Abril 7, 2024

Relatório SSI - TP1 - Grupo 26

Ivan Ribeiro (a100538) - IVSOP

Pedro Ferreira (a100709) - pedromeruge

Duarte Afonso Freitas Ribeiro (a100764) - DuarteRibeiroo

Sumário

1) Introdução

Neste trabalho foi desenvolvido um serviço de Message Relay focado na componente de segurança. O serviço é suportado por dois servidores: o primeiro com função de receber, armazenar e reencaminhar as mensagens entre os utlizadores do sistema; o segundo para registo e validação de timestamps. Foram implementadas as funcionalidades base e de valorização do enunciado.

2) Estrutura da solução

O projeto está estruturado em quatro pastas: `common/`, `client/`, `server/` e `timestamp_server/`. Destas pastas destacam-se alguns ficheiros.

`client/` inclui:

- `client_class.py` - parse e processamento de comandos
- `client_msgs.py` - mensagens enviadas de cliente para servidor
- `client_socket.py` - enviar e receber mensagens pela socket; handshake com o servidor
- `msg_client.py` - main do cliente

`server/` inclui:

- `client_record.py` - registos de cada cliente no servidor
- `server_class.py` - classe com todos os registos de clientes
- `logs/` - onde se guardam os logs resultantes de comunicacoes dos clientes com o servidor
- `msg_server.py` - main do servidor
- `server_msgs.py` - mensagens enviadas de servidor para cliente; mensagens guardadas no servidor
- `server_worker.py` - parse e processamento de pedidos

`timestamp_server/` inclui:

- `timestamp_server_worker.py`- parse e processamento de pedidos
- `timestamp_server.py`- main do servidor de timestamp

`/common` inclui:

- `timestamp_server_msgs.py` - mensagens entre cliente e servidor de timestamp

Para o correto funcionamento do serviço, é necessário iniciar simultaneamente o servidor principal e o servidor de timestamps. Os comandos descritos de seguida devem ser executados na pasta root do projeto "TPs/TP1".

Para começar o servidor base, executar:

- `python3 main.py msg_server`

Para começar o servidor de timestamps, executar:

- `python3 main.py timestamp_server`

Para começar um cliente, executar:

- `python3 main.py client [client args]`

2.1) Protocolo e formato dos dados

As mensagem de comunicação entre cliente e servidor têm id e dados. O id permite a distinção dos diferentes tipos de mensagem. As mensagens são serializadas e desserializadas com recurso a BSON.

Apresentamos de seguida uma visão geral da comunicação entre clientes e servidor para cada tipo de comando da aplicação. Os pormenores e justificação dos comandos surgirão no tópico 2.2.

- Para **enviar uma mensagem**, inicialmente, cliente envia <UID> de destino e assunto da mensagem; servidor devolve-lhe chave pública do destinatário; cliente envia o corpo da mensagem encriptado; servidor devolve mensagem de sucesso ou insucesso.
- Para **solicitar a fila de espera**, cliente não envia dados adicionais na mensagem (apenas o id da mensagem); servidor devolve queue na forma de string.
- Para **solicitar mensagem específica**, cliente envia <NUM> da mensagem; servidor devolve campos da mensagem (inclusive corpo da mensagem encriptado) e chave pública de emissor; cliente desencripta e valida mensagem recebida.

2.2) Requisitos

Por tópicos, apresentamos as decisões tomadas e arquitetura obtida que permitem cumprir os requisitos do enunciado.

2.2.a) Comunicação entre cliente e servidor

Para criar uma ligação segura para comunicação entre servidor e cliente, recorremos ao protocolo *station-to-station*, adaptado da ficha prática S7. Este segue os seguintes passos:

- Simultaneamente, cliente e servidor utilizam parâmetros DH fixos para criar segredo privado/público e carregam as suas chaves RSA privadas.
- Cliente envia o seu segredo público a servidor.
- Servidor guarda segredo público de cliente.
- Servidor envia a cliente: o seu segredo público, assinatura de segredo público de ambos com chave RSA privada, certificado do servidor.
- Cliente guarda segredo público de servidor.
- Cliente valida certificado e assinatura de servidor.
- Cliente envia assinatura de segredo público de ambos com chave RSA privada, e certificado de cliente.
- Servidor valida certificado e assinatura de cliente.
- Cliente e servidor derivam segredo partilhado com HDKF, utilizando o seu segredo partilhado e segredo público do outro.

Toda a interação posterior envolve uma ligação com encriptação AES GCM (usando como chave o segredo partilhado), garantindo confidencialidade, integridade a autenticação para todas as mensagens enviadas entre cada cliente e o servidor. Assim sendo, a comunicação é protegida contra acesso ilegítimo e/ou manipulação de outros, inclusive utilizadores do sistema.

2.2.b) Conteúdo das mensagens

Por sua vez, para garantir autenticidade, integridade e confidencialidade do conteúdo das mensagem entre o cliente recetor e o cliente destinatário, visto que o servidor não deve poder manipular o conteúdo e destino das mensagens, seguimos a estratégia referida de seguida, por passos (ver figuras 1,3,4,5). Os dados relativos ao corpo da mensagem são ilustrados na figura 2.

Na comunicação origem-servidor (figura 1):

- Após receber do emissor quem será o destinatário de mensagem, servidor manda certificado do destinatário ao emissor.
- Emissor valida certificado e extrai chave pública do destinatário
- Emissor assina o corpo da mensagem com a sua chave privada
- Emissor cria chave simétrica randomizada de 32 bytes
- Emissor encripta corpo+assinatura com a chave simétrica e algoritmo AES
- Emissor encripta chave simétrica com chave pública do destinatário

- Emissor envia corpo+assinatura encriptado e chave simétrica encriptada ao servidor, que guarda os dados (figura 2)

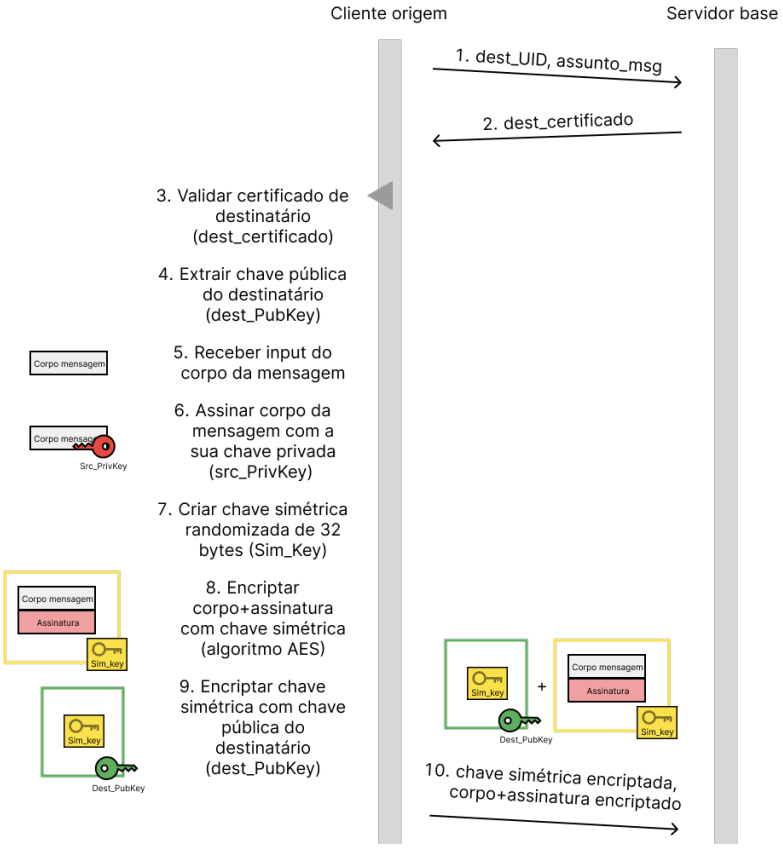

Figura 1: Comunicação cliente origem-servidor base

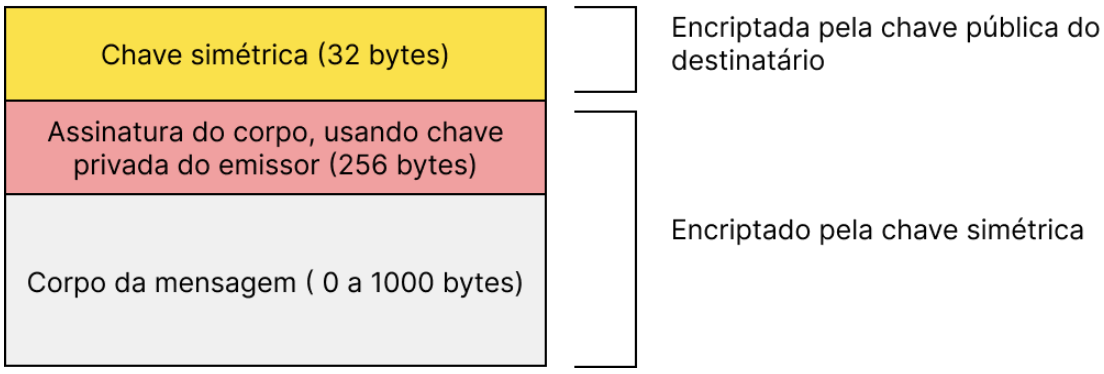

Figura 2: Conteúdo final relativo a corpo da mensagem enviado de cliente para servidor base no passo 10

O servidor, ao receber a mensagem, irá então certificar a data de entrega da mesma através de um servidor de *timestamping* externo (figura 3). Para isso, o servidor irá calcular a *hash*[1] da mensagem ao servidor de *timestamping* e o mesmo irá assiná-la, junto com a data em que a recebeu, com a chave privada do seu certificado.

[1]Não é necessário enviar a mensagem toda para assinatura, uma hash comprova igualmente que o conteúdo foi de facto recebido e poupa largura de banda e evita a partilha da mensagem, ainda que encriptada, a outro agente (algo que deve sempre ser evitado, pois não sabemos quando um nó da rede se pode tornar maligno)

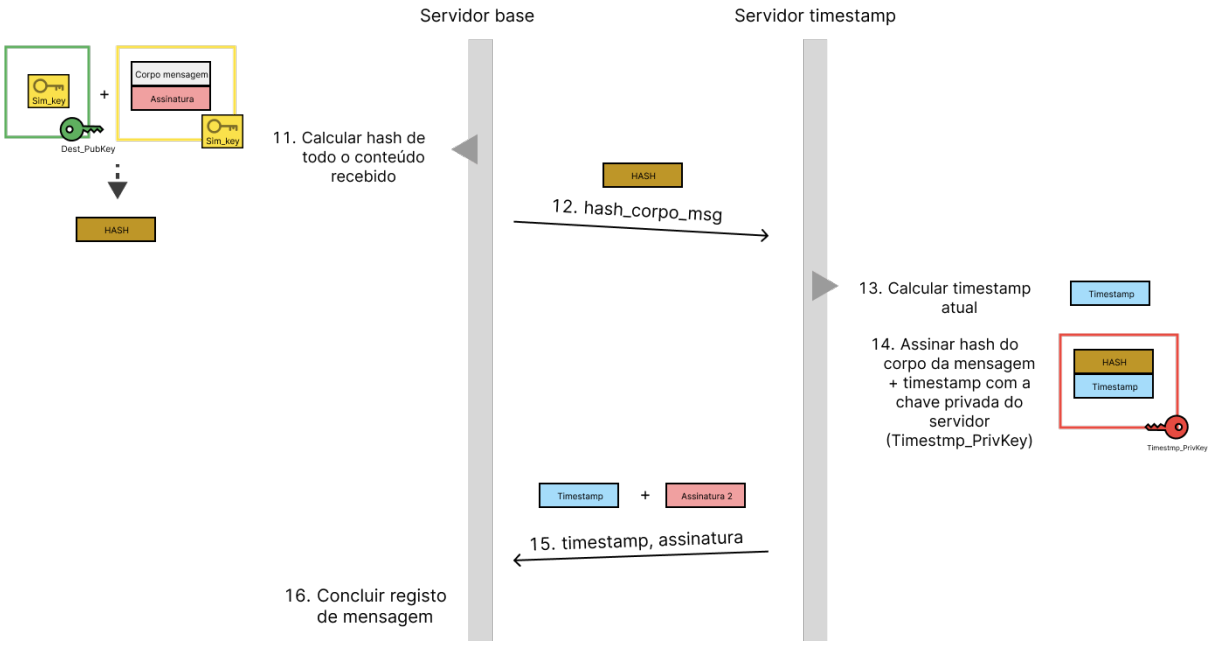

Figura 3: Comunicação servidor base-servidor timestamp

Na comunicação servidor-destino (figuras 4 e 5):

- Após receber do destinatário pedido de mensagem específica, servidor manda mensagem correspondente e certificado do emissor.
- Destinatário valida certificado e extrai chave pública de emissor
- Destinatário valida a *timestamp* recebida, verificando se foi assinada pelo servidor reconhecido para esse efeito, e se corresponde à *hash* da mensagem efetivamente recebida
- Destinatário desencripta chave simétrica com a sua PrivKey
- Destinatário desencripta corpo+assinatura com a chave simétrica
- Destinatário verifica assinatura com PubKey de emissor
- Destinatário lê conteúdo da mensagem

Assim, conseguimos garantir as seguintes propriedades:

- Assumindo hipoteticamente que o corpo e a assinatura não estão encriptados, ainda assim o servidor não pode manipular o corpo da mensagem, pois o destinatário verificaria que a assinatura era incompatível com o conteúdo da mensagem alterada. Garante-se assim integridade da mensagem.
- O servidor não pode manipular o destino da mensagem. Apenas o dono da chave privada correspondente à que se usou para encriptar a chave simétrica poderá descodificar a chave, logo enviando a mensagem para o utilizador incorreto, este não seria capaz de descodificar o corpo+assinatura e aceder aos conteúdos da mensagem. Garante-se confidencialidade (outros clientes não conseguem ler a mensagem)
- Um cliente sabe que a mensagem foi enviada pelo emissor especificado porque mensagem tem assinatura com chave privada do emissor (que só o emissor conhece). Garante-se autenticidade e não repúdio. Também garante-se que a mensagem é a si dirigida porque a chave simétrica foi encriptada com a sua chave pública e só ele a pode descodificar.
- O servidor nao consegue sequer ler o corpo da mensagem, visto precisar da chave simétrica e, portanto, da chave privada do destinátario. Garante-se confidencialidade (também o servidor não consegue ler a mensagem)

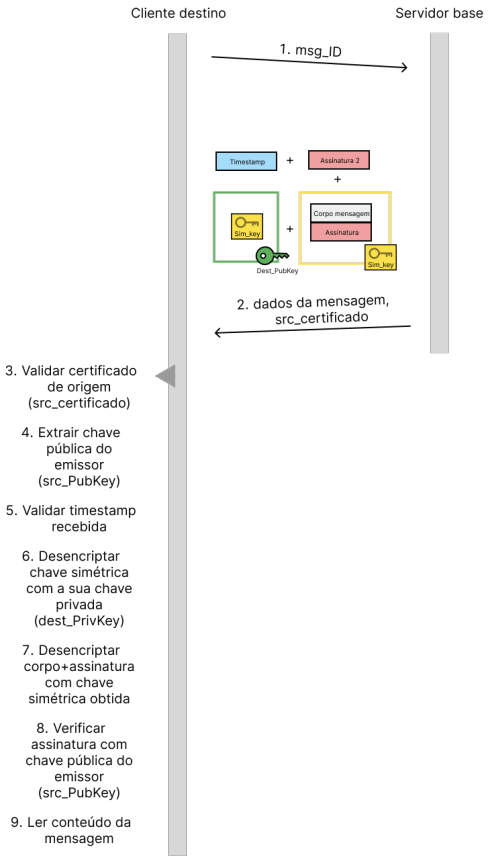

Figura 4: Comunicação cliente destinatário-servidor base

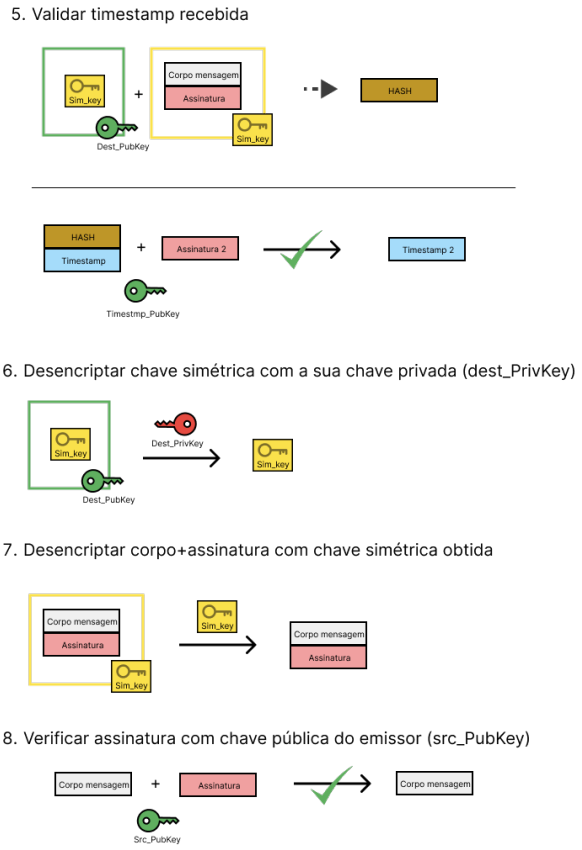

Figura 5: Passos de validação e desencriptação do cliente destinatário (passos 5 a 8)

Referir também que, em vez de encriptar body+assinatura simplesmente com a chave pública do destinatário, optamos por utilizar uma chave simétrica e encriptar essa chave simétrica com a chave pública do destinatário, visto que chaves RSA têm limitações quanto ao tamanho de dados encriptável, bem como de performance para grandes quantidades de dados. Assim, sendo necessário garantir a capacidade de encriptar 256 bytes da assinatura mais 0 a 1000 bytes de conteúdo da mensagem (0 a 1000 caractéres), recorremos a uma chave simétrica para poder encriptar mensagens com elevado conteúdo de forma segura, e apenas precisamos de encriptar uma chave simétrica pequena (32 bytes) com a chave pública do destinatário.

2.2.c) Funcionalidades extra

Para além das funcionalidade base, implementamos também recibos simplificados, um logger, codificação em BSON e criação de certificados.

- Para atestar que mensagens foram submetidas ao sistema, implementamos mensagens de resposta que servidor envia para o cliente para confirmar se envio de mensagem foi ou não bem sucedido. Tendo em conta que a comunicação entre cliente e servidor se processa por TCP e comunicação utiliza chave secreta acordada no protocolo station-to-station, não houve necessidade de criar MACs para garantir integridade da mensagem ao enviar entre cliente e servidor.
- Todas as transações do servidor são registadas em ficheiros na pasta `server/logs/`. Para cada cliente que se conecta ao servidor é criado um ficheiro ".log" com o histórico de interação com servidor.
- Como referido antes, utilizou-se um encoding binário com BSON para estruturar as mensagens do protocolo de comunicação.
- Desenvolvemos também um script "generate_certs.py" para geração de certificados. Permite criar certificados para o CA, servidor e clientes, com os mesmos atributos no campo subject e as mesmas extensões que os presentes nos certificados fornecidos. Na versão final submetida do projeto, todas as keystores PKCS12 e certificados utilizados foram gerados por este script.

  Para criar chave privada e certificado de CA:
  ▸ `python3 common/generate_certs.py ca <CA_PSEUDONYM>`

  Para criar keystores PKCS12 de servidor:
  ▸ `python3 common/generate_certs.py server <CA_PSEUDONYM> <SERVER_PSEUDONYM>`

  Para criar keystores PKCS12 do cliente
  ▸ `python3 common/generate_certs.py server <CA_PSEUDONYM> <CLIENT_PSEUDONYM>`

  Se não for especificado, os parâmetros <CA_PSEUDONYM>, <SERVER_PSEUDONYM>, <CLIENT_PSEUDONYM> assumem, respetivamente, os valores "MSG_CA","MSG_SERVER","MSG_CLI1″ por padrão. Todos os ficheiros criados vão para a pasta "otherCA/".

3) Considerações

Decidimos que não seria permitido que clientes enviassem mensagens a clientes que ainda não estão registados no sistema, visto que não seria possível obter a chave pública do destinatário para encriptar corretamente a mensagem na origem. Assim sendo, para enviar uma mensagem a outro cliente, o outro cliente tem antes de realizar um pedido de qualquer tipo ao servidor, por exemplo a pedir a fila de espera. Caso o cliente destinatário não esteja registado é lançada uma mensagem de erro.

4) Ideias não concluídas

Uma grande melhoria que não chegou a ser implementada, mas que no entanto estivemos muito perto de concluir o desenvolvimento foi um sistema de melhor encriptação das mensagens entre origem e destino. A solução atual, apesar de não permitir ao servidor ler o conteúdo das mensgagens, não possui *forward secrecy*, ou seja, se as chaves privadas dos certificados que estão a ser usados forem comprometidas, poderão ser lidas todas as mensagens anteriores trocadas entre os clientes, pois existe apenas um único ponto de falha. Nós tentamos implementar uma adaptação do procotolo X3DH, que é o padrão pelo qual a maioria das aplicações de mensagens encriptadas se regem. A nossa implementação iria diferir na obrigatoriedade do uso de OPKs *(One-Time Prekeys)* e na IK *Identity Key* e SPK *(Signed Prekey)* serem assinadas pelo certificado do utilizador. Estas mudanças tinham como objetivo simplificar o protocolo de modo a acelerar a sua implementação. No entanto, não foi possível completar a sua implementação, ficando no entanto o progresso conseguido até à data disponível na *branch* do repositório denominada por *tripleDH*.

5) Conclusão

Com este trabalho prático consolidamos conceitos relativos à utilização de chaves simétricas e assimétricas, algoritmos de hash, encriptação e assinatura de dados e refletimos sobre as suas vantagens e desevantagens. Com o desenvolvimento de um serviço de Message Relay, aplicamos conceitos de autenticidade, integridade e confidencialidade, no contexto do envio de mensagens entre clientes e um servidor intermediário, recorrendo também a um servidor de timestamps. Ultimamente, assentamos numa arquitetura que combina chaves simétricas e assimétricas para construir um serviço com segurança.