universidade de aveiro

Segurança Informática e Nas Organizações

Projeto 2/3

Digital Rights Management

Grupo: Leandro Silva (93446)
Mário Silva (93430)

Disciplina: Segurança Informática e Nas Organizações

Índice

Introdução

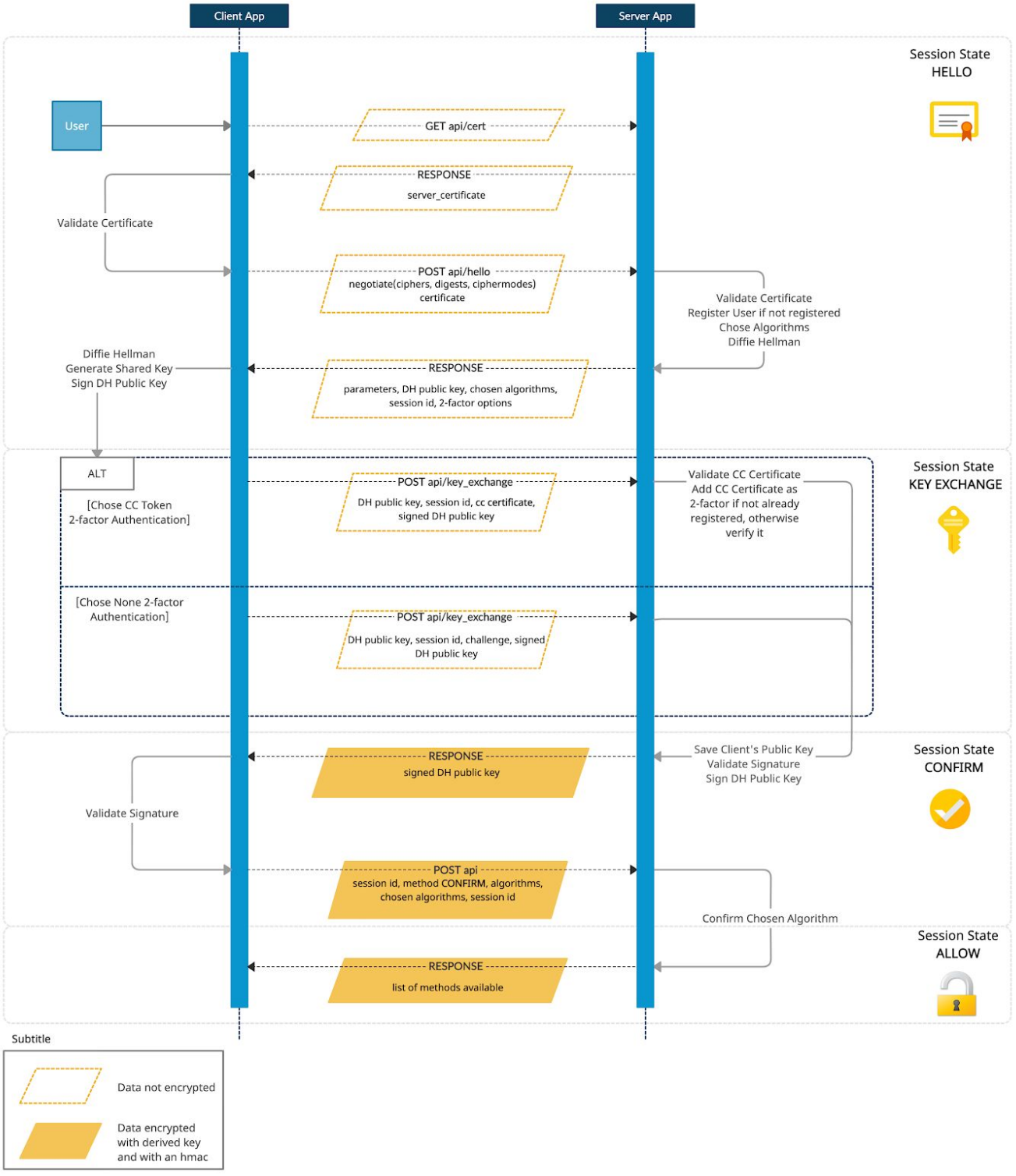
Este projeto tem como finalidade a criação de um protocolo que permita o estabelecimento de uma ligação segura entre duas entidades com o propósito de distribuir seguramente ficheiros multimédia.

A primeira parte do projeto (2) estabelece uma sessão segura entre as duas entidades e a segunda parte (3) trata da autenticação, controle de acesso e confinamento.

O servidor dispõe de uma API que recebe os pedidos do cliente através do protocolo de transporte HTTP.

Arquitetura

Comunicações Iniciais de uma Sessão



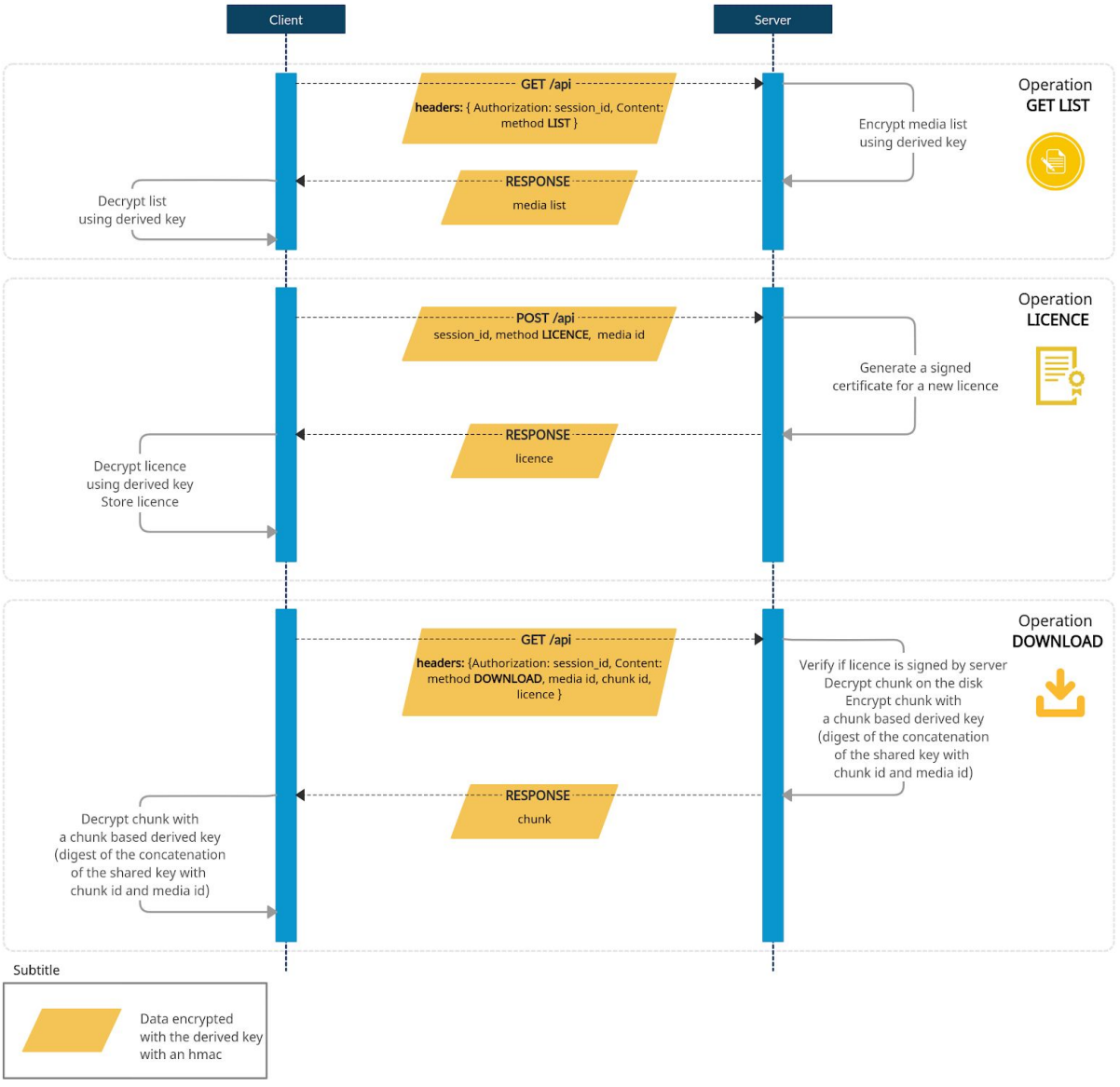
Descrição

Numa primeira fase, o cliente e o servidor trocam de **certificados** validando a autenticidade de cada um, fazem também a **negociação** dos algoritmos e de obter uma chave secreta partilhada através de Diffie-Hellman.

De seguida, é possível escolher outro fator de **autenticação** se o **utilizador** estiver a registar-se no servidor, e se estiver a fazer *log in*, terá que completar o *log in* **obrigatoriamente** com o 2-fator **se o tiver registado** no servidor para poder conectar-se.

Depois de ter a chave secreta é verificado se o servidor escolheria os mesmos algoritmos que escolheu inicialmente pois agora **todas as mensagens trocadas a partir deste ponto são encriptadas**.

Operações



Descrição

Existem três operações possíveis de efetuar, uma para obter a lista dos ficheiros multimédia, outra para um cliente conseguir obter novas licenças e outra para fazer o *download* da multimédia.

Implementação

Na implementação iremos descrever o protocolo seguindo a ordem das *features* pedidas no guião. Será também usado como referência os diagramas finais da arquitetura.

1. Estruturas

O servidor utiliza dois dicionários ***users*** e ***sessions*** para suportar múltiplos utilizadores e sessões ao mesmo tempo.

É atribuído a cada cliente, quando realizada o Diffie-Hellman um **id**, esta parte deveria ser realizada em concorrência para evitar que uma sessão fique com o mesmo id.

Numa **sessão**, é guardado as chaves do Diffie-Hellman do cliente e servidor, a chave pública do cliente e o seu certificado, os protocolos usados para as comunicações e o **estado** da sessão.

```
class Session:
    PUB_KEY = 0
    PRI_KEY = 1
    CLIPUB_KEY = 2
    SHARED_KEY = 3
    CIPHER = 4
    DIGEST = 5
    MODE = 6
    STATE = 7
    CERT = 8
```

Em cada mensagem depois de ter sido atribuído um id ao cliente, este tem que enviar o id no **header** para o servidor poder identificá-lo e ter acesso a estes atributos.

Para registar e conectar os utilizadores é utilizado um dicionário que guarda o certificado do Cartão de Cidadão se o utilizador utilizar esse fator de autenticação.

```
class User:
    CC_TOKEN = 0
```

2. Ordem das operações

Para uma sessão ter acesso às operações que o servidor dispõe tem de chegar a um estado **ALLOW**, passando pelos estados de **HELLO**, onde se fazem as trocas iniciais de negociação dos algoritmos, seguindo para o próximo estado **KEY EXCHANGE**, onde são trocados os valores do Diffie-Hellman e de seguida passa para o estado **CONFIRM**, onde se verifica que não houve alterações dos dados trocados anteriormente como explicado acima.

```
class State:
    HELLO = 0
    KEY_EXCHANGE = 1
    CONFIRM = 2
    ALLOW = 3
```

3. Negociação de Cifras

A primeira implementação realizada foi um protocolo que promovesse um acordo entre o cliente e o servidor da *cipher*, *digest* e *cipher mode* a serem utilizados para estabelecer uma comunicação encriptada.

O cliente começa por fazer um POST com o método **HELLO**, no qual apresenta uma lista com os seus protocolos suportados, de seguida, o **servidor escolhe** os que são compatíveis e que considera **mais seguros e eficientes**.

Como as cifras transmitidas não são encriptadas, por serem trocadas antes da existência de uma chave secreta partilhada, existe o risco de um atacante adulterar a mensagem e alterar a cifra escolhida pelo servidor. Por isso, depois do estabelecimento de uma chave secreta partilhada, o cliente faz um POST para a API com o método **CONFIRM** com os protocolos que enviou inicialmente, e o servidor confirma se escolheria os mesmos aos que tem registados no seu dicionário.

4. Negociação de Chaves Efémeras

Juntamente com a negociação de cifras, também estabelecemos uma negociação de chaves para a encriptação das mensagens trocadas com uma chave em comum entre ambas as partes.

Para isso, precisamos de um método que estabelecesse um compartilhamento de chaves secreto. O **Diffie-Hellman** é um exemplo prático disso mesmo, onde a chave partilhada pode ser usada para encriptar mensagens usando uma cifra de **chave simétrica** decidida na negociação de cifras.

O servidor é a entidade que inicia a geração dos parâmetros necessários para gerar uma chave em comum. Depois de gerar as suas chaves pública e

privada a partir dos parâmetros, estes são passados ao cliente desencriptados junto com a chave pública, também desencriptada.

O cliente, por sua vez, cria as suas chaves pública e privada a partir dos parâmetros recebidos e gera uma **chave secreta** com base na chave pública do servidor e na sua chave privada. Depois, o cliente manda na próxima mensagem a sua chave pública com a qual o servidor cria a chave secreta compartilhada por ambos.

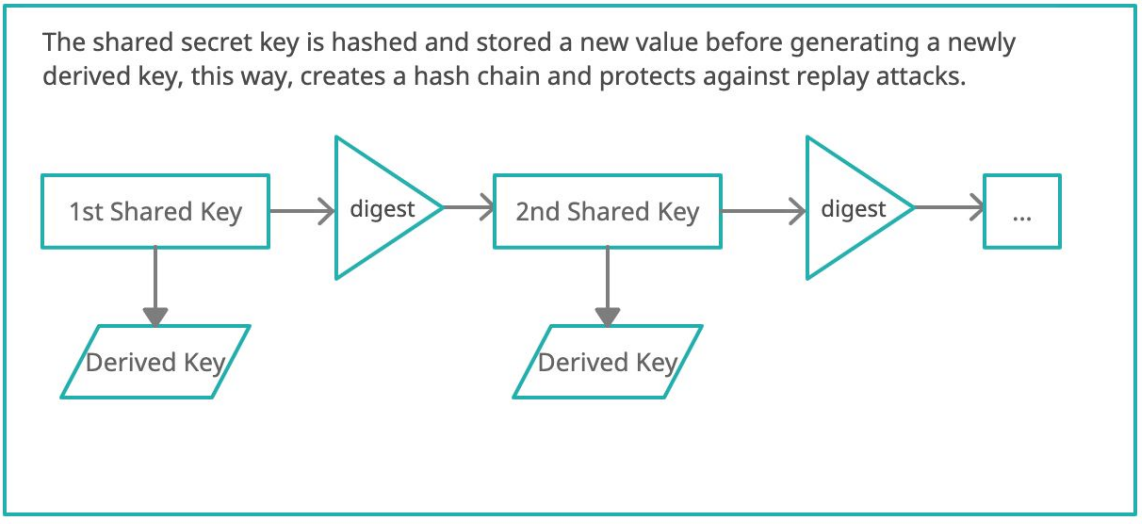
A geração da chave secreta partilhada é efetuada sempre que é iniciada uma nova sessão, resultando assim a **efemeridade** das chaves.

5. Rotação de chaves

Tendo a chave e cifra partilhada entre o cliente e o servidor, torna-se fácil a encriptação das mensagens trocadas. Porém, para cifras simétricas é recomendado mudar a chave como uma boa prática de segurança, pois ajuda a prevenir ataques de força bruta e limita o número de mensagens vulneráveis no caso da chave ser comprometida em algum momento.

Seguindo os requisitos do projeto, implementamos uma rotação de chaves baseada em *chunks*, onde concatenamos à chave secreta partilhada o media id e o chunk id e é realizada uma *Hash Key Derivation Function* sobre esse resultado para derivar uma nova chave utilizada na encriptação e na desencriptação dos dados.

Decidimos também introduzir uma rotação mais geral de chaves, onde a chave secreta partilhada é alterada para a *hash* dela própria sempre que é trocada uma mensagem. Esta cadeia de *hashes* permite evitar *replay attacks*, onde um atacante reenviaria os pedidos para obter o seu efeito 2 vezes. Isto poderia ser perigoso, por exemplo, se o servidor possuísse licenças baseadas em número de visualizações, pois um *replay attack* poderia consumir tal licença sem proveito do utilizador.



6. Integridade e Autenticidade do Conteúdo

Para alcançar a **integridade** das mensagens, precisamos de incluir nas mensagens enviadas algo como uma *digest* dela própria (**MIC**), de modo a que o destinatário possa conferir se a *digest* da mensagem feita por ele corresponde à *digest* enviada. Caso isto se verifique, este pode concluir que não houve alterações na mensagem.

Contudo, isto não fornece **autenticidade** ao conteúdo, pois um atacante, conhecendo a *digest* usada pelo cliente e servidor, poderia facilmente alterar as mensagens sem ser percebido.

O **HMAC** não detém esse problema, pois usa gera um MAC a partir de uma chave secreta. Como esta não é conhecida por entidades exteriores, torna-se impossível forjar MACs, garantindo assim tanto a integridade como a autenticidade das mensagens.

O HMAC difere das assinaturas digitais, pois os valores dos MACs são gerados e verificados usando a mesma chave secreta em vez da chave privada de um par de chaves. Por essa razão, os MACs **não** provêm a **não repudiação** do conteúdo.

7. Licenças

Para um cliente ter acesso a qualquer ficheiro de multimédia necessita de obter primeiro uma licença que comprove que este pode ter acesso.

As licenças são certificados auto-assinados pelo servidor, com um **prazo de validade** de 10 minutos.

Sempre que o cliente compra uma licença, o servidor retorna-a ao cliente para este guardá-la no seu dicionário de licenças.

Já tendo a licença para determinado ficheiro multimédia, o cliente envia no primeiro pedido de *download* do primeiro *chunk* a licença.

O servidor, recebendo este primeiro pedido, antes de responder com o respectivo *chunk*, verifica se o certificado foi assinado pela chave do servidor, se não se encontra expirado e se pertence ao cliente em questão.

8. Infraestrutura de Chaves Públicas

Um certificado emitido por uma **autoridade de certificação CA** serve para autenticar a sua chave pública. Dito de uma forma simples, um certificado é a sua chave pública assinada pela chave privada da CA.

As CAs operam numa hierarquia estrita, com as **root CAs** no topo da hierarquia, e com os **intermediate CAs** formando assim uma estrutura em árvore. Estas CAs intermediárias são bastante necessárias, considerando uma situação na qual a chave privada de uma root CA é comprometida por algum motivo, uma solução para isso seria atualizar o software para substituir a chave pública extinta da root CA, mas esta solução não é ótima pois existem milhões de dispositivos nos quais o software é raramente atualizado ou nunca. No entanto, isto não é necessário quando é uma **intermediate CA** a ser comprometida, pois os certificados afetados podem ser simplesmente adicionados a uma **Lista de Revogação de Certificados CRL** de uma CA de nível superior.

Em relação a como usamos a CA para validar um certificado:

- ➔ Carrega-se o certificado e verifica-se que não está expirado.
- ➔ É atualizada a **CRL** inicialmente.
- ➔ É construída uma *chain* verificando se chave pública da CA que validou esse certificado está ou nas root ou intermediate CAs, este processo é feito de forma recursiva até que se chega a um certificado **self-signed**.
- ➔ Chegando a um self-signed verifica-se se é uma **trusted root CA.**
- ➔ Depois valida-se a chain, verificando se os certificados estão assinados corretamente, e também verifica-se se na lista de **CRLs** o certificado atual encontra-se revogado.
- ➔ Neste ponto é possível validar o certificado.

Esta **root CA**, tem de ser confiada tanto pelo cliente como pelo servidor, para ambos poderem validar os certificados um do outro.

Para facilitar a leitura e escrita do código, criámos uma classe **CA** no ficheiro *certificate_authority.py* que trata destes processos.

9. Autenticidade das Entidades

Na negociação de chaves usamos o Diffie-Hellman para gerar uma chave secreta compartilhada pelo cliente e servidor. Contudo, numa implementação simples como esta, existe o risco de **ataques Man-In-The-Middle**. Para evitar estes ataques, fazemos também um **Authenticated Diffie-Hellman**.

Para chegarmos a tal autenticidade, precisamos que cada entidade tenha adquirido e validado um certificado da outra anteriormente. Depois, a "**chave pública**" do Diffie-Hellman que cada um envia é **assinada** usando a **chave privada** do seu próprio **certificado**, desta forma, o remetente da chave, consegue verificar com certeza que esta chave não foi alterada por nenhuma entidade no meio da comunicação.

Mesmo que um atacante intercepte os parâmetros enviados pelo servidor, não consegue iludir nem o servidor nem o cliente, a trocarem a chave secreta com ele, uma vez que não conseguirá forjar a assinatura de nenhum dos dois (isto se o certificado de ambos for antes validado por uma root CA confiável).

Dito isto, podemos afirmar que o Authenticated Diffie-Hellman garante a **não repudiação**, o que é importante para evitar, por exemplo, falsas alegações de clientes a dizer que não compraram determinada licença.

Outra abordagem que pensámos foi encriptar com a chave pública da outra entidade, pelo que apenas ela seria capaz de decifrá-la, no entanto, seria desnecessário visto que o objetivo do DH é mesmo conseguir obter um segredo partilhado ao transmitir dados públicos, no entanto, seria uma possibilidade também.

10. Autenticação de 2 Fatores

É dada a opção ao cliente de escolher um fator de autenticação extra se quiser, neste caso, pode optar por autenticação apenas pelo **certificado** emitido pela root CA através da **assinatura** na chave pública do Diffie-Hellman, a outra opção é através do **Cartão do Cidadão (CC)**.

Este fator extra tem ainda 2 fatores por si só, pois para autenticar com o CC, é preciso ter o objeto físico, o **cartão**, que não é algo fácil de replicar e uma pessoa estando do outro lado do mundo, dificilmente consegue obter o cartão, e também precisa de ter o **Pin** para poder assinar um *challenge* gerado aleatoriamente.

O servidor ao receber o certificado do cartão primeiro valida o certificado através das funções presentes no *certificate_authority.py*, para tal, é preciso ter os certificados intermédios das entidades certificadoras do estado Português, para testar o código fizemos *download* de apenas aquelas que assinaram os nossos cartões, no entanto, como existem várias entidades, metemos essa parte do código em comentário para o código ser utilizável com outros cartões.

A seguir a esta validação, é verificada se a assinatura do *challenge* foi realmente efetuada pelo certificado, garantindo assim que o cliente realmente o cliente tem também acesso ao **Pin**.

11. Proteção do Conteúdo

A encriptação do conteúdo é fundamental para obtermos o mínimo de segurança. Todas as mensagens trocadas depois da negociação das cifras e chave secreta devem ser encriptadas, de modo que uma entidade exterior não consiga perceber nem o seu conteúdo, nem o seu propósito.

Em vista disso, a constituição das mensagens encriptadas segue-se da seguinte maneira:

```
def encrypt_request(self, data):
      """Encrypt a request with integrity validation."""

      derived_key, hmac_key, salt = self.gen_derived_key()
      data, iv = self.encrypt_data(derived_key, data)
      data = self.gen_MAC(hmac_key, data)

      return json.dumps({
          'salt': binascii.b2a_base64(salt).decode('latin').strip(),
          'data': binascii.b2a_base64(data).decode('latin').strip(),
          'iv': binascii.b2a_base64(iv).decode('latin').strip(),
      }).encode('latin')
```

- O **IV** e o **salt** são sempre enviados em conjunto com a **data** e sem nenhuma encriptação (são seguros de serem públicos), para o destinatário conseguir desencriptar o conteúdo desta.
- A encriptação é feita noutra função que usa as cifras e chave decididas na negociação.
- Outra função concatena ao final da **data** o MAC usado para verificar a integridade e autenticidade do conteúdo.

O *IV* também é usado para identificar o **nonce** usado do **ChaCha20**.

No lado do servidor, podem ser acrescentados mais 2 atributos, o **media_id** e o **chunk_id**, usados para determinar a chave derivada num momento da rotação de chaves.

Percebemos que seria melhor também concatenar o *IV* e o *salt* à data encriptada em vez de usar identificadores para estes. Apesar dos identificadores não oferecerem nenhum risco significativo, contribuiria para uma maior entropia nas mensagens trocadas.

Também o conteúdo do **disco** deve ser encriptado, caso o acesso a este seja comprometido. Para isso, usamos a cifra **AES**, por ser rápida e criptograficamente forte, juntamente com o modo **CBC**.

É necessário guardar o *IV* e o *salt* utilizado na encriptação, para depois ser possível a sua desencriptação.

A chave usada é derivada de uma palavra passe secreta usando o **PBKDF2HMAC**.

Conclusão

Este projeto permitiu-nos aprofundar os conhecimentos e explorar mais sobre os temas lecionados, tais como, os processos envolventes de uma PKI, e de confidencialidade, integridade e autenticidade dos dados transmitidos.

Para conseguirmos construir protocolos e mecanismos seguros, colocamos-nos várias vezes no papel dos atacantes para encontrarmos pontos vulneráveis na nossa implementação. Imaginamos todas as maneiras em que as mensagens trocadas poderiam ser manipuladas por agentes exteriores, assim como os seus efeitos e impactos causados.

Por estas razões apresentadas, estamos bastante satisfeitos com o nosso trabalho e consideramos ter sido fundamental.

Bibliografia

https://engineering.purdue.edu/kak/compsec/NewLectures/Lecture13.pdf
https://cryptography.io/
http://sweet.ua.pt/jpbarraca/page/teaching/sio/2020/