CS 1653: Project Phase 3

Group Members: Solomon Astley, John Fahnestock, Alex Mesko

Introduction

For this phase of the project, our group will utilize a variety of cryptographic techniques in order to protect against the threats outlined in the project description. For starters, we will use a **hybrid authentication protocol** in order to do some forms of server authentication and to establish a **shared session key**. This hybrid protocol will take advantage of both **symmetric** and **asymmetric cryptographic encryption** algorithms. Additionally, to perform user authentication we will be implementing a password-based protocol which uses **cryptographic hashing** in order to maintain the integrity of user passwords on the group server. In order to prevent users from meddling with group-server-issued tokens, we will be utilizing a **signature** from the group server on a token which can be efficiently checked by file servers using the group server's **public key** in order to verify the token's integrity. These techniques and protocols will be discussed in more detail later in the report.

Mechanisms

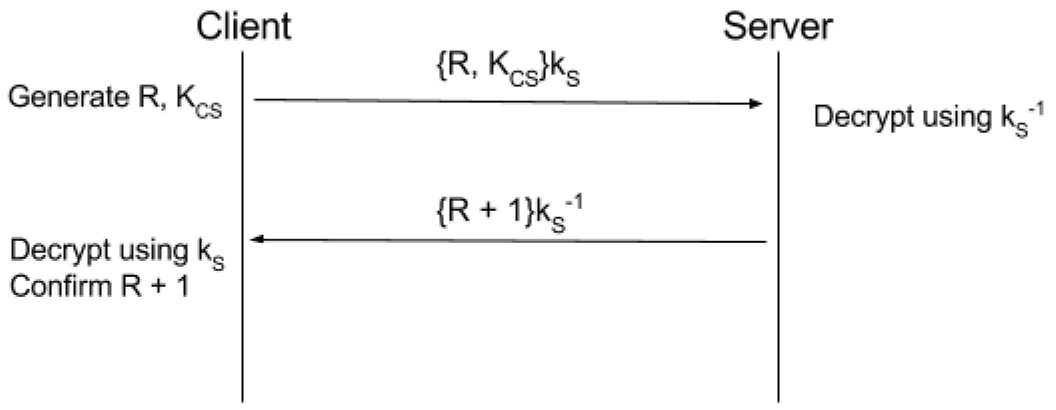
This section of the report will consist of a description of each of the mechanisms that will be deployed in order to protect against the threats outlined in the project description. Each threat will be described, followed by a description of the mechanism being used to protect against the threat and a short argument explaining the correctness of the preceding mechanism to protect against the given threat.

Protocol 1

Before the threats are described, however, it would be best to describe a protocol now which will be referenced by several of the protection mechanisms later on. In particular, this protocol will be used in order to authenticate a server to a client and establish a shared secret between the two. Specifically, the protocol uses hybrid cryptographic encryption to do so. The following assumptions are made for this protocol, and all protocols that implement this one:

1. The server has a public RSA key k_S which is known by the client.
2. The server has a private RSA key k_s^{-1} which is only known by the server.

To begin, the client generates a random number R and a cryptographically strong AES key K_{CS}. The client then encrypts these two values using the server's public key, k_S, and sends that message to the server. Upon receipt of this message, the server decrypts it using its private key, k_S^{-1}. At this point, the client and the server both have access to a shared secret, K_{CS}. The server then uses k_S^{-1} to encrypt the number R + 1 and sends that back to the client. When the client receives this message, it decrypts it using k_S. At this point, the client has authenticated the server.



This protocol works because of the underlying cryptographic techniques used to execute it. Without the private key k_S^{-1}, it would not be feasible to decrypt the initial message sent from the client to the server containing the randomly chosen number R. Thus, when the client receives a response from the server which, when decrypted, contains the number R + 1, it knows that the response must be from the server which has knowledge of its private key. For a similar reason, this reasoning also explains why only the client and the server have knowledge of the symmetric key K_{CS} at the end of the protocol.

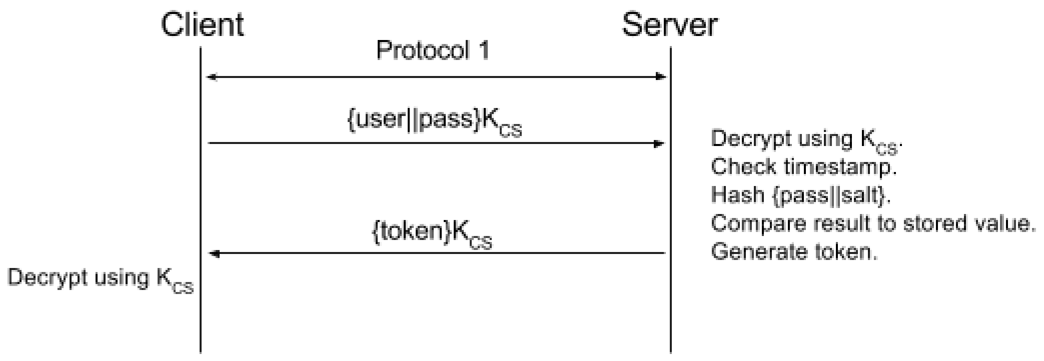
A 128 bit AES key should be infeasible for a modern adversary to launch a brute force attack against. Given that the K_{CS} will potentially be used to encrypt file data, the CBC mode of operation should be used to help prevent replay attacks. Since AES has gone through rigorous testing and attacking through use by the NSA and in the private sector we deem AES to be a safe encryption standard to use. In general, the algorithm is fast and easy to implement while remaining safe. Additionally, a 4096 bit RSA key will be used. This key length removes any question about the strength of the RSA key against brute force attacks for the time being and provides a long enough key to encrypt all the values required for this protocol in one encryption operation (4096 / 8 = 512 bytes). RSA for public/private key encryption is also a widely used and accepted algorithm. With respect to its security, it has been posited that breaking RSA would be reduced to the hardness of factoring large primes which is currently infeasible given modern hardware. This is also infeasible to break through knowledge of efficient algorithms as this problem is considered to be in the class NP, but has not been proven to be NP-Complete.

Threat 1: Unauthorized Token Issuance

In this phase of the project, all clients are assumed to be untrustworthy. As such, they may attempt to obtain tokens from the group server which do not belong to them. This may include users that either have an existing account on the group server or do not have an existing account. For example, while Alice may have an account on the group server, she should not have permission to obtain Bob’s token. In the current implementation from phase 2 of the project, there are no restrictions preventing one user from accessing another user's token. All the malicious user would need to do is enter another user's name in the prompt for a token.

To protect against this threat, the group server will use a password-based user authentication protocol. In particular, when a user asks for a token, they will be prompted for a username and password by the client application. The client application will then encrypt these values using a shared symmetric key between the client and server (see Protocol 1 for the establishment of this shared secret) and send the result to the group server. When the group server receives this message, it will decrypt it using the shared secret key. Before any password checking is performed, the group server will first check for an existing timestamp for this username. Login attempts will be limited to one per second in order to prevent online brute-force attacks.

If there is no existing timestamp for this user or the existing timestamp is more than one second old, the group server will check the user's password. To do so, the group server will compute a cryptographically secure hash of the user's password concatenated with a randomly generated salt value for that user and compare the result to a value it has stored for that particular user. The salting and hashing is done in order to prevent offline attacks in the event of a compromised group server. If the compared values match, the group server will generate a token for the user, encrypt it using the shared secret, and send the result to the client. Then, the client will decrypt this message also using the shared secret and will have successfully received a token from the group server. This protocol is demonstrated in the figure below.



This protocol works because Protocol 1 works and because it protects against both online and offline attacks from malicious users. An online attack might consist of a malicious user repeatedly trying to guess a user's password by communicating with the group server. The protocol protects against this by limiting login attempts to one per second, making it infeasible to try all possible passwords for a given user. An offline attack might consist of a malicious user who has compromised the group server database and has access to the table of usernames and hashed passwords. The protocol protects against this by not storing plaintext passwords and by salting passwords before hashing them.

Threat 2: Token Modification/Forgery

Not only will users attempt to gain access to other users' tokens (such as in threat model 1), but they will also attempt to modify their own token and/or forge fake tokens for themselves in order to give themselves higher access privileges to the file server. It is unrealistic for the file server to validate a user's token by communicating with the group server for every action that the user takes on the file server, so a protocol must be developed which will allow the file server to verify the validity of a user's token without talking to the group server. In the current implementation, if a user that also happens to be a skilled programmer were to maliciously modify their token, the file server would have no way of discovering this modification.

In order to protect against this threat, the group server will cryptographically sign each token that it issues. This will be done by computing a digest of the token and then encrypting the digest with the server's private key, k_s^{-1}. The result of this encryption will be added to the token itself. When the token is issued and used to access a file server, the file server may verify the token's validity by decrypting the signature using the group server's public key, k_S, and then comparing the result to a digest of the token.

In order to sign tokens such that each token is unique and will produce different hashes as long as they are not identical, the User's data will be concatenated in a StringBuilder object in the form (Issuer || Subject || groups…) since the username's are necessarily unique the hashes of these tokens will be unique.

This protocol works assuming the digest of the token is computed using an algorithm which has all the properties of a good cryptographic hash algorithm. If that is the case, then it will be infeasible for an attacker to generate a fake token (or modify an existing token) which will hash to the same digest as the one computed and encrypted by the group server. Without a token which, when hashed, generates a digest which is equal to the one provided by the signature of the group server, the file server will not permit any actions to be taken by a user.

Threat 3: Unauthorized File Servers

As per the project description, any number of file servers may exist and any user may make a new file server if they wish to do so. Given this, it would be theoretically possible for a malicious user to pretend to be a trustworthy file server. In doing so, they would potentially be able to gain access to the private information of users that are fooled into connecting with them, e.g. the user's private token. To combat this threat, a protocol must be developed to authenticate the identity of an arbitrary file server in the system.

The mechanism for achieving this is nearly achieved by Protocol 1 alone. Recall that Protocol 1 is used to establish a shared secret between a client and a server and to authenticate the identity of the server to the client. One of the assumptions of Protocol 1, however, was that the server has a public key k_S which is known to the world. When there is only one group server, it is reasonable to make the assumption that client applications would have knowledge of this public key. Given that new file servers may be created at any time, this assumption is no longer valid for this particular threat.

In order to use Protocol 1 as described the user must be able to first authenticate the file server as stated above. When a client initiates a connection with a file server for the first time, the file server will return its public key to the application which will then present it to the user. The user will then be able to take the public key and manually check it to verify that it belongs to the file server to which the user believes it is speaking. Once the user is satisfied in the authenticity of the file server's public key, the client can then use that public key to encrypt the first message in Protocol 1 and continue with authentication and establishment of the shared key. The client will store the file server's public key for the remainder of that session and will use it for any other connections to that server. After shutdown of the app, if the client reconnects to this server they will again be presented with the public key and asked if they are sure they want to connect. This protects against possible comprised keys of a file server so that if a file server needs to generate a new keypair it can do so and a client won't be connecting using a compromised keypair. If a file server was not using its own public key then it would not be able to decrypt the client's initial message.

Threat 4: Information Leakage via Passive Monitoring

This phase of the project assumes that there are passive attackers that will attempt to compromise information as it is in transit between clients and servers. In other words, attackers may "listen on the wire" and attempt to discover information about other users and their files. In the previous phase of the project, this would be a major issue for a secure system because all of the communication that is done between clients and servers is unencrypted. To protect against this threat, a method should be developed to prevent passive attackers from gaining any useful information from the data they receive by listening on the wire.

One method for achieving this sort of protection is to encrypt any data that passes between a client and a server using a shared secret key. Recall that Protocol 1 is used in order to establish a shared secret between a client and server. Before any other communication is done, Protocol 1 may be employed to establish a shared AES key. Afterwards, any further communication between a client and server should be encrypted using this key. Given that the key is known only to the client and server, it will be infeasible for a passive attacker to decrypt or gain knowledge from the data passing through the communication medium.

This protocol works because Protocol 1 works and because of the cryptographic strength of AES encryption. Using Protocol 1, only the client and server shall have knowledge of the key which can be used to decrypt the data being passed between the two. Given that the encryption was done using AES, a cryptographically strong encryption algorithm, it will be impossible for someone without the key (i.e. a passive attacker) to gain any information from the encrypted data.

Conclusion

Given that it was an essential part of protecting against most of the threats, Protocol 1 turned out to be especially useful. Not only does it serve to establish a shared secret between a client and server (which is used for various other purposes), but it also authenticates the server to the client. This versatile protocol was really helpful in developing a secure system for these particular threat models, and we believe that it will continue to be useful in the future.