deti universidade de aveiro
departamento de eletrónica,
telecomunicações e informática

Segurança Informática e nas Organizações

Authentication and Access Control

António Gomes Nº 80124
Carlos Marques Nº 84912

December 14, 2019

Contents

List of Figures

Chapter 1

Introduction

1.1 Introductory Note

This report was developed by António Gomes and Carlos Marques for the SIO (Segurança Informática e nas Organizações) subject having as the main goal explore the concepts related with the establishment of a secure session between two parties. Also explores concepts related with the authentication of these parties in the communication and the control of client access. This document contains all the information on the design and implementation of a protocol that allows secure communication (confidential and complete) between two points with mutual authentication with the help of the code created in the previous project (SIO Project 2).

1.2 Abstract

Over the last decades, the Internet has been growing more and more, now at an incredible rate making cyber security an extremely necessary subject.

Cryptography helps us, the programmers, to encrypt the data in various different ways with the objective of hide this data from possible attackers.

Authentication with the citizen card and/or X.509 protocols are a good way to start building a secure session between two points. But how it works?

Chapter 2

Access Control

Access Control is a security technique that regulates who or what can view or use resources in a computing environment. [1] This minimizes the risk of hacker attacks and data leak which is fundamental for any business or organization.

In our implementation we use Access Control in both parties - client and server. So both client and server send each other a control challenge to ask for access and also each one process this request to check if access is granted or not.

2.1 Client

Our implementation has for the client access control two function, on in the client side and other in the server side for, respectively, sending a challenge to check if the server can access the client and to process this challenge sent by the client.

The following figure shows the function from the client side with the job of sending a challenge(message of type CLIENT-ACCESS-CONTROL) to the server based on the client ID number of his Citizen Card and also a nonce - an arbitrary number.

```
def send_client_acess_control(self):
    """
    Client sends a challenge to check if the server can access client.
    Sends a challenge with the client's CC id number that is read from
    the CC and a nonce value
    """
    serial_number = self.citizen_card.get_id_number()
    nonce = os.urandom(12).decode('iso-8859-1')
    challenge = security.challenge_serial_number(serial_number,nonce)
    message = {"type": "CLIENT_ACCESS_CONTROL", 'challenge':challenge.decode('iso-8859-1'),'nonce':nonce}
    self._send(message)
    self.state = STATE_CLIENT_ACCESS_CONTROL
```

Figure 2.1: Client access control request

When the server receive the message of type CLIENT-ACCESS-CONTROL from the client, the following function is called. This function job is to verify if both challenges match in order to give the client access. A message of type OK is sent if everything goes right.

```
def process_client_access_control(self,message) -> bool:
    """
        Server verifies if client can access the server
        Solves the challenge and compares with the one
        sent by the client
    """
    self.state = STATE_CLIENT_ACCESS_CONTROL
    challenge = message['challenge'].encode('iso-8859-1')
    nonce = message['nonce']
    if security.verify_challenge_serial_number(self.allowed_users,nonce,challenge) != True:
        return False
    message = {'type':'OK'}
    self._send(message)
    return True
```

Figure 2.2: Process client access control

2.2 Server

A similar thing is done in the server control access. It also has two functions, one in each party. In the server we create a challenge that will be sent to the client. This challenge uses the server's own certificate fingerprint, hashes this and sends to the client with a message of type SERVER-ACCESS-CONTROL.

```
def send_server_access_control(self) -> None:
    """
        Sends an access control challenge to the client
        It uses its own certificate fingerprint and generates a digest
        that is sent to the client.
    """
    server_cert = security.load_cert('server_cert.pem')[0]
    challenge = security.hash_fingerprint(server_cert)
    message = {'type':'SERVER_ACCESS_CONTROL','digest':challenge}
    self.state = STATE_SERVER_ACCESS_CONTROL
    self._send(message)
```

Figure 2.3: Server access control request

In the client side, we check if the digest is the same, e.i., verify if the hashes with the digest given from the client and the certificates fingerprint are equal. A message of type OK is sent and True is returned if this happens, otherwise we return False.

```
def process_server_access_control(self,message) -> bool:
    """
        Client verifies if the server can access the client
        Checks if the the digest sent by the server is
        a product of the hash of server cert fingerprint
    """
    self.state = STATE_SERVER_ACCESS_CONTROL
    digest = message['digest']
    if security.verify_hashes(digest,self.cert_fingerprints) != True:
        return False
    message = {'type':'OK'}
    self._send(message)
    return True
```

Figure 2.4: Process server access control

Chapter 3

Server Certificates Authentication

First we generated generate a random value that is going to be the content to be signed and load both the Server private key that only the Server has access to, With the private key using an algorithm we sign the content.

The certificate is loaded, serialized and decoded and is sent to client along with signature and the content.[Figure 3.1] The client now has to authenticate the server and to this it loads the signature, the content and the server certificates. Creates the certificate chain using its own trusted certificates and the server certificate.

The chain goes throw a validation process using an algorithm[Figure 3.3] (that checks if the certificates are valid and if the chain is also well built) and the signature is also verified. If both of these are True the server is authenticated.

```
def send_x509_server_authentication(self) -> bool:
    self.private_key = security.loadPrivateKey('server_privkey')
    #content to signed
    content = os.urandom(12)
    #sign content private key
    signature = security.sign(self.private_key,content)
    #send server certificate
    server_cert = security.load_cert('server_cert.pem')[0]
    server_cert = security.serialize(server_cert).decode('iso-8859-1')
    message = {'type':'SERVER_CERT_AUTH','signature': signature.decode('iso-8859-1'),
     'content':content.decode('iso-8859-1'),'server_cert':server_cert}
    self._send(message)
    self.state = STATE_SERVER_CERT_AUTH
    return True
```

Figure 3.1: X.509 Authentication Request to the Server - Code

```
def process_x509_server_authentication(self,message: str) -> bool:
    #server cert priv_key signature
    signature = bytes(message['signature'], encoding='iso-8859-1')
    #content that was signed by server cert priv_key
    content = message['content'].encode('iso-8859-1')
    #server certificate
    certificate = message['server_cert'].encode('iso-8859-1')
    certificate = security.deserialize(certificate, security.load_pem_x509_certificate)
    self.server_pub_key = certificate.public_key()
    #load trusted_certificates
    trusted_certificates =  security.load_cert('PTEID.pem') + security.load_cert('ca.pem')
    #build certification chain
    chain = security.build_certification_chain(certificate,trusted_certificates)
    #verify certification chain
    if security.valid_certification_chain(chain,[{
        'KEY_USAGE': lambda ku: ku.value.digital_signature and ku.value.key_agreement
    }] + [{
        'KEY_USAGE': lambda ku: ku.value.key_cert_sign and ku.value.crl_sign
    }] * 3, check_revogation = [ True ] * 3 + [ False ]) != True:
        return False

    #verify signature
    if security.verify(certificate,signature,content) != True:
        return False

    message = {'type': 'OK'}

    self._send(message)

    return True
```

Figure 3.2: Server X.509 Chain and Signature verification - Code

```
def valid_certification_chain(certification_chain, vkwargs, backend = backend, check_revogation = None):
    """
    Validates a certification chain
    """
    if check_revogation is None:
        check_revogation = [ True ] * len(certification_chain)
    if not all([ (not to_revokate or revogation_status(certificate, backend) == 0)
        and not_expired(certificate) and valid_attributes(certificate, kwargs)
        for certificate, kwargs, to_revokate in zip(certification_chain, vkwargs, check_revogation) ]):
        return False
    for i in range(len(certification_chain) - 1):
        try:
            certificate0, certificate1 = certification_chain[i], certification_chain[i + 1]
            certificate1.public_key().verify(certificate0.signature, certificate0.tbs_certificate_bytes,
            asymmetric.padding.PKCS1v15(), certificate0.signature_hash_algorithm)
        except cryptography.exceptions.InvalidSignature:
            return False
    if certification_chain[-1].subject == certification_chain[-1].issuer:
        try:
            certificate0, certificate1 = certification_chain[-1], certification_chain[-1]
            certificate1.public_key().verify(certificate0.signature, certificate0.tbs_certificate_bytes,
            asymmetric.padding.PKCS1v15(), certificate0.signature_hash_algorithm)
        except cryptography.exceptions.InvalidSignature:
            return False
    return True
```

Figure 3.3: Certificate chain validation - Code

Chapter 4

Citizen Card Authentication

To help us with the Citizen Card Authentication, we used a specific cryptography.io X.509 library - Object Identifiers [7].

The client starts by sending the Citizen Card Authentication request to the Server. The client creates a signature with a random value content using the CC private key(using a CC reader), loads the certificates in the CC and sends all of them server. The server processes the Citizen

```
def send_citizen_card_auth(self) -> None:
    #read Citizen card
    self.citizen_card = security.CitizenCard()
    security.store_public_key(self.citizen_card.get_public_key(),"client")
    #content to be signed
    content = os.urandom(12)
    #sign content private key from citizenCard
    self.signature = self.citizen_card.sign(content)[0]
    signature = bytes(self.signature)
    #need to send certificate chain
    chain = self.citizen_card.get_x509_certification_chains()[0]
    certificates = [security.serialize(certificate).decode('iso-8859-1') for certificate in chain]
    message = {'type': 'CITIZEN_CARD_AUTH', 'signature': signature.decode('iso-8859-1'),
     'content':content.decode('iso-8859-1'), 'certificates':certificates}
    self._send(message)
    self.state = STATE_AUTH_CARD
```

Figure 4.1: Citizen Card Authentication Request - Code

Card Authentication request (the message received from the client). The server loads the self-signed trusted certificates that are available and loads the certificates sent by the client. With these certificates the server creates a chain that is gonna be validated first and then the signature is also verified if both validations are correct the client is authenticated.

```
def process_citizen_card_authentication(self,message) -> bool:
    self.state = STATE_CLIENT_CARD_AUTH
    #client citizen card signature
    signature = bytes(message['signature'], encoding='iso-8859-1')
    #content that was signed by citizen card
    content = message['content'].encode('iso-8859-1')
    #load trusted_certificates
    trusted_certificates =   security.load_cert('PTEID.pem') + security.load_cert('ca.pem')
    #load client certificates
    certificates = message['certificates']
    certificates = [ cert.encode('iso-8859-1') for cert in certificates]
    certificates = [ security.deserialize(certificate, security.load_pem_x509_certificate)
                    for certificate in certificates]
    #build certification chain
    chain = security.build_certification_chain(certificates,trusted_certificates)
    #verify certification chain
    if security.valid_certification_chain(chain,[{
        'KEY_USAGE': lambda ku: ku.value.digital_signature and ku.value.key_agreement
    }] + [{
        'KEY_USAGE': lambda ku: ku.value.key_cert_sign and ku.value.crl_sign
    }] * 3, check_revogation = [ True ] * 3 + [ False ]) != True:
        return False
    #verify signature
    if security.verify(certificates[0],signature,content) != True:
        return False
    message = {'type': 'OK'}
    self._send(message)
    return True
```

Figure 4.2: Citizen Card Authentication Process and Response - Code

Chapter 5

Challenge-Response Authentication

5.1 OTP - One Time Password

One time password (OTP) is a password that is valid for only one login session or transaction. In contrast to static passwords, OTP's are not vulnerable to replay attacks, which is the most important advantage and we use it in our scenario for that same reason. [4]

The server starts by sending a request to the client to authenticate [Figure 5.1].

The client solves the otp challenge using the arguments sent by the client and the shared password between the two. Encrypts the solution with server cert public key and sends it back to the server to process [Figure 5.2].

The server decrypts the solution, solves it as well and compares both. Generates a new random index value for further authentications [Figure 5.3].

```
def send_otp_authentication(self) -> bool:
    """
    Sends an one time password authentication
    request with some arguments used for to
    solve the challenge.
    """
    message = {'type':'OTP_AUTH','indice':None ,'raiz':None}
    message['indice'] = self.index
    self.raiz = security.generate_raiz()
    raiz = self.raiz.decode('iso-8859-1')
    message['raiz'] = raiz
    self._send(message)
    self.state = STATE_AUTH_OTP
    return True
```

Figure 5.1: OTP - Request - Code

```python
def send_otp_solution(self,message: str) -> None:
    """
    Send client otp authentication solution
    to the server
    """
    raiz = message['raiz']
    indice = message['indice']
    solution = security.otp(index= indice-1,root= raiz, password=self.password)
    solution = security.encrypt(self.server_pub_key,solution)[0]
    message = {'type':'OTP_AUTH', 'solution':solution.decode("iso-8859-1") }
    self._send(message)
    self.state = STATE_AUTH_OTP
```

Figure 5.2: OTP - Solve Solution - Code

```python
def process_otp_authentication(self,message) -> bool:
    """
        Returns true if the attemp sent by the
        client corresponds to the solution expected
    """
    solution = message['solution'].encode('iso-8859-1')
    solution = security.decrypt(self.private_key,solution)[0]
    if solution == security.otp(index=self.index,root=self.raiz,password=self.password) != True:
        return False
    message = {'type':'OK'}
    #generate new random index for next authentication
    self.index = random.randint(3,9)
    self._send(message)
    return True
```

Figure 5.3: OTP - Process the Solution - Code

```python
def otp(index,root,password,data=None):
    cont = 0
    if data != None:
        return hash(data=data)
    data = (password + str(root)).encode("utf8")
    while cont != index -1:
        result = hash(data=data)
        data = result
        cont+=1
    return data
```

Figure 5.4: OTP - Algorithm - Code

5.2 CHAP- Challenge-Handshake Authentication Protocol

Server creates a challenges and sends it over to the client to solve [Figure 5.4].

Client solves the challenge: hashes the challenge with a nonce value and a password (shared secret). Encrypts the solution of the challenge using the public key from the server certificate previously exchanged with the nonce value [Figure 5.5].

Finally the server has to verify if the challenge was correctly solved. To do this the server decrypts the clients solution, solves the challenge itself and then compares both and checks whether they are the same or not.[Figure 5.6]

```
def send_challenge(self) -> bool:
    """Create challenge and send
        it to the client
    """
    self.challenge = security.create_challenge()
    message = {'type':'CHAP','challenge':self.challenge.decode('iso-8859-1')}
    self._send(message)
    self.state = STATE_CHAP
    return True
```

Figure 5.5: CHAP - Create Challenge - Code

```
def send_challenge_solution(self,message: str) -> None:
    """
    Send client CHAP challenge authentication
    solution to the server
    """
    challenge = message['challenge'].encode('iso-8859-1')
    nonce = os.urandom(12).decode("iso-8859-1")
    solution = security.solvePasswordChallenge(self.password,challenge,nonce)
    solution = security.encrypt(self.server_pub_key,solution)[0]
    message = {'type':'CHAP','nonce':nonce,'solution':solution.decode("iso-8859-1")}
    self._send(message)
    self.state = STATE_CHAP
```

Figure 5.6: CHAP - Challenge Authentication Solution - Code

```
def process_challenge(self,message) -> bool:
    """
        Returns true if the attemp sent by the
        client corresponds to the solution
    """
    nonce = message['nonce']
    solution = message['solution'].encode('iso-8859-1')
    solution = security.decrypt(self.private_key,solution)[0]
    if security.verifyPasswordChallenge(self.password,self.challenge,nonce,solution) != True :
        return False
    message = {'type': 'OK'}
    self._send(message)
    return True
```

Figure 5.7: CHAP - Process the Challenge sent by the Client - Code

Chapter 6

Conclusion

The internet is full of dangers, hacker-vulnerable data that can lead to catastrophic events. Normal users should not have many permissions. Every permission given is a privilege and we need to give as few privileges as possible in order to keep our "home" safe by filtering unwanted users. Our best to do is to create authentication methods for being able to identify different types of users and maybe their intentions.

In our implementation to create a secure session between a client and a server with mutual authentication we started with the planning on how authentication in the communications would happen - by challenge - response mechanisms. After that we needed to think about the access control, i.e., if a specific user could or not transfer files. In the Challenge-Response authentication we used OTP and CHAP. Finally, we made use of X.509 Certificates and Citizen Card Authentication, both implementations explained throughout this document.

We hope in the future we can take all the knowledge acquired during the development of this project and apply it in a "real world" case and that we can keep our data safe on the Internet.

Chapter 7

Source Code

https://github.com/showzen/SIO_project_2.git

References

[1] Access Control https://searchsecurity.techtarget.com/definition/access-control

[2] How to Deliver Public Keys with X.509 Digital Certificates [Available online at] https://www.youtube.com/watch?v=_IF5JQUkUgQ

[3] X.509 [Available online at] https://en.wikipedia.org/wiki/X.509

[4] One-time password [Available online at] https://en.wikipedia.org/wiki/One-time_password

[5] Whats is an OTP? [Available online at] https://www.cm.com/blog/what-is-otp-one-time-password/

[6] Challenge-Handshake Authentication Protocol [Available online at] https://en.wikipedia.org/wiki/Challenge-Handshake_Authentication_Protocol

[7] cryptography.x509.oid [Available online at] https://cryptography.io/en/latest/_modules/cryptography/x509/oid/