

Universidade de Aveiro

Sistema Seguro de Mensagens Instantâneas

Nuno Humberto

Relatório no âmbito de Segurança

Dezembro, 2016
Aveiro

Sistema Seguro de Mensagens Instantâneas

Nuno Humberto

Dezembro 2016, Universidade de Aveiro

Conteúdo

Lista de Figuras	2
0.1 Resumo	3
1 Introdução	4
2 Preparação	5
2.1 Linguagem utilizada	5
2.2 Funcionamento básico (1ª entrega)	5
3 Procedimento (1ª entrega)	7
3.1 Troca de chaves	7
3.2 Cifragem de mensagens	9
3.3 Integridade das mensagens	11
3.4 Interface	12
4 Procedimento	13
4.1 Validação da identidade	13
4.1.1 Fase de ligação	13
4.1.2 Troca de mensagens	15
4.2 Validação da entrega	16
4.3 Controlo do fluxo de informação (Bell-LaPadula)	17
4.4 Opcional: Validação offline	18
4.5 Opcional: Identificação do dispositivo	20
5 Análise de Resultados	22
6 Conclusão	23
Glossário	24
Bibliografia	26

Lista de Figuras

3.1	Função responsável pela geração das chaves	7
3.2	Conversão para PEM e emissão de chave pública do cliente ao servidor	7
3.3	Importação de chave pública e cálculo de segredo comum no servidor	8
3.4	Derivação de chave com utilização de salt gerado pelas duas partes	9
3.5	Geração de um nonce/IV para uso num dos dois modos de cifragem	9
3.6	Definições de renegociação de chaves	11
3.7	Exemplo da verificação de um HMAC recebido numa mensagem	11
3.8	Aspecto da interface do cliente	12
4.1	Actualização da cache de certificados	13
4.2	Função para produção de assinaturas no servidor, recebendo o caminho da chave privada	14
4.3	Função para produção de assinaturas no cliente, utilizando uma sessão do Cartão do Cidadão	14
4.4	Função para verificação de assinaturas geradas pelo servidor . . .	15
4.5	Função para verificação de assinaturas geradas pelo cliente	15
4.6	Exemplo de verificação de assinatura gerada pelo servidor	16
4.7	Exemplo de verificação de assinatura gerada pelo cliente	16
4.8	Configuração (segundos) do <i>delay</i> até ao reenvio, juntamente com outros parâmetros configuráveis	16
4.9	Decisão do nível de um utilizador	17
4.10	Verificação do nível de dois utilizadores	17
4.11	Listagem de utilizadores	17
4.12	Ecrã de seleção dos <i>logs</i> disponíveis	18
4.13	Validação com sucesso de uma cadeia de mensagens	19
4.14	Falha na validação de uma cadeia de mensagens	20
4.15	Método utilizado para o cálculo do <i>hardware ID</i>	21
5.1	Exemplo da interface do utilizador	22
5.2	Exemplo da interface do utilizador - deteção de slots	22

0.1 Resumo

Tratando-se da fase final do desenvolvimento do projecto prático da disciplina de Segurança, foi concluída a criação de um sistema de troca segura de mensagens instantâneas.

Na entrega inicial do projecto, foram cobertos todos os aspectos relacionados com a comunicação entre entidades, incluindo todas as cifras, controlos da integridade e da confidencialidade das mensagens transmitidas.

Mantendo o esquema da primeira entrega, na troca de chaves entre o servidor e um cliente ou entre dois cliente, é usado [ECDHE](#), uma variante do protocolo Diffie-Hellman que utiliza [ECC](#).

Foi utilizada a cifra por blocos [AES](#) com o objectivo de cifrar as mensagens trocadas.

Para a garantia de integridade, foi utilizado um [HMAC](#) com o algoritmo de [hashing SHA](#).

Os elementos do relatório referentes à primeira entrega do projecto estão devidamente identificados no título do capítulo ou secção. Além disso, o procedimento referente à primeira entrega será isolado num capítulo distinto. ([Capítulo 3](#))

Adicionados na entrega final do projecto prático, estão a preservação da identidade, com recurso ao uso do Cartão do Cidadão, a validação da entrega das mensagens, o controlo do fluxo da informação e ainda, propostas para funcionalidades opcionais, a identificação do dispositivo de cada participante e a possibilidade da validação offline da correcta ordem das mensagens recebidas de cada cliente.

Para o controlo do fluxo da informação, foi utilizado o modelo de Bell-LaPadula.

Na verificação e geração de assinaturas emitidas pelo servidor, foi utilizado o esquema de assinaturas [PSS](#).

Em contraste, na verificação e geração de assinaturas pela parte do utilizador, com recurso ao Cartão do Cidadão, foi utilizado o esquema de assinaturas [PKCS1 v1.5](#), devido à falta de suporte a mecanismos de assinaturas mais modernos.

Capítulo 1

Introdução

Este relatório tem como objectivo principal a descrição do procedimento seguido no âmbito do desenvolvimento da totalidade do projecto proposto. A segurança na troca de mensagens trata-se de um tema importante e actual pois, mais que nunca, a troca de mensagens é utilizada diariamente por todo o mundo, através de implementações cujos detalhes de segurança são muitas vezes desconhecidos.

Neste relatório são descritos os métodos e bibliotecas utilizadas para cumprir os objectivos propostos.

São também descritas detalhadamente as implementações da geração e troca de chaves e certificados, da verificação da identidade tanto do servidor como dos utilizadores, da cifra simétrica das mensagens e da validação da sua receção, dos mecanismos de garantia de integridade e, também, da interface da aplicação.

Por fim, este relatório inclui uma breve análise dos resultados obtidos e conclui com um comentário final ao progresso do desenvolvimento das funcionalidades da aplicação.

Capítulo 2

Preparação

2.1 Linguagem utilizada

O sistema de troca de mensagens instantâneas consiste num cliente e num servidor, ambos desenvolvidos na linguagem *Python*. www.python.org

O servidor utilizado foi baseado no exemplo em *Python* inicialmente desenvolvido pelo Professor João Paulo Barraca. O cliente foi desenvolvido de raíz.

É também incluído um módulo para a verificação offline de alguns aspectos das mensagens obtidas por cada cliente, cujo funcionamento será explicado em detalhe na [Seção 4.4](#).

Para a implementação de muitas das funcionalidades relacionadas com o projecto, foram usadas ferramentas e bibliotecas externas:

- *cryptography.io*[1] - geração e derivação de chaves, utilização de cifras simétricas, geração e verificação de [HMACs](#).
- *pyOpenSSL*[2] - utilização geral de certificados X.509 e [CRLs](#)
- *pyCrypto*[3] - geração e verificação de assinaturas [PKCS1 v1.5](#) e [PSS](#), geração de hashes [SHA](#)
- *XCA*[4] - geração de [CA](#) e certificados utilizados na verificação da identidade do servidor
- *PyKCS11* [5] - interface com o Cartão do Cidadão, incluindo extração de certificados e geração de assinaturas
- *dmidecode*[6] - acesso à [DMI](#) para obtenção de detalhes de hardware

2.2 Funcionamento básico (1ª entrega)

Antes do desenvolvimento dos aspectos referentes a segurança, tratou-se de preparar uma versão básica tanto do servidor como do cliente utilizados, de

modo a permitir a interação básica e em aberto, sem qualquer preocupação com a segurança da informação transmitida.

Esta dispõe de métodos para, de uma forma suficiente:

- Efectuar ligações cliente-servidor.
- Obter uma lista dos clientes ligados ao servidor.
- Estabelecer ligações cliente-cliente.
- Enviar mensagens de texto em aberto, entre clientes.
- Remover ligações entre clientes.

Após a realização das versões inseguras do servidor e cliente, foram adicionadas as capacidades de transmissão segura de mensagens à aplicação.

Capítulo 3

Procedimento (1ª entrega)

3.1 Troca de chaves

Para a troca de chaves, tanto entre o cliente e o servidor como entre clientes, foi usado [ECDHE](#), uma variante do protocolo Diffie-Hellman que utiliza [ECC](#). O problema matemático por trás da utilização de curvas elípticas na criptografia é denominado [ECDLP](#) e é de resolução muito mais difícil que outros algoritmos como, por exemplo, [RSA](#).[\[7\]](#) Quebrar uma chave de curva elíptica de 228-bit iria requerer tanto poder computacional como quebrar uma chave [RSA](#) de 2380-bit.[\[8\]](#)

Tendo isto em conta, a utilização de [ECC](#) permite uma poupança no armazenamento, memória e largura de banda dos dispositivos que a utilizam.

No âmbito do trabalho desenvolvido, as chaves privadas e públicas baseadas em [ECC](#) são geradas tanto no cliente como no servidor através da função presente na [Figura 3.1](#).

```
def generateTransportKeyPair(self, algorithm, interclient=False):
    if algorithm == 'ECDHE':
        self.privkey_tls = ec.generate_private_key(ec.SECP256K1, default_backend())
        self.pubkey_tls = self.privkey_tls.public_key()
```

Figura 3.1: Função responsável pela geração das chaves

Após a geração das chaves, tanto o cliente como o servidor enviam as suas chaves públicas, como ilustra a [Figura 3.2](#).

```
self.send({'type': 'connect', 'phase': 3, 're-exchange': reexchange, 'name': 'Nuno Humberto',
'id': base64.urlsafe_b64encode(os.urandom(8)), 'uid': base64.urlsafe_b64encode(uid),
'ciphers': [self.serverCipherSpec],
'data': {'pubkey': self.pubkey_tls.public_bytes(encoding=serialization.Encoding.PEM,
format=serialization.PublicFormat.SubjectPublicKeyInfo)} })
```

Figura 3.2: Conversão para PEM e emissão de chave pública do cliente ao servidor

Em seguida, tanto o cliente como o servidor utilizam as chaves públicas recebidas para calcular um segredo comum, como ilustra a [Figura 3.3](#).

```
imported_key = serialization.load_pem_public_key(str(next['data']['pubkey']), backend=default_backend())
self.secret = self.privkey_tls.exchange(ec.ECDH(), imported_key)
del self.privkey_tls
```

Figura 3.3: Importação de chave pública e cálculo de segredo comum no servidor

Logo após a obtenção do segredo comum por ambas as partes, devem ser eliminados da memória os elementos que já não são necessários, como está também incluído na [Figura 3.3](#).

Este também é o método utilizado para acordar um segredo comum entre dois clientes mas, neste caso, as mensagens de negociação estarão encapsuladas em mensagens do tipo SECURE, que são cifradas no cliente e decifradas no servidor, ou vice-versa.

3.2 Cifragem de mensagens

Para a cifragem das mensagens entre o cliente e o servidor, é calculada uma chave derivada do segredo comum. Esta chave derivada é obtida através da função de derivação de chaves [PBKDF2](#). Para a utilização da função, é gerado um [salt](#) em que participam tanto o cliente como o servidor, cada um com 16 bytes de dados aleatórios.

A derivação da chave é demonstrada na [Figura 3.4](#).

```
saltA = os.urandom(16)
saltB = base64.urlsafe_b64decode(str(request['data']['saltB']))

derived_key = PBKDF2HMAC(algorithm=hashes.SHA512(), length=64, salt=saltA+saltB,
    iterations=100000, backend=default_backend()).derive(sender.sa_data['secret'])
```

Figura 3.4: Derivação de chave com utilização de salt gerado pelas duas partes

A derivação da chave é efectuada através da aplicação consecutiva de 100000 iterações de uma função de [hashing](#) ([SHA512](#)) em que participa o segredo comum das duas entidades.

A chave gerada corresponde a uma sequência de 512 bits, dos quais os primeiros 256 irão participar na cifragem da mensagem e os últimos 256 na função de verificação de integridade, explicada na [Seção 3.3](#).

Após calculada a chave a usar na cifragem, é aplicada à mensagem uma cifra por blocos ([AES256](#)). O modo a utilizar para a cifragem é extraído do [cipherspec](#), como mostra a figura [Figura 3.5](#), que é acordado entre o cliente e o servidor no início da sua ligação.

```
counternonce = os.urandom(cipher_algo.block_size/8)

bc_spec = self.getBlockCipherModeFromSpec(sender.chosenSpec)
if bc_spec == "CTR":
    ciphermode = modes.CTR(counternonce)
elif bc_spec == "OFB":
    ciphermode = modes.OFB(counternonce)

countercipher = Cipher(cipher_algo, ciphermode, backend=default_backend())
ciphertext_gen = countercipher.encryptor()

ciphertext = ciphertext_gen.update(plaintext) + ciphertext_gen.finalize()
```

Figura 3.5: Geração de um nonce/IV para uso num dos dois modos de cifragem

No âmbito desta aplicação, foram implementados os modos **CTR** e **OFB**. Na utilização do modo **CTR**, deve ser fornecido um **nonce** com um comprimento igual às dimensões dos blocos do algoritmo em uso. (Sempre 128 bits, no caso do **AES**)

Na utilização do modo **OFB**, deve ser fornecido um **IV** com um comprimento também igual às dimensões dos blocos do algoritmo em uso.

Como tal, é necessária a geração de 16 bytes de dados aleatórios para a produção de um **IV/nonce**, dependendo do modo de cifragem a utilizar.

Sempre que é necessária a geração de números aleatórios, é utilizada a fonte de entropia do sistema operativo através da função *os.random(n)*.

Para cada mensagem, as chaves utilizadas para as cifras por blocos são sempre negociadas novamente a partir do segredo comum definido entre o servidor e o cliente. Dois clientes que desejem comunicar entre si, irão negociar as suas chaves por cada mensagem que desejem enviar mas, neste caso, a própria negociação das chaves a utilizar para as cifras por blocos, estará encapsulada dentro de mensagens do tipo **SECURE**.

Isto significa que uma mensagem cifrada de um cliente para outro será sempre:

- Cifrada no remetente (chave remetente-destinatário)
- Cifrada no remetente (chave remetente-servidor)
- Decifrada no servidor (chave remetente-servidor)
- Cifrada no servidor (chave servidor-destinatário)
- Decifrada no destinatário (chave servidor-destinatário)
- Decifrada no destinatário (chave remetente-destinatário)

Além das chaves de cifras por blocos, também os segredos comuns originários do [ECDHE](#) são renegociados de forma configurável.

```
MAX_SERVER_MESSAGES = 5
MAX_ENDPOINT_MESSAGES = 5
MAX_CLIENT_SESSION_SECONDS = 30
MAX_SERVER_SESSION_SECONDS = 45
```

Figura 3.6: Definições de renegociação de chaves

Com as configurações da [Figura 3.6](#), todas as sessões cliente-servidor terão as suas geradas novamente passadas 5 mensagens ou 45 segundos, dependendo do qual acontecer primeiro. E, de forma semelhante, todas as sessões cliente-cliente terão as suas geradas novamente passadas 5 mensagens ou 30 segundos.

Desta forma, este sistema de mensagens tem implementada *forward and backward secrecy*, visto que mesmo que sejam comprometidas as chaves de um determinado momento, as chaves apenas serão válidas por uma janela de tempo muito pequena (configurável), tornando extremamente difícil a obtenção de mensagens anteriores ou posteriores a esse momento.

3.3 Integridade das mensagens

Para garantir a integridade das mensagens entre todas as entidades, é utilizado um [HMAC](#). Através da utilização dos últimos 256 bits da chave gerada na [Seção 3.2](#) e de uma mensagem cifrada, é gerada uma sequência de bits que possibilita a verificação da integridade da mensagem ao ser recebida por uma das entidades.

A utilização de um HMAC gerado a partir de dados cifrados denomina-se de *encrypt-then-MAC*, e permite verificar a integridade tanto da mensagem cifrada como do seu conteúdo original.

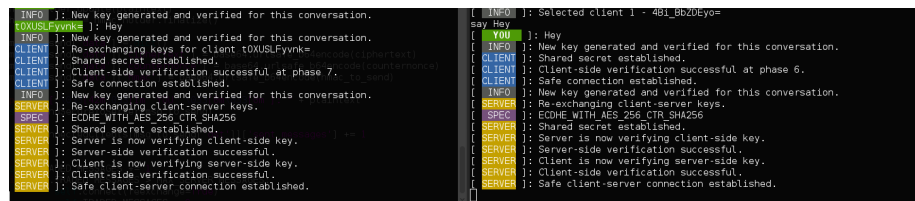
```
hmac_holder = hmac.HMAC(derived_key[32:64], hashes.SHA256(), backend=default_backend())
hmac_holder.update(ciphertext)
try:
    hmac_holder.verify(received_hmac)
except:
    print "[\033[41m\033[1m ERROR \033[0m\033[0m]: Client-side verification failed."
    return
```

Figura 3.7: Exemplo da verificação de um HMAC recebido numa mensagem

3.4 Interface

Para interagir com o cliente do sistema de mensagens, apenas são necessários os comandos:

- whoami - devolve o nome (ID) do utilizador actual
- list - devolve a lista de utilizadores ligados ao servidor
- select NUMERO - selecciona o utilizador NUMERO da lista
- connect - estabelece uma ligação com o utilizador seleccionado
- say MENSAGEM - envia a MENSAGEM ao utilizador seleccionado
- disconnect - termina a ligação com o utilizador seleccionado
- exchange - força a renovação das chaves cliente-servidor
- clientexchange - força a renovação das chaves com o cliente seleccionado



```
[INFO] : New key generated and verified for this conversation.
[TOXUSLFYVKE] : Hey
[INFO] : New key generated and verified for this conversation.
[CLIENT] : Re-exchanging keys for client TOXUSLFYVKE.
[CLIENT] : Shared secret established.
[CLIENT] : Client-side verification successful at phase 7.
[CLIENT] : Safe connection established.
[INFO] : New key generated and verified for this conversation.
[SERVER] : Re-exchanging client-server keys.
[SPICE] : ECDHE WITH AES_256_CTR_SHA256
[SERVER] : Shared secret established.
[SERVER] : Server is now verifying client-side key.
[SERVER] : Server-side verification successful.
[SERVER] : Client is now verifying server-side key.
[SERVER] : Client-side verification successful.
[SERVER] : Safe client-server connection established.

[INFO] : Selected client 1 - 481_BbZbEyo=
say Hey
[TOXUSLFYVKE] : Hey
[INFO] : New key generated and verified for this conversation.
[CLIENT] : Shared secret established.
[CLIENT] : Client-side verification successful at phase 6.
[CLIENT] : Safe connection established.
[INFO] : New key generated and verified for this conversation.
[SERVER] : Re-exchanging client-server keys.
[SPICE] : ECDHE WITH AES_256_CTR_SHA256
[SERVER] : Shared secret established.
[SERVER] : Server is now verifying client-side key.
[SERVER] : Server-side verification successful.
[SERVER] : Client is now verifying server-side key.
[SERVER] : Client-side verification successful.
[SERVER] : Safe client-server connection established.
```

Figura 3.8: Aspecto da interface do cliente

Capítulo 4

Procedimento

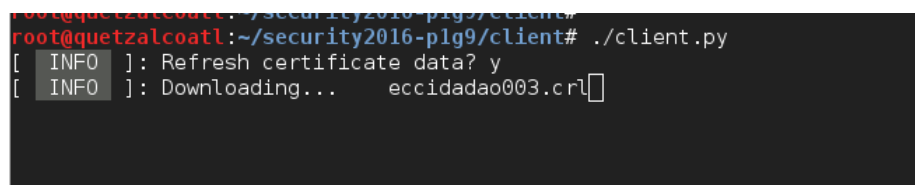
4.1 Validação da identidade

Para a troca de mensagens com o servidor ou um cliente, é necessária uma validação da identidade de ambas as partes.

Para tal, no caso do servidor, foi criada uma CA utilizando a ferramenta *XC*[4] e, em seguida, foi criado um certificado para o servidor, assinado pela CA recentemente criada. O certificado desta CA é, posteriormente, adicionado como confiável nos utilizadores do sistema de troca de mensagens.

No caso do cliente, o certificado a utilizar provém do Cartão do Cidadão e é extraído através da biblioteca *PyKCS11*[5]. Este certificado, quando recebido pelo servidor ou por outro cliente, é sucessivamente verificado até uma raiz confiável, sendo que a informação dos certificados e CRLs a utilizar são extraídos através do próprio certificado do utilizador.

Para evitar o download constante destes certificados e CRLs, é dada a possibilidade ao utilizador a actualização da cache destes dados no arranque da aplicação, como mostra a Figura 4.1.



```
root@quetzalcoatl:~/security2016-plg9/client# ./client.py
[ INFO ]: Refresh certificate data? y
[ INFO ]: Downloading... eccidadao003.crl
```

Figura 4.1: Actualização da cache de certificados

4.1.1 Fase de ligação

Para estabelecer uma ligação segura com o servidor ou com outro cliente é necessário efectuar a troca dos certificados de ambas as entidades.

No processo de ligação, após a troca de certificados, as entidades produzem um *challenge* composto por 16 bytes aleatórios e armazenam o valor produzido.

Em seguida, o *challenge* é enviado à entidade oposta que, por sua vez, produz uma assinatura (PKCS1 v1.5, se gerada por um cliente ou PSS, se gerada pelo

servidor), denominada *response*, e a envia de regresso à entidade emissora do *challenge*.

Os métodos responsáveis pela geração de assinaturas encontram-se na [Figura 4.2](#), para o caso do servidor e na [Figura 4.3](#), para o caso do cliente.

```
def sign_pss(key_filename, data):
    priv_key = RSA.importKey(open(key_filename).read())
    data_hash = SHA.new()
    data_hash.update(data)
    data_signer = PKCS1_PSS.new(priv_key)
    return data_signer.sign(data_hash)
```

Figura 4.2: Função para produção de assinaturas no servidor, recebendo o caminho da chave privada

```
if 'challenge' in msg:
    challenge_from_client = base64.urlsafe_b64decode(str(msg['challenge']))
    response_to_client = self.cardSession.sign(self.cardSession.findObjects()[0],
                                              challenge_from_client, mecha=PyKCS11.Mechanism(PyKCS11.CKM_SHA1_RSA_PKCS, None))
    response_to_client = ''.join(chr(signature_byte) for signature_byte in response_to_client)
```

Figura 4.3: Função para produção de assinaturas no cliente, utilizando uma sessão do Cartão do Cidadão

Ao receber a resposta ao seu *challenge*, a entidade deve utilizar a chave pública presente no certificado enviado pela entidade oposta para verificar a validade da resposta produzida, junto do valor aleatório anteriormente gerado e armazenado.

Os métodos responsáveis pela verificação de assinaturas encontram-se na [Figura 4.4](#), para o caso do servidor e na [Figura 4.5](#), para o caso do cliente.


```
def verifySignature_pss(cert_str, signature, data):
    cert = load_certificate(FILETYPE_PEM, cert_str)
    pub_key = OpenSSL.crypto.dump_publickey(FILETYPE_PEM, cert.get_pubkey())
    pub_key = RSA.importKey(pub_key)
    data_hash = SHA.new()
    data_hash.update(data)
    signature_verifier = PKCS1_PSS.new(pub_key)
    if signature_verifier.verify(data_hash, signature):
        return True
    else:
        return False
```

Figura 4.4: Função para verificação de assinaturas geradas pelo servidor

```
def verifySignature(cert_str, signature, data):
    cert = load_certificate(FILETYPE_ASN1, cert_str)
    pub_key = OpenSSL.crypto.dump_publickey(FILETYPE_PEM, cert.get_pubkey())
    pub_key = RSA.importKey(pub_key)
    data_hash = SHA.new()
    data_hash.update(data)
    signature_verifier = PKCS1_v1_5.new(pub_key)
    if signature_verifier.verify(data_hash, signature):
        return True
    else:
        return False
```

Figura 4.5: Função para verificação de assinaturas geradas pelo cliente

A verificação com sucesso da resposta ao *challenge* enviado, permite a conclusão da validação da identidade na fase de ligação.

4.1.2 Troca de mensagens

Além da verificação da identidade de ambas as partes na fase da sua ligação, todas as mensagens seguras que ambas as entidades trocam transportam uma assinatura da quem as emite.¹

Novamente, **PKCS1 v1.5**, se gerada por um cliente ou **PSS**, se gerada pelo servidor.

Exemplos das invocações dos métodos responsáveis pela verificação de assinaturas encontram-se na [Figura 4.6](#), para o caso do servidor e na [Figura 4.7](#), para o caso do cliente.

¹No caso de se tratar, por exemplo, de uma mensagem *client-com* a ser transportada entre dois clientes, transportará então as assinaturas do servidor e do cliente.

```

if common_cert.verifySignature_pss(self.serverCert, signature, json.dumps(msg)):
    print "[ \033[43m\033[1m DEBUG \033[0m\033[0m ]: Secure message successfully verified!"
else:
    print "[ \033[45m\033[1m SRS ERROR \033[0m\033[0m ]: Bad identity signature!"
    return

```

Figura 4.6: Exemplo de verificação de assinatura gerada pelo servidor

```

if common_cert.verifySignature(sender.userCert, signature, json.dumps(request)):
    print "[ \033[43m\033[1m DEBUG \033[0m\033[0m ]: Secure message successfully verified!"
else:
    print "[ \033[45m\033[1m SRS ERROR \033[0m\033[0m ]: Bad identity signature!"
    return

```

Figura 4.7: Exemplo de verificação de assinatura gerada pelo cliente

4.2 Validação da entrega

Na fase de desenvolvimento final do projecto, foram adicionadas funcionalidades úteis às mensagens do tipo [ACK](#).

No seu interior contém uma assinatura da mensagem original e a sua recepção é aguardada pelo emissor original. Permitem, desta forma, a validação da entrega da mensagem ao destinatário correcto.

Foi também implementado um esquema de reenvio, na eventualidade do emissor da mensagem original não receber um [ACK](#) da sua mensagem num espaço de tempo configurável, como mostra a [Figura 4.8](#).

```

RESEND_IF_NOT_ACKED = 15
MAX_SERVER_MESSAGES = 5
MAX_ENDPOINT_MESSAGES = 3
MAX_CLIENT_SESSION_SECONDS = 60
MAX_SERVER_SESSION_SECONDS = 60

```

Figura 4.8: Configuração (segundos) do *delay* até ao reenvio, juntamente com outros parâmetros configuráveis

Para efeitos de demonstração, na apresentação, do reenvio automático, o servidor está configurado para ignorar a primeira mensagem *list* que receber, de forma a que o cliente a tente reenviar automaticamente.

4.3 Controlo do fluxo de informação (Bell-LaPadula)

A aplicação descrita neste relatório implementa um controlo do fluxo de informação baseado no modelo de Bell-LaPadula.

Assim sendo, é associado um nível a cada utilizador, no processo da sua ligação. A atribuição do nível é feita tendo em conta a informação armazenada num ficheiro, importado para um dicionário no momento de arranque do servidor, que indica que nível atribuir a cada utilizador, dependendo do seu número do Cartão do Cidadão.

Como mostra a [Figura 4.9](#), se o número do Cartão do Cidadão do utilizador não estiver presente na base de dados do servidor, então é automaticamente atribuído, ao utilizador em questão, o nível 1.

```
sender.ccID = common_cc.getID(temp_cert)
if sender.ccID[2:] in self.userLevels:
    sender.level = self.userLevels[sender.ccID[2:]]
else:
    sender.level = 1
```

Figura 4.9: Decisão do nível de um utilizador

Segundo o modelo de Bell-LaPadula, um utilizador apenas poderá enviar mensagens a outros utilizadores se a estes estiver associado um nível igual ou superior ao seu. Esta implementação é aplicada apenas a mensagens do tipo *client-com* e está exposta na [Figura 4.10](#).

```
if request['payload']['type'] == 'client-com' and int(sender.level) > int(dst.level):
    print "[ \033[44mCLIENT\033[0m ]: Blocked message from \033[1m\033[32m" + # Block message
          sender.name + "\033[0m\033[0m to \033[1m\033[32m" + dst.name + "\033[0m\033[0m."
    return
```

Figura 4.10: Verificação do nível de dois utilizadores

Ao executar um comando *list*, o servidor inclui o nível de cada utilizador nos seus resultados, como mostra a [Figura 4.11](#).

```
[ SERVER ]: Available clients:
[ SERVER ]: 1: HrLxTndHCQo= - Nuno Humberto (BI1 ) - LEVEL 2
```

Figura 4.11: Listagem de utilizadores

4.4 Opcional: Validação offline

Nesta aplicação, está implementada uma funcionalidade que permite a validação offline da ordem correcta das mensagens recebidas de cada cliente. Para tal, cada cliente tem associado a cada um dos clientes aos quais está ligado, um contador incrementado por cada mensagem que envia.

Para a validação da cadeia de mensagens ser possível, sempre que enviar uma mensagem para outro cliente, cada cliente envia também uma assinatura sobre três componentes: a mensagem, o contador e a assinatura da última mensagem, os quais são exportados para um ficheiro pelo cliente que os recebe. Ao receber estes dados, o cliente adiciona-lhes também a mensagem recebida, criando assim um *log* das mensagens enviadas por cada cliente. Juntamente com o *log* das mensagens, é também armazenado o certificado do utilizador que as enviou, por modo a poder verificar a assinatura das mesmas.

Para permitir esta validação, foi desenvolvido um módulo extra, responsável pela leitura dos *logs* e certificados.

O módulo automaticamente procura os *logs* disponíveis no directório actual e apresenta um ecrã de seleção ao utilizador, como mostra a [Figura 4.12](#).

```
root@quetzalcoatl:~/security2016-plg9/client# ./verify.py
Choose a chat log:

1 - u24qIkgi--o=
    Name: Nuno Humberto
    HWID: 542320FCEE95A0604FEBB39368115D18E0947C70

2 - zeHc0F6HR6k=
    Name: Ana Isabel
    HWID: 542320FCEE95A0604FEBB39368115D18E0947C70

3 - 43MxEtdFH08=
    Name: Humberto Jorge
    HWID: 542320FCEE95A0604FEBB39368115D18E0947C70

Pick a log: ☐
```

Figura 4.12: Ecrã de seleção dos *logs* disponíveis

O utilizador, neste ecrã, pode seleccionar um cliente com o qual estabeleceu uma ligação previamente, sendo fornecido o *ID*, nome e *hardware ID* de cada um. (explorado em detalhe na [Seção 4.5](#))

Após selecionar um cliente, são mostradas ao utilizador as mensagens recebidas na conversa, pela ordem de chegada e, além disso, é mostrado o resultado do estado da cadeia de verificação das mensagens para cada mensagem armazenada.

Segue-se na [Figura 4.13](#) um exemplo de validação com sucesso de uma cadeia de mensagens.

```
root@quetzalcoat1:~/security2016-plg9/client# ./verify.py
Choose a chat log:

1 - u24qIkgi--o=
    Name: Nuno Humberto
    HWID: 542320FCEE95A0604FEBB39368115D18E0947C70

2 - zeHc0F6HR6k=
    Name: Ana Isabel
    HWID: 542320FCEE95A0604FEBB39368115D18E0947C70

3 - 43MxEtdFH08=
    Name: Humberto Jorge
    HWID: 542320FCEE95A0604FEBB39368115D18E0947C70

Pick a log: 1
1 - Hi there! - OK
2 - This is a secure message. - OK
3 - Qwertyuiop. - OK
4 - Another one - OK

Message chain validation succeeded.
```

Figura 4.13: Validação com sucesso de uma cadeia de mensagens

De seguida, foi manualmente alterada a assinatura da mensagem número 2. A Figura 4.14 mostra o resultado obtido.

```
root@quetzalcoatl:~/security2016-plg9/client# ./verify.py
Choose a chat log:

1 - u24qIkgi--o=
    Name: Nuno Humberto
    HWID: 542320FCEE95A0604FEBB39368115D18E0947C70

2 - zeHc0F6HR6k=
    Name: Ana Isabel
    HWID: 542320FCEE95A0604FEBB39368115D18E0947C70

3 - 43MxEtdFH08=
    Name: Humberto Jorge
    HWID: 542320FCEE95A0604FEBB39368115D18E0947C70

Pick a log: 2
1 - One, two, three. - OK
2 - "Message example." - FAIL
3 - Potato. - FAIL

Message chain validation failed.
```

Figura 4.14: Falha na validação de uma cadeia de mensagens

4.5 Opcional: Identificação do dispositivo

Numa tentativa de fornecer aos utilizadores informação sobre a consistência do dispositivo de um cliente com o qual o mesmo pretende comunicar, é calculada e apresentada ao utilizador uma sequência de caracteres, gerada através da obtenção de vários atributos do *hardware* do cliente que possam identificar o seu dispositivo de modo minimamente consistente.

São usados os seguintes atributos:

- CPU ID de todos os processadores encontrados
- Part Number de todos os módulos de memória RAM instalados no dispositivo
- System **UUID** do dispositivo
- Endereço **MAC** da placa de rede do dispositivo

Os valores dos processadores, módulos de RAM e System **UUID** são obtidos através da **DMI**, que expõe dados do *hardware* do dispositivo a uma interface para a qual existe uma **API** para Python.

Logo após a obtenção dos valores, é gerado um [SHA](#) através dos valores obtidos.

A [Figura 4.15](#) expõe o método utilizado para o cálculo do *hardware ID*.

```
def getHardwareID(self):
    all_dmi = dmidecode.parse_dmi(dmidecode._get_output())
    cpu_ids = [] # Add found CPU IDs
    ram_partNO = [] # Add part numbers of every RAM module found
    system_UUID = [] # Add system UUID
    for dmi in all_dmi:
        if dmi[0] == 1:
            if 'UUID' in dmi[1]:
                system_UUID.append(dmi[1]['UUID'])
        if dmi[0] == 4:
            if 'ID' in dmi[1]:
                cpu_ids.append(dmi[1]['ID'].replace(" ", ""))
        elif dmi[0] == 17:
            if 'Part Number' in dmi[1]:
                if dmi[1]['Part Number'].upper() != "[EMPTY]":
                    ram_partNO.append(dmi[1]['Part Number'])
    hardwareID = SHA.new()
    for part_id in (cpu_ids + ram_partNO + system_UUID):
        hardwareID.update(part_id)
    hardwareID.update(str(uuid.getnode())) # Add network interface MAC address
    return hardwareID.hexdigest().upper()
```

Figura 4.15: Método utilizado para o cálculo do *hardware ID*

Capítulo 5

Análise de Resultados

Terminando o projecto prático, é possível observar que o resultado do desenvolvimento deste projecto é um sistema seguro para a troca de mensagens entre utilizador, que permite a utilização directa de um documento de identificação pessoal para a validação da identidade dos participantes.

O sistema possui também uma interface de utilizador minimamente agradável e que permite uma mais fácil distinção entre diferentes tipos de mensagens através da utilização de cores.

```
USERID ]: Certified ID detected.
USERID ]: Authenticating as Humberto Jorge.
SERVER ]: Received a valid certificate from the server.
SPEC ]: ECDHE_WITH_AES_256_GCM_SHA384
SERVER ]: Server correctly responded to challenge.
SERVER ]: Connected to message server.
INFO ]: Ready.
SERVER ]: Resending non-ACKED message (secure)
DEBUG ]: Secure message successfully verified!
DEBUG ]: ACK from server OK.
DEBUG ]: Secure message successfully verified!
SERVER ]: Available clients:
SERVER ]: 1: HrLxTndHCQo= - Nuno Humberto (B11) - LEVEL 2
```

Figura 5.1: Exemplo da interface do utilizador

```
[ INFO ]: Refresh certificate data? n
[ INFO ]: Detecting smartcard slots...
>>> q
[ INFO ]: Slot 0: Alcor Micro AU9520 (HUMBERTO JORGE)
[ INFO ]: Slot 1: Bit4id miniLector (NUNO HUMBERTO)
[ INFO ]: Select a slot: 0
>>> quit()
ECDHE (fres)...
```

Figura 5.2: Exemplo da interface do utilizador - deteção de slots

Capítulo 6

Conclusão

Graças aos conhecimentos adquiridos ao longo do semestre, foi possível a realização de todos os objectivos propostos no enunciado do trabalho prático. Foi possível concluir, sobretudo, que é facilmente possível implementar um sistema seguro para um uso específico, através da utilização de ferramentas e bibliotecas externas, desde que estejam presentes os conceitos referentes à sua utilização.

A realização deste trabalho prático permitiu também adquirir conhecimentos de vários ataques comuns a todo o tipo de sistemas informáticos e sobre quais os métodos utilizados para os evitar.

Glossário

ACK Acknowledgement. [16](#)

AES Advanced Encryption Standard. [3](#), [9](#), [10](#)

API Application Programming Interface. [20](#)

CA Certification Authority. [5](#), [13](#)

cipherspec String que permite identificar os algoritmos de troca de chaves, cifragem e autenticação. [9](#)

CRL Certificate Revocation List. [5](#), [13](#)

CTR Counter Mode. [10](#)

DMI Desktop Management Interface. [5](#), [20](#)

ECC Elliptic Curve Cryptography. [3](#), [7](#)

ECDHE Elliptic Curve Diffie-Hellman Ephemeral. [3](#), [7](#), [11](#)

ECDLP Elliptic Curve Discrete Logarithm Problem. [7](#)

hashing Geração irreversível de uma sequência de bits fixa derivada de uma informação ou ficheiro. [3](#), [9](#)

HMAC Hash-based Message Authentication Code. [3](#), [5](#), [11](#)

IV Initialization Vector. [10](#)

MAC Media Access Control. [20](#)

nonce Número aleatório que só pode ser usado uma vez. [10](#)

OFB Output Feedback Mode. [10](#)

PBKDF2 Password-based Key Derivation Function 2. [9](#)

PKCS1 v1.5 Esquema de assinaturas introduzido em 1993, não utiliza elementos aleatórios. [3](#), [5](#), [13](#), [15](#)

PSS Esquema de assinaturas moderno, os mesmos dados produzem constantemente assinaturas diferentes. [3](#), [5](#), [15](#)

RSA Rivest Shamir Adleman. [7](#)

salt Dado aleatório utilizado como fonte adicional de dados numa função que não é reversível. [9](#)

SHA Secure Hash Algorithm. [3](#), [5](#), [9](#), [21](#)

UUID Universally Unique Identifier. [20](#)

Bibliografia

- [1] Cryptography.io. cryptography. <https://cryptography.io/>, 2016.
- [2] pyOpenSSL. pyOpenSSL. <https://pyopenssl.readthedocs.io/en/stable/>, 2016.
- [3] Dwayne Litzenberger. Python Cryptography Toolkit. <http://pythonhosted.org/pycrypto/>, 2016.
- [4] Christian Hohnstädt. X Certificate and key management. <http://xca.sourceforge.net/>, 2016.
- [5] Ludovic Rousseau. PKCS11 Wrapper for Python. <https://ludovicrousseau.blogspot.com>, 2016.
- [6] Unknown. dmidecode Python API. <https://pypi.python.org/pypi/dmidecode>, 2016.
- [7] WSTEIN. The Elliptic Curve Discrete Logarithm Problem. <http://wstein.org/edu/2007/spring/ent/ent-html/node89.html>, 2016.
- [8] ARSTechnica. A (relatively easy to understand) primer on elliptic curve cryptography. <http://arstechnica.com/security/2013/10/a-relatively-easy-to-understand-primer-on-elliptic-curve-cryptography/2/>, 2016.