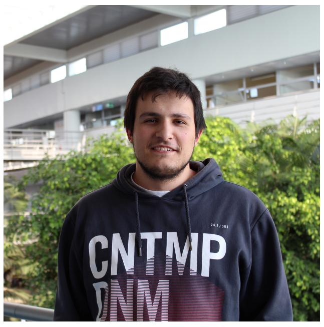
Automatic Vulnerability Detector
SIRS

Last updated on December 2019

Campus TagusPark

Group T13



Inês Albano 87664
ines.albano@
tecnico.ulisboa.pt

Miguel Oliveira 87689
moreira.oliveira@
tecnico.ulisboa.pt

Viviana Bernardo 87709
vivianabernardo@
tecnico.ulisboa.pt

1 Problem

The program in development stores and receives data from group members, which means there is a need for authentication on the behalf of each user. Only the leader of the group should be able to visualize the information stored in a data base relative to the vulnerabilities exploited and fingerprints submitted.

1.1 Requirements

- Confidentiality - sensitive data, such as the exploited vulnerabilities and the fingerprints. Guarantee that only the leader should be able to see the scoreboard and the exploits of each member.
- Integrity - to ensure that the contents should not change state, which means unauthorized people can not change it. For example the SQL databases can not be dropped or deleted. Ensure that the data exchanged between the machines is not changed.
- Authenticity - assurance that the vulnerability and fingerprint were submitted by one person in particular, this involves proof of identity.
- Freshness - to avoid rediscovery of the same vulnerability.
- Availability - guarantee that the information is always available, because we trust the server is always up and running.

1.2 Trust assumptions

- There are no loss of data when a team member submits the vulnerabilities exploited and fingerprints.
- The server is always up and running.
- The programs that the vulnerability detector is analysing does not fail.
- The certificates emitted by the authentication server are always trustworthy (self-signed certificates).
- The public key sent by the server (used in custom protocol) is always trustworthy and server cannot be spoofed.
- The network between the host and the machines is ignored as a security problem because in a real system this network wouldn't exist.

2 Prepared Solution

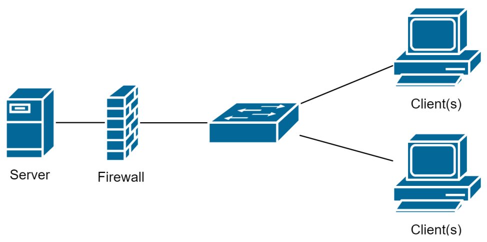
2.1 Deployment

Only three virtual machines (VMs) are needed: one to run the server for both the website and the custom protocol, one for the firewall and one for the client to submit or see the scoring board (being the leader or not). The server and the client will communicate over https for the website and costume protocol for the vulnerability submission system. The firewall protects the server network of any connection that is not https or custom protocol. It is important to note that the firewall only protects the server, the client still has access to the internet. The VMs will communicate over a isolated network. The communication between host and VMs will be done over another network.

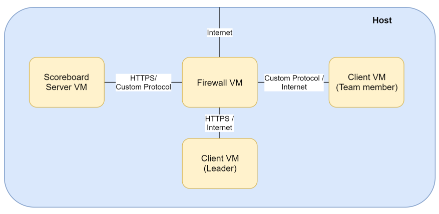
2.2 Secure channels and protocols

The three VMs will have a connection to the host system via SSH. The communication between the server and the client to see the website is done via HTTPS. When submitting a vulnerability, the client will communicate to the scoreboard server via a custom secure protocol:

1. The server sends the certificate to the client.
2. The client extracts the public key from the certificate, this public key will work as a KEK.
3. The client generates a symmetric key and encrypts it with the public key.
4. The server decrypts the message from the client with the private key and stores the symmetric key of the client.
5. The client and the server communicate using the symmetric key to encrypt the messages.



(a) Network diagram



(b) Machines diagram

Figure 1: Project infrastructure

3 Results

sha-256 When submitting a vulnerability, the communication will be done through a socket that is available for 2 seconds and starts when a client wants to establish a connection. After this the server sends the self-signed certificate (notice that if this was a real system, we would have a CA signing the server entity, but for reasons of simplification we are using a self-signed certificate assuming the server is trustworthy). The client will extract the public key from the certificate and generate a symmetric key. It will then send that symmetric key encrypted with the public key. As the server is the only entity that has the private key, it will also be the only one able to decrypt the symmetric key generated by the client. After the acknowledgment of the received symmetric key, all the communication will the done encrypted with that key. Then the client has to log in, submitting the user name and its password. After the user is accepted, it can send the vulnerability found with the fingerprint and the explanation. This message contains the vulnerability encrypted with the symmetric key and that first part of the message hashed with SHA-256 to ensure integrity. Finally the server will answer with the number of points the client has and then close the connection.

Implementation notes

- We used vagrant to help setting up the project in different computers, because with vagrant we just need to configure the files with the setup we want and it will create the machines.

- We choose a self-signed certification to simplify the implementation, although in reality we would need a CA to sing our certificate. We used certificates in order to prove the trustworthiness of an entity (server) and avoid man in the middle.

- Every time a client wants to send a new found vulnerability, a new connection is created in a new socket. This sockets is only opened for 2 seconds, and if the information is not sent within this time frame, the connection closes and the client needs to make a new request and generate a new symmetric key. Therefore we didn't find necessary to use a timestamp in order to prove the clients identity, because even if a packet is intercepted by an attacker, he would need to use brute force in order to falsify the client's identity which is very unlikely for the time frame the user has to complete the submission of the vulnerability.

- We decided to use a combination between asymmetric and symmetric keys, because:

  1. We used the public key as a KEK in order to establish a connection server-client and share the symmetric key between them.

2. We used the symmetric key to encrypt all the data shared between the server and the client.

This way we ensured the authenticity of the client, because for each connection there is only one symmetric key generated and this key is only shared between the server and the client for the max of 2 seconds. We also guarantee confidentiality for the same reason. To ensure integrity for the vulnerability message we also send that encrypted message hashed with SHA-256.

We were able to achieve:

- Secure communication between client and server through the custom protocol. Taking advantage of the existence of public/private keys and symmetric keys.
-
- Secure communication between the leader and the website due to the usage of https.
- Support of XSS attacks, because of the sanitization of the variables displayed on the screen with *htmlspecialchars.*
- Decrease of DDoS attacks due to the implementation of a firewall, once it filters all the packets directed to the server and the default rule is to drop the packets apart from the ones from https (443) and the custom protocol (4347).
- Protection against man in the middle attacks when submitting vulnerabilities by using a certificate.
- Assure the integrity of the message that has the vulnerability by sending it with an SHA256 hash of itself.
- Protection against SQL Injection attacks, due to the usage of prepared statement, this way the attacker can't inject any code into the DB nor delete them.

Weaknesses that can be found:

- Our program allows Replay Attacks, meaning the packets can be intercepted and stored and later send all in the same order and the program will allow it. Even though this is a possibility, the program does not allow the submission of the same vulnerability for the same. The actual Replay Attack will only be possible if the attacker changed the credentials within the packets.

- Since our server is critical, if it becomes compromised, we would need to restore its previous state by hand.
- Our program doesn't guaranty Perfect Forward Secrecy, meaning that if an attacker discovers the private key it can read all the previous packets sent, discovering all the vulnerabilities submitted and all the users passwords. Even thought this is possible the probability of an attacker discovering the private key is very low because the difficulty of discovering it is extremely high due to the fact that the key pair is only used as a KEK and doesn't encrypt data, only keys.

4 References

References

[1] Angr: Python framework.
`https://angr.io/`