CS 1653 Project P3

Fall 2020

Group: No such assignment

Writeup (50)

44 some details mentioned in our meeting did not get included, other flaws as mentioned

Approval in discussion (10)

10

Code (35)

35

Scheduling demo (5)

5

Miscellaneous

+6 great work!

Total

100

CS 1653 Project P3 Writeup

Dane Halle dmh148@pitt.edu | Kyle O'Malley jko12@pitt.edu | Quinnan Gill qcg1@pitt.edu

Overview

We mostly used variants of the security protocols we learned in class to overcome the threat models given to us in this project. Those include asymmetric cryptography using a variant of Diffie-Hellman key exchange using RSA, symmetric cryptography using AES in CBC mode, public key fingerprinting.

Threat T1: Unauthorized Token Issuance

The fear for this threat is that unauthorized and illegitimate clients request tokens from the Group Server. An example of this threat is a user attempting to gain access to an ADMIN token/account for malicious purposes. We need some way to authenticate and determine legitimacy of clients before giving them further access to the system by issuing them an existing token.

Being that a user account is created by an ADMIN user, we do not expect, nor do we encourage the user to tell the ADMIN their password to their account for setup purposes. We can assume some form of external communication outside of the system is reasonable due to the fact that an ADMIN will not make a user account without being prompted. We will continue under the assumption that User U will contact ADMIN A over a phone call to request an account. Over that call A will verbally prompt U for a username and then make a simple, yet unique temporary password that the User account made for U will be initialized with. A will then verbally deliver the temporary password to U. When U signs into their account for the first time with the temporary password, it will prompt them for a new, different password so that A cannot access U's account at a later point.

We determined that a Diffie-Hellman password sharing approach would be effective. The group we will be using is group 21 which is a 521-bit random elliptic curve. This provides adequate security against modern threats as well as being strong enough to encompass a 256-bit AES-CBC key. The password sharing approach we decided

to go with is Encrypted Key Exchange or EKE for short. T1.1, which is taken from our lecture notes, details how EKE works:

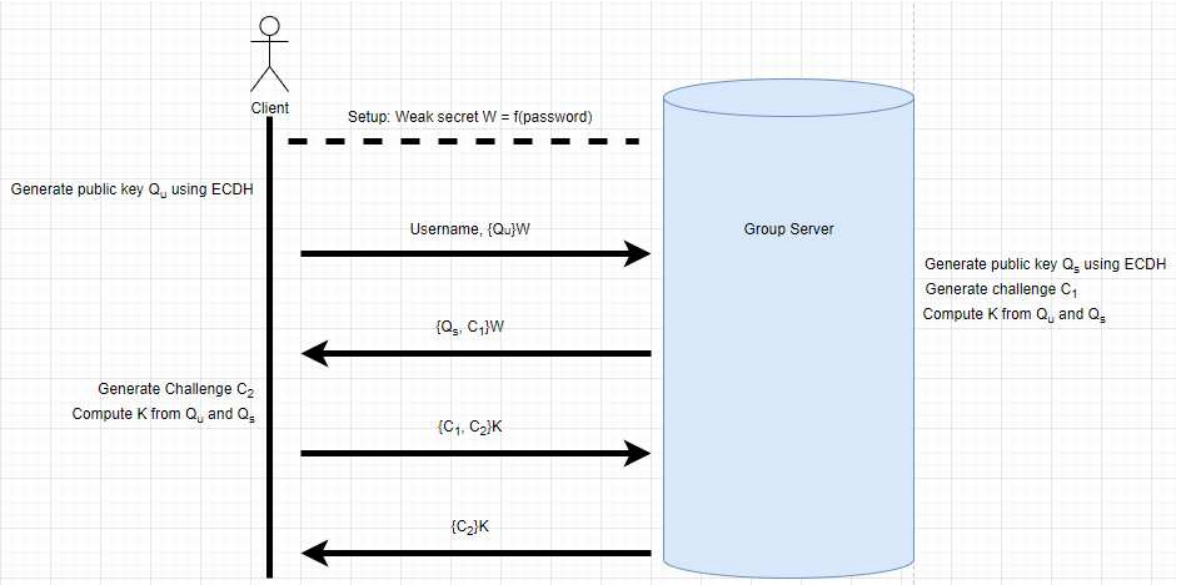

Diagram T1.1

how? 521 bit → 256?

This would work through a user providing a password after being prompted which will be used to generate a weak secret that is known between the User and the Server. That secret will be generated using the Password Based Key Derivation Function V2 (PBKDF2). Then the Client will generate some public key Q_u and send their username unencrypted and Q_u encrypted by the weak secret W to the Group Server. The Group server will then unencrypt using the W associated with the given username and generate its own public key, Q_s, some challenge C_1, and then compute K from the unencrypted Qu and the generated Q_s. The server will encrypt both Q_s and C_1 with W to send back to the client. Client will unencrypt both, generate some challenge C_2, and compute K from the unencrypted Q_s and the generated Q_u. It will then return the response to C_1 and it's own challenge C_2 encrypted with K to the Group server. The Group server will unencrypt both, determine if the response to the challenge was valid, and then send back the response to challenge C_2 encrypted with K. The client will ensure that is the proper response. If all is successful, both user and server are authenticated and this allows for secure communication for the entire instance that the client is connected to the server with K.

This key exchange works because it not only authenticates the server to the user but it authenticates the user to

the server. If another user attempts to impersonate a user, there is no practical way to ensure they can bypass the system. The imposter also does not know when they have correct information as everything looks randomized if the random b is unknown. The initializations of user accounts works because it assumes that Users will use a phone to call the ADMIN's in order to request an account where they will be given an temporary unique password generated by the ADMIN. Then the prompt for a new password after the first time signing in with the temporary password will ensure that no one other than the server and User know that secret and the ADMIN can't access information that account gains access to later. The initialization of the very first weak secret within the system is secure as the Group Server itself asks for user information instead of wanting information sent to it. When the ADMIN goes to sign in to that account, K will be established for that instance of the client and be used to encrypt every communication they make to either Group or File server. This means that creating users with temporary passwords, and thus weak secrets is a secure act.

Sources:

Diffie-Hellman Groups

Threat T2: Token Modification/Forgery

The group server and the file server are only able to communicate via UserTokens. It is assumed that users may attempt to modify or forge these UserTokens. This will provide the illegitimate users, who forged or modified UserTokens, unauthenticated access to upload, download, modify, or delete files on the file server. The attacker can do this by changing the group values in the UserTokens to be whatever group they desire. The integrity of the UserToken shall be made possible with the use of RSA signatures. These signatures provide a verification that the UserToken was in fact generated by the group server.

RSA signatures prevent users from modifying or forging UserTokens without access to the group server's private keys. When a UserToken is generated the information will be signed by a 2048-bit RSA private key. This signature will be sent along with the UserToken to provide verification for when it arrives at the file server (via the client).

hash and sign?

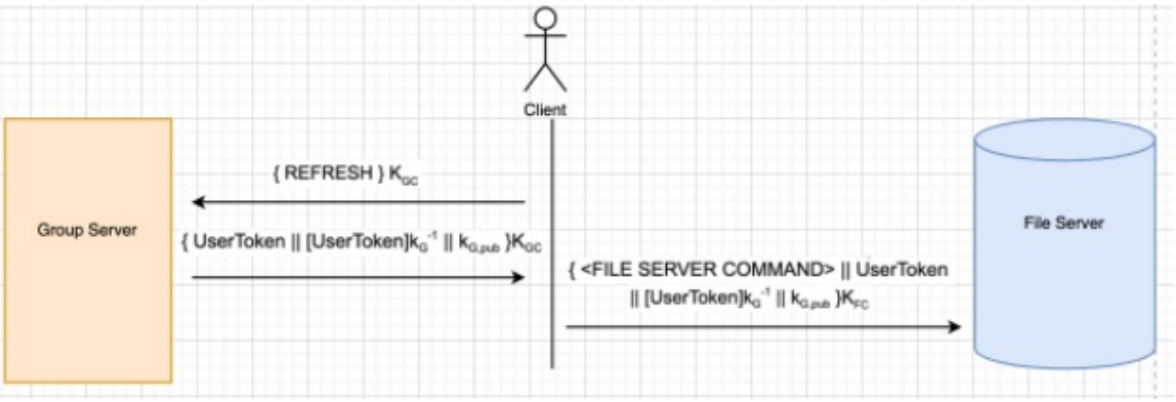

Diagram T2.1

Here it assumes there exists a method to verify that the public key being sent does in fact belong to the group server (i.e. an out of bounds request to the admin). The diagram above demonstrates how the UserToken will be used in communicating with the file server. When requested, the UserToken is sent with the UserToken's signature, signed with the file server's private key. Whenever the client performs a command the signature will provide a permission check to guarantee that the client has the correct permissions on both servers.

When the UserToken is signed, the Java will be converted into a string where almost all instance variables are added to the string. The instance variables not added to the string are the signature of the Token itself and the public key used to sign it. This is then encoded into Base64 to prevent invalid encoding of special characters. For verification it checks ~~if the Base64 encoded to~~String is the same as the signature.

The RSA signature provides integrity for the UserToken because the client will be unable to modify the UserToken without the group server's private key. If the client does end up modifying the UserToken, then it will then render the UserToken useless. A new signature cannot be generated without the group server's private key, and when the UserToken is validated it will see that the signature does not match.

Threat T3: Unauthorized File Servers

This threat is that an adversary could attempt to impersonate file server s with another file server s'. Therefore, the user must be able to authenticate the file server it is communicating with before it shares any information with it. It is assumed that once a file server is authenticated, it is to be trusted. If the user can't authenticate a file server, then there's a possibility of them connecting to an imposter file server. The ramifications of this are that

the imposter file server could then leak any files the user uploads, corrupt the files, or steal the user's token.

We can protect against this threat using a signed Diffie-Hellman encrypted key exchange using a 2048-bit RSA key pair. During the initial User/File Server Connection we will set up a secure channel (as described in T4).

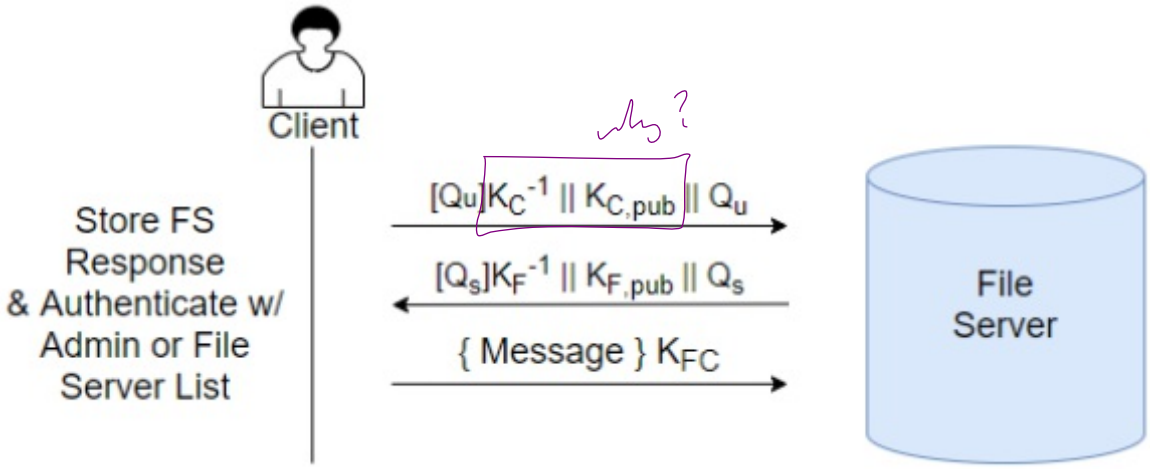

Diagram T3.1

In figure T3.1 we see the authentication process between the client and a file server. The client sends Q_u the elliptic curve public key. When the file server receives this they respond with Q_S signed with their private key, along with their public key appended at the end. During the initial connection, the client will contact the admin to verify that they have received the correct public key for the file server they want to connect to. This can be justified since to the client the admin is a trusted third party. If the public key cannot be verified, we will terminate the connection. If we can verify the public key, we will store it in a list of trusted file servers. We will then compute the verification of the signed Q_u using that public key. In following connections the authentication process will be virtually identical except that we will use the public key stored for that file server to verify that they aren't an imposter.

This approach sufficiently solves the problem of determining if the file server we are connected to is authentic or not. Without the file server's private key they will have no way to sign Q_u in a way that it will successfully be verified by the client since it is hard to factor the public key to derive the private key. The user contacting the admin to make sure they have received the correct public key for that file server, and not a fake one, ensures that a malicious file server can't just make up its one RSA key pair. It also expects the user to accept the verification

using only the public key that the admin verifies.

Sources:

How is the server authenticated in ssh

Threat T4: Information Leakage via Passive Monitoring

A passive attacker has the ability to read all communications between the client and the servers. This attacker can either be an external threat or just a nosy administrator or client. Sensitive data such as UserTokens, passwords, and commands are being sent between the clients and the servers. Access to this confidential data could lead to a malicious actor performing an impersonation attack against the system. Encryption is needed in order to deter a passive attacker from being able to interpret the contents of what is being communicated between the client and the servers.

Several encryption schemes will be used to ensure communication confidentiality. Along with what is mentioned above in T1, T2, and T3, normal communication between the client and the server's 256-bit AES-CBC with PKCS7Padding will be used to encrypt all messages. The reason for the mode is to for a handful of reasons. The first is that it is rarely prone to errors and is self-synchronizing. CTR is rarely prone to errors; however, the increment state of the initialization vector needs to be passed between the client and the servers for encryption and decryption. Another reason is that it is assumed that the messages being passed will be considerably large for a block cipher and not need a CFB 8-bit cipher. The reasoning for this is that UserToken and files that transfer will be large enough to use an entire block. Also, the UserToken will change frequently enough to provide some form of semantic security. For instructions smaller than the block size, the data will be padded.

This will work by having a wrapper around the ObjectInputStream and ObjectOutputStream to send encrypted SealedObjects. When SealedObjects is provided with the AES CBC cipher it will encrypt and send any data within the SealedObject. The only time unencrypted data will be sent is for the key exchange protocol and EKE.

This approach will provide a way to efficiently and securely transmit data. Having all of the communications be encrypted with AES will help provide a mechanism to ensure only parties with the key access to the messages. If a message is intercepted, the passive attacker will only be able to view unreadable text because of the CBC

cipher.

Sources:

Cipher Feedback Mode

What is a good AES mode to use on file encryption

Discussion

There would be quite a bit of overlap within all of the systems we are proposing. We are utilizing some form of Diffie-Hellman key exchange for all of the threats. The key generated by a Diffie-Hellman key exchange done during a GET will be utilized for encryption of communications from that point after. We started the processes of designing these mechanisms by going through each threat and spit-balling/whiteboarding ideas that could potentially work to address or solve them. After an initial look, we went back and tried to flesh out each idea. During this process, we could determine if it fell under one of the following buckets: “Good idea with potential”, “Could potentially work but has some flaws”, “Insecure or Has too many flaws to work”. We then worked further with the ideas in the first two buckets and didn’t put effort into ones that fell into the last bucket.

We discussed and whiteboard quite a few ideas prior to solidifying what we decided to go with. For T1, we talked about the potential to use some form of 2-factor authentication along with a password to increase security of the system. We talked about different types of 2-factor auth and how we could get it to work for free as most providers (like DUO or other mobile authenticators) can cost quite a bit of money. For T2, we talked about the potential to hash a User’s token within the File Server so there would be some form of validation. However, anything we would hash the token with was either insecure or “annoying” (like entering a password every time). This led us to thinking about having some talk with the group server which is the approach we ended up going with.

Extra Credit

EKE, which was implemented for T1, provides several security benefits and mitigate threats that we are not explicitly required to provide a response for. EKE provides protection against active attacks where an attacker attempts to guess the password. It also provides protection from dictionary attacks.

what kind? doesn't prevent offline attack / theft

As part of this, we had to extend the function of the GUI to encompass the new base functionality for the system. This included adding more prompts for GET and CUSER buttons in order to accept a password. Then we got the GUI to be able to prompt for a new password should the user's account have a temporary password set to it. Much like the CLI, this will prompt as many times as it takes for the user to input a different password than was previously set. We also added a way for the Client to verify keys through the GUI.