



UNIVERSIDADE DE AVEIRO

DEPARTAMENTO DE ELECTRÓNICA, TELECOMUNICAÇÕES E
INFORMÁTICA

SEGURANÇA

Sistema de Mensagens Seguras

Relatório Final - V2

8240 - MESTRADO INTEGRADO EM ENGENHARIA DE
COMPUTADORES E TELEMÁTICA

Diogo Ferreira
NMec: 76425 | P1G12

Pedro Martins
NMec: 76551 | P1G12

2017-2018

Conteúdos

Introdução	3
Especificações da Plataforma	4
1 Algoritmos/Sistemas de Cifra	4
2 Processo de <i>Handshake</i> entre Cliente-Servidor	7
2.1 Troca de Chaves de Sessão	7
2.2 Autenticação das Partes	7
2.3 <i>Handshake</i>	9
3 Mensagens RESOURCE	13
4 Criação de Utilizadores	14
5 Envio de Mensagens para Outros Utilizadores	16
6 Leitura de Mensagens	18
7 Envio e Receção de Recibos	19
8 Verificação do Estado de Recibos	21
9 Detalhes de Implementação	22
9.1 Servidor	22
9.2 Cliente	23
9.3 Gestão de certificados	24
9.4 Utilidades	26
10 Execução da Plataforma	27
10.1 Instalação e Configuração	27
10.2 Manual de instruções	27
Conclusão	29

Capítulo

Introdução

Nos dias de hoje, a segurança nas redes informáticas é um tema controverso, sendo necessária uma correta definição e implementação das suas políticas.

Neste trabalho, pretende-se desenvolver um sistema de mensagens seguras entre diversos utilizadores, autenticados com o seu Cartão de Cidadão (CC). Vários mecanismos foram implementados, tais como autenticação dos intervenientes através de assinaturas digitais, derivação de chaves de sessão entre clientes e servidor para criação de um túnel seguro, cifras híbridas para troca de mensagens e recibos de leitura entre utilizadores, todos eles descritos neste relatório.

Este documento é uma versão modificada de um relatório anteriormente desenvolvido, sendo que o foco do trabalho é o mesmo, mas com diversas alterações que tinham em mente tornar a plataforma mais segura e robusta.

Capítulo

Especificações da Plataforma

1 Algoritmos/Sistemas de Cifra

Nas trocas de informação entre servidor e clientes e apenas entre clientes, é sempre definido o "*cipher suite*" que será usado. Ele seguirá o formato *KKK-SSS_MMM-AAA_PPP-XXX_NNN_HHH_CCC_HHH-III-HHH*, correspondente a:

- **KKK**, o algoritmo de troca de chaves entre servidor e cliente:
 - **Ephemeral Elliptic Curve Diffie-Hellman (EECDH)**. O uso de curvas elípticas permite obter o mesmo nível de segurança que outros sistemas criptográficos assimétricos, mas usando chaves mais pequenas, como Rivest–Shamir–Adleman (RSA). Além disso, operações com curvas elípticas são, por norma, mais rápidas; Elliptic Curve Diffie-Hellman (ECDH) é mais rápido do que o clássico mecanismo Diffie-Hellman. Adicionalmente, o uso de um valor aleatório e a constante mudança de chaves permite evitar ataques de repetição. Em conjunção com o RSA, usado para autenticação dos intervenientes, podemos chamar a este processo de EECDH. [1]
- **SSS_MMM**, o algoritmo de cifra simétrica e o seu modo, usado para cifrar mensagens. Optou-se por apenas usar AES192 e AES256 porque, apesar de uma maior perda de performance em relação ao uso de Advanced Encryption System (AES) com chaves de 128 bits, o reforço de segurança é compensatório. Optou-se ainda pelo uso de modos de cifra contínuas, evitando-se assim a manutenção de padrões da mensagem original. Para controlo de integridade das mesmas, todas as mensagens cifradas com AES serão posteriormente assinadas usando

RSA, garantindo assim autenticação e controlo de integridade. Os modos escolhidos são:

- **AES_CBC**, que não reproduz padrões, provoca confusão na entrada da cifra, a alteração determinística de bits é dificilmente obtida, permite paralelização na decifra e recuperação após perdas de blocos inteiros;
 - **AES_CTR**, que não reproduz padrões e provoca confusão na entrada da cifra (considerando o Initialization Vector (IV) secreto) e permite paralelização na cifra e na decifra.
- **AAA_PPP**, o algoritmo de cifra assimétrica e o seu modo de *padding*, usado para cifrar chaves (processo de cifra híbrido). Optou-se apenas pelo uso de RSA (1024 e 2048 bits) face ao ElGamal, uma vez que este apenas usa uma complexidade da ordem de logaritmos modulares, enquanto que o primeiro usa também fatorização. Em relação aos modos de *padding*:
 - **RSA2048_OAEP**, o recomendado para sistemas atuais;
 - **RSA1024_PCKS1v15**, não recomendado para sistemas atuais, mas que pode atuar para retro compatibilidade.
 - **XXX_NNN_HHH_CCC_HHH**, o algoritmo de assinatura e verificação da mesma (**XXX**) e os modos de *padding* e *hashing* para interação entre servidor e cliente (**NNN_HHH**) e entre clientes (**CCC_HHH**). Optou-se por usar novamente RSA, mas apenas com chaves de 2048 bits (o atualmente recomendado para assinaturas). Quanto os modos de *padding*, optou-se por usar a melhor combinação de entre as possíveis disponibilizadas pelo CC, que é RSA2048 usando PKCS1v15 (com auxílio de SHA256) para *padding*. No entanto, o uso de PKCS1v15 atualmente já não é o recomendado, em favor do uso de Probabilistic Signature Scheme (PSS). Como o CC não suporta este mecanismo de *padding*, optou-se apenas por usá-lo nas assinaturas das mensagens por parte do servidor. As combinações possíveis são portanto:
 - **RSA2048_PSS_SHA256_PKCS1v15_SHA256**;
 - **RSA2048_PSS_SHA384_PKCS1v15_SHA256**;
 - **III**, o algoritmo de controlo de integridade. Para o efeito usar-se-á apenas Hash-based Message Authentication Code (HMAC), sendo que a função de *hashing* usada pelo algoritmo é a função genérica definida por **HHH**.

- **HHH**, o algoritmo de *hashing* usado para síntese de mensagens (geração de recibos, por exemplo). Usaram-se funções de síntese de vários tamanhos, para quando possível usar um valor maior, para diminuir a probabilidade de colisões. Portanto:
 - **SHA256**;
 - **SHA384**;

Resumindo, quando um cliente se registra na plataforma, escolhe de entre uma das seguintes especificações de cifra suportadas:

- EEC DH-AES192_CFB-RSA1024_PKCS1v15-RSA2048_PSS_SHA256_PKCS1v15_SHA256-HMAC-SHA256;
- EEC DH-AES192_CFB-RSA2048_OAEP-RSA2048_PSS_SHA256_PKCS1v15_SHA256-HMAC-SHA256;
- EEC DH-AES256_CFB-RSA2048_OAEP-RSA2048_PSS_SHA384_PKCS1v15_SHA256-HMAC-SHA384;
- EEC DH-AES192_CTR-RSA1024_PKCS1v15-RSA2048_PSS_SHA256_PKCS1v15_SHA256-HMAC-SHA256;
- EEC DH-AES192_CTR-RSA2048_OAEP-RSA2048_PSS_SHA256_PKCS1v15_SHA256-HMAC-SHA256;
- EEC DH-AES256_CTR-RSA2048_OAEP-RSA2048_PSS_SHA384_PKCS1v15_SHA256-HMAC-SHA384.

2 Processo de *Handshake* entre Cliente-Servidor

Para que as trocas de informação entre cliente e servidor possam ser feitas de forma confidencial e autenticada, é primeiramente necessário realizar um processo de *handshake* entre ambos, garantindo assim um canal seguro de comunicação.

Para tal, é necessária uma troca de chaves para que, no final deste processo, os dois intervenientes tenham um segredo comum partilhado com o qual irão cifrar todas as mensagens, evitando assim *eavesdropping attacks* e observadores passivos. No entanto, ataques Man-in-The-Middle (MiTM) não serão prevenidos apenas cifrando o canal de comunicação, os quais apenas serão contornados com a autenticação entre as partes, usando chaves assimétricas (com os respetivos certificados dos intervenientes) para assinar e verificar as assinaturas sobre as mensagens trocadas.

O processo de autenticação e troca de chaves é designado por Ephemeral Elliptic Curve Diffie-Hellman Exchange (EECDHE).

2.1 Troca de Chaves de Sessão

Para cada sessão, o servidor e o cliente têm de estabelecer uma chave de sessão para estabelecer uma ligação segura. O processo de *handshake* seguirá as linhas do EECDHE, como referido anteriormente.

O processo consiste numa troca de chaves públicas sobre um canal inseguro de modo a que, no final do processo, ambos possuam um segredo comum, do qual se poderá derivar uma chave para uso em sistemas criptográficos simétricos (como AES) para troca de mensagens. Como o segredo comum não deverá ser, em princípio, resistente a ataques *brute force*, é necessário usar-se uma Key Derivation Function (KDF) (um processo também conhecido como *key stretching*, de forma a obter uma chave de determinado tamanho a partir de outra para ser usada em determinados algoritmos criptográficos). No contexto deste trabalho, usar-se-á Hash-based Message Authentication Key Derivation Function (HKDF), pois permite-nos, de forma facilitada, evitar os ataques de repetição através do uso de um *salt* (valor aleatório), e derivar a chave para usar na cifra das mensagens.

Mais especificamente, o processo utilizado para a troca de chaves é conhecido como Ratchet Diffie-Hellman Exchange (RDHE), no qual se deriva um novo segredo a cada mensagem trocada e não a cada sessão.

2.2 Autenticação das Partes

Uma ligação segura, por si só, não garante autenticação entre intervenientes, nem previne um ataque MiTM. Para tal, é necessária a troca de certificados de autenticação e proceder à verificação das chaves públicas e

assinaturas das partes.

O CC usa chaves assimétricas RSA de 2048 bits, portanto, assumir-se-á que o servidor também funcionará de maneira equivalente, sendo a chave e os certificados do mesmo gerados usando X Certificate and key management (XCA) (foi gerado uma ServerCA que serve como Certification Authority (CA) e um certificado para o próprio servidor, assim como as correspondentes chaves; é também assumido que a ServerCA não emite quaisquer Certificate Revocation List (CRL)s, portanto, desde que o certificado do servidor não esteja expirado, será sempre válido).

O *payload* da mensagem de estabelecimento de sessão enviada pelo cliente já vai assinada pela chave de autenticação do CC do mesmo (assim como o respetivo certificado). Na receção desta mensagem, o servidor verifica a autenticidade do cliente: é primeiramente verificado se o certificado ainda não expirou e qual o seu propósito, e, de seguida, a validade é verificada junto de uma CA, usando Online Certificate Status Protocol (OCSP). Caso não exista indicação de um endereço de um serviço OCSP, deve ser descarregada a CRL e a *delta* correspondente mais recente referenciada pelo certificado. Esta verificação é levada a cabo para todos os certificados da cadeia de certificação, para que, num cenário em que todos são válidos, se possa finalmente verificar a própria cadeia (validação das assinaturas dos certificados).

De notar que, tanto os clientes como o servidor, possuem uma *cache* para armazenar temporariamente certificados considerados válidos, isto é, para cada certificado é guardada a data da última validação do mesmo e o seu *serial number*, e, de cada vez que se pretender verificar um certificado, este é procurado na *cache*. Se um certificado estiver presente em *cache*, é verificado se já passaram 24 horas desde a última validação ou se o seu *serial number* é diferente, altura em que a sua cadeia de certificação é novamente verificada para que a *cache* esteja sempre num estado integro e válido. De notar que um certificado é referenciado na cache através de um identificador que, no caso dos servidores, é o Common Name e, no caso dos utilizadores, é o número do seu Bilhete de Identidade, um número único que nunca será alterado.

Se o cliente for autenticado no servidor, o servidor gera uma resposta, calcula o Message Authentication Code (MAC), assina esse mesmo *payload* e, juntamente com o seu certificado, envia tudo isso para o cliente, para que o mesmo possa autenticar o servidor.

Se o servidor se autenticar no cliente, a sessão segura e autenticada está estabelecida e nenhuma das seguintes mensagens entre servidor e cliente necessita de ser assinada por nenhuma das partes.

A geração do MAC serve para garantir a integridade nas trocas de mensagens entre servidor e clientes. No entanto, visto que mais nenhuma mensagem trocada entre ambos, excetuando as duas primeiras, é assinada, é necessário

um mecanismo para garantir a genuinidade e sequencialidade da "conversa". Para tal, todos os MAC são gerados concatenando o *payload* da mensagem a enviar com o MAC da última mensagem recebida, usando a chave de sessão da mensagem prestes a enviar (excetuando no primeiro MAC gerado pelo servidor em que o mesmo é apenas concebido a partir da mensagem atual concatenada com o *nonce* enviado pelo cliente).

2.3 *Handshake*

O processo de estabelecimento de um canal de comunicação seguro é assim descrito:

1. Para estabelecer uma ligação segura, o cliente irá criar uma instância de **ClientSecure** (responsável por todas as funções criptográficas), que gera um par de valores ECDH e um valor aleatório (*salt*), dos quais envia o seu valor público e o *salt* para o servidor; o *payload* desta mensagem irá ser assinado pela chave privada de autenticação do CC do cliente e enviada juntamente com o seu certificado;
2. O servidor, ao receber a mensagem, irá criar uma instância de **Client** para esse cliente, no qual irá armazenar os valores ECDH gerados, *salt*, entre outros, na respetiva instância de **ServerSecure**;
3. Primeiro, o servidor irá autenticar o cliente, verificando a validade do seu certificado e a autenticidade da sua assinatura;
4. De seguida, gera também uma chave pública e privada (ECDH) e um valor aleatório para evitar ataques de repetição (*salt*);
5. Recebendo o valor público do cliente e o seu valor privado, o servidor pode gerar um segredo comum, que advém da combinação da sua chave privada e da chave pública do cliente;
6. A partir de uma combinação dos valores de *salt* (evita ataques de repetição) de ambos e do segredo comum, o servidor deriva uma chave de cifra simétrica para uso no envio da sua próxima mensagem; a partir deste momento a comunicação será cifrada (mas, teoricamente, não totalmente autenticada);
7. Juntamente com a sua mensagem cifrada (a primeira apenas levará o respetivo **user_id** no seu conteúdo caso o cliente já tenha uma conta criada, caso contrário leva uma mensagem vazia), o servidor envia para o cliente o seu valor público e o seu *salt*;

8. A mensagem criada anteriormente é assinada pela chave privada do servidor e é gerado o respetivo MAC, usando a chave de sessão obtida. Juntamente com o seu certificado, todos estes valores são enviados para o cliente;
9. O cliente, ao receber a mensagem do servidor, autentica a sua mensagem verificando o seu certificado e a respetiva assinatura;
10. Caso o servidor se autentique perante o cliente, o último deriva a chave usada para cifrar a mensagem, através do seu valor privado e do novo valor público recebido do servidor, assim como de ambos os *salt* e, sendo esta a primeira mensagem recebida, caso tenha conta criada, armazena o seu *user_id*; a geração desse segredo implica o *reset* de um *index* (*number_of_hash_derivations*) que será incrementado a cada geração de segredos comuns e que não será reinicializado até receber um novo valor público do servidor (isto garante que o processo de troca de chaves é sempre coerente entre as partes, mesmo que os clientes enviem múltiplas mensagens para o servidor sem obter resposta); esse *index* irá indicar o número de vezes que uma chave é derivada a partir dos mesmos valores ECDH e deverá ser enviado para o servidor para o mesmo efetuar o mesmo número de derivações. A cada mensagem segura recebido do servidor, é também gerado um novo par de valores ECDH para a seguinte troca de mensagens;
11. Depois de obtida a chave de sessão usada pelo servidor para enviar a mensagem, o cliente autentica a mensagem usando o MAC usado pelo servidor e guarda-o para ser posteriormente usado na geração do MAC da sua próxima mensagem.

Este ciclo repete-se enquanto houver mensagens trocadas entre um cliente e o servidor, estando descrito na Figura .1.

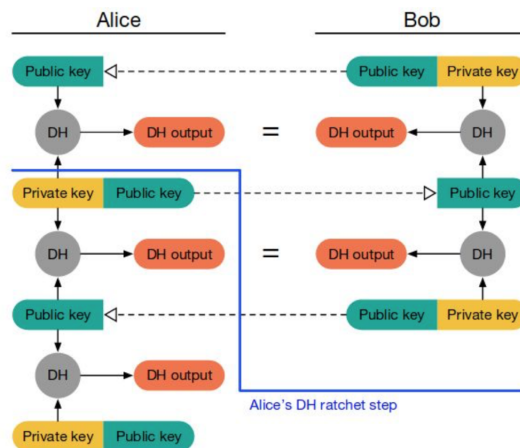


Figura .1: Processo de troca de chaves RDHE.

A primeira mensagem trocada, enviada do cliente para o servidor de maneira a estabelecer uma sessão, segue o seguinte formato:

```
{
  "type": "init",
  "payload": base64.b64encode(json.dumps({
    "uuid": <user UUID>,
    "secddata": {
      "dhpvalue": <diffie-hellman public value>,
      "salt": <128 bit random value>,
      "index": <number of hash derivations>
    },
    "nonce": <128 bit random value>
  })).encode().decode(),
  "signature": <payload signed by authentication private key>,
  "certificate": <authentication public key certificate>,
  "cipher_spec": <used cipher specification>
}
```

De notar que campos que não sejam *strings* ou valores numéricos terão de ser serializados para *bytes* e convertidos para base 64 e de seguida para *strings*, para que as mensagens JavaScript Object Notation (JSON) sejam efetivamente válidas.

Depois do envio desta primeira mensagem, e existindo um segredo comum (chave para uso em sistemas criptográficos simétricos) partilhado por ambos, pode confirmar-se que está estabelecida uma ligação segura e que as mensagens entre cliente e servidor podem ser cifradas segundo o *cipher spec* acordado. A partir desse momento, *eavesdropping attacks* e observadores passivos já não serão um problema.

Relativamente à troca de chaves, o RDHE garante *forward and backward secrecy*, visto que as chaves usadas no envio de uma mensagem são inde-

pendentes das chaves usadas nas mensagens seguintes ou anteriores, no caso de ser descoberta uma chave, não será possível obter nada relativo a outras trocas de informação. De referenciar que a derivação de uma nova chave a cada troca de mensagens é uma funcionalidade extra, mas que impõe mais uma camada de segurança na troca de mensagens entre cliente e servidor.[2]

A primeira resposta do servidor ao cliente segue tanto com uma assinatura como com um MAC. Apesar de a assinatura já providenciar controlo de integridade neste caso, o MAC ajuda a garantir a genuinidade e sequencialidade da troca de mensagens com o cliente:

```
{
  "type": "secure",
  "payload": base64.b64encode(json.dumps({
    "message": <ciphred payload of the message>,
    "secddata": {
      "dhpubkey": <public value of diffie-hellman exchange>,
      "salt": <salt used on diffie-hellman>,
      "iv": <AES initialization value>,
      "index": <number of key derivations>
    }
  })).encode().decode(),
  "mac": <hmac(nonce|payload, session_key)>,
  "signature": <payload signed by authentication private key>,
  "certificate": <authentication public key certificate>,
  "cipher_spec": <used cipher specification>
}
```

A estrutura do pacote depois de estabelecida a sessão segura e autenticada (i.e., a partir do envio da segunda mensagem do cliente) será:

```
{
  "type": "secure",
  "payload": base64.b64encode(json.dumps({
    "message": <ciphred payload of the message>,
    "secddata": {
      "dhpubkey": <public value of diffie-hellman exchange>,
      "salt": <salt used on diffie-hellman>,
      "iv": <AES initialization value>,
      "index": <number of key derivations>
    }
  })).encode().decode(),
  "mac": <hmac(nonce|payload, session_key)>,
  "cipher_spec": <used cipher specification>
}
```

3 Mensagens RESOURCE

No contexto de comunicação segura entre clientes, foi necessário introduzir um novo tipo de mensagens trocadas entre cliente e servidor, **RESOURCE**, para que estes possam pedir recursos ao servidor que necessitam para gerar a mensagem que realmente pretendem enviar.

Estas têm como intuito eventuais pedidos de chaves públicas, certificados e *cipher suites* que um cliente faça ao servidor, pois este último armazena temporariamente chaves públicas RSA e certificados do CC dos utentes. Estes recursos são eventualmente necessários para cifras, decifras, validação de assinaturas, entre outros.

Como já referido, o cliente também dispõe de uma *cache* para que possa armazenar temporariamente informações de outros utentes, evitando assim ter de realizar este pedido ao servidor a cada troca de mensagens que se dê.

O formato de uma mensagem **RESOURCE** será:

```
{
  "type": "resource",
  "ids": [<identifier of the user>, ...]
}
```

O servidor irá responder com:

```
{
  "result": [
    {
      "id": <identifier of the user>,
      "secdata": base64.b64encode(json.dumps({
        "rsapubkey": <RSA generated public key>,
        "cccertificate": <CC authentication public key certificate>,
        "cipher_spec": <cipher spec choosed by the client>
      })).encode().decode(),
      "signature": <signature over secdata by CC authentication key>
    }, ...
  ]
}
```

Quando um cliente recebe uma mensagem **RESOURCE**, o certificado do CC de **secdata** é validado (tal como explicado em 2.2) e a assinatura sobre **secdata** é verificada. Caso seja legítima, o cliente armazena a informação na sua *cache*.

De referir que todas as mensagens do tipo **RESOURCE** serão, como todas as outras, encapsuladas no **payload** das mensagens do tipo **SECURE**, referenciadas anteriormente. Uma outra possibilidade é o envio deste tipo de informação anexado a respostas do servidor a outras mensagens, de tal maneira que uma mensagem **RESOURCE** vá embutida noutra.

4 Criação de Utilizadores

No contexto da aplicação, para que um utilizador possa enviar e receber mensagens, é necessário primeiro registar-se no servidor para que lhe sejam atribuídas uma *message box* e uma *receipt box*. Portanto, logo que se inicie a aplicação do cliente, é pedido para ser efetuado o *login*. As credenciais de acesso são o UUID do utilizador (gerado a partir de uma síntese do certificado da chave de autenticação do CC, logo é necessário ter a interface com o mesmo funcional logo à partida) e uma *password*. Caso já tenha uma contra criada, o cliente irá iniciar sessão normalmente e estabelecerá a ligação segura e autenticada ao servidor. Por outro lado, caso não tenha uma conta, a mesma é criada automaticamente partir do UUID, *password* e outras informações necessárias.

Antes de comunicar com o servidor, deve ser gerado um par de chaves assimétricas RSA para, a título de exemplo, cifrar as chaves de cifra simétrica das mensagens que serão armazenadas no servidor, sendo, portanto, o primeiro passo à criação de uma conta a geração destas mesmas.

As chaves privadas anteriormente geradas necessitam de ser guardadas de forma segura, logo cada utilizador terá de usar a sua *password* para cifrar e decifrar o valor privado. Para o efeito, usar-se-ão as próprias funções de serialização da biblioteca Python `cryptography`, que permitem a salvaguarda e o carregamento para memória de uma chave privada RSA protegida por uma *password*.

De uma maneira sumária, cada utilizador terá:

- Um par de chaves RSA para cifra e decifra de chaves e outros valores sensíveis de mensagens;
- Um par de chaves RSA do respetivo CC, usadas para assinaturas de conteúdo.

Para a efetiva criação de um utilizador no sistema, é necessário enviar uma mensagem de *payload* `CREATE` para o servidor com:

- O UUID do utilizador, que consiste numa síntese do certificado da chave pública, extraído do CC;
- Um campo (`secdata`) que engloba o certificado da chave pública do CC, a chave pública RSA gerada e o *cipher spec* escolhido pelo utilizador;
- Uma assinatura sobre o campo `secdata`, gerada usando a chave privada de autenticação do CC, para que os dados do cliente no servidor não possam ser adulterados.

O formato da mensagem será:

```
{
  "type": "create",
  "uuid": <user uuid>,
  "secdata": base64.b64encode(json.dumps({
    "rsapubkey": <RSA generated public key>,
    "cccertificate": <CC authentication public key certificate>,
    "cipher_spec": <cipher spec choosed by the client>
  })).encode().decode(),
  "signature": <signature over secdata by CC authentication key>
}
```

O servidor, ao processar a mensagem **CREATE**, irá, primeiramente, verificar se o **uuid** já existe na sua base de dados. Se tal não se verificar, irá passar à validação do certificado efetuando os passos já referidos na secção 2.2. Caso esta validade se verifique, irá registar um novo perfil de utilizador com os seguintes campos:

```
{
  "id": <identifier>,
  "uuid": <user universal identifier>,
  "mbox": <message box path>,
  "rbox": <sent box path>,
  "secdata": base64.b64encode(json.dumps({
    "rsapubkey": <RSA generated public key>,
    "cccertificate": <CC authentication public key certificate>,
    "cipher_spec": <cipher spec choosed by the client>
  })).encode().decode(),
  "signature": <signature over secdata by CC authentication key>
}
```

5 Envio de Mensagens para Outros Utilizadores

Quando se pretende enviar uma mensagem para um utilizador, é primeiramente necessário cifrar a respetiva mensagem, para que apenas o destinatário e o emissor a possam ler. No entanto, as chaves utilizadas para cifrar os conteúdos terão de ser diferentes, isto é, o emissor apenas poderá decifrar a mensagem da sua *receipt box* e não a presente na *message box* do destinatário, e vice-versa.

Para se poder enviar uma mensagem, como já referido anteriormente, se o cliente não tiver o certificado e o *cipher spec* usado pelo destinatário (pois é usado sempre o *cipher spec* do destinatário para cifrar as mensagens a ele destinadas), é primeiramente necessário enviar uma mensagem **RESOURCE** para o servidor. De seguida, o processo de envio de uma mensagem segue as seguintes instruções:

1. Gerar um valor aleatório de 128 bits (*nonce*). Este valor serve para distinguir mensagens com igual conteúdo do mesmo emissor para o mesmo recetor. Também será usado para comprovar se o destinatário leu efetivamente a mensagem e será posteriormente guardado nos recibos;
2. Gerar uma chave de cifra simétrica e um IV, segundo o algoritmo especificado no *cipher suite* escolhido pelo emissor;
3. Cifrar a mensagem (cifra simétrica) usando a chave e o IV gerados;
4. Cifrar o IV, a chave e *nonce* (concatenados) com a chave pública RSA presente no perfil do destinatário;
5. Construir uma mensagem JSON com os resultados anteriores, o ID do emissor e do recetor, a mensagem cifrada e o *cipher suite*;
6. Por fim, o **payload** é assinado com a chave privada de autenticação do CC e concatenada à mensagem.

O formato resultante da mensagem seria:

```
{
  "payload": {
    "src": <id of the user that sent the message>,
    "dst": <id of the destination user>,
    "message": <ciphered message>,
    "nonce_key_iv": <ciphered iv|key|nonce used to cipher original message>
  },
}
```



```
"signature": <payload signed by authentication private key>,  
"cipher_spec": <used cipher specification>  
}
```

Os identificadores de origem e destino são particularmente úteis para verificar a validade dos recibos, pois estes permitem-nos efetivamente verificar quem enviou ou recebeu a mensagem, já que, estando cifrados, não podem ser alterados.

Por outro lado, como é necessário guardar também a mensagem na caixa de recibos do emissor, todos os passos anteriormente descritos devem ser repetidos.

Por fim, as duas "mensagens" são englobadas noutra mensagem JSON, encapsuladas num pacote **SECURE** (como anteriormente referido), e enviadas para o servidor. O servidor ao receber a mensagem, separa-as e guarda-as na *message box* do destinatário e na *receipt box* do emissor, respetivamente.

6 Leitura de Mensagens

Quando um utilizador pretende ler uma das mensagens na sua *message box*, é necessário:

1. Escolher a mensagem a ler e enviar o respetivo pedido ao servidor;
2. O servidor irá carregar o ficheiro da *message box* do utilizador e todas as informações de recursos relativos aos emissor (chaves, certificados, etc.) e enviar tudo numa só mensagem para o cliente (vai uma mensagem do tipo **RESOURCE** embutida na resposta à mensagem **RECV**);

O cliente ao receber a mensagem:

1. Faz o *parsing* da mensagem **RESOURCE** recebida juntamente com a resposta à mensagem **RECV** e caso seja necessário, armazena os valores na sua *cache*;
2. Valida o certificado, se este não estiver em *cache*;
3. Separa a assinatura da restante mensagem e, juntamente com o certificado recebido anteriormente, verifica a sua validade;
4. Obtém os restantes componentes: o *cipher spec*, as informações da cifra (chave e IV), o *nonce* e a mensagem cifrada;
5. Separa o *nonce* (usado para posterior envio de um recibo), a chave AES e o IV, depois de decifrados com a sua chave privada RSA;
6. Em concordância com o *cipher spec* da mensagem, decifra a mensagem com a chave e IV obtidos anteriormente.

Caso um cliente tente ler uma mensagem que não se encontre na sua *message box*, não será possível a decifra da chave AES, do IV e do *nonce*, logo será imediatamente mostrada uma mensagem de erro.

7 Envio e Receção de Recibos

Depois da leitura de uma mensagem, é automaticamente produzido um recibo de leitura da mesma, mas, para tal, é necessário o identificador de origem e destino da mensagem, o conteúdo decifrado da mesma e o respetivo *nonce*.

Para enviar um recibo, é criado um dicionário JSON com o seguinte formato (conteúdo do recibo):

```
{
  "src": <id of the user that sent the message>,
  "dst": <id of the destination user>,
  "message": <deciphered message>,
  "timestamp" <timestamp of the moment of receipt generation>,
  "nonce": <nonce grabbed from read message>
}
```

Este dicionário é convertido para base 64 e assinado pela chave privada de autenticação do CC do leitor da mensagem. O resultado dessa assinatura é englobado num novo dicionário JSON com o *timestamp* usado para a criação do dicionário anterior (o *timestamp* garante a frescura e a singularidade dos recibos):

```
{
  "timestamp" <timestamp of the moment of receipt generation>,
  "signature": <signature over receipt contents by CC authentication key>
}
```

Por fim, tal como uma mensagem, também o recibo (o dicionário com o *timestamp* e assinatura) deve ser cifrado para que apenas o emissor da mensagem original tenha acesso a ele, sendo necessário:

1. Gerar uma chave de cifra simétrica e um IV, segundo o algoritmo *cipher spec* escolhido anteriormente pelo emissor;
2. Cifrar o *payload* (dicionário com *timestamp* e *signature*) do recibo (cifra simétrica) usando a chave e o IV gerados;
3. Cifrar a chave e o IV com a chave pública RSA presente no perfil do remetente;
4. Criar uma mensagem JSON com os resultados anteriores, o recibo cifrado e o *cipher spec* usado;
5. Por fim, toda a mensagem é assinada com a chave privada de autenticação do CC para garantir, principalmente, que tanto o recibo cifrado como a chave e o IV não são alterados, isto é, para controlo de integridade.

O formato resultante do recibo cifrado seria:

```
{  
  "receipt": <ciphered receipt>,  
  "key_iv": <ciphered key|iv used to cipher receipt>,  
  "cipher_spec": <used cipher specification>  
}
```

Uma vez que a verificação da validade de um recibo é feita pelo emissor da mensagem original quando todos eles são requisitados, o servidor, quando recebe um recibo do destinatário da mensagem, apenas o guarda sem qualquer tipo de pós-validação.

8 Verificação do Estado de Recibos

Quando um utilizador pretende verificar o estado de uma mensagem na sua *receipt box*, basta enviar um pedido com o `id` da caixa e o `msg_id`.

O servidor ao receber o pedido, carrega o ficheiro da mensagem da *receipt box* do utilizador e todos os recibos correspondentes e envia-os para o cliente.

Ao receber a resposta à mensagem **STATUS** que previamente enviara, o cliente tem de, inicialmente, decifrar a mensagem guardada na *receipt box*. Caso não consiga decifrar a mensagem ou se, de seguida, não conseguir validar o certificado do emissor do recibo, é imediatamente apresentada uma mensagem de erro. Por fim, cada recibo deve ser individualmente decifrado e validado, da seguinte forma:

1. Obtendo a chave AES e o IV que cifraram o recibo usando a sua chave pública RSA;
2. Decifrando o recibo, obtendo o *timestamp* do momento de geração do recibo e a assinatura sobre o conteúdo do mesmo;
3. Reconstruindo o dicionário original JSON (conteúdo do recibo) que continha os identificadores de origem e destino da mensagem original, o respetivo conteúdo da mesma decifrado, o respetivo *nonce* e o *timestamp* já referido;
4. Separando a assinatura da restante mensagem e, juntamente com o certificado recebido, verificar a sua validade;
5. Se a validade da assinatura se verificar, o recibo é então apresentado como válido ao utilizador.

De notar que, o uso da assinatura sobre todo o conteúdo (os identificadores de origem e destino da mensagem original, o respetivo conteúdo da mesma decifrado, o respetivo *nonce* e o *timestamp*) garante não só que é impossível enviar um recibo sem conhecer a mensagem original e o *nonce*, mas também a frescura e unicidade do mesmo, pois é armazenado o momento da sua criação.

9 Detalhes de Implementação

9.1 Servidor

Importa referir algumas das classes e métodos usados do lado do servidor para a gestão dos detalhes de segurança.

A classe `Client` anteriormente referida usada no servidor é assim construída:

```
class Client:
    count = 0

    def __init__(self, socket, addr, registry, certs):
        self.socket = socket          # client socket
        self.bufin = ""               # in buffer
        self.bufout = ""              # out buffer
        self.addr = addr               # client address
        self.id = None                 # client id - different from user_id
        self.secure = ServerSecure(
            registry=registry, certs=certs) # instance of server side secure, where
            it is managed all secure related features, such as storing DH values
            and ciphering messages
```

Como já referido, todas as operações relativas aos aspetos de segurança são geridas do lado do servidor por uma instância de `ServerSecure`, que segue a seguinte estrutura:

```
class ServerSecure:

    def __init__(self, registry, certs):
        self.uuid = None               # user UUID
        self.cipher_spec = None        # string cipher_spec
        self.cipher_suite = {}         # dict cipher suite

        self.salt = None                # DH salt
        self.server_cert = certs.cert  # server certificate
        self.priv_value = None          # DH private value
        self.pub_value = None           # DH public value
        self.peer_pub_value = None      # last received DH pub value from client
        self.peer_salt = None           # last received DH salt from client
        self.number_of_hash_derivations = None # last number of hash derivations
            used by KDF to derive AES key received from the client
        self.prev_mac = None            # MAC of the last received message
        self.nonce = None               # random value received from client only used to
            generate first MAC

        self.private_key = certs.priv_key # Private key used in assymetric
            deciphering operations
        self.public_key = certs.pub_key   # Public key used in assymetric ciphering
            operations

        self.registry = registry         # reference to server's user registry
        self.certs = certs               # reference to server's certificate manager

    def uncapusulate_insecure_message(self, payload):
        # Uncapusulate first insecure received message from user
```

```

def encapsulate_secure_message(self, payload):
    # Encapsulate a secure message to a user

def uncapusulate_secure_message(self, message):
    # Uncapsulate a secure message from a user

```

Apesar de nos ser fornecido já uma base de um servidor inseguro, foram feitas as alterações para que se pudesse estabelecer uma ligação segura com o cliente, como já referido ao longo deste trabalho. No entanto, manteve-se o comportamento do código fornecido em que, se um cliente enviasse várias mensagens para um mesmo destinatário, todas elas seriam substituídas pela mais recente.

Por outro lado, inseriu-se um reforço de segurança em que apenas são aceites mensagens e recibos cujo o ID do remetente seja igual ao do emissor da mensagem, assim como a leitura de mensagens e a verificação das mesmas nas *message* e *receipt boxes* só é permitida se o ID do cliente da *socket* seja idêntico ao da caixa à qual ele pretende aceder.

9.2 Cliente

Do lado do cliente, a classe `ClassSecure` está encarregue da maioria das operações de segurança, apesar da classe `Client` guardar algumas informações relativas ao utilizador, tratar da ligação ao servidor, de construir todo o tipo de mensagens (CREATE, SEND, etc.) e de interagir com o cliente.

```

class ClientSecure:

    def __init__(self, uuid, private_key, public_key, cipher_spec=None,
                  cipher_suite=None, pin=None):
        self.uuid = uuid # user UUID
        self.cipher_spec = cipher_spec # string cipher_spec
        self.cipher_suite = cipher_suite # dict cipher suite
        self.number_of_hash_derivations = 1 # number of hash derivations used by KDF
        to derive AES key
        self.salt_list = [] # list of salts used to derive keys
        self.nonce = None # random value to validate MAC, used on init
        message with server
        self.cc_cert = cc.get_pub_key_certificate() # cc authentication pub key
        certificate
        self.certificates = certificates.X509Certificates() # instance of certificate
        management class

        self.priv_value = None # DH private value
        self.pub_value = None # DH public value
        self.peer_pub_value = None # last received DH pub value from server
        self.peer_salt = None # last received DH salt from server
        self.private_key = private_key # RSA private key
        self.public_key = public_key # RSA public key
        self.prev_mac = None # MAC of the last received message

        self.cc_pin = pin # CC pin

        self.user_resources = {} # cache of other users' resources

```

```

def encapsulate_init_message(self):
    # Encapsulate first message in order to establish a secure session with the
    # server

def encapsulate_secure_message(self, payload):
    # Encapsulate a secure message to the server

def uncapusulate_secure_message(self, message):
    # Uncapsulate a secure message to the server

def encapsulate_resource_message(self, ids):
    # Encapsulate a resource payload
    # The result of this method is also being encapsulated in a secure payload

def uncapusulate_resource_message(self, resource_payload):
    # Uncapsulate the response to a resource message sent to the server
    # If the client already has the resources relative to that user in the cache
    # those will be replaced

def cipher_message_to_user(self, message, src_id, dst_id,
                           peer_rsa_pubkey=None, nonce=None,
                           cipher_suite=None):
    # Cipher the message that will be saved in other users' message box
    # Or even cipher the one that will be saved in the receipt box

def decipher_message_from_user(self, payload, peer_certificate=None):
    # Decipher a message saved in a client's receipt or message box

def generate_secure_receipt(self, src_id, dst_id, message, nonce,
                           peer_rsa_pubkey, cipher_suite):
    # Generate a secure receipt using the provided message nonce

def decipher_secure_receipt(self, payload):
    # Decipher a secure receipt

def verify_secure_receipts(self, src_id, dst_id, deciphered_message,
                           nonce, peer_certificate, receipts):
    # Calls decipher_message_from_user() method to decipher the message present
    # in the receipt box and deciphers and validates all the receipts

```

9.3 Gestão de certificados

De forma a poder fazer uma melhor gestão dos certificados que são usados como objeto de validação das várias mensagens trocadas, foi criada uma classe `X509Certificates` dotada de vários métodos úteis para manipulação de certificados OpenSSL X509, como o download de uma CRL, execução de pedidos OCSP, importação dos certificados dos utilizadores, entre outros. De todos os métodos disponíveis, destacam-se os seguintes:

```

def check_expiration_or_revoked(self, cert_entry):
    # Checks if a cert has already expired
    # First tries to validate it using OCSP
    # Then, if OCSP is not available, downloads CRLs and deltas
    # Checks if the certificate has been revoked

```



```
def validate_cert(self, cert):  
    # Checks if certificate is already in cache  
    # Otherwise, checks if it has extension KeyUsage with a checked digital_signature  
    # field  
    # Checks if all certificates in the chain are valid (not expired or revoked)  
    # Checks if the chain is valid
```

É neste módulo que está integrada a *cache* de certificados válidas e nele que se faz a gestão da mesma, como explicado em 2.2.

9.4 Utilidades

Foram também desenvolvidos vários módulos:

- `cipher_utils`, que contém métodos gerais para geração de valores aleatórios, chaves, sínteses, execução de cifras e decifras, importação de chaves, entre outros;
- `cc_interface`, que comunica diretamente com o *middleware* do CC e que permite a importação de certificados, chaves públicas e ainda permite a construção de assinaturas com as chaves privadas do mesmo;
- `log`, que cria um *logger* que facilita o debug da aplicação (para ligar/-desligar o mesmo, basta alterar o valor de `self.logger.propagate`);
- `lib`, que contém vários o caminho para vários diretórios essenciais ao bom funcionamento da aplicação.

10 Execução da Plataforma

10.1 Instalação e Configuração

Para a correta execução da plataforma, primeiro é necessário garantir que se têm instalado Python3 (foi testado usando Python 3.6.4). De seguida devem ser instaladas todas as bibliotecas usadas:

```
$ pip3 install -r requirements.txt
```

Deve-se também garantir que se tem o *middleware* do CC corretamente instalado e configurado, pois este é um sistema de mensagens de seguras entre utilizadores do CC. Como tal, também os certificados do mesmo devem ser corretamente descarregados, no entanto, os mesmos já são disponibilizados na pasta **certs** (tanto do lado do servidor como do cliente).

Para executar o servidor, basta garantir que a porta 8080 está disponível e executar:

```
$ python3 src/Server/server.py
```

Para abrir uma consola de cliente, basta:

```
$ python3 src/Client/client.py
```

Foi ainda criado um pequeno *script* (*delete_accounts.sh*), que permite fazer um *reset* às contas de utilizador registadas no sistema (tanto no cliente como no servidor), e que poderá ser útil para efeitos de teste (e.g., mudar de *cipher_spec*), uma vez que a quantidade de contas de utilizadores que podemos criar é muito limitada, devido ao número de CC que alguém possua no momento:

```
$ chmod +x src/delete_accounts.sh  
$ ./src/delete_accounts.sh
```

10.2 Manual de instruções

Depois de efetuada a instalação dos módulos necessários à plataforma, e de colocado em execução o servidor e cliente, poderá passar então a usufruir das funcionalidades oferecidas pelo mesmo. De relembrar que muitas das transações efetuadas entre servidor e cliente têm como objeto chaves e certificados presentes no CC, pelo que é essencial que o mesmo esteja conectado ao seu computador.

Primeiro é oferecida ao cliente a possibilidade de entrar em modo *debug*. Este modo irá mostrar todos os *logs* gerados e irá permitir enviar mensagens para o servidor com identificadores diferentes do do cliente. No entanto, serão sempre exibidas mensagens de erro, pois estas operações não são permitidas.

Se nunca se tiver registado na plataforma, o utilizador será prontamente convidado a escolher uma *password* que servirá de proteção dos seus valores

privados no espaço de armazenamento. Cada vez que iniciar sessão, esta *password* será pedida, para que estes valores possam ser usados. A par da *password*, e apenas no processo de registo, será convidado igualmente a escolher um dos *cipher_specs* disponíveis (indicados em 1), que será utilizado daí em diante. Será, ainda, perguntado se se pretende guardar em *cache* o seu PIN do CC, uma vez que este será necessário em várias etapas durante o uso da plataforma. Se se indicar que não deseja manter em *cache*, cada vez que for necessária a sua utilização, ser-lhe-á pedido para introduzir o mesmo.

Depois de estar registado e autenticado no servidor, terá à sua disposição as seguintes opções:

1. **Listar as *message boxes* dos utilizadores registados**, podendo igualmente indicar o ID de um utilizador em específico para listar uma em específico, ou simplesmente listá-las todas;
2. **Listar novas mensagens**, indicando o ID do utilizador que deseja verificar;
3. **Listar todas as mensagens (enviadas e recebidas)**, indicando o ID do utilizador em causa;
4. **Enviar uma nova mensagem**, indicando o ID do recetor e a mensagem a enviar. Depois do envio será-lhe indicado o ID da mensagem no sistema, bem como o do recibo, para que possa consultar mais tarde.
5. **Ler uma mensagem**, indicando o ID da *message box* e da mensagem que deseja ler (de lembrar que o ID de uma mensagem é alterado quando esta é lida, pelo que se deve consultar a listagem de mensagens se não se recordar do mesmo). Depois de lida a mensagem, será automaticamente enviado um recibo de leitura para o remetente.
6. **Verificar o estado de uma mensagem**, indicando o ID da *receipt box* e da mensagem cujo estado deseja consultar (por estado subentende-se verificar se uma mensagem já foi lida pelo seu destinatário). Se a mensagem já tiver sido efetivamente lida, ser-lhe-ão apresentados os vários recibos de leitura (com a indicação da sua validade), com algumas informações sobre o mesmo (data, ID do remetente, assinatura sobre a mensagem, entre outros).

Capítulo

Conclusão

Neste relatório pretendeu-se elaborar um plano para a elaboração de um sistema de comunicação de mensagens seguras entre utilizadores. Ao longo do desenvolvimento deste projeto foram sendo colocados em debate vários aspetos técnicos relativos à segurança, muitos dos quais nos fizeram efetivamente pensar e duvidar do quão seguros são os sistemas que usamos no nosso dia-a-dia, mais em particular os de *instant messaging* e *email* e do quão anónimos poderemos ser nesta imensa rede que é a Internet.

Pretendeu-se também tornar o sistema mais robusto, principalmente ao nível de geração e verificação de recibos seguros.

Acrónimos

MiTM Man-in-The-Middle

RSA Rivest–Shamir–Adleman

PSS Probabilistic Signature Scheme

ECDH Elliptic Curve Diffie-Hellman

EECDH Ephemeral Elliptic Curve Diffie-Hellman

EECDHE Ephemeral Elliptic Curve Diffie-Hellman Exchange

RDHE Ratchet Diffie-Hellman Exchange

AES Advanced Encryption System

KDF Key Derivation Function

HKDF Hash-based Message Authentication Key Derivation Function

MAC Message Authentication Code

HMAC Hash-based Message Authentication Code

IV Initialization Vector

CC Cartão de Cidadão

CRL Certificate Revocation List

OCSP Online Certificate Status Protocol

CA Certification Authority

XCA X Certificate and key management

JSON JavaScript Object Notation

Bibliografia

- [1] Entrust. Zero to ecdh in 30 minutes. [Online; acedido em Novembro 2017].
- [2] Robert Picciotti. Signal & the double ratchet. [Online; acedido em Novembro 2017].
- [3] André Zúquete and João P. Barraca. Slides segurança 2017-2018. [Online; acedido em Novembro 2017].
- [4] Individual Contributors. Cryptography.io. [Online; acedido em Novembro 2017].
- [5] Paul Bakker. Why use ephemeral diffie-hellman. [Online; acedido em Novembro 2017].