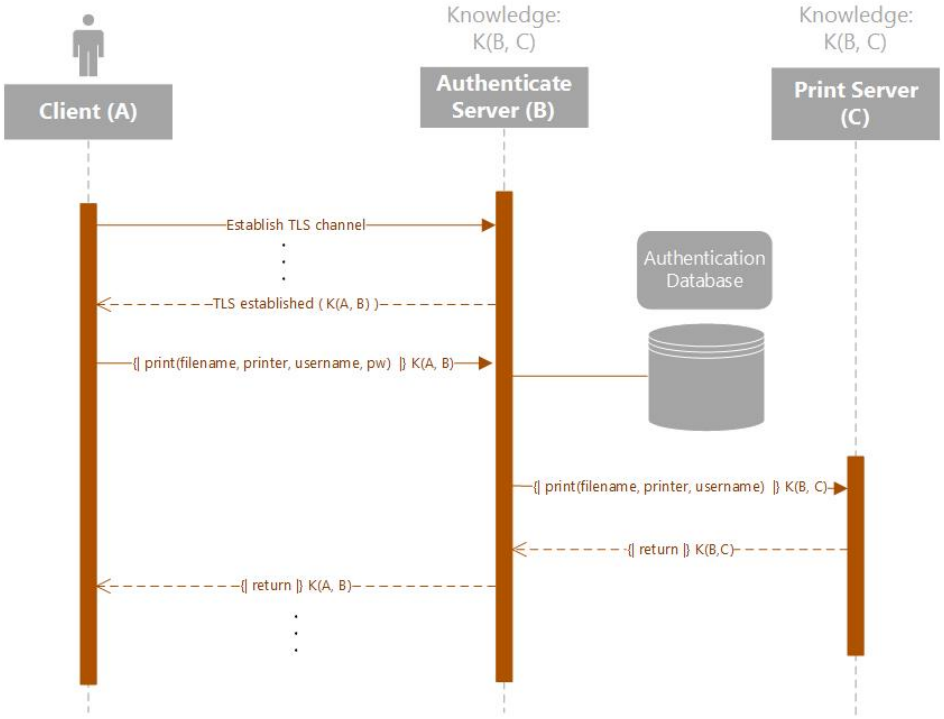

Figure 3: Isolated Individual Request Authentication

Since the user is always authenticated when invoking methods, then the accountability goal has been achieved. The other goals are confidentiality, integrity, availability as well as preventing other parties from using another users authentication.

Confidentiality and integrity are all ensured by TLS, though Kerberos also has mechanisms to ensure them using the keys that are generated (with the exception of the client-authentication message (1)).

Availability is considered in terms of how resilient the service can be against DoS. As established in the implementation section, Kerberos has some problems scaling, in particular because keys must be duplicated, or a new set must be created. Individual requests do not have this issue assuming all servers are properly certified, and even the extended version in figure 3 scales better since it is just one key (K(A,B)) which there must be two of, instead of in Kerberos where an extra TGS server will require a new key to the authentication server as well as for each of the services.

In terms of preventing replay attacks or man-in-the-middle attacks, this is ensured by TLS, but beyond that, Kerberos also provides additional security by using timestamps, but the storage of the ticket in the client side might be potentially vulnerable. Similarly in Kerberos the servers are authenticated by proving that they can decrypt the messages between AS and TGS and TGS and the service to get the key, and respond by using this key to encrypt the returning message (thus the need for message 6 which responds t+1).

5 Conclusion

In conclusion both the implementation for session based authentication and individual request authentication achieve the goals assuming there are no flaws in the implementation of the database, the authentication between two servers (assumption that K(AS, TGS) and K(TGS, B) are only known by both parties), storage of the ticket and keys in the client as well as proper implementation of TLS. Also this does not take into account any attacks such as a Trojan horse in the client in order to get the password.

In regards to which solution is preferred, it is my opinion that Kerberos ensures many security goals that are already covered by TLS (both encrypt messages and prevent replay attacks), giving perhaps more overhead than necessary, while implementing Kerberos without the use of TLS would leave many vulnerabilities open (objections where raised in regards to using the password as encryption key in lecture 9 of the course). Also session authentication will never be safer than individual request authentication [1], but is a compromise to achieve usability.

Thus I would say individual request authentication is a simpler and in my opinion better solution, and session authentication should only be used only if the assumptions are all considered, and the damage caused by an external party getting a valid ticket is not so big.

5.1 Future work

To improve the solutions, the following are some improvements that can be made:

1. Replace PBKDF2WithHmacSHA1 to PBKDF2WithHmacSHA512
2. Implement TLS to both solutions (requires servers to have a certificate)
3. Implement a secure database
4. Consider the temporary storage of keys and ticket in the client, and make sure it is done safely
5. Implement safe user enrolment policies
6. Implement several servers (AS, TGS or service servers), and define if it will duplicate keys or use a new set of keys.

References

[1] Shari Lawrence Pfleeger Charles H. Pfleeger. *Security in Computing*. Pearson Education, 2007.

[2] Defuse Security. Salted password hashing - doing it right. https://crackstation.net/hashing-security.htm.