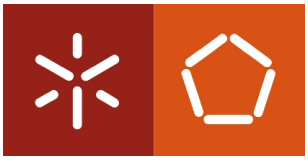

Universidade do Minho

Segurança de Sistemas Informáticos

jPostMan

Desenvolvido por:
Isac Meira PG31577
Paulo Queiroga A72204

Docentes :
José Bacelar
Vitor Fontes

3 de Janeiro de 2018

Conteúdo

1 Introdução

No âmbito da componente prática da avaliação da Unidade Curricular de Segurança de Sistemas Informáticos, foi-nos proposto o desenvolvimento de uma aplicação cliente-servidor que suporte a funcionalidade de um serviço de caixas de correio.

Os requisitos impostos à aplicação foram um servidor que concentra a responsabilidade de gerir as ligações aos clientes do serviço e que armazena e gere as mensagens depositadas pelos utilizadores. O servidor tem que permitir múltiplas instâncias de clientes que se conectam de modo a enviarem uma mensagem a outro utilizador do serviço e recolherem as mensagens depositadas na sua caixa de correio.

A comunicação entre o cliente e servidor é regulada por um protocolo de comunicação e teria-se que se assegurar a integridade, confidencialidade e autenticidade de todas as mensagens trocadas, sendo a comunicação, tanto quanto possível, anónima para outros utilizadores do sistema.

Assume-se que o servidor é *honest but curious*, isto é, segue sempre o protocolo estabelecido, mas não lhe deve ser possível ter acesso ao conteúdo de mensagens trocadas entre utilizadores do sistema.

Adicionalmente o serviço deveria ser preparado para ser instalável num Ubuntu Server 16.04 LTS, devendo integrar-se de forma mais completa possível com a infraestrutura de gestão do sistema, pretendendo-se que o serviço opere com o mínimo de privilégios possível. Era também pedido o desenvolvimento de um ou mais *scripts* de reconfiguração do sistema de modo a minimizar a sua superfície de vulnerabilidade.

2 Arquitetura da Solução

De modo a acomodar a exigência de termos várias instâncias de clientes o servidor foi criado utilizando o modelo de atores.

Quando um cliente se tenta conectar ao servidor por um *Socket TCP* o servidor aceita a sua conexão e cria um ator para lidar com a criação de um canal seguro entre o cliente e o servidor, seguindo o protocolo *Station-to-Station* e volta a escutar o Socket de modo a aceitar mais ligações.

O ator que corre o Protocolo, se conseguir estabelecer o canal seguro, cria por sua vez 3 outros atores, os atores *Reader*, *Writer* e *Manager*, caso contrário termina a ligação com o cliente.

O ator *Reader* possuí no seu estado interno a *InputStream* do Socket TCP. Este ator recebe as mensagens oriundas do Socket, decifra as mesmas e verifica a integridade das mensagens. De seguida verifica qual o tipo de cada mensagem recebida por parte do cliente e embrulha a mesma conforme o seu tipo numa mensagem para o *Manager* tratar do pedido.

Já o ator *Manager* recebe na sua mailbox os diferentes tipos de mensagens, que poderão ser pedidos de registo, pedidos de login, pedidos de envio de mensagens a outros clientes como pedidos de verificação de novas mensagens e trata os mesmos em conformidade, tendo no seu estado interno dois *Database Acess Objects* sendo estes o *MessageDAO* e o *ClientDAO*, que possuem o código para as transações com uma base de dados *MySQL* que guarda o estado do servidor *offline*.

Quando o *Manager* termina de processar um pedido, envia uma mensagem para a mailbox do ator *Writer* que contem uma resposta a ser dada ao cliente.

Por fim, o ator *Writer* recebe as mensagens do *Manager*, cifra as mesmas com a chave de sessão AES256 acordada com o cliente e transmite a mesma juntamente com o vetor de inicialização utilizado no modo *Counter Mode* da cifra AES e o MAC do vetor de inicialização concatenado com o criptograma, isto é o MAC da mensagem na na forma *encrypt-then-mac*.

Desta feita, o ator *Writer* faz uso do *OutputStream* do Socket que possuí no seu

estado interno envia a resposta ao cliente.

As mensagens que transitam no Socket, fazem todas uso do *Protocol Buffers* que é um mecanismo flexível, eficiente e automático de serializar dados estruturados de dados.

O cliente depois de concluir o protocolo poderá registar-se no serviço ou fazer login no mesmo.

Após ter iniciado sessão, poderá mandar mensagens a outros clientes, visualizar as suas mensagens e poderá também a qualquer momento mandar um pedido ao Servidor para o mesmo verificar se existem mais mensagens a serem-lhe encaminhadas.

Todas as mensagens que recebeu na sua máquina são armazenadas num ficheiro binário, podendo desta feita quando voltar a iniciar a sessão ter acesso às mesmas.

3 Protocolo Station-to-Station

No inicio do *Protocolo Station-to-Station* o servidor cria uma instância de um gerador de números aleatórios utilizando a implementação *NativePRNGNonBlocking* que utiliza para gerar um número primo aleatório com 2048 bits a ser usado no acordo de chaves *Diffie-Helman*.

Um número primo é encontrado fazendo uso do método probablePrime da classe java *BigInteger*, que retorna um número primo com alta percentagem de sucesso.

Usa-se como gerador o número 2 que é uma raiz primitiva e cria-se a classe *DHParameterSpec* com os parâmetros a serem usados para a geração de um par de chaves públicas e privadas para o acordo de chaves Diffie-Helman.

Neste ponto o servidor envia ao cliente o seu objeto *PublicKey*, que contém tanto a sua chave pública como os parâmetros g e p.

O cliente recebe a chave pública do servidor e os parâmetros utilizados para a geração da mesma, procedendo então à criação do seu par de chaves pública e privada e utilizando a sua chave privada com a chave pública do servidor gera a chave secreta de sessão.

Nesta fase o cliente irá carregar a sua *KeyStore* para memória e irá fazer uso da sua chave privada e da sua *Certificate Chain* do seguinte modo:

1. Concatena a sua chave pública com a chave pública do servidor;
2. Assina a mensagem concatenada utilizando *hash-then-sign* para prevenir ataques à chaves privada;
3. Utiliza o algoritmo SHA-256 na chave partilhada de sessão;
4. Cifra a mensagem assinada com a cifra simétrica AES256 no modo Counter Mode com padding PKCS5, com um vetor inicial de 128 bits gerado aleatoriamente e como chave da cifra o hash da chave calculado na alínea anterior;
5. Utiliza o algoritmo HMAC com SHA-256 passando como chave, a chave de sessão utilizada no AES no passo 4, para calcular o MAC do vetor de inicialização concatenado com o criptograma, isto é, utiliza-se *encrypt-then-MAC*;

6. Envia para o servidor da sua chave pública, a sua Certificate Chain e juntamente com o criptograma o vetor de inicialização e o MAC.

De notar que neste processo alternativamente poderíamos ter utilizado o algoritmo de hashing SHA512, utilizar os primeiros 256 bits do mesmo para a chave do AES256 e os restantes 256 bits para a chave do algoritmo de *Message Authentication Code* HMAC com SHA-256, sendo que preferimos utilizar a forma anterior.

O servidor receberá a Certificate Chain do cliente, juntamente com um criptograma um vetor de inicialização e o MAC. Agora o servidor carregará o certificado da Autoridade de Certificação em que confia e que gerou os certificados dos clientes, estabelece-a como *Trust Anchor* e verifica se o certificado enviado pelo cliente é validado. Caso tal se verifique cria a chave partilhada Diffie-Helman, elevando a chave pública do cliente com a sua chave privada.

Analogamente, ao já relatado na fase do cliente, o servidor irá utilizar o algoritmo de hashing SHA-256 de modo a possuir uma chave de 256bits para a cifra simétrica AES de 256bits. De seguida, instancia o HMAC com o SHA-256 com a chave simétrica do AES e utiliza o HMAC para verificar se o MAC do vetor de inicialização concatenado com o criptograma é igual ao MAC que recebeu.

Caso tal se verifique, decifra o criptograma com o AES256, concatena a chave pública do cliente com a sua chave pública e utiliza a chave pública do certificado do cliente para verificar a assinatura que o cliente enviou. A verificação da assinatura, utiliza a técnica *hash-then-verify*.

Se em algum destes passos,algum dos passos anteriores não se verificar a ligação é interrompida.

De forma semelhante ao cliente será agora a vez do servidor enviar o seu certificado ao cliente e a assinatura cifrada da mensagem da sua chave pública concatenada com a chave pública do cliente, juntamente com o vetor de inicialização e o MAC.

O cliente irá então fazer as verificações já relatadas na parte do servidor, se tudo correr bem o canal seguro é estabelecido com sucesso.

Daqui em diante toda a comunicação realizada entre cliente-servidor é cifrada usando

a chave simétrica de sessão do AES256. As mensagens serão cifradas e decifradas com esta chave e os criptogramas devem levar consigo o vetor de inicialização gerado aleatoriamente o MAC calculado. O recetor da mensagem verificará sempre que o MAC por ele calculado e o MAC da mensagem são coincidentes.

Desta forma, garantimos a integridade, confidencialidade e autenticidade de todas as mensagens trocadas.

A integridade é garantida pois se is MACs condizerem temos a garantia que a mensagem não foi manipulada por um atacante ativo.

A confidencialidade é garantida pois apenas o servidor e o cliente possuem a chave de sessão acordada.

A autenticidade é garantida, pois o certificado estabelece um mapeamento entre a identidade digital e a identidade civil do cliente, que como iremos ver adiante irá ser usado no envelope digital para assegurar que só o remetente e o destinatário conseguem ver a mensagem que o servidor encaminha entre as partes.

4 Envelope Digital

Quando um cliente pretende enviar um email a um outro cliente do serviço o mesmo irá pedir ao servidor que lhe seja facultado o certificado do futuro recetor da mensagem.

Assim que o cliente recebe o certificado do outro cliente por parte do servidor através do canal seguro, executando os passos já explicados anteriormente, o cliente gerará uma chave gerada aleatoriamente de 256 bits a ser usada na cifra simétrica AES256.

Mais uma vez utiliza-se o *encrypt-then-mac* da mensagem e então com a chave pública do certificado do destinatário da mensagem ciframos a chave usada para cifrar com o AES256 a mensagem a ser enviada. Que só poderá ser decifrada pela chave privada do destinatário.

Pegamos no criptograma, no vetor de inicialização, no MAC e na chave cifrada e cifrámos tudo agora com o AES256 com a chave de sessão. Enviamos pelo canal o criptograma gerado juntamente com o seu MAC e vetor de inicialização.

O servidor verificará a integridade da mensagem, decifrar-lha-á e guardarar-lha-á offline na sua base de dados MySQL até que a possa enviar ao destinatário.

A mensagem será entregue ao destinatário quando o mesmo se logar na aplicação, ou quando já logado fizer *Refresh* para verificar se possuí mensagens novas.

Quando o destinatário recebe mensagens novas, depois enviará pelo processo já descrito diversas vezes de troca de mensagens no canal os ids das mensagens recebidas ao servidor, para este assim saber quais aquelas que já foram devidamente entregues e eliminar as mesmas da base de dados.

5 Conclusão

Na elaboração da nossa aplicação tentámos por diversas vezes criar a nossa Autoridade de Certificação com o openssl e gerar certificados com a mesma, mas sem sucesso. Na nossa última tentativa, quando o cliente tentava cifrar a chave com que cifrou a mensagem era lançada uma exceção *InvalidKeyUsage* que não sabemos atribuir se devido à configuração do openssl o certificado não tinha especificado utilização da chave para cifra ou se pela tentativa de colocar o certificado juntamente com a sua chave privada numa keystore pkcs12.

Por esse motivo, a quando do registo de um cliente na aplicação em que guardamos o seu certificado juntamente com o username que nos fornece e password, não podemos verificar que o username escolhido coincide com o o seu nome no certificado, pois tivemos que utilizar os certificados gerados pelo professor Bacelar.

Retirando esse aspeto, penso que cumprimos na integra o que nos era pedido para realizar na parte do projeto do professor Bacelar.

Quanto à parte do professor Vitor Fontes, apenas instalámos a máquina virtual com o Ubuntu Server 16.04 LTS. Não tínhamos muitos conhecimentos para desenvolver o que era esperado de nós nessa parte do projeto, o máximo que conseguiríamos fazer seria criar um novo utilizador sem permissões de sudo e limitar as escritas e execuções de ficheiros, pequena parte essa que não realizamos, pois como referido anteriormente utilizamos atores no nosso desenvolvimento, mais concretamente a biblioteca de atores *Quasar* da *Parallel Universe* que necessitava de instrumentação do código através de um *javaagent*.

Ora, acontece que com o maven a gerir-nos as dependências, conseguíamos através de um plugin de execução instrumentar o código, mas não tínhamos destreza suficiente com o maven para conseguir gerar o jar do servidor a ser o seu código instrumentado durante a sua execução.

Contudo, fazemos uma análise positiva do nosso desempenho neste trabalho graças ao código por nós produzido na realização da parte do projeto do Professor Bacelar.