IntraGIT

A SaaS secure VCS (based on Git) for confidential repositories

Project Members:

1. Ashirwad Pradhan (2019H1120043P)
2. Vishal Singh Kushwaha (2019H1120047P)

Brief Overview:

This project provides a solution to fields of defence, confidential drug development or any other field which requires secure code development with VCS. The users can set up their own in-house git repository and host in their own network or their preferred cloud provider. The stored code/data is always in their control and no third party can read or write to it unless either the keys are compromised or access rules allow the action. All communications between the VCS server and the developers is secured by TLS.

There exist solutions like Github and Bitbucket which mainly focus on open source project development and most of the code/data is stored unencrypted. This restricts many potential users, such as those mentioned above, from using the system. Though the idea of private repositories is provided as a solution but it is often flawed as the third party is still in control of the data which is stored in their servers.

Our SaaS proposes an architecture which enables an organization to set up their own VCS based on Git. The solution is expected to have the following features.

1. A private VCS (based on Git) hosted on their own in-house infrastructure
2. The information stored in VCS should be encrypted.
3. A gateway which connects the private repository to the outside world. This gateway also can act as ACL (access control layer) to dictate which entities are authorized for what actions on the private repo.
4. The communication with gateway happens over TLS so that all data that goes through it is encrypted.
5. The variables used in encryption should be cryptographically secure PRN.

Components of the Architecture:

The solution presented by us has the following components in the architecture to build the features expected above.

1. **PKI (Public Key Infrastructure)**

   We have created our own PKI which provides the following functionalities.

   - creates and manages public keys and private keys of CA (certificate authority)

- creates and manages private keys and CSR (certificate signing request) for the server
- manages CA to sign the CSR of the server and issues server public key based on CA's private and public key

The PKI is used to create public and private keys for the **Gateway.** As this is a private solution the organization hosting the repo acts as a CA. The CA certificates are self-signed for the previously mentioned reason. This is required to set up **HTTPS** connection to the server(Gateway). **HTTPS** traffic is encrypted.

2. **Gateway**

The Gateway acts as the entry point to the server hosting the repo. All the IntraGIT clients connect to it. The Gateway provides the following functionalities.

- Registers the User
  - Registers the users and authenticates them.
    Uses **salting** to store passwords.
- Log in/out the User.
  - Act as an authentication mechanism to allow only authorized users to connect and use the service. (ACL)
- Process the create repo request
  - The repo which is created has a set of following attributes for each repo that is created.
    - admin - sets the user who is the owner of the repo
    - session_key - facilitates as a key during **envelope encryption** and when communicating with the client this key is used to encrypt the repo related data communicated to the client
    - server_random - is required during **envelope encryption**
    - users - contains a list of users who can interact with the client (ACL). Handles Authorization.
- Process the Push repo request
  - Allows only authorized users to push into a repo. The function receives encrypted data from the client which is encrypted using the **session_key** mentioned before.
  - It decrypts the encrypted data received from the client and sends it to the envelope encryption routine.
  - It then stores the envelope encrypted data into the remote **Repository**.
- Process the Pull repo request
  - Allows only authorized users to pull into the repo. The function takes the data from the remote **Repository** and then performs **envelope decryption.**
  - It then sends the data encrypted with the **session_key** to the client.

3. **KMS (Key Management Service)**

   This is the crucial part of the entire architecture. The KMS creates and manages all the keys and algorithms required for the cryptography used in our solution. Our proposed KMS is based on **HSM**. An HSM is a **hardware security module** which is responsible for key generation and management and as well as cryptographic number crunching. HSM is preferred for reasons including:

   - It's behind a secure and limited network access. It can be only accessed through **Gateway**
   - The processing of cryptographic material is very fast.
   - They provide entropy based cryptographically secure PRN.

   Our current implementation is software based due to lack of hardware.

   To generate **session_key**(aka **root key** in **envelope encryption**):

   - Takes **client_random** and **server_random** as input. They are random bits provided by client and server respectively.
   - Pass the client_random and server_random through a KDF (key derivation function) mentioned in **RFC 2898 (PKCS #5)** to get a key length of 512 bits. (Parameters used: count - 1000000, hash_module - **SHA512**)
   - Divide the key into two parts of 256bits each.
   - Pass the individual key parts into the **bcrypt** algorithm to get two different keys.
   - Now combine the above two keys and pass it to **SHA512** hash module to get a 512 bits session_key

   To generate data_key for **envelope encryption**:

   - Takes the **session_key** and **server_random** as input as generated above.
   - Put the session_key as a **paraphrase** parameter and server_random as a **salt** parameter to the **scrypt** algorithm.
   - Other parameters as suggested by ***`Colin Percival`*** in 2009 for file encryption are used. **N = 2^20, r = 8 , p = 1**
   - The key obtained from the above is put into the SHA512 hash module to get the **data_key**.

4. **Repository**

   The repository is a remote file system which keeps the data after it has been encrypted. The data in the remote repository is encrypted using **envelope encryption**.

   Our solution proposed that the **Repository** should be a VCS based on Git.

   Our **current implementation status** stands that the repository is a text file based solution.

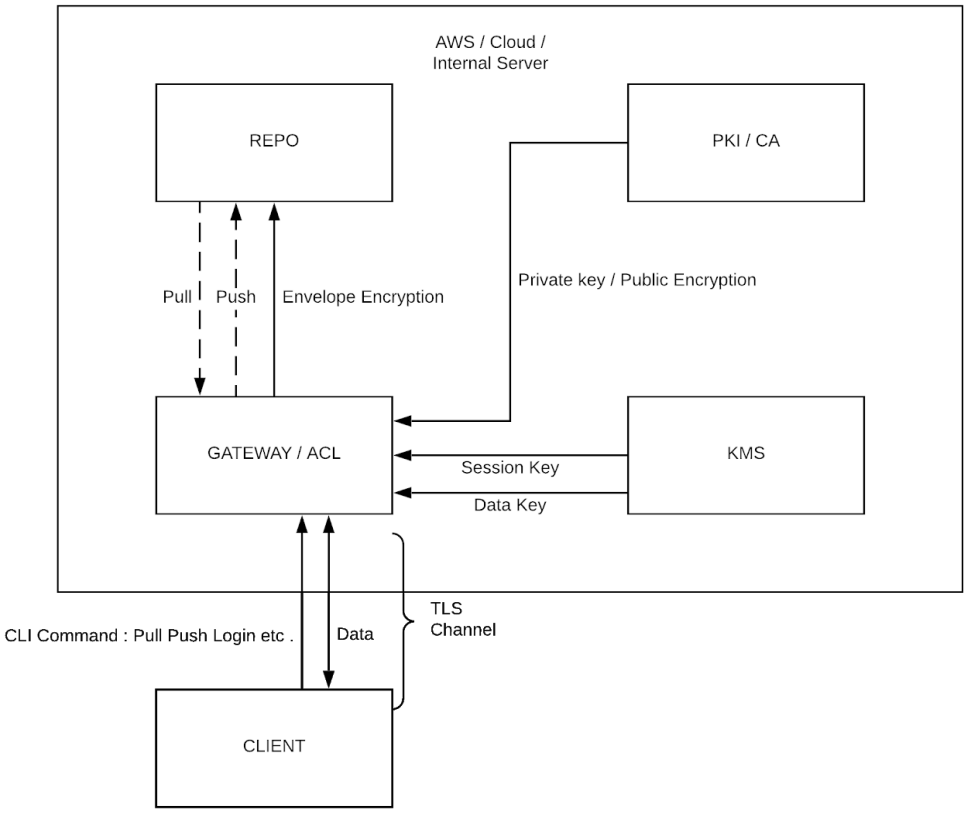
   In addition to the remote repository there is also a local repository which is the working directory of the user.

5. **IntraGIT Client**

   The IntraGIT client is the module which handles all the user's requests pertaining to the interaction with the remote repository. These are the following functions handled by the module.

- register <username>
  Registers the username with the IntraGIT service
- login <username>
  Manages login of the user to the IntraGIT service
- logout <username>
  Manages logout of the user to the IntraGIT service
- creater <repo name>
  Sends a request to the **gateway** to create a repo. Also produces a client_random which is required for generating **session_key**.
- pushr <repo name>
  Send a request to the **gateway** to push local repository changes to the remote repository. Takes the local data changes and encrypts it with the **session_key**.
- pullr <repo name>
  Send a request to the **gateway** to pull remote repository changes to synchronize the local repository. Takes the incoming data from the **gateway** and decrypts it using the session_key and stores the data in human readable form in the local repository.

Architecture Diagram:



IntraGIT Architecture

Encryption Schemes:

We have used the following encryption schemes in our architecture.

Envelope Encryption

Envelope encryption uses two or more cryptographic keys to secure a message. Typically, one key is derived from a **session key** or **root key 'Kss'**. The other key that is used is the **data key 'Kd'** which is generated to encrypt the message.
The envelope is formed by encrypting the plaintext as,

ciphertext = E(Kd, plaintext) with **ct_iv** as IV

We then go onto encrypt the data key **'Kd'** with the **session_key 'Kss'** as,

encKey = E(Kss, msgKey) with **k_iv** as IV

We then pack the two values **(encKey, ciphertext)** into a single structure, or envelope encrypted message. This is then stored in the remote repository.
The recipient, with access to **Kss**, can open the enveloped message by first decrypting the data key and then decrypting the message.Our KMS provides the ability to manage these session_key(root keys) and automate the process of envelope encryption of data.

Our implementation uses **AES256** with **CBC** mode with padding **PKCS7**
Our data is stored in following format: **| ct_iv | k_iv | encKey | ciphertext |**
All the **iv** are of 16 bytes in length and **encKey** is of 48 bytes (due to padding)

Broadcast Encryption

We are using this scheme to share the **session_key** with all the clients registered to a particular repository. The **session_key** is generated w.r.t to a repository and a **client_random** generated by the admin of the repository in addition to the **server_random.** This encryption scheme helps us to solve the problem of a collaborative repository where once a message is encrypted all the members in collaboration should be able to decrypt the message **independently**.
Our current solution implements this scheme to share remote repository data among several authorized users.

All encryption using session_key is done using **AES256** with **GCM** (Galois Counter Mode). This provides us with **AAD** (additional authenticated data) hence we can know that if the data received is tampered or not. We can verify with the tag received in the encrypted message.
The data is stored in the given format: **| nonce | tag | ciphertext |**
nonce and tag are of 16 bytes each.

Salting of password

The passwords for the users are not stored in plaintext but rather hashed. But they can still be prone to **rainbow table attacks**. By salting the passwords our solution is resistant to **brute force**, **dictionary** and as well as **rainbow table attack**.

The encryption scheme used in **KMS** is explained in the previous section.

Progress Report:

According to the document submitted earlier, we identified the following macro milestones.

1. Realize an efficient encryption scheme to encrypt the repository data.
2. Create a workflow for the multi key encryption/decryption which enables a team to work securing on the same repository simultaneously.
3. Access control layer for authorization of users attached to a repository.
4. Decide on the interaction flow between the keys mentioned above.
5. Securing the git transport protocol messages with TLS.
6. Realize a system to efficiently manage the keys.
7. Command line client tool which will enable developers to perform operations (create, clone, push, pull) to the secure repository

All the points mentioned in the green highlight (i.e. 1, 2, 3, 4, 6, 7) are **completed** as per our mid term report status.
Point mentioned in the red highlight (i.e. 5) is **not required** anymore as it is redundant. All the traffic in our implementation is flowing through **https** which is a secured channel communication.

Next Schedule:

1. **Implement the Git based Repository in place of text-file based repository as in the current implementation**
2. **Deploy the entire SaaS on Amazon EC2**
3. **Implement session_key renewal when an authorized member is removed from the repository's authorized member list or removal can be time based in broadcast encryption scheme.**

The above two milestones are **in progress** and are expected to get completed before final presentation.

Security Considerations:

Security aspects that **can** be guaranteed are as follows

1. Communication between **IntraGIT** client and **Gateway** is encrypted using **TLS** as they communicate over **HTTPS**. This protects from **man-in-the-middle** attacks.
2. The **Gateway** also acts as an **ACL** therefore providing **Authentication** and **Authorization**.
3. All encryption of data between client and gateway is with session_key with AES256 with GCM mode which provides us with **AAD**. Therefore **data integrity** is protected.
4. **Confidentiality** of data is maintained by the use of both **symmetric** and **asymmetric** encryption.
5. Salting of passwords protects us from **rainbow tables attack** even if our user data is compromised.
6. The KMS which is based on **HSM** also provides us **cryptographically secure PRN** which serves our purpose.
7. The repository data is secure until **session_key** is secure.

8. If the some user goes rogue then he can be immediately removed from the authorized users lists and the session_key for the repo is renewed. (After implementing point no. 3 mentioned in the previous section in yellow)

Security aspects that **cannot** be guaranteed

1. If the session_key is compromised then the whole solution is compromised. The client therefore is **never** in charge of **session_key**. The session_key is provided to the client on demand and it doesn't know the renewal pattern.
2. If the KMS is compromised, then HSM can be prone to **timing attacks** or other **applied cryptanalysis attacks**.
3. If the client password is compromised through **social engineering attack** then also the system is compromised.

Project Repository

The project repository can be found at https://github.com/AshirwadPradhan/intragit
All the necessary steps to set up the service can be found there.