deti universidade de aveiro
departamento de eletrónica, telecomunicações e informática

SIO: relatório final - Comunicações Seguras

Aluno: Vinícius Benite Ribeiro (82773)
Docentes: Professor João Paulo Barraca e Professor Vitor Cunha.

Conteúdo

1 Introdução

1.1 Contexto da aplicação

Este trabalho prático visa estabelecer uma conexão segura entre um servidor - cliente para transferência de ficheiros. Explora conceitos relacionados com a troca de chaves, cifras simétricas, controlo de integridade, autenticação e autorização.

1.2 Limitações

- Não foi demonstrada a presença de uma vulnerabilidade e resultante mecanismo de confinamento;
- Não foi implementado a verificação da cadeia de certificados;
- Não foi implementado outros mecanismos relevantes para a segurança, como exemplo, autenticação por Cartão do Cidadão e rotação de chaves.

2 Descrição geral

Esta aplicação consiste nos seguintes mecanismos/etapas:

- Estabelecimento de uma conexão TCP entre cliente - servidor;
- Negociação dos algoritmos a serem usados;
- Troca de chaves;
- Cifra e decifra;
- Verificação de integridade (MAC);
- Autenticação dos clientes (password);
- Autenticação do servicor (x509)
- Autorização.

3 Implementação

3.1 Negociação

É escolhido aleatoriamente qual algoritmo será usado (3DES ou AES), qual o modo (ECB ou CBC) e qual função de síntese (SHA-256 ou MD5).

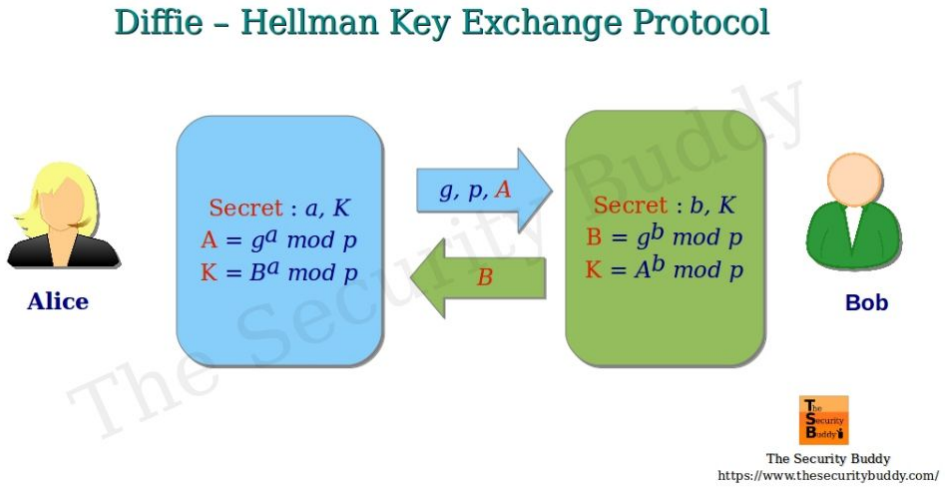
```
def do_agreement(self) -> None:
        self.algorithm = self.protocols.get('algorithms')[random.randint(0, 1)]
        self.mode = self.protocols.get('modes')[random.randint(0, 1)]
        self.hash_function = self.protocols.get('hash_functions')[random.randint(0, 1)]

        if self.algorithm == 'AES':
            self.iv = os.urandom(16)
        elif self.algorithm == '3DES':
            self.iv = os.urandom(8)

        msg = {'type': 'AGREEMENT',
                    'algorithm': self.algorithm,
                    'mode': self.mode,
                    'hash_functions': self.hash_function,
                    'iv': base64.b64encode(self.iv).decode('utf-8')}

        self.send(msg)

        self.state = STATE_KEYEX
```

Função de negociação no cliente

```
def process_agreement(self, message: str) -> bool:
        self.state = STATE_AGREEMENT

        # Definir alg
        if message['algorithm'] not in self.protocols['algorithms']:
            logger.info('Algorithm not found!')
            return False
        self.algorithm = message["algorithm"]

        # Definir modo
        if message['mode'] not in self.protocols['modes']:
            logger.info('Mode not found!')
            return False
        self.mode = message["mode"]

        # Definir sinstese
        if message['hash_functions'] not in self.protocols['hash_functions']:
            logger.info('hash_function not found!')
            return False
        self.hash_function = message["hash_functions"]

        # Definir iv
        self.iv = base64.b64decode(message['iv'])

        logger.info("NEGOTIATION ONGOING. RECEIVED FROM CLIENT:")
        logger.info( "Algorithm -> {}, mode -> {}, hash function -> {} and iv ->
{}".format(self.algorithm, self.mode, self.hash_function, self.iv))
        message = {'type': 'AGREEMENT_OK'}
        self.send(message)
        # Advance state
        logger.info("ADVANCING TO STATE_KEYEX")
        self.state = STATE_KEYEX
        return True
```

Função de negociação no servidor

Em seguida, temos a troca de chaves para cifra e decifra das mensagens. Para tal, utilizamos Diffie-Hellman.



Protocólo Diffie-Hellman

São geradas a chave privada (tendo em consideração primo e o coprimo) e a chave pública (a partir da privada) do cliente. É enviado para o servidor o p, g e a sua chave pública.

```
        def do_keyexchange(self):
                logger.info("STATE: KEYEX")

                # do key exchange
                params = dh.generate_parameters(generator=2, key_size=512,
backend=default_backend())

                # Private key
                self.private_key = params.generate_private_key()

                # Public key
                self.public_key = self.private_key.public_key()

                public_key_bytes = self.public_key.public_bytes(serialization.Encoding.DER,
                                                serialization.PublicFormat.SubjectPublicKeyInfo)

                # Prime numbers
                p = params.parameter_numbers().p
                g = params.parameter_numbers().g

                msg = {'type': 'KEYEX',
                'public_key': base64.b64encode(public_key_bytes).decode('utf-8'),
                'p': p,
                'g': g}
                self.send(msg)

                data = self.read()
                logger.debug("KEYEX got -> {}".format(data))
                mtype = data.get('type', 'UNKNOWN')
                if mtype != 'AGREEMENT_OK':
                    raise Exception("MSG != SERVER_PUBLIC_KEY")

                # Advance state
                logger.info("Advancing to STATE_AUTHN")
                self.state = STATE_AUTHN
```

Key exchange no cliente

No servidor p e g são utilizados para gerar a chave privada e consequentemente a chave pública. É calculada a chave partilhada a partir da privada do servidor com a pública do cliente. É enviada para o cliente a chave pública do servidor.

```
        def process_keyex(self, message: dict) -> bool:
                if self.state != STATE_KEYEX:
                    logger.warning("Invalid state. Discarding")
                    return False
                logger.info("STATE: KEY_EXCHANGE")



                # do key exchange
                # Primes
                p = message['p']
```

```
            g = message['g']
            params_number = dh.DHParameterNumbers(p, g)
            params = params_number.parameters(default_backend())

            # Private key
            self.private_key = params.generate_private_key()
            # Public key
            self.public_key = self.private_key.public_key()

            client_pub_key_bytes = base64.b64decode(message['public_key'])
            client_pub_key = serialization.load_der_public_key(
                client_pub_key_bytes, backend=default_backend())

            shared_key = self.private_key.exchange(client_pub_key)

            # Derivation
            self.key = HKDF(algorithm=hashes.SHA256(), length=16, salt=None,
info=b'derivation', backend=default_backend()).derive(shared_key)

            server_pub_key_bytes = self.public_key.public_bytes(serialization.Encoding.DER,
serialization.PublicFormat.SubjectPublicKeyInfo)
            msg = {"type": 'SERVER_PUBLIC_KEY', 'server_public_key':
base64.b64encode(server_pub_key_bytes).decode('utf-8')}

            self.send(msg)
            # In the last message of this process advance the state
            logger.info("ADVANCING TO STATE_AUTHN")
            self.state = STATE_AUTHN
            return True
```

Key exchange no servidor

Em seguida, o cliente irá gerar uma chave compartilhada através da chave pública do servidor e enviar para o servidor. Para a derivação dessa chave usei HKDF.

```
    def do_authenticate(self) -> None:
            logger.info("Begining authn process")
            # Gen key
            d = self.read()

            b_server_pub_key = base64.b64decode(d['server_public_key'])
                        server_pub_key   =   serialization.load_der_public_key(b_server_pub_key,
backend=default_backend())
            shared_key = self.private_key.exchange(server_pub_key)

            # Derivation
            self.key = HKDF(algorithm=hashes.SHA256(),
            length=16,
            salt=None,
            info=b'derivation',
            backend=default_backend()).derive(shared_key)
```

Geração da chave compartilhada pelo cliente

3.2 Cifra e decifra

Para a cifra de mensagens é utilizado o algoritmo, modo e a função de síntese definidos anteriormente na negociação.

```
def encrypt(self, message: dict) -> dict:
        cipher = None
        block_size = 0
        # Algorithm
        if self.algorithm == 'AES':
            alg = algorithms.AES(self.key)
            block_size = alg.block_size
        elif self.algorithm == '3DES':
            alg = algorithms.TripleDES(self.key)
            block_size = alg.block_size
        # Mode
        if self.mode == 'ECB':
            modo = modes.ECB()
        elif self.mode == 'CBC':
            modo = modes.CBC(self.iv)
        cipher = Cipher(alg, modo, backend=default_backend())
        # Synthesis
        if self.hash_function == 'SHA-256':
            sintese = hashes.SHA256()
        elif self.hash_function == 'MD5':
            sintese = hashes.MD5()
        encryptor = cipher.encryptor()
        # Convert to base64
        msg = base64.b64encode(message)
        padder = padding.PKCS7(block_size).padder()
        p_data = padder.update(msg) + padder.finalize()
        text = encryptor.update(p_data) + encryptor.finalize()
        # Integrity control
        h_mac = self.add_integrity_control(text, sintese)
        return text, h_mac
```

A decifra é exatamente o processo inverso, adicionando verificação do hmac. Se falhar, é porque a mensagem foi comprometida.

```
        def decrypt(self, text, mac, message: dict) -> dict:
                mtype = message.get('type', '')

                if mtype != "SECURE":
                    logger.error("Cannot decrypt message")
                    raise Exception("Cannot decrypt message")

                cipher = None
                block_size = 0
                # Algorithm
                if self.algorithm == 'AES':
                    alg = algorithms.AES(self.key)
                    block_size = alg.block_size
                elif self.algorithm == '3DES':
```

```
            alg = algorithms.TripleDES(self.key)
            block_size = alg.block_size
        # Mode
        if self.mode == 'ECB':
            modo = modes.ECB()
        elif self.mode == 'CBC':
            modo = modes.CBC(self.iv)
        cipher = Cipher(alg, modo, backend=default_backend())
        # Synthesis
        if self.hash_function == 'SHA-256':
            sintese = hashes.SHA256()
        elif self.hash_function == 'MD5':
            sintese = hashes.MD5()
        # Decrypt msg
        decryptor = cipher.decryptor()
        unpadder = padding.PKCS7(block_size).unpadder()
        if self.verify_integrity_control(text, sintese, mac) is False:
            raise Exception ("Integrity control failed")
        p_data = decryptor.update(text) + decryptor.finalize()
        data = unpadder.update(p_data) + unpadder.finalize()
        final_data = base64.b64decode(data)
        return final_data
```

3.3 Verificação de integridade

Para a adição e verificação de integridade nas mensagens, a aplicação usa o hmac, implementado com a biblioteca Cryptography.

```
    def verify_integrity_control(self, text, sintese, mac):
        try:
            h = hmac.HMAC(self.key, sintese, backend=default_backend())
            h.update(text)
            h.verify(mac)
        except Exception as e:
            logger.exception("Integrity control failed -> {}".format(e))
            return False
        return True

    def add_integrity_control(self, text, sintese):
        h = hmac.HMAC(self.key, sintese, backend=default_backend())
        h.update(text)
        h_mac = h.finalize()
        return h_mac
```

3.4 Autenticação e autorização

Primeiramente, a aplicação fará a autenticação do utente por password e desafios. A base de dados de usuários foi representada como um simples dicionário. As permissões referentes a cada usuário foi definida aqui.

```
    user_list = {'vinicius':
                 {
                     'fullname': 'vinicius ribeiro',
                     'certificate': '',
```

```
                    "password": "vinicius",
                    'can_connect': True,
                    'can_read': False
                },
                'joao': {
                    'fullname': 'joao barraca',
                    'certificate': '',
                    "password": "joao",
                    'can_connect': True,
                    'can_read': True
                }
        }
```

É feita, também, a autenticação do servidor através de certificados x509. Para isso, foram gerados dois certificados, um para a root e outro para o servidor, através da ferramenta OpenSSL. O cliente vai enviar um pedido ao servidor contendo o nonce para começar o processo de autenticação. O servidor, por sua vez, irá carregar o seu certificado, o certificado da root e sua chave privada do certificado e assinar o nonce com sua chave privada do servidor, que serão enviadas ao cliente. Em seguida, o cliente irá assinar o nonce com a chave pública do servidor recebido e comparar com o nonce recebido pelo servidor. Se essa etapa falhar, a autenticação falha e a conexão termina.

```
    def do_authenticate(self) -> None:

            GEN KEY PROCESS ….

            # Send authentication request
            self.server_nonce = os.urandom(16)
                        text = str.encode(json.dumps({ "type": 'AUTHN_REQ', "nonce":
base64.b64encode(self.server_nonce).decode('utf-8') }))
            payload, mac = self.encrypt(text)
              msg = { "type": 'SECURE', "payload": base64.b64encode(payload).decode('utf-8'),
"h_mac": base64.b64encode(mac).decode('utf-8') }
            self.send(msg)

            # Authenticate server
            data = self.read()
            logger.debug("KEY_GEN got -> {}".format(data))
            mtype = data.get('type', 'UNKNOWN')
            if mtype != 'KEY_GEN_OK':
                raise Exception("Something went wrong in key gen")

            sv_nonce = base64.b64decode(data["nonce"])
            b_sv_crt = base64.b64decode(data["sv_cert"])
            self.sv_crt = x509.load_der_x509_certificate(b_sv_crt, backend=default_backend())

            with open("./certs/rootCA.crt", "rb") as f:
                data = f.read()
                                    self.rootCA_crt = x509.load_pem_x509_certificate(data,
backend=default_backend())

            self.sv_crt_pub_key = self.sv_crt.public_key()
            self.rootCA_crt_pub_key = self.rootCA_crt.public_key()

            hs = hashes.Hash(hashes.SHA256(), backend=default_backend())
```

```
                hs.update(self.server_nonce)
                digest = hs.finalize()

                # Verify the signature validation
                try:
                                                        self.sv_crt_pub_key.verify(sv_nonce,    digest,
padder.PSS(mgf=padder.MGF1(hashes.SHA256()),                    salt_length=padder.PSS.MAX_LENGTH),
utils.Prehashed(hashes.SHA256()))
                    logger.info("Server authenticated")
                except Exception as e:
                    #TODO fix this ... DONE!
                    logger.exception("Invalid signature {}".format(e))
                    exit(1)

                #TODO VALIDATE SERVER CHAIN ???

                if self.auth_mode == "pass":
                        self.password_validation_request()
                else:
                    self.send({ "type": "ERROR" })
                    raise Exception("Authentication mode not suported")

                # Check if authentication went ok
                p_reply = self.read()
                logger.info(p_reply)
                logger.debug("AUTHN got -> {}".format(p_reply))
                mtype = p_reply.get('type', 'UNKNOWN')
                if mtype != 'AUTHN_OK':
                    raise Exception("Authentication fail!!")

                logger.info("Advancing state")
                self.state = STATE_AUTHZ
```

Processo de autenticação pelo cliente

```
        def process_authn(self, message: dict) -> bool:

                if self.state != STATE_AUTHN:
                    logger.warning("Invalid state (AUTHN). Discarding")
                    return False

                logger.info("STATE: AUTHENTICATE")

                # do authenticate
                logger.debug("Got: {}".format(message))
                self.server_nonce = base64.b64decode(message['nonce'])

                # Load server pri_key
                with open("./certs/server.key", "rb") as f:
                    data = f.read()
                    self.sv_crt_pri_key = serialization.load_pem_private_key(data, password=None,
backend=default_backend())

                # Get pub_key
                self.sv_crt_pub_key = self.sv_crt_pri_key.public_key()

                hs = hashes.Hash(hashes.SHA256(), backend=default_backend())
                hs.update(self.server_nonce)
                digested_hash = hs.finalize()
```

```
            nonce = self.sv_crt_pri_key.sign(digested_hash,
padder.PSS(mgf=padder.MGF1(hashes.SHA256()), salt_length=padder.PSS.MAX_LENGTH),
utils.Prehashed(hashes.SHA256()))

            # sv certificate
            with open("./certs/server.crt", "rb") as f:
                data = f.read()
                self.sv_crt = x509.load_pem_x509_certificate(data, backend=default_backend())
            b_sv_cert = self.sv_crt.public_bytes(serialization.Encoding.DER)

            # Send all to client
            text = str.encode(json.dumps( { "type": "KEY_GEN_OK", "nonce":
base64.b64encode(nonce).decode("utf-8"), "sv_cert": base64.b64encode(b_sv_cert).decode('utf-8')
}))
            # Encrypted message
            payload, mac = self.encrypt(text)
            msg = {"type": "SECURE", "payload": base64.b64encode(payload).decode("utf-8"),
"h_mac": base64.b64encode(mac).decode("utf-8")}

            self.send(msg)
            logger.info("CERETIFICATE SENDED")

            # In the last message of this process advance the state
            logger.info("ADVANCING STATE")
            self.state = STATE_CHALLENGE
            return True
```

Processo de autenticação no servidor

O cliente envia um pedido de password challenge, o servidor responde com o challenge e o cliente enviará a resposta deste challenge para ser verificado pelo servidor. Se tudo correr bem, o cliente está autenticado.

```
    def password_validation_request(self):
            logger.info("REQUESTING PASSWORD CHALLENGE")
              self.rsa_pri_key = rsa.generate_private_key(public_exponent=65537, key_size=2048,
backend=default_backend())
            self.rsa_pub_key = self.rsa_pri_key.public_key()
                    b_rsa_pub_key = self.rsa_pub_key.public_bytes(serialization.Encoding.DER,
serialization.PublicFormat.SubjectPublicKeyInfo)
                text = str.encode(json.dumps({ "type": "PWD_CHALLENGE_REQ", "RSA_PUB_KEY":
base64.b64encode(b_rsa_pub_key).decode("utf-8") }))
            payload, mac = self.encrypt(text)
              msg = { "type": "SECURE", "payload": base64.b64encode(payload).decode("utf-8"),
"h_mac": base64.b64encode(mac).decode("utf-8") }
            self.send(msg)

            try:
                reply = self.read()
                mtype = reply.get("type")
                if mtype == "CHALLENGE_PASS":
                    self.password_validation_reply(reply)
            except Exception as e:
                logger.exception("Something went wrong ... {}".format(e))
```

```
        def password_validation_reply(self, message: str) -> None:
            logger.info("REPLYING PASSWORD CHALLENGE")
            self.chalange_nonce = base64.b64decode(message["nonce"])
            self.user = input("Type you name: ")
            pwd = input("Type your password: ")
            p = pwd.encode() + self.chalange_nonce
            hs = hashes.Hash(hashes.SHA256(), backend=default_backend())
            hs.update(p)
            digest = hs.finalize()
                                        pass_signed     =     self.rsa_pri_key.sign(digest,
padder.PSS(mgf=padder.MGF1(hashes.SHA256()),                  salt_length=padder.PSS.MAX_LENGTH),
utils.Prehashed(hashes.SHA256()))
              text = str.encode(json.dumps( { "type": "CHALLENGE_PWD_REP", "user": self.user,
"password": base64.b64encode(pass_signed).decode("utf-8") }))
            payload, mac = self.encrypt(text)
              msg = { "type": "SECURE", "payload": base64.b64encode(payload).decode("utf-8"),
"h_mac": base64.b64encode(mac).decode("utf-8") }
            self.send(msg)
```

Challenge request e reply pelo cliente

```
       def send_challenge_pass(self, message: str) -> None:
            if self.state != STATE_CHALLENGE:
                logger.warning("Invalid state (CHALLENGE). Discarding")
                raise Exception("Something went wront while sending chalange pass")
            logger.info("SENDING CHALLENGE PASSWORD")
            b_rsa_client_pub_key = base64.b64decode(message['RSA_PUB_KEY'])
              self.rsa_client_pub_key = serialization.load_der_public_key(b_rsa_client_pub_key,
backend=default_backend())
            self.chalenge_nonce = os.urandom(16)
                      text   =   str.encode(json.dumps({   "type":   'CHALLENGE_PASS',   "nonce":
base64.b64encode(self.chalenge_nonce).decode("utf-8") }))
            payload, mac = self.encrypt(text)
              msg = { "type": "SECURE", "payload": base64.b64encode(payload).decode("utf-8"),
"h_mac": base64.b64encode(mac).decode("utf-8") }
            self.send(msg)

        def process_challenge_pass(self, message: str) -> None:
            if self.state != STATE_CHALLENGE:
                logger.warning("Invalid state (CHALLENGE). Discarding")
                raise Exception("WRONG STATE!!")
            logger.info("PROCESSING CHALLENGE PASSWORD")
            self.user = message["user"]
            self.pwd = base64.b64decode(message["password"])
            if self.user not in user_list.keys():
                self.send( { "type": "ERROR", "payload": "Wrong username/password" } )
                raise Exception("Something went wrong. Check your user/pass")
            else:
                              password   =   user_list.get(self.user).get("password").encode()   +
self.chalenge_nonce
                hs = hashes.Hash(hashes.SHA256(), backend=default_backend())
                hs.update(password)
                digest = hs.finalize()
                try:
                                            self.rsa_client_pub_key.verify(self.pwd,   digest,
padder.PSS(mgf=padder.MGF1(hashes.SHA256()),                  salt_length=padder.PSS.MAX_LENGTH),
utils.Prehashed(hashes.SHA256()))
                    self.send({ "type": "AUTHN_OK" })
```

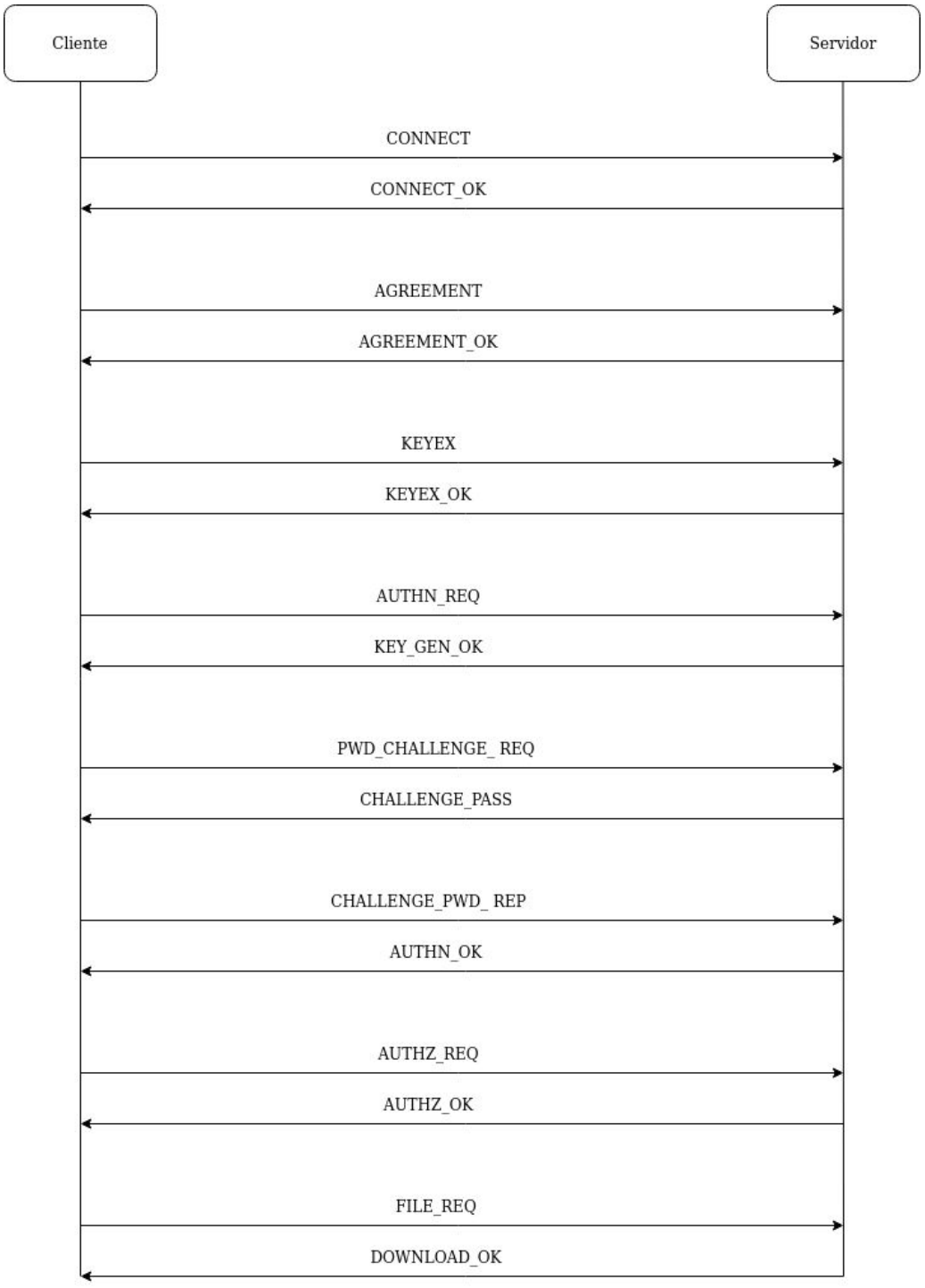
```
                    logger.info("User {} authenticated".format(self.user))
                    self.state = STATE_AUTHN
                except Exception as e:
                    logger.info("Something went wrong... {}".format(e))
                    self.send({ "type": "ERROR", "payload": "Invalid signature" })
```

Envio de processamento do challenge pelo servidor

Certos clientes não possuem autorização para fazer o download de ficheiros do servidor. Isto é checado e, em caso de não autorização, o cliente é desconectado.

```
    def process_authz(self, message: dict) -> bool:
            if self.state != STATE_AUTHZ:
                logger.warning("Invalid state. Discarding")
                return False

            logger.info("STATE: AUTHORIZE")
            user = message['user']

            # do authorization
            if user_list.get(user).get("can_read") is False:
                self.send({ "type": "ERROR", "payload": "User can't download files" })
                raise Exception("User can not download files from server")

            logger.info("USER IS AUTHORIZED TO DOWNLOAD FILES")
            self.send({'type': 'GET_OK', "payload": "AUTHZ_OK"})

            # In the last message of this process advance the state
            logger.debug("ADVANCING TO GET STATE")
            self.state = STATE_GET
            return True
```

4 Protocolo



Protocolo da aplicação

Assumindo sucesso em todos os passos.

5 Considerações finais

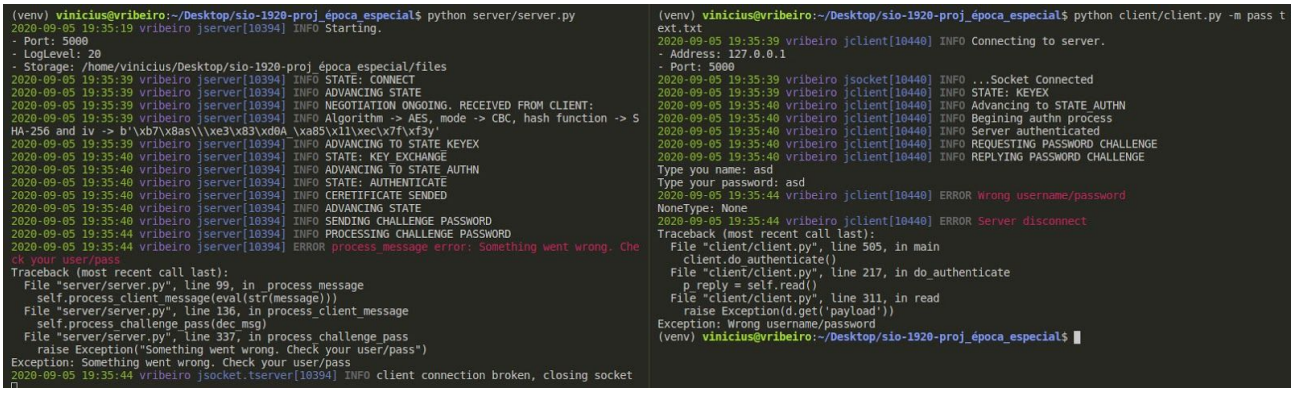
5.1 Considerações importantes

Ao final deste trabalho, penso que obtive um nível satisfatório, mesmo não cumprindo todos os objetivos estabelecidos pelos docentes. Foi possível aprofundar meus conhecimentos de criptografia, negociação de chaves, verificação de integridade, autenticação e autorização de clientes/servidor.

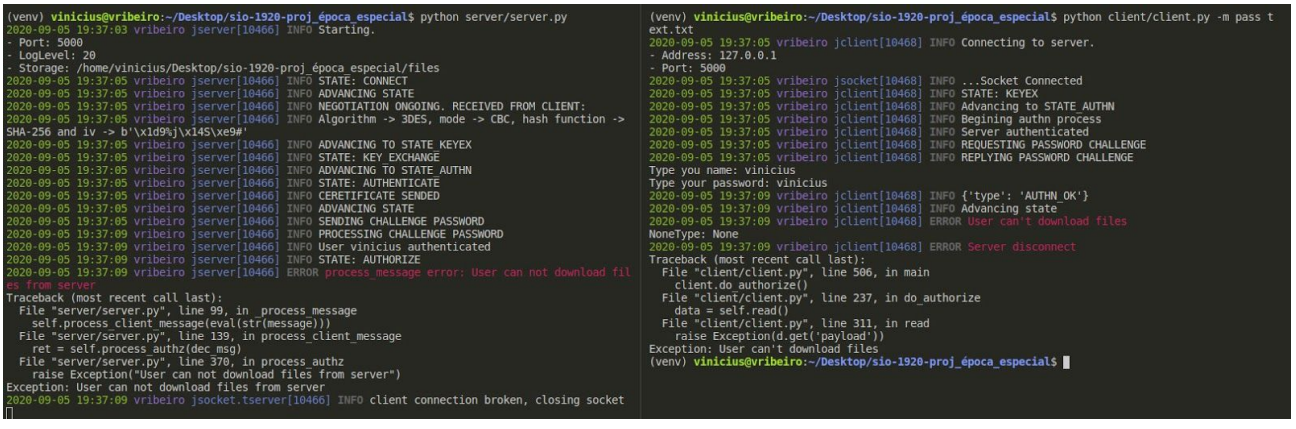
Afim de melhorar este projeto, os próximos passos seriam a implementação da verificação da cadeia de certificados e o refactor do código para reduzir o número de código duplicado, ou seja, a criação de funções genéricas.

5.2 Resultados esperados

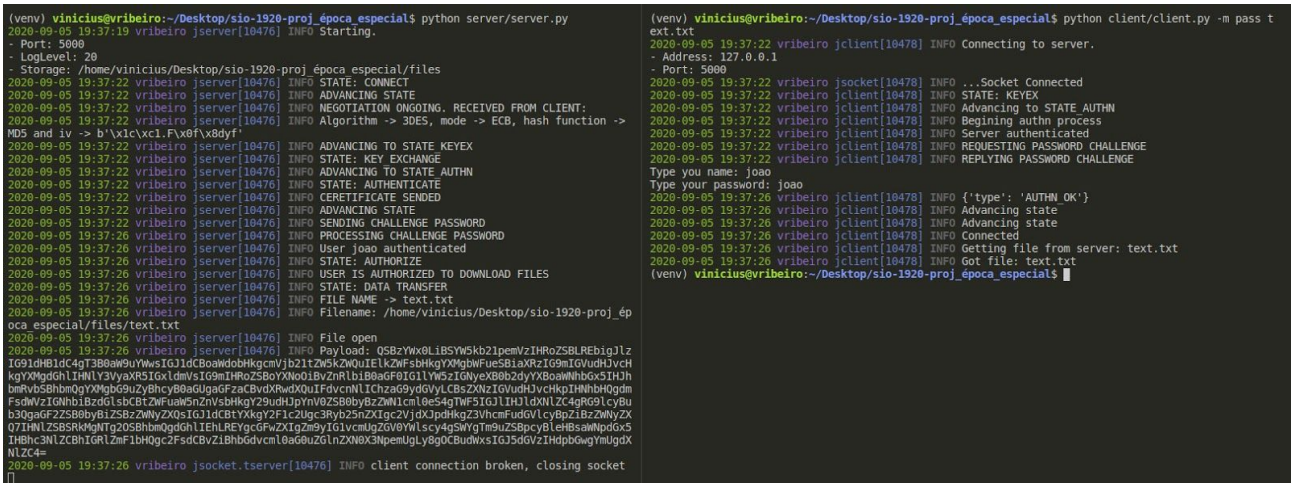
Em caso de user/password incorretos:



Em caso de user sem permissões:



Em caso de user com permissões:



6 Referências e recursos

https://cryptography.io/
https://gist.github.com/fntlnz/cf14feb5a46b2eda428e000157447309
Slides da disciplina.