View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Repositório Institucional da UFSC

UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Pedro Henrique Lenzi Soares

ALTERNATIVA PARA EMISSÃO DE CERTIFICADOS POR AUTORIDADE CERTIFICADORA ONLINE

Florianópolis

2017

Pedro Henrique Lenzi Soares

ALTERNATIVA PARA EMISSÃO DE CERTIFICADOS POR AUTORIDADE CERTIFICADORA ONLINE

Tese submetida ao Programa de Graduação em Ciências da Computação para a obtenção do Grau de Bacharel em Ciências da Computação.
Orientador: Prof. Dr. Jean Everson Martina

Florianópolis

2017

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Lenzi Soares, Pedro Henrique
Alternativa para Emissão de Certificados por Autoridade Certificadora Online / Pedro Henrique Lenzi Soares ; orientador, Jean Everson Martina, 2017.
70 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Ciências da Computação, Florianópolis, 2017.

Inclui referências.

1. Ciências da Computação. 2. Criptografia. 3. Certificação Digital. I. Martina, Jean Everson. II. Universidade Federal de Santa Catarina. Graduação em Ciências da Computação. III. Título.

Pedro Henrique Lenzi Soares

ALTERNATIVA PARA EMISSÃO DE CERTIFICADOS POR AUTORIDADE CERTIFICADORA ONLINE

Esta Tese foi julgada aprovada para a obtenção do Título de "Bacharel em Ciências da Computação", e aprovada em sua forma final pelo Programa de Graduação em Ciências da Computação.

Florianópolis, 01 de Maio 2017.

Prof. Dr. Rafael Luiz Cancian
Coordenador do Curso

Banca Examinadora:

M.Sc Lucas Pandolfo Perin
Presidente

Prof. Dr. Jean Everson Martina
Orientador

Marlon Trapp

Leonardo Meurer

RESUMO

As áreas de estudo em criptografia e certificação digital apresentam diversas ferramentas que poderiam ser utilizadas como mecanismos de comprovação de identidade e autenticação por uma larga variedade de softwares. Não obstante, é comumente custoso para o time de desenvolvimento implementar tais ferramentas de forma fácil de usar e o mais portável possível. O objetivo principal desta tese é criar ferramentas que podem ser facilmente integradas com a maior variedade de softwares possível, mantendo o menor número possível de dependências, para minimizar o esforço que os usuários precisam fazer para ter acesso à funcionalidades de criptografia e certificação digital. As soluções aqui apresentadas como resultado deste estudo foram testadas e integradas em um software existente, para que o processo de integração pudesse ser descrito de forma mais precisa.
Palavras-chave: Criptografia. Certificação Digital.

ABSTRACT

The concepts of cryptography and digital certification present various tools that could be used as identity verification and authentication mechanisms by a wide assortment of software. It is commonly costly, however, for the development team to implement these tools in an easy-to-use and as portable as possible manner. The main goal of this thesis is to create tools that can be easily integrated with the widest possible variety of software, while having as few dependencies as viable, to mitigate the effort the users need to do to perform useful functions of cryptography and digital certification. The solutions presented as result of this study were tested and integrated into an existent software, for the integration process to have higher trustworthiness.
Keywords: Cryptography. Digital Certification.

LISTA DE FIGURAS

Figura 1 Comunicação sem uso de criptografia.................... 23
Figura 2 Comunicação com criptografia simétrica................. 24
Figura 3 Comunicação com criptografia assimétrica............... 24
Figura 4 Assinatura digital.. 26
Figura 5 Device Server (TRAPP, 2016) 31
Figura 6 Funções do módulo criptográfico 32
Figura 7 Emissão em browser. 33
Figura 8 Emissão em token... 35

LISTA DE ABREVIATURAS E SIGLAS

LabSEC	Laboratório de Segurança em Computação	19
UFSC	Universidade Federal de Santa Catarina	19
LCR	Lista de Certificados Revogados	27
ICP	Infraestrutura de Chaves Públicas	27
AC	Autoridade Certificadora	27
PKCS	Public Key Cryptography Standards	28
MSC	Módulos de Segurança Criptográfica	28
SAEC	Sistema Automatizado de Emissão de Certificados	29
HTML	HyperText Markup Language	29
HTTPS	HyperText Transfer Protocol	32
HTTPS	HyperText Transfer Protocol Secure	32
SSL	Secure Sockets Layer	32

SUMÁRIO

1 INTRODUÇÃO 19
1.1 JUSTIFICATIVA 19
1.2 OBJETIVOS 20
1.2.1 Objetivo Geral 20
1.2.2 Objetivos Específicos 20
1.3 METODOLOGIA 20
1.4 ORGANIZAÇÃO DOS CAPÍTULOS 20
2 FUNDAMENTAÇÃO TEÓRICA 23
2.1 CRIPTOGRAFIA 23
2.1.1 Criptografia Simétrica 23
2.1.2 Criptografia Assimétrica 25
2.2 RESUMO CRIPTOGRÁFICO 25
2.3 ASSINATURA DIGITAL 25
2.4 CERTIFICADO DIGITAL 26
2.5 LISTA DE CERTIFICADOS REVOGADOS 26
2.6 INFRAESTRUTURA DE CHAVES PÚBLICAS 27
2.6.1 Autoridade Certificadora 27
2.7 TOKEN 27
2.7.1 Smart Card 28
2.8 PUBLIC KEY CRYPTOGRAPHY STANDARD 28
2.8.1 PKCS #11 28
2.8.2 PKCS #12 28
3 PROPOSTA 29
3.1 ROTINA DE CERTIFICAÇÃO 29
3.2 CONEXÃO À DISPOSITIVOS 30
4 DEVICE SERVER 31
4.1 MÓDULO PARA COMUNICAÇÃO COM DISPOSITIVOS CRIPTOGRÁFICOS 31
4.2 PROTOCOLO HTTPS 32
5 IMPLEMENTAÇÃO 33
5.1 EMISSÃO EM BROWSER 33
5.1.1 Integração com o SAEC 34
5.2 EMISSÃO EM TOKEN 34
5.2.1 Integração com o SAEC 34
6 CONCLUSÃO 37
6.1 TRABALHOS FUTUROS 37
REFERENCIAS 39

1 INTRODUÇÃO

A certificação digital é comumente requerida em diversos tipos de software, como por exemplo, softwares de Gestão Eletrônica de Documentos que fazem uso de assinatura digital. Devido à grande variedade de ambientes utilizados pelos usuários, seja por possuírem diferentes sistemas operacionais, browsers, dispositivos ou tokens criptográficos, muitas vezes é difícil achar uma forma de desenvolver uma solução com ótima portabilidade.

Aplicações que requerem operações criptográficas e de certificação digital encontram problemas ao necessitarem que tais operações sejam executadas no lado do usuário, tanto de compatibilidade quanto de segurança. Soluções conhecidas utilizam métodos inviáveis para esses fins específicos, ou ferramentas para as quais os browsers mais utilizados atualmente deixarão de oferecer suporte.

A disponibilidade de uma ferramenta que possa ser facilmente integrada em qualquer sistema para auxiliar em tarefas como a criação de chaves criptográficas, de requisições de certificado, a instalação de certificados em browser ou tokens e a comunicação com tokens criptográficos facilitaria o desenvolvimento de quaisquer aplicações que desejam ter funcionalidades referentes à certificação digital.

Tal facilidade, gerada pela existência dessa ferramenta, traria consigo certo incentivo à utilização de certificação digital como forma de autenticação, encorajando o desenvolvimento de uma quantidade mais ampla de projetos que a envolvem.

1.1 JUSTIFICATIVA

A falta de uma ferramenta compatível com a grande maioria dos browsers e sistemas operacionais utilizados torna difícil a realização de várias operações no lado cliente da aplicação, o que, por sua vez, restringe as possibilidades de implementação de várias aplicações. No Laboratório de Segurança Em Computação (LabSEC) da Universidade Federal de santa Catarina (UFSC) estão em andamento projetos que necessitam das soluções propostas neste trabalho como alternativa às tecnologias utilizadas atualmente, que não são mais suportadas por diversos ambientes e vem sendo depreciadas ao longo dos últimos anos. Este trabalho poderá, também, ser facilmente integrado para uso em projetos futuros.

1.2 OBJETIVOS

1.2.1 Objetivo Geral

Desenvolver uma forma de realizar operações criptográficas e de certificação digital que possa ser facilmente utilizada na camada de usuário da aplicação, com segurança e sem problemas de compatibilidade. O material produzido deve, também, ser facilmente integrado com qualquer projeto existente que necessite tais operações. Utilizando-se do resultado, integrar as ferramentas implementadas em uma aplicação existente para a realização de tarefas como a geração de chaves e emissão de certificados, instalando-os em browsers ou smartcards.

1.2.2 Objetivos Específicos

- Implementação das ferramentas para execução das operações necessárias.

- Integração das ferramentas criadas com uma aplicação existente (SAEC).

1.3 METODOLOGIA

A metodologia de pesquisa baseia-se em encontrar soluções para problemas parecidos, que possam conter material útil para o desenvolvimento do trabalho proposto. Tais soluções serão estudadas e suas idéias, adaptadas aos objetivos deste trabalho, serão implementadas e testadas para se encontrar uma alternativa adequada para desenvolver o trabalho proposto.

1.4 ORGANIZAÇÃO DOS CAPÍTULOS

A organização dos capítulos do trabalho será da seguinte forma:

- Introdução: Aqui o problema será apresentado, juntamente com os motivos que tornam a sua resolução importante e os objetivos que pretende-se alcançar com o trabalho.

- Fundamentação teórica: Apresentação de diversos conceitos julgados necessários para uma melhor compreensão do trabalho.
- Proposta: Aqui serão detalhadas as soluções encontradas e os estudos feitos para encontrá-las.
- Implementação: Detalhes das tarefas executadas para o desenvolvimento das soluções propostas.
- Conclusão: Principais pontos constatados durante a realização do trabalho, juntamente com os resultados obtidos.
  - Trabalhos futuros: Possíveis melhorias ou usos para o material criado que não entraram no escopo deste trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

Para o melhor entendimento do trabalho, aqui são apresentados conceitos de segurança em computação julgados necessários.

2.1 CRIPTOGRAFIA

Criptografia consiste em cifrar dados de forma que somente determinadas entidades podem ter acesso ao real conteúdo dos dados. Um exemplo simples consiste na análise de troca de mensagens via web. Ao enviar uma mensagem para Bob por canal não seguro, Alice corre o risco de que um agente mal intencionado, Eve, obtenha a mensagem e a leia. A criptografia oferece formas de codificar tal mensagem, para que Eve não consiga entendê-la, mesmo se a obtiver. (HOUSLEY R.; POLK, 2001)

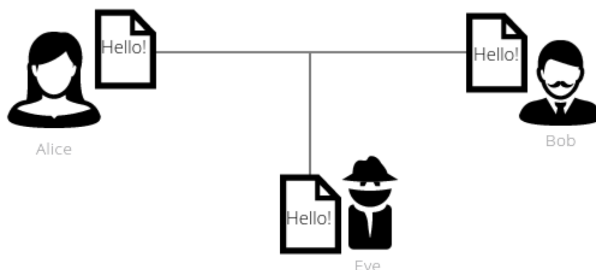

Figura 1 – Comunicação sem uso de criptografia.

2.1.1 Criptografia Simétrica

A criptografia simétrica utiliza uma única chave secreta, que ambos remetente e destinatário devem possuir, e é utilizada para cifrar e decifrar dados. Neste caso, Alice, antes de enviar a mensagem, deve cifrá-la, tornando seu conteúdo ilegível. Ao recebê-la, Bob deve utilizar a mesma chave utilizada por Alice para decifrar e obter o conteúdo real da mensagem. Para poder ler a mensagem, além de interceptá-la pelo canal inseguro, seria necessário que Eve possuísse a chave.

Uma dificuldade encontrada na criptografia simétrica consiste na troca de chaves entre ambas as entidades, que deve ser feita por canal seguro, pois se alguém obtiver a chave, poderá decifrar todas as mensagens trocadas que forem cifradas com ela. Uma alternativa viável é a utilização de um algoritmo de chaves públicas, como por exemplo o Diffie-Hellman.

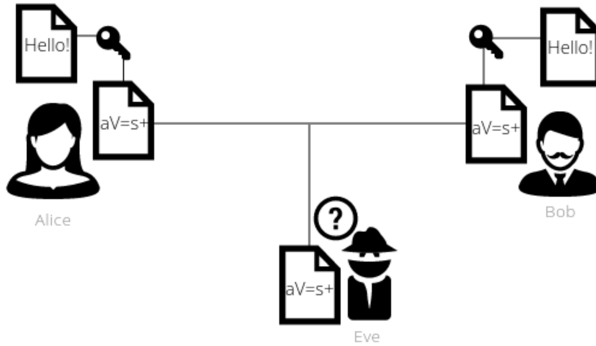

Figura 2 – Comunicação com criptografia simétrica.

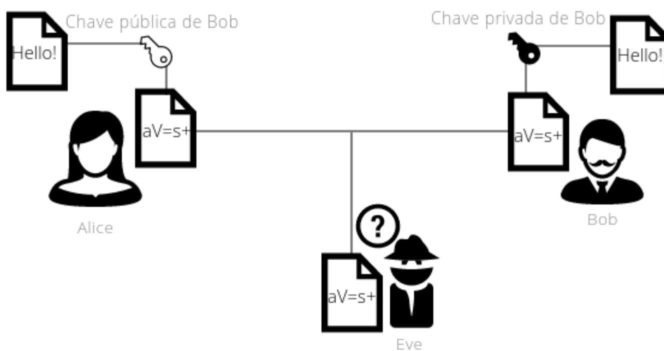

Figura 3 – Comunicação com criptografia assimétrica.

2.1.2 Criptografia Assimétrica

A criptografia assimétrica consiste na geração de um par de chaves, de forma que tudo o que for cifrado com uma delas só possa ser decifrado com a outra. Elas são nomeadas chave pública e privada. Ao gerar um par de chaves, a entidade guarda seguramente sua chave privada, e disponibiliza sua chave pública para quem desejar utilizá-la. Dessa forma, Alice deve obter a chave pública de Bob para enviar uma mensagem à ele, e ele deve utilizar sua própria chave privada para decifrar a mensagem.

2.2 RESUMO CRIPTOGRÁFICO

O resumo criptográfico é gerado a partir de um texto submetido à uma função matemática, geralmente muito maior que o resumo. Tal função é necessariamente unidirecional, ou seja, a partir de um resumo não deve ser possível obter o texto que o gerou. O resumo é chamado de hash, ou impressão digital do texto de entrada, sendo possível dizer se determinado resumo foi gerado a partir de determinada entrada ou não. (HOUSLEY R.; POLK, 2001)

2.3 ASSINATURA DIGITAL

A criptografia assimétrica fornece a base para a assinatura digital. Para assinar uma mensagem, uma entidade deve cifrá-la com sua chave privada. Outra entidade pode utilizar a chave pública para decifrar a mensagem e confirmar a identidade do remetente. Em aplicações reais, por questões de custo de processamento, a assinatura é feita em um resumo da mensagem, e não na mensagem em si. Para se comunicar com Bob, Alice faz um resumo da mensagem, o cifra com sua chave privada e envia tanto a mensagem como a assinatura (hash cifrado) para Bob. Ele, por sua vez deve decifrar a assinatura utilizando a chave pública de Alice e gerar o resumo da mensagem. Comparando os resultados, se ambos os hashes forem iguais, há a confirmação de que foi Alice que enviou a mensagem e de que a mensagem não foi modificada antes de chegar em Bob.

Não obstante, para que isso funcione, é necessário que Bob tenha a certeza de que a chave pública que utilizou é realmente a que pertence ao par de chaves de Alice. Caso contrário, não é possível afirmar que

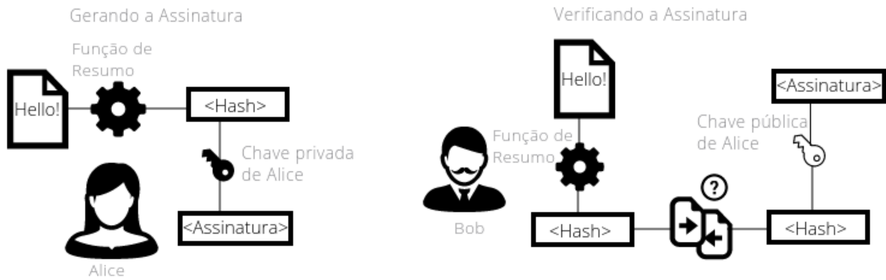

Figura 4 – Assinatura digital.

a mensagem foi enviada por Alice. Para ligar uma chave pública à sua chave privada correspondente e evitar tal problema, utiliza-se o Certificado Digital.

2.4 CERTIFICADO DIGITAL

O certificado digital é um objeto puramente digital que possui uma chave pública e dados da entidade à qual a chave privada correspondente à ela pertence. Além desses dados, que podem conter nome, CPF e informações para contato entre outros, existem duas datas e informações sobre o órgão emissor responsável pelo certificado. As datas especificam o período de validade do certificado. O órgão emissor deve ser uma entidade confiável que atesta a posse da chave privada correspondente à pública presente no certificado pela entidade cujos dados do certificado descrevem. Como prova de que tal órgão confirma as informações contidas no certificado, este contém uma assinatura digital daquele. (HOUSLEY R.; POLK, 2001)

2.5 LISTA DE CERTIFICADOS REVOGADOS

Visto que o certificado é um objeto puramente digital, não espera-se que seja possível recolhê-lo caso ele não seja mais confiável. Existem alguns motivos pelos quais um certificado deve ser revogado, entre eles estão o comprometimento da chave privada correspondente ou do órgão emissor.

Para controlar quais certificados não são mais válidos, apesar de não terem expirado, existe a Lista de Certificados Revogados (LCR), que é emitida pelo mesmo órgão emissor dos certificados. Nela estão todos os certificados que não são mais confiáveis, juntamente com as datas de sua emissão e de validade. Cabe ao órgão emitir sempre uma nova LCR quando a anterior expirar, assim como definir um período de validade coerente com a urgência de divulgação da revogação de certificados que emitiu.

2.6 INFRAESTRUTURA DE CHAVES PÚBLICAS

Uma Infraestrutura de Chaves Públicas (ICP) é desenhada para facilitar o uso da certificação digital. Ela é dividida em componentes que executam tarefas distintas, tais como a Autoridade Certificadora e a Autoridade Registradora.

2.6.1 Autoridade Certificadora

A Autoridade Certificadora (AC) é o principal elemento de uma infraestrutura de chaves públicas. É ela a responsável por emitir certificados e LCRs, assim como manter informação sobre eles. A AC também possui um certificado e utiliza a sua chave privada para assinar cada certificado que emite. O seu certificado pode ser auto assinado, caso ela seja uma AC Raiz, ou assinado por uma entidade superior à ela na hierarquia da ICP.

2.7 TOKEN

Um token é um objeto que pode ser utilizado para processos de autenticação, junto com ou no lugar de senhas. Eles podem guardar chaves criptográficas, dados biométricos ou senhas. Existem diversos modelos de token, alguns com visor, teclado numérico e mecanismos para prevenir que agentes mal intencionados obtenham as informações contidas nele, entre outros componentes. Em especial para este projeto, nos interessam os smart cards.

2.7.1 Smart Card

O smart card é um cartão que contém circuitos integrados, que o permitem armazenar dados e executar funções criptográficas diretamente no cartão. Leitoras de cartão são utilizadas para a comunicação entre o cartão e softwares. Os cartões são uma ferramenta muito importante para a criptografia assimétrica, pois possuem mecanismos de proteção de dados e podem gerar chaves privadas diretamente em seus chips, sem a necessidade da chave ser enviada ao cartão por algum meio não tão seguro, e podem utilizar as chaves que contém para todas as funções criptográficas permitidas pela chave.

2.8 PUBLIC KEY CRYPTOGRAPHY STANDARD

Public Key Cryptography Standards (PKCS) são grupos de regras e padrões para a criptografia de chaves públicas definidos pela empresa de segurança em redes e computadores RSA Security. (RSA...,)

2.8.1 PKCS #11

O padrão PKCS #11 define uma API para a realização de funções criptográficas relacionadas à diferentes tipos de chaves e certificados em dispositivos, tais como smart cards e Módulos de Segurança Criptográfica (MSC). (RSA...,)

2.8.2 PKCS #12

O PKCS #12 define um formato padrão para arquivos que contenham diferentes objetos criptográficos dentro de um único arquivo. É comumente utilizado para guardar um certificado junto com sua chave privada. Tal arquivo também pode ser criptografado, com o objetivo de obter-se mais segurança, e assinado. (RSA...,)

3 PROPOSTA

O trabalho surgiu com a necessidade de encontrar uma alternativa à forma como estava implementado o processo de emissão de certificados pelo projeto de autoridade certificadora online Sistema Automatizado de Emissão de Certificados (SAEC) do Laboratório de Segurança Em Computação. Tal forma fazia uso da tag HTML *keygen* que vem sendo depreciada nos últimos anos, e cada vez mais browsers estão deixando de aceitá-la, por apresentar risco de segurança. A tag permite que o processo de importação de certificado no browser seja automatizado, permitindo que servidores mal intencionados importem certificados indesejáveis no browser dos clientes, utilizando-os posteriormente para forjar uma autenticação que não deveria ser considerada válida.

A proposta foi dividida em duas partes. A primeira consiste em encontrar tal alternativa, que torne novamente possível a emissão de certificados diretamente no browser do cliente, utilizando somente ferramentas já fornecidas pelos navegadores e sistemas operacionais mais utilizados na atualidade. Já para a segunda parte, por envolver comunicação com dispositivos de segurança, consiste na implementação de uma aplicação Java que seja executada no ambiente do cliente, se comunique com o seu navegador e realize operações criptográficas utilizando-se de dispositivos conectados à máquina do usuário, como smart cards.

Assim como a primeira parte da proposta, a segunda também possui uma solução largamente utilizada que vem sendo cada vez menos suportada pelos navegadores atuais, os Java Applets. Como alternativa, será utilizada a forma proposta por Marlon Trapp em seu trabalho de conclusão de curso. (TRAPP, 2016).

Como forma de analisar e aprimorar o processo de integração do trabalho desenvolvido com outros softwares, ambas as partes foram integradas ao SAEC. Para as dificuldades encontradas foram desenvolvidas soluções, e o processo simplificado descrito aqui, para que sirva de guia para a utilização do trabalho.

3.1 ROTINA DE CERTIFICAÇÃO

No processo de certificação digital por uma autoridade certificadora online o par de chaves deve ser gerado no lado do cliente, visto que somente ele deve possuir sua chave privada. Gerar o par de chaves no

servidor e mandá-lo via internet abre um possível caminho de ataque para agentes mal intencionados. O usuário deve então gerar uma requisição de certificado, que contém seus dados e a chave pública do par gerado, assiná-la para que a autoridade certificadora possa confirmar a autenticidade da requisição e a posse da chave privada correspondente e enviá-la ao servidor.

No lado servidor da aplicação, a autoridade certificadora deve emitir o certificado, assinando-o com sua chave privada e enviá-lo de volta para o cliente, para que este possa guardá-lo e utilizá-lo. Existem diferentes métodos para armazenar um certificado, como por exemplo em formato PKCS #12, importado em um browser e em token magnético, que são todos abordados neste trabalho.

3.2 CONEXÃO À DISPOSITIVOS

Para realizar a comunicação entre o computador do cliente e dispositivos, como por exemplo leitoras de cartão, prevista na segunda parte da proposta, foi utilizada a alternativa aos applets em java para acesso a dispositivos proposta por Marlon Trapp em (TRAPP, 2016).

Tal alternativa sugere a execução de um programa na máquina do usuário, chamado de Device Server, que realiza a comunicação entre o browser e qualquer dispositivo conectado ao computador do cliente. Tendo como proposta ser facilmente adaptável a quaisquer novos dispositivos, o trabalho desenvolvido propõe a criação de diferentes módulos para diferentes tipos de dispositivo. Como protótipo, foi também desenvolvido por Marlon Trapp um módulo para que o Device Server tenha acesso à tokens criptográficos, que foi utilizado e ampliado durante o desenvolvimento deste trabalho.

A execução da operação se dá de forma parecida com a apresentada anteriormente, com a diferença de que para esta solução o par de chaves é gerado diretamente em um smart card, onde também é armazenada a chave privada. O certificado devolvido pelo servidor é então inserido no token e linkado com a chave privada correspondente.

4 DEVICE SERVER

O device server é uma aplicação Java que deve ser executada no ambiente do cliente, servindo de meio para a comunicação entre o navegador e quaisquer dispositivo conectado ao computador.

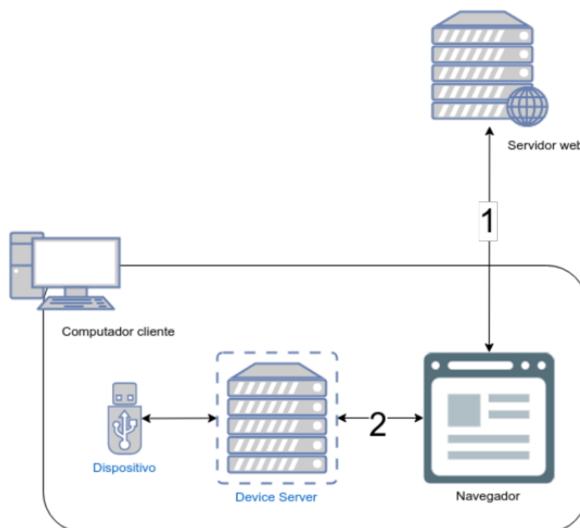

Figura 5 – Device Server (TRAPP, 2016)

4.1 MÓDULO PARA COMUNICAÇÃO COM DISPOSITIVOS CRIPTOGRÁFICOS

O módulo desenvolvido para o Device Server para a comunicação com dispositivos criptográficos continha funções para listar dispositivos compatíveis, listar certificados de um dispositivo e realizar assinatura digital. Neste trabalho, foram desenvolvidas três novas funcionalidades para módulo.

A primeira tem o objetivo de gerar um par de chaves, diretamente no token, e exportar a chave pública para que seja utilizada na criação de um certificado. Para que seja enviada ao Web Server uma requisição de certificado e não apenas a chave pública, deve-se utilizar

o método já existente para realizar a assinatura digital da requisição de certificado.

A segunda recebe como entrada um certificado digital, procura no token a chave privada que é o par da chave pública utilizada no certificado e instala o certificado no token, associando-o com a chave privada.

A terceira e última funcionalidade adicionada não é necessária para o processo de emissão de certificado, mas foi julgada útil. Ela tem como único objetivo exportar um certificado dentre os contidos no token, escolhido pelo usuário.

Métodos		Descrição
Existentes	list	Lista os dispositivos compatíveis conectados
	listCerts	Lista os certificados contidos no dispositivo escolhido
	sign	Assina uma mensagem com a chave escolhida
Adicionados	generateKey	Gera um par de chaves no dispositivo escolhido
	installCert	Importa um certificado no dispositivo escolhido
	getCert	Exporta certificado escolhido

Figura 6 – Funções do módulo criptográfico

4.2 PROTOCOLO HTTPS

O device server foi desenvolvido para ser executado seguindo o protocolo de comunicação HTTP , porém, visto que as aplicações para as quais este trabalho é direcionado, em sua maioria, utilizam um protocolo HTTPS , é necessário que o Device Server seja executado também em HTTPS, para poder se comunicar com o Web Server. Como solução, foi criada uma AC Raiz SSL para a emissão de um certificado para o Device Server. Tal certificado deve ser enviado para cada cliente e instalado em seus browsers como confiável, para que estes aceitem realizar a comunicação com o Device Server.

5 IMPLEMENTAÇÃO

Para a primeira parte da proposta, foi utilizado JavaScript juntamente com a biblioteca Forge (DIGITAL...,), de código livre disponível na internet, desenvolvida pela Digital Bazaar. Essa biblioteca utiliza a Web Cryptography API, que é suportada por grande parte dos browsers utilizados (CAN...,), para realizar diversas funções criptográficas.

Para a segunda parte da proposta, foram utilizados os protótipos desenvolvidos por Marlon Trapp em (TRAPP, 2016). Para as alterações necessárias, foram utilizadas as mesmas tecnologias já adotadas no desenvolvimento dos protótipos.

5.1 EMISSÃO EM BROWSER

Na implementação da primeira parte da proposta, foi utilizada a rotina de certificação descrita em 3.1, que consiste em duas etapas. O cenário é representado pela figura 7. A primeira tem a tarefa de gerar o par de chaves, salvar a chave privada no session storage do browser e montar a requisição de certificado assinada para enviar ao servidor. Tanto a chave privada quanto a requisição são convertidas para o formato PEM, que é facilmente reconhecido pela maioria das tecnologias que podem ter sido utilizadas na aplicação que deseja integrar o código desenvolvido neste trabalho. O session storage do browser é utilizado para o armazenamento temporário da chave privada, por ser acessível somente da própria máquina do cliente e ser deletado sempre que o browser é encerrado.

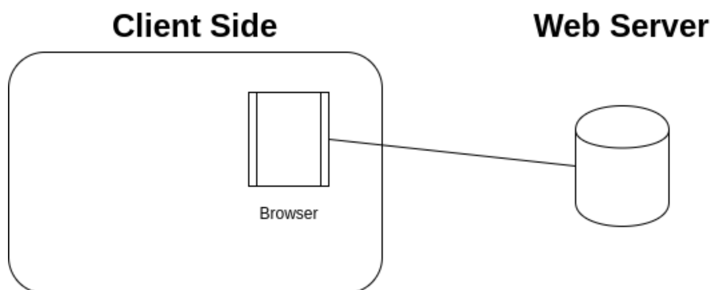

Figura 7 – Emissão em browser.

Já a segunda etapa, que deve esperar pela resposta do servidor com o certificado, é responsável por buscar a chave privada do session storage e juntá-la com o certificado em um objeto PKCS #12, cifrado com uma senha escolhida pelo usuário. O objeto é convertido para o formato padrão de arquivos PKCS #12, deixando para a aplicação se preocupar apenas com a disponibilização de um link para download do arquivo, que já estará em um formato facilmente reconhecível pelo sistema operacional e browser. Após fazer o download deste arquivo, o cliente pode importá-lo em seu navegador, manualmente.

5.1.1 Integração com o SAEC

O código desenvolvido consiste em duas funções simples, uma para cada uma das etapas descritas acima. Para a primeira parte, foi necessário utilizar JavaScript para buscar o valor do campo que especifica o tamanho da chave e passá-lo por parâmetro para a primeira função, que retorna uma requisição de certificado assinada para ser enviada ao servidor. For preciso, também, adaptar o código do back-end do SAEC, visto que este não esperava por uma requisição de certificado, mas apenas uma chave pública para a emissão de um certificado.

Para a segunda parte foi também necessário utilizar JavaScript, dessa vez para chamar o segundo método com o certificado emitido pelo servidor como parâmetro, em formato PEM, além da criação de um link de download para que o usuário pudesse baixar um arquivo em formato p12, tendo como conteúdo o retorno daquele método.

5.2 EMISSÃO EM TOKEN

O cenário da emissão em token, descrito pela figura 8, é muito parecido com o descrito pela figura 7, tendo como diferença o fato do par de chaves ser gerado em um dispositivo criptográfico, e não diretamente no browser do cliente.

5.2.1 Integração com o SAEC

Juntamente com o código desenvolvido para o módulo criptográfico do Device Server, foi alterada parte da aplicação web exemplo desenvolvida por Trapp para que fosse utilizada como biblioteca para ajudar na integração, disponibilizando métodos para realizar a conexão

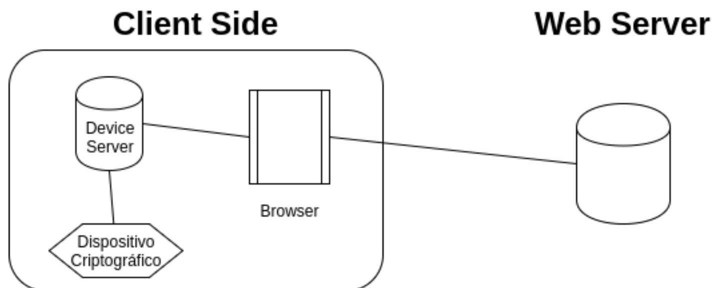

Figura 8 – Emissão em token.

com o Device Server, a troca de mensagens, e a chamada das funções existentes no módulo criptográfico.

Com o acesso aos métodos listados no parágrafo anterior, as únicas alterações necessárias na aplicação foram relacionadas à interface com o usuário, se resumindo à decidir por meio de que interação cada função seria chamada, e como os dados seriam apresentados ao cliente.

6 CONCLUSÃO

Neste trabalho foram apresentadas soluções para a emissão de certificados digitais por autoridades certificadoras online que podem ser facilmente utilizadas como alternativa para soluções já existentes e largamente utilizadas para as quais em breve não haverá mais suporte. Como exemplo, podemos citar a tag html keygen, usada para a geração de pares de chaves criptográficas, que já não é mais suportada pelo navegador google chrome, e a utilização de Java Applets para a comunicação com tokens criptográficos, que também vem sendo depreciada.

No capítulo 3 é descrita detalhadamente a proposta deste trabalho, juntamente com a separação e explicação dos passos escolhidos para uma rotina de certificação, em 3.1, que foi seguida na implementação da solução. Na seção 3.2 é apresentado o trabalho de Marlon Trapp, que foi utilizado como alternativa viável aos Java Applets para atingir os resultados esperados.

No capítulo 5 são apresentados detalhes sobre a implementação do trabalho proposto, explicando-se cada funcionalidade desenvolvida. As seções 5.1.1 e 5.2.1 contêm os passos que foram necessários para integrar as ferramentas à um software já existente, e servem de exemplo para demonstrar a simplicidade do processo de integração, assim como de base para outros softwares que desejam adotar as soluções aqui propostas.

6.1 TRABALHOS FUTUROS

Apesar de utilizável, o modo como foi proposta a utilização do Device Server em HTTPS gera um trabalho indesejável para o cliente, que além de fazer o download do Device Server e executá-lo, deve também fazer o download de um certificado e importá-lo em seu navegador. Uma trabalho futuro interessante seria estudar uma nova forma, simplificada, de instalar o certificado no browser do usuário, ou uma outra forma de executar o servidor java em HTTPS, evitando a necessidade de importar um novo certificado no navegador de cada cliente.

REFERĘNCIAS

CAN I Use? <http://caniuse.com/feat=cryptography>. Acessado em 24/07/2016.

DIGITAL Bazaar inc. <http://new.digitalbazaar.com/forge/>. Acessado em 20/11/2016.

HOUSLEY R.; POLK, T. *Planning for PKI: Best Practices Guide for Deploying Public Key Infrastructure.* [S.l.]: John Wiley Sons, Inc, 2001. 327 p.

RSA LABORATORIES. Public Key Cryptography Standards. <http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/public-key-cryptography-standards.htm>. Acessado em 21/11/2016.

TRAPP, M.

Uma Alternativa Aos Applets Em Java Para Acesso A Dispositivos — Universidade Federal de Santa Catarina, 2016.

Anexo A -- Código Fonte JavaScript/Forge

```
1.  /**
2.   * Generates a key pair with given size
3.   * Returns the public key's PEM and saves the private key's PEM on a session storage
4.   *
5.   * @param keySize the size the keys shall have
6.   * @param storageName the name with which the private key's PEM will be saved on the
    session storage
7.   *
8.   * @return publicKeyPem the public key's PEM
9.   */
10. function generateKeys(keySize, storageName) {
11.     var pki = forge.pki;
12.     var keys = forge.pki.rsa.generateKeyPair(keySize);
13.     var publicKeyPem = pki.publicKeyToPem(keys.publicKey);
14.     var privateKeyPem = pki.privateKeyToPem(keys.privateKey);
15.     sessionStorage.setItem(storageName, JSON.stringify(privateKeyPem));
16.
17.     var csr = forge.pki.createCertificationRequest();
18.     csr.publicKey = keys.publicKey;
19.     csr.sign(keys.privateKey);
20.     var certReqPem = forge.pki.certificationRequestToPem(csr);
21.
22.     return certReqPem;
23. };
24.
25. /**
26.  * Generates a certificape pkcs#12
27.  * Returns the certificate in a p12 base64 format
28.  *
29.  * @param storageName the name with which the private key's PEM was saved on the
    session storage
30.  * @param certPem the PEM of the certificate that shall be converted into p12 format
31.  * @param pw the passphrase with which the p12 shall be locked
32.  *
33.  * @return p12b64 the certificate together with it's private key in a p12b64 format
34.  */
35. function generateP12(storageName, certPem, pw) {
36.     var keyPem = JSON.parse(sessionStorage.getItem(storageName));
37.     var pki = forge.pki;
38.     var key = pki.privateKeyFromPem(keyPem);
39.     var cert = pki.certificateFromPem(certPem);
40.     var p12Asn1 = forge.pkcs12.toPkcs12Asn1(key, cert, pw, {algorithm: '3des'});
41.     var p12Der = forge.asn1.toDer(p12Asn1).getBytes();
42.     var p12b64 = forge.util.encode64(p12Der);
43.     return p12b64;
44. };
```

Anexo B -- Alterações no Device Server

```
1.      /**
2.       * Starts Grizzly HTTP server exposing JAX-RS resources defined in this
3.       * application.
4.       *
5.       * @return Grizzly HTTP server.
6.       */
7.      public static HttpServer startServer() {
8.
9.          try {
10.             final ResourceConfig rc = new ResourceConfig(ServerResource.class);
11.             rc.register(ConnectionFilter.class);
12.
13.             HttpServer server =
    GrizzlyHttpServerFactory.createHttpServer(URI.create(BASE_URI), rc, true,
    getSslEngineConfig(),false);
14.
15.             Logger l =
    Logger.getLogger("org.glassfish.grizzly.http.server.HttpHandler");
16.             l.setLevel(Level.ALL);
17.             l.setUseParentHandlers(false);
18.             ConsoleHandler ch = new ConsoleHandler();
19.             ch.setLevel(Level.ALL);
20.             l.addHandler(ch);
21.
22.             NetworkListener listener = server.getListeners().iterator().next();
23.             TCPNIOTransport transport = listener.getTransport();
24.             ThreadPoolConfig workerThreadPoll =
    transport.getWorkerThreadPoolConfig();
25.             workerThreadPoll.setCorePoolSize(1);
26.             workerThreadPoll.setMaxPoolSize(2);
27.             transport.setSelectorRunnersCount(1);
28.             server.start();
29.             return server;
30.         } catch (Throwable e) {
31.             e.printStackTrace();
32.             System.exit(0);
33.         }
34.         return null;
35.     }
36.
37.
38.     private static class TrustAllCerts implements X509TrustManager {
39.         @Override public void checkClientTrusted(X509Certificate[] x509Certificates,
    String s) throws CertificateException { }
40.         @Override public void checkServerTrusted(X509Certificate[] x509Certificates,
    String s) throws CertificateException { }
41.         @Override public X509Certificate[] getAcceptedIssuers() { return new
    X509Certificate[0]; }
42.     }
43.
44.     private final static TrustManager[] trustAllCerts = new TrustManager[] { new
    TrustAllCerts() };
45.
```

```
46.     public static SSLEngineConfigurator getSslEngineConfig() throws
    KeyManagementException, NoSuchAlgorithmException,IOException,
    UnrecoverableKeyException, KeyStoreException, CertificateException {
47.         SSLContext sc = SSLContext.getInstance("TLSv1.2");
48.
49.         KeyStore ks = KeyStore.getInstance("JKS");
50.         InputStream ksIs = new FileInputStream("./localhost.jks");
51.         try {
52.             ks.load(ksIs, "123456".toCharArray());
53.         } finally {
54.             if (ksIs != null) {
55.                 ksIs.close();
56.             }
57.         }
58.
59.         KeyManagerFactory kmf =
    KeyManagerFactory.getInstance(KeyManagerFactory.getDefaultAlgorithm());
60.         kmf.init(ks, "123456".toCharArray());
61.
62.         sc.init(kmf.getKeyManagers(), trustAllCerts, null);
63.
64.         SSLServerSocket ssl = (SSLServerSocket)
    sc.getServerSocketFactory().createServerSocket(40445);
65.         ssl.setEnabledProtocols(new String[] {"TLSv1", "TLSv1.1", "TLSv1.2",
    "SSLv3"});
66.
67.         SSLEngineConfigurator sslEngineConfigurator = new SSLEngineConfigurator(sc,
    false, false, false);
68.         return sslEngineConfigurator;
69.     }
```

Anexo C -- Alterações no Cryptographic Module

Arquivo CryptographicModule.java

```
1.    public DeviceMessage incomingMessage(DeviceMessage message) throws DeviceException
    {
2.       switch (message.getMethod()) {
3.              case "list":
4.                  return this.list();
5.              case "listCerts":
6.                  return this.listCerts(message);
7.              case "sign":
8.                  return this.sign(message);
9.              case "hashsign":
10.                 return this.hashsign(message);
11.             case "verify":
12.                 return this.verify(message);
13.             case "generateKey":
14.                 return this.generateKey(message);
15.             case "installCert":
16.                 return this.installCertificate(message);
17.             case "getCert":
18.                 return this.getCertificate(message);
19.             default:
20.                 break;
21.         }
22.         return new DeviceMessage("return", "result: {}");
23.     }
24.
25.     private DeviceMessage hashsign(DeviceMessage message) throws DeviceException {
26.         String jsonInString = null;
27.         try {
28.             ObjectMapper mapper = new ObjectMapper();
29.             SignatureRequest signReq = null;
30.             signReq = mapper.readValue(message.getMessage(), SignatureRequest.class);
31.             byte[] signedBytes = CryptoDevices.hashsign(signReq);
32.             jsonInString =
    mapper.writeValueAsString(Base64.getEncoder().encodeToString(signedBytes));
33.         } catch (IOException e) {
34.             throw new DeviceException(e.getMessage(), e);
35.         }
36.         return new DeviceMessage("hashsign", jsonInString);
37.     }
38.
39.     private DeviceMessage verify(DeviceMessage message) throws DeviceException {
40.         String jsonInString = null;
41.         try {
42.             ObjectMapper mapper = new ObjectMapper();
43.             SignatureRequest signReq = null;
44.             signReq = mapper.readValue(message.getMessage(), SignatureRequest.class);
45.             byte[] signedBytes = CryptoDevices.sign(signReq);
46.             jsonInString =
    mapper.writeValueAsString(Base64.getEncoder().encodeToString(signedBytes));
47.         } catch (IOException e) {
48.             throw new DeviceException(e.getMessage(), e);
49.         }
```

```
50.         return new DeviceMessage("verify", jsonInString);
51.     }
52.
53.     private DeviceMessage generateKey(DeviceMessage message) {
54.         String pubKeyPem = null;
55.         try {
56.             ObjectMapper mapper = new ObjectMapper();
57.             GenKeyRequest genkeyreq = null;
58.             genkeyreq = mapper.readValue(message.getMessage(), GenKeyRequest.class);
59.             pubKeyPem = CryptoDevices.generateKeyPair(genkeyreq);
60.         } catch (IOException e) {
61.             e.printStackTrace();
62.         }
63.         return new DeviceMessage("generateKey", pubKeyPem);
64.     }
65.
66.     private DeviceMessage installCertificate(DeviceMessage message) {
67.         try {
68.             ObjectMapper mapper = new ObjectMapper();
69.             InstallCertRequest req = null;
70.             req = mapper.readValue(message.getMessage(), InstallCertRequest.class);
71.             CryptoDeviceInfo deviceInfo = new CryptoDeviceInfo(req.getTokenSerial(),
null);
72.             CryptoDevices.installCertificate(deviceInfo, req.getCertPem());
73.
74.         } catch (IOException e) {
75.             e.printStackTrace();
76.         }
77.         return new DeviceMessage("installCertificate", "happiness");
78.     }
79.
80.     private DeviceMessage getCertificate(DeviceMessage message) throws
DeviceException {
81.         String jsonInString = null;
82.         try {
83.             ObjectMapper mapper = new ObjectMapper();
84.             SignatureRequest signReq = null;
85.             signReq = mapper.readValue(message.getMessage(), SignatureRequest.class);
86.             CertificateInfo certInfo = new CertificateInfo();
87.             certInfo.setDeviceSerial(signReq.getTokenSerial());
88.             certInfo.setId(signReq.getKeyId());
89.             certInfo.setLabel(signReq.getKeyLabel());
90.             CryptoDeviceInfo deviceInfo = new CryptoDeviceInfo();
91.             deviceInfo.setSerial(signReq.getTokenSerial());
92.             Certificate certificate = CryptoDevices.getCertificate(deviceInfo,
certInfo);
93.             jsonInString =
mapper.writeValueAsString(Base64.getEncoder().encodeToString(certificate.getEncoded()
));
94.         } catch (IOException | CertificateEncodingException e) {
95.             throw new DeviceException(e.getMessage(), e);
96.         }
97.         return new DeviceMessage("getCert", jsonInString);
98.     }
```

Arquivo CryptoDevices.java

```
1.      public static String generateKeyPair(GenKeyRequest genkeyreq) {
2.          if (lastListingTokens == null) {
3.              CryptoDevices.list();
4.          }
5.          CryptoDeviceInfo deviceinfo = new CryptoDeviceInfo();
6.          deviceinfo.setLabel(genkeyreq.getLabel());
7.          deviceinfo.setSerial(genkeyreq.getSerial());
8.          Token token = lastListingTokens.get(deviceinfo);
9.          Session session;
10.         RSAPublicKey pk = null;
11.         try {
12.             session = token.openSession(Token.SessionType.SERIAL_SESSION,
    Token.SessionReadWriteBehavior.RW_SESSION, null, null);
13.
14.             char[] psw = Utils.getPassword();
15.             session.login(Session.UserType.USER, psw);
16.
17.             Mechanism m = Mechanism.get(PKCS11Constants.CKM_RSA_PKCS_KEY_PAIR_GEN);
18.
19.             RSAPublicKey pubKeyTemplate =
    generatePublicKeyTemplate(genkeyreq.getKeyid());
20.             RSAPrivateKey privKeyTemplate =
    generatePrivateKeyTemplate(genkeyreq.getKeyid());
21.
22.             KeyPair kp = session.generateKeyPair(m, pubKeyTemplate, privKeyTemplate);
23.             pk = (RSAPublicKey) kp.getPublicKey();
24.             RSAPublicKeySpec spec = new RSAPublicKeySpec(new
    BigInteger(pk.getModulus().getByteArrayValue()),
    newBigInteger(pk.getPublicExponent().getByteArrayValue()));
25.             KeyFactory factory;
26.             factory = KeyFactory.getInstance("RSA");
27.             byte[] pubKeyPem = factory.generatePublic(spec).getEncoded();
28.
29.             return new
    String(org.bouncycastle.util.encoders.Base64.encode(pubKeyPem));
30.
31.         } catch (Exception e) {
32.             e.printStackTrace();
33.         } finally {
34.             try {
35.                 token.closeAllSessions();
36.             } catch (TokenException e) {
37.             }
38.         }
39.
40.         return null;
41.     }
42.
43.     public static String installCertificate(CryptoDeviceInfo tokenSerial, String
    certPEM) {
44.         if (lastListingTokens == null) {
45.             CryptoDevices.list();
```

```
46.         }
47.         Token token = lastListingTokens.get(tokenSerial);
48.         Session session;
49.         try {
50.             session = token.openSession(Token.SessionType.SERIAL_SESSION,
Token.SessionReadWriteBehavior.RW_SESSION, null, null);
51.
52.             char[] psw = Utils.getPassword();
53.             session.login(Session.UserType.USER, psw);
54.
55.             byte [] decoded =
Base64.getDecoder().decode(certPEM.replaceAll("-----BEGIN CERTIFICATE-----",
"").replaceAll("-----END CERTIFICATE-----", ""));
56.             X509Certificate x509Certificate =
(X509Certificate)CertificateFactory.getInstance("X.509").generateCertificate(newByteA
rrayInputStream(decoded));
57.             java.security.PublicKey pk = x509Certificate.getPublicKey();
58.
59.             java.security.interfaces.RSAPublicKey rsaPublicKey =
(java.security.interfaces.RSAPublicKey) pk;
60.             RSAPrivateKey rsaPrivateKeySearchTemplate = new RSAPrivateKey();
61.             byte[] modulus =
Util.unsignedBigIntergerToByteArray(rsaPublicKey.getModulus());
62.             rsaPrivateKeySearchTemplate.getModulus().setByteArrayValue(modulus);
63.             Key keyTemplate = rsaPrivateKeySearchTemplate;
64.             Object searchTemplate = keyTemplate;
65.
66.             Object[] objects;
67.             session.findObjectsInit(searchTemplate);
68.             objects = session.findObjects(1);
69.             Key foundKey = (Key) objects[0];
70.             session.findObjectsFinal();
71.
72.             byte [] objectID = foundKey.getId().getByteArrayValue();
73.
74.             X509PublicKeyCertificate pkcs11X509PublicKeyCertificate = new
X509PublicKeyCertificate();
75.             byte[] asn1encodedSubjectName =
x509Certificate.getSubjectX500Principal().getEncoded();
76.             byte[] asn1EncodedIssuerName =
x509Certificate.getIssuerX500Principal().getEncoded();
77.             String commonName = x509Certificate.getSubjectX500Principal().getName();
78.             char[] label = commonName.toCharArray();
79.
80.             byte[] asn1SerialNumber = new
DERInteger(x509Certificate.getSerialNumber()).getEncoded();
81.
82.             pkcs11X509PublicKeyCertificate.getToken().setBooleanValue(Boolean.TRUE);
83.
pkcs11X509PublicKeyCertificate.getPrivate().setBooleanValue(Boolean.FALSE);
84.             pkcs11X509PublicKeyCertificate.getLabel().setCharArrayValue(label);
85.             pkcs11X509PublicKeyCertificate.getId().setByteArrayValue(objectID);
```

```
86.
    pkcs11X509PublicKeyCertificate.getSubject().setByteArrayValue(asn1encodedSubjectName)
    ;
87.
    pkcs11X509PublicKeyCertificate.getIssuer().setByteArrayValue(asn1EncodedIssuerName);
88.
    pkcs11X509PublicKeyCertificate.getSerialNumber().setByteArrayValue(asn1SerialNumber);
89.
    pkcs11X509PublicKeyCertificate.getValue().setByteArrayValue(x509Certificate.getEncode
    d());
90.
91.             session.createObject(pkcs11X509PublicKeyCertificate);
92.         } catch (Throwable e) {
93.             e.printStackTrace();
94.         }
95.
96.         return "Success?";
97.     }
98.
99.
100.        private static RSAPrivateKey generatePrivateKeyTemplate(String keyid) {
101.            RSAPrivateKey rsaPrivateKeyTemplate = new RSAPrivateKey();
102.            byte[] bytearray = keyid.getBytes();
103.            rsaPrivateKeyTemplate.getId().setByteArrayValue(bytearray);
104.            char[] chararray = keyid.toCharArray();
105.            rsaPrivateKeyTemplate.getLabel().setCharArrayValue(chararray);
106.
107.
108.            rsaPrivateKeyTemplate.getToken().setBooleanValue(true);
109.            rsaPrivateKeyTemplate.getPrivate().setBooleanValue(true);
110.            rsaPrivateKeyTemplate.getSensitive().setBooleanValue(true);
111.            rsaPrivateKeyTemplate.getNeverExtractable().setBooleanValue(true);
112.            rsaPrivateKeyTemplate.getAlwaysSensitive().setBooleanValue(true);
113.            rsaPrivateKeyTemplate.getLocal().setBooleanValue(true);
114.
115.            rsaPrivateKeyTemplate.getDecrypt().setBooleanValue(true);
116.            rsaPrivateKeyTemplate.getSign().setBooleanValue(true);
117.            rsaPrivateKeyTemplate.getSignRecover().setBooleanValue(true);
118.            rsaPrivateKeyTemplate.getUnwrap().setBooleanValue(true);
119.
120.            return rsaPrivateKeyTemplate;
121.        }
122.
123.            /**
124.             * @param keyPairAttributes the attributes of the key pair
125.             * @param id the object ID
126.             *
127.             * @return a RSA public key template with the given parameters
128.             */
129.        private static RSAPublicKey generatePublicKeyTemplate(String keyid) {
130.
131.            long modulusBits = Long.valueOf(1024);
132.            RSAPublicKey rsaPublicKeyTemplate = new RSAPublicKey();
133.            byte[] publicExponentBytes = { 0x01, 0x00, 0x01 };
```

```
134.            rsaPublicKeyTemplate.getModulusBits().setLongValue(modulusBits);
135.
rsaPublicKeyTemplate.getPublicExponent().setByteArrayValue(publicExponentBytes);
136.
137.            System.out.println(keyid);
138.            byte[] bytearray = keyid.getBytes();
139.            System.out.println(bytearray);
140.            rsaPublicKeyTemplate.getId().setByteArrayValue(bytearray);
141.            char[] chararray = keyid.toCharArray();
142.            rsaPublicKeyTemplate.getLabel().setCharArrayValue(chararray);
143.
144.
145.            rsaPublicKeyTemplate.getToken().setBooleanValue(true);
146.            rsaPublicKeyTemplate.getLocal().setBooleanValue(true);
147.
148.            rsaPublicKeyTemplate.getVerify().setBooleanValue(true);
149.            rsaPublicKeyTemplate.getVerifyRecover().setBooleanValue(true);
150.            rsaPublicKeyTemplate.getEncrypt().setBooleanValue(true);
151.            rsaPublicKeyTemplate.getWrap().setBooleanValue(true);
152.
153.            return rsaPublicKeyTemplate;
154.        }
155.
156.
157.
158.        /**
159.         * This method will return all certificates that have a public key with the
160.         * same ID of it, if the device contains multiple certificates with the same
161.         * ID it can cause strange effects.
162.         *
163.         * @param tokenSerial
164.         *            the serial of the token that will receive this operation
165.         *
166.         * @return a list of ID and Label of each certificate that match the
167.         *         conditions
168.         */
169.        public static List<CertificateInfo> listCertsWithMatchingKey(CryptoDeviceInfo
tokenSerial) {
170.            if (lastListingTokens == null)
171.                CryptoDevices.list();
172.            Token token = lastListingTokens.get(tokenSerial);
173.            Session session;
174.            List<CertificateInfo> matchedCerts = new ArrayList<CertificateInfo>();
175.            try {
176.                session = token.openSession(Token.SessionType.SERIAL_SESSION,
Token.SessionReadWriteBehavior.RO_SESSION, null, null);
177.
178.                Object[] objects;
179.                // FIND public keys
180.                List<PublicKey> publicKeys = new ArrayList<PublicKey>();
181.                session.findObjectsInit(new PublicKey());
182.                while ((objects = session.findObjects(10)).length > 0) {
183.                    for (Object object : objects) {
184.                        publicKeys.add((PublicKey) object);
```

```
185.                        }
186.                    }
187.                    session.findObjectsFinal();
188.
189.                    // FIND certificates
190.                    List<X509PublicKeyCertificate> certificates = new
ArrayList<X509PublicKeyCertificate>();
191.                    session.findObjectsInit(new X509PublicKeyCertificate());
192.                    while ((objects = session.findObjects(10)).length > 0) {
193.                        for (Object object : objects) {
194.                            certificates.add((X509PublicKeyCertificate) object);
195.                        }
196.                    }
197.                    session.findObjectsFinal();
198.
199.                    for (X509PublicKeyCertificate certificate : certificates) {
200.                        for (PublicKey key : publicKeys) {
201.                            if
(key.getAttribute(Attribute.ID).equals(certificate.getAttribute(Attribute.ID))) {
202.                                CertificateInfo certInfo = new CertificateInfo();
203.
certInfo.setId(certificate.getAttribute(Attribute.ID).toString());
204.
certInfo.setLabel(certificate.getAttribute(Attribute.LABEL).toString());
205.                                certInfo.setDeviceSerial(tokenSerial.getSerial());
206.                                matchedCerts.add(certInfo);
207.                            }
208.                        }
209.                    }
210.
211.                } catch (Throwable e) {
212.                    System.out.println("shit happens"+e.getMessage());
213.                    e.printStackTrace();
214.                } finally {
215.                    try {
216.                        token.closeAllSessions();
217.                    } catch (TokenException e) {
218.                    }
219.                }
220.                return matchedCerts;
221.            }
222.
223.            public static Certificate getCertificate(CryptoDeviceInfo tokenId,
CertificateInfo certificateInfo) {
224.                if (lastListingTokens == null)
225.                    CryptoDevices.list();
226.                Token token = lastListingTokens.get(tokenId);
227.                Session session;
228.                Certificate certificate = null;
229.                try {
230.                    session = token.openSession(Token.SessionType.SERIAL_SESSION,
Token.SessionReadWriteBehavior.RO_SESSION, null, null);
231.
232.                    Object[] objects;
```

```
233.                // FIND certificates
234.                CertificateTemplate template = new CertificateTemplate();
235.                if (certificateInfo.getId() != null)
236.                    template.setIdAttribute(certificateInfo.getId());
237.                if (certificateInfo.getLabel() != null)
238.                    template.setLabelAttribute(certificateInfo.getLabel());
239.
240.                session.findObjectsInit(template);
241.                objects = session.findObjects(1);
242.                session.findObjectsFinal();
243.                if (objects.length > 0) {
244.                    X509PublicKeyCertificate cert = (X509PublicKeyCertificate)
    objects[0];
245.                    Attribute attr = cert.getAttribute(Attribute.VALUE);
246.                    byte[] certBytes =
    DatatypeConverter.parseHexBinary(attr.toString());
247.                    ByteArrayInputStream bais = new ByteArrayInputStream(certBytes);
248.                    certificate =
    CertificateFactory.getInstance("x509").generateCertificate(bais);
249.                }
250.
251.            } catch (TokenException e) {
252.                e.printStackTrace();
253.            } catch (CertificateException e) {
254.            } finally {
255.                try {
256.                    token.closeAllSessions();
257.                } catch (TokenException e) {
258.                }
259.            }
260.            return certificate;
261.
262.        }
263.
264.        public static byte[] sign(SignatureRequest signReq) throws DeviceException {
265.            if (lastListingTokens == null)
266.                CryptoDevices.list();
267.            Token token = lastListingTokens.get(new
    CryptoDeviceInfo(signReq.getTokenSerial(), null));
268.            Session session;
269.            try {
270.                session = token.openSession(Token.SessionType.SERIAL_SESSION,
    Token.SessionReadWriteBehavior.RW_SESSION, null, null);
271.                char[] psw = Utils.getPassword();
272.                if (psw != null) {
273.                    session.login(Session.UserType.USER, psw);
274.
275.                    PrivateKeyTemplate templateSignatureKey = new
    PrivateKeyTemplate();
276.                    templateSignatureKey.getSign().setBooleanValue(Boolean.TRUE);
277.                    templateSignatureKey.setIdAttribute(signReq.getKeyId());
278.
279.                    byte[] bytes = Base64.getDecoder().decode(signReq.getData());
280.                    session.findObjectsInit(templateSignatureKey);
```

```
281.                    Object[] objects = session.findObjects(1);
282.                    session.findObjectsFinal();
283.                    if (objects.length > 0) {
284.                        PrivateKey key = (PrivateKey) objects[0];
285.                        MessageDigest digestEngine =
MessageDigest.getInstance(signReq.getHashAlgorithm(), "BC");
286.                        // String sha1Oid = CMSSignedGenerator.DIGEST_SHA1;
287.                        Field field =
CMSSignedGenerator.class.getDeclaredField("DIGEST_" + signReq.getHashAlgorithm());
288.                        String hashOid = (String) field.get(null);
289.
290.                        // be sure that your token can process the specified
291.                        // mechanism
292.                        Mechanism signatureMechanism =
Mechanisms.getCorrespondentMechanism(key);
293.                        if (signatureMechanism != null) {
294.                            // initialize for signing
295.                            session.signInit(signatureMechanism, key);
296.
297.                            //byte[] digest = digestEngine.digest(bytes); FIXME now it
expects a hash to sign
298.                            DigestInfo di = new DigestInfo(new AlgorithmIdentifier(new
ASN1ObjectIdentifier(hashOid)), bytes/*digest*/);
299.                            byte[] signatureValue = session.sign(di.getEncoded());
300.                            session.closeSession();
301.                            return signatureValue;
302.                        } else {
303.                            throw new DeviceException("Key type not supported");
304.                        }
305.                    }
306.                } else {
307.                    throw new DeviceException("The user didn't allow the signature");
308.                }
309.            } catch (Throwable e) {
310.                throw new DeviceException(e.getMessage(), e);
311.            } finally {
312.                try {
313.                    token.closeAllSessions();
314.                } catch (TokenException e) {
315.                }
316.            }
317.            return null;
318.
319.        }
320.
321.        public static byte[] hashsign(SignatureRequest signReq) throws DeviceException
{
322.            if (lastListingTokens == null)
323.                CryptoDevices.list();
324.            Token token = lastListingTokens.get(new
CryptoDeviceInfo(signReq.getTokenSerial(), null));
325.            Session session;
326.            try {
```

```
327.                session = token.openSession(Token.SessionType.SERIAL_SESSION,
    Token.SessionReadWriteBehavior.RW_SESSION, null, null);
328.                char[] psw = Utils.getPassword();
329.                if (psw != null) {
330.                    session.login(Session.UserType.USER, psw);
331.
332.                    PrivateKeyTemplate templateSignatureKey = new
    PrivateKeyTemplate();
333.                    templateSignatureKey.getSign().setBooleanValue(Boolean.TRUE);
334.                    templateSignatureKey.setIdAttribute(signReq.getKeyId());
335.
336.                    byte[] bytes = Base64.getDecoder().decode(signReq.getData());
337.                    session.findObjectsInit(templateSignatureKey);
338.                    Object[] objects = session.findObjects(1);
339.                    session.findObjectsFinal();
340.                    if (objects.length > 0) {
341.                        PrivateKey key = (PrivateKey) objects[0];
342.                        MessageDigest digestEngine =
    MessageDigest.getInstance(signReq.getHashAlgorithm(), "BC");
343.                        // String sha1Oid = CMSSignedGenerator.DIGEST_SHA1;
344.                        Field field =
    CMSSignedGenerator.class.getDeclaredField("DIGEST_" + signReq.getHashAlgorithm());
345.                        String hashOid = (String) field.get(null);
346.
347.                        // be sure that your token can process the specified
348.                        // mechanism
349.                        Mechanism signatureMechanism =
    Mechanisms.getCorrespondentMechanism(key);
350.                        if (signatureMechanism != null) {
351.                            // initialize for signing
352.                            session.signInit(signatureMechanism, key);
353.
354.                            byte[] digest = digestEngine.digest(bytes);
355.                            DigestInfo di = new DigestInfo(new AlgorithmIdentifier(new
    ASN1ObjectIdentifier(hashOid)), digest);
356.                            byte[] signatureValue = session.sign(di.getEncoded());
357.                            session.closeSession();
358.                            return signatureValue;
359.                        } else {
360.                            throw new DeviceException("Key type not supported");
361.                        }
362.                    }
363.                } else {
364.                    throw new DeviceException("The user didn't allow the signature");
365.                }
366.            } catch (Throwable e) {
367.                throw new DeviceException(e.getMessage(), e);
368.            } finally {
369.                try {
370.                    token.closeAllSessions();
371.                } catch (TokenException e) {
372.                }
373.            }
374.            return null;
```

```
375.
376.         }
```

Arquivo InstallCertRequest.java

```
1.  package br.com.trapp.deviceserver.cryptographicmodule.objects;
2.
3.  public class InstallCertRequest {
4.
5.      private String tokenSerial;
6.      private String certPem;
7.
8.      public InstallCertRequest() {
9.
10.     }
11.
12.     public InstallCertRequest(String tokenSerial, String certPem) {
13.         this.tokenSerial = tokenSerial;
14.         this.certPem = certPem;
15.     }
16.
17.     public String getTokenSerial() {
18.         return tokenSerial;
19.     }
20.
21.     public void setTokenSerial(String tokenSerial) {
22.         this.tokenSerial = tokenSerial;
23.     }
24.
25.     public String getCertPem() {
26.         return certPem;
27.     }
28.
29.     public void setCertPem(String certPem) {
30.         this.certPem = certPem;
31.     }
32.
33.     @Override
34.     public String toString() {
35.         return "SignatureRequest [tokenSerial=" + tokenSerial + ", certPem=" +
    certPem + "]";
36.     }
37.
38. }
```

Anexo D -- Biblioteca da Aplicação Web Exemplo

Arquivo device-server.js

```
1.  function DeviceServer(base64Cert) {
2.      this.cert = base64Cert;
3.  }
4.
5.  DeviceServer.prototype.connectToModule = function (moduleName, privKey, callback,
    errorCallBack) {
6.
7.      var req = this.createRequest();
8.      var instance = this;
9.
10.     req.onreadystatechange = function () {
11.         if (req.readyState != 4) return; // Not there yet
12.         if (req.status != 200) {
13.             errorCallBack();
14.         }
15.         // Request successful, read the response
16.         instance.establishConnection(req.responseText, privKey, callback);
17.     };
18.
19.     data = {};
20.     data.certificate = this.cert;
21.     // req.open("POST", "http://localhost:9977/deviceserver/startsession/" +
    moduleName);
22.     req.open("POST", "https://localhost:9977/deviceserver/startsession/" +
    moduleName);
23.     req.setRequestHeader("Content-Type", "application/json");
24.     req.send(JSON.stringify(data));
25. };
26.
27. //END DeviceServer class
28.
29. DeviceServer.prototype.establishConnection = function (authProcess, pem, callback) {
30.     authObject = JSON.parse(authProcess)
31.     instance = this;
32.     this.session = authObject.session;
33.     this.dh = new DiffieHellman({
34.         p: authObject.p,
35.         g: authObject.g
36.     });
37.     myY = this.dh.generateY().toString();
38.     this.dh.setOtherY(new forge.jsbn.BigInteger(authObject.y));
39.
40.     var req = this.createRequest();
41.     req.onreadystatechange = function () {
42.         if (req.readyState != 4) return; // Not there yet
43.         if (req.status != 200) {
44.             // Handle request failure here...
45.         }
46.         // Request successful, read the response
47.         instance.calculateSessionKey(req.responseText);
48.         callback();
49.     }
```

```
50.
51.
52.     data = {};
53.     data.y = myY;
54.     var pki = forge.pki;
55.     // convert a PEM-formatted private key to a Forge private key
56.     var privateKey = pki.privateKeyFromPem(pem);
57.
58.     var md = forge.md.sha256.create();
59.     md.update(this.session.toString());
60.     md.update(ds.dh.p.toString());
61.     md.update(ds.dh.g.toString());
62.     md.update(ds.dh.y.toString());
63.     var signature = privateKey.sign(md);
64.
65.     data.signature = new String(forge.util.encode64(signature));
66.     // req.open("POST", "http://localhost:9977/deviceserver/establishsession/" +
    this.session);
67.     req.open("POST", "https://localhost:9977/deviceserver/establishsession/" +
    this.session);
68.     req.setRequestHeader("Content-Type", "application/json");
69.     req.send(JSON.stringify(data));
70.
71. }
72.
73. DeviceServer.prototype.calculateSessionKey = function () {
74.     var md = forge.md.sha256.create();
75.     md.update(ds.dh.k.toString());
76.     this.key = md.digest();
77. }
78.
79. DeviceServer.prototype.encrypt = function (data, iv) {
80.     var keyCopy = this.key.copy();
81.     var cipher = forge.cipher.createCipher('AES-CBC', keyCopy);
82.     cipher.start({
83.         iv: iv
84.     });
85.     sb = new forge.util.ByteStringBuffer(data);
86.     cipher.update(sb);
87.     cipher.finish();
88.     var encrypted = cipher.output;
89.     return encrypted
90. }
91.
92. DeviceServer.prototype.decrypt = function (data, iv) {
93.
94.     var keyCopy = this.key.copy();
95.     var decipher = forge.cipher.createDecipher('AES-CBC', keyCopy);
96.     decipher.start({
97.         iv: iv
98.     });
99.     sb = new forge.util.ByteStringBuffer(data);
100.        decipher.update(sb);
101.        decipher.finish();
```

```
102.        var decrypted = decipher.output;
103.        return decrypted
104.    }
105.
106.    DeviceServer.prototype.sendData = function (method, data, callBack) {
107.        var req = this.createRequest();
108.        instance = this;
109.        req.onreadystatechange = function () {
110.            if (req.readyState != 4) return; // Not there yet
111.            if (req.status != 200) {
112.                // Handle request failure here...
113.            }
114.            // Request successful, read the response
115.            resp = JSON.parse(req.responseText);
116.            var iv = forge.util.decode64(resp.iv);
117.            var message = forge.util.decode64(resp.message);
118.            try{
119.                callBack(JSON.parse(instance.decrypt(message, iv)));
120.            } catch(e) {
121.                callBack(instance.decrypt(message, iv).data);
122.            }
123.            // callBack(message);
124.        }
125.
126.        var iv = forge.random.getBytesSync(16);
127.        message = {};
128.        message.message = new String(forge.util.encode64(this.encrypt(data,
    iv).getBytes()));
129.        message.method = new String(forge.util.encode64(this.encrypt(method,
    iv).getBytes()));
130.        message.iv = forge.util.encode64(iv);
131.
132.        req.open("POST", "https://localhost:9977/deviceserver/" + this.session);
133.        req.setRequestHeader("Content-Type", "application/json");
134.        req.send(JSON.stringify(message));
135.    }
136.
137.    DeviceServer.prototype.createRequest = function () {
138.        var result = null;
139.        if (window.XMLHttpRequest) {
140.            // FireFox, Safari, etc.
141.            result = new XMLHttpRequest();
142.        } else if (window.ActiveXObject) {
143.            // MSIE
144.            result = new ActiveXObject("Microsoft.XMLHTTP");
145.        } else {
146.            return;
147.            // No known mechanism -- consider aborting the application
148.        }
149.        return result;
150.    }
151.
152.    DeviceServer.prototype.setSessionID = function (session) {
153.        this.session = session;
```

```
154.    }
155.
156.
157.    //BEGIN DiffieHellman class
158.    function DiffieHellman(dhParameters) {
159.        this.p = new forge.jsbn.BigInteger(dhParameters.p);
160.        this.g = new forge.jsbn.BigInteger(dhParameters.g);
161.    }
162.
163.    DiffieHellman.prototype.generateY = function () {
164.        var myPrng = forge.random.createInstance();
165.        this.x = new forge.jsbn.BigInteger(myPrng.generate(this.p.bitLength()));
166.        this.y = this.g.modPow(this.x, this.p);
167.        return this.y;
168.    }
169.
170.    DiffieHellman.prototype.setOtherY = function (otherY) {
171.        this.k = otherY.modPow(this.x, this.p);
172.        return this.k;
173.    }
```

Arquivo crypto-module.js

```
var ds = new DeviceServer("-----BEGIN CERTIFICATE-----\n" +
    "MIIHADCCAuigAwIBAgIBAzANBgkqhkiG9w0BAQsFADB/MQswCQYDVQQGEwJCUjEX\n" +
    "MBUGA1UECAwOU2FudGEgQ2F0YXJpbmExFjAUBgNVBAcMDUZsb3JpYW5vcG9saXMx\n" +
    "DjAMBgNVBAoMBVRyYXBwMRQwEgYDVQQLDAtEZXZlbG9wbWVudDEZMBcGA1UEAwwQ\n" +
    "RGV2aWNlIFNlcnZlciBDQTAeFw0xNjAzMDMyMDI4NDdaFw0xNzAzMDMyMDI4NDda\n" +
    "MHUxCzAJBgNVBAYTAkJSMRcwFQYDVQQIDA5TYW50YSBDYXRhcmluYTEWMBQGA1UE\n" +
    "BwwNRmxvcmlhbm9wb2xpczEOMAwGA1UECgwFVHJhcHAxFDASBgNVBAsMC0RldmVs\n" +
    "b3BtZW50MQ8wDQYDVQQDDAZKUyBBcHAwggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAw\n" +
    "ggEKAoIBAQDCp9GxEhE9YE5rhxOiT6uYEPpJKIJ7SPn6qS1Kn2OGCw2GDZlc1JTP\n" +
    "xWQh/Q1LEzsIJoh05H88wPKHocbljn/WXqexlYIQ/QX4doCKFg2ePuxtzdQ+wQU1\n" +
    "oj1b0mkUDEzEu1qKuf93EauFr76JyHISf6HGE/rytAVdYujT+H8a/DqO0/+kHvSR\n" +
    "qUAIdpw1u2Cvcd5Rli5tIu4ClucLn2c1WV40LNymnsSbJ1c4UbGGed9ID5eEPzhS\n" +
    "6rf/ILaSlKnb0D2o2fhBDx7XHwEM/vZX++wXvW05VnktjDPSyIMAx5fscZtGZ7GU\n" +
    "xPpNypluD4y4wRNGn2mx8mUr65z+z+OHAgMBAAGjgZAwgY0wCQYDVR0TBAIwADAd\n" +
    "BgNVHQ4EFgQUGvDmfSLIYKfyvYblEt1V0FCrBEkwHwYDVR0jBBgwFoAUqe5wCBqZ\n" +
    "/u4TwaX0TXMfTtK5whUwQAYDVR0fBDkwNzA1oDOgMYYvaHR0cDovL3d3dy5pbmYu\n" +
    "dWZzYy5ici9+bWFybG9uLnRyYXBwL2NhLWNybC5wZW0wDQYJKoZIhvcNAQELBQAD\n" +
    "ggQBACtNYKe+jK6PzE5wSLGtE0OSVqn7v45wVNsFmUpPqZizTHHkwgix+RnJ4Iui\n" +
    "TGKhhdXK8F7Arp0/5wbr+SIOqpywEhnls17OYc4cUAyTLNi1I0UIj4rVVNkcyLH5\n" +
    "cHFD3aLgok5ZbwraWE9xfoID1T59acovHef0NwuUNQGjdlCPa/abXSANS2tHLDCP\n" +
    "UJqBpt/N+/n3v0p+r5PNOzI85JEkWB7iQ8Qo5TerzlLArCiz3IO9SyTf3jjnOwyx\n" +
    "0vSwKhqo8K0OEfFQA9nwKoSG/16KWTrOh/3Iv2K7tF6456Vl4lRA+Ntvgu4be9EU\n" +
    "X8HNMhUYWG3PZGNo+7afq4tJuQPjNV50llj+PvYB/GGPccm5iSSHHq+yJ8e8gFlt\n" +
    "kLJK/7BpUs4f9bdWQLmnPOdXKQFJ7ezMN0eM44qrUr2kVF0qSb47Ng3cp2f52MaL\n" +
    "uOXViW8DMLK+CXuhUwPSI4ydA3JTQHgPrkOm3QMoJgK3M68/DGrZP4FbFeTkiwLC\n" +
    "uQOsq8U8uNX8DuI32MquUiJxMC1veGQ/2xNlW6HI1C31L1oOVb5S0kO0BG87Q3dK\n" +
    "H8EgT8IPssiUNBToTwS7ZQsBLYpTBahPvwNCYwOk15hJTagT1gx9ueYZiT49CJb4\n" +
    "Hfu2ZvlPXGeKL6wwjXpLULl43FOikRqXsHPzkfwswkh+2/pCHxgTK+9P6dlTLy+i\n" +
    "StUgJBVXOeFjZURUIWzqJINU4e8MonMwetRnYOsz1LSrSIlRMSCS6lr1TuPpPMMS\n" +
    "jdVavMyOs2w43/93ejyC3vHmrIdVKrMCIj724Myhg1XnG8+tUGyqWbxTAOf5r24d\n" +
    "QukIB6kur1Uw0kRUDf5Jvt18MYw2+XsWfc5R18TMoto+j/cFvJ1fmPDnTRmh1s3S\n" +
    "yc4NhW/A9WUlYnqrDxUxzAuFweo+RrIke0Nhip3kAo386+mRxwZvraZ3xJr9jvZG\n" +
    "SKufSwopwY3EeeNmRS+D+hpEvnqT58+/eFSrs0A/tfKQIVbQCcsA7bXfZChZcht7\n" +
    "LTfF87B3c/kWCzU3fzua6K7qNe5F8RNSmhG0lkOZAJMPZKE4I0q+Mi1JakB0N00v\n" +
    "6DeNP4hxHUpcEvMqC2DvVOTyCg8ryPrcBkyFhbH94Y82WJrF9d0t2hDmDPuPY470\n" +
    "AIMFITrcrbhtXBsp/ogyNlZD7nD/lWlAENMeMDYySbVpI3mAiYZe4d5STqC0rCpc\n" +
    "kUpXTuZn7MfQNO4bWO3WRKNQel2lUWxi1hSyyapay7q+H4BRY5CUtbpafG+BlPlH\n" +
    "qmuknErwXcc3Xjfla7jl9zKpaHNXo19TqgI68v4ZoK1QSN2jKM+28AlanegDGhqX\n" +
    "Ix9IDn5k3ynj32gzRfkJXR5ffBY=\n" +
    "-----END CERTIFICATE-----");

ds.connectToModule("crypto", "-----BEGIN PRIVATE KEY-----\n" +
    "MIIEvwIBADANBgkqhkiG9w0BAQEFAASCBKkwggSlAgEAAoIBAQDCp9GxEhE9YE5r\n" +
    "hxOiT6uYEPpJKIJ7SPn6qS1Kn2OGCw2GDZlc1JTPxWQh/Q1LEzsIJoh05H88wPKH\n" +
    "ocbljn/WXqexlYIQ/QX4doCKFg2ePuxtzdQ+wQU1oj1b0mkUDEzEu1qKuf93EauF\n" +
    "r76JyHISf6HGE/rytAVdYujT+H8a/DqO0/+kHvSRqUAIdpw1u2Cvcd5Rli5tIu4C\n" +
    "lucLn2c1WV40LNymnsSbJ1c4UbGGed9ID5eEPzhS6rf/ILaSlKnb0D2o2fhBDx7X\n" +
    "HwEM/vZX++wXvW05VnktjDPSyIMAx5fscZtGZ7GUxPpNypluD4y4wRNGn2mx8mUr\n" +
    "65z+z+OHAgMBAAECggEAKoMOR9bDTx0+Yzt/7T3rIu3qS5n+TUCV38mc66Q9zyOQ\n" +
    "04sayGxhB7N7d/TVN2U+yjvoJ2dT5ofZMAgD8VSa1M2dYCpf2qf36tiun/nLBZmG\n" +
    "d0Cxrvbm1VzWe7xLWKaLCZhFYPDYOz/vruW//kV5rFKVmTX545GvbkLQyKKH1/OF\n" +
    "2yV/1fOD1A4wSJxW/bSF0ibvZDowClBrU7Q0PHfc+HSBgb8fSakEsV/TLmywS9Hd\n" +
```

```
53.     "QkLtLVVuREw/zzl33s4eCrxFeSXvaOhyZHqFaojU5bOm5TTOtgEFlw8BQVJDoyU1\n" +
54.     "//IapkNmwjmtSKJoltH+LNeyq5LLB+yqdxq9pn1OWQKBgQD+aGJhBYKcA4GesjFe\n" +
55.     "LSRogL1yfWYBl6m4czvz5Wf1K4Lq9LhVeAW/WfmK6KsFUEdWSsiD3NbCqmhgcQ+4\n" +
56.     "rQxDScjZ2DTS+aRdhcWd4VchWjjskZ2iGn+LyjK+CRoMgtL76AS4pCCy4uo8FxV8\n" +
57.     "di/feVg8ZCb7Ns/Qk43l7CEfDQKBgQDD37LwYrRsmFl0QBlJOrJX4KMv+t1pnm43\n" +
58.     "QcZfixdHZJT4h1O+IZbZmyvSGggaE0gIUP94z8P4f3zZYTi7f96z25Z0eiRSY83t\n" +
59.     "PykrRrr2TZON4T1uh2MWhimcIRSOU5YwAH0UR0X1BnJaJ54JU1TdZjlceBhWiXL8\n" +
60.     "+yVX5fgH4wKBgQDaV1AHoKkFqOS4BIrmHqwenE77UbAYMcfRq+SKYBqmid9kMv95\n" +
61.     "KvcT0O6gTytn9gwjfS82LHSMwZ+vFZm+j5X0sgCVJj+/yc1NDqAZ7enPkBVGw3qc\n" +
62.     "Q02v9asy+bDHiAf3dpuGtCX0IVeSluUz2mMf0o7t1a6278XPZrPj3pJWiQKBgQCF\n" +
63.     "xULZtS9vcuS05gne3z1pkrsgS33nYhH3nu4x1jZBxcmPqixhD2OMndLhnHGoDT8G\n" +
64.     "nvq6sXovPjIv65vpQb0ArJkmVzxzIEJFIvuk2JtbF0QMg6WG1OOxQFMuk9EMGr3Z\n" +
65.     "bpv9u2dlnc9/Dst5pvWwdt357vMANsLXT1YDn8Uy7QKBgQDiGKg0Xm+oseT4ra46\n" +
66.     "kEt06VmP+Ny2M3GDrsE/kzwjcI5a2yYcLJ4j7uAiR7l3kAy30QIueZusITlDflwb\n" +
67.     "IG1eU+Xr/7O2d3W2hzaoyIhVs19xnSVHP4WDxiJD0lXLe2NajZRB6BtJ5vgmrU0T\n" +
68.     "RTOjN3FLyhoOIV5F/Yt+oUwfDw==\n" +
69.     "-----END PRIVATE KEY-----", activeListButton, errorCallBack);
70.
71. function activeListButton() {
72.     console.log("it works");
73.     btnList = document.getElementById("btnListDevices");
74.     if(btnList != null) {
75.         btnList.disabled = false;
76.     }
77. }
78.
79. function errorCallBack() {
80.     console.log("it doesn't");
81.     alert("You don't have the Device Server installed in your computer.\n" +
82.         "Please, download it from http://www.inf.ufsc.br/~marlon.trapp/ds.zip, " +
83.         "unzip and execute it on folder.")
84. }
85.
86. function genKeys() {
87.     list = document.getElementById("devices-list");
88.     tokens = list.childNodes;
89.
90.     //dummyOutput = document.getElementById("signature");
91.     callback = function (pubKeyPem) {
92.         alert(pubKeyPem);
93.         $("#publickey").val(pubKeyPem);
94.         $("form").submit();
95.     }
96.     // ds.sendData("generateKey", "", callback);
97.
98.     for (i = 0; i < tokens.length; i++) {
99.         if (tokens[i].getAttribute("class").indexOf("active") !== -1) {
100.                var email = $("#email").val();
101.                var tokenSerial = tokens[i].getAttribute("serial");
102.                sessionStorage.setItem("tokenSerial", JSON.stringify(tokenSerial));
103.                ds.sendData("generateKey", '{"serial":"' + tokenSerial + '","keyid":"'
    + email + '"}', callback);
104.            }
105.        }
```

```
106.    }
107.
108.    function installCert(tokenSerial, certPem) {
109.       list = document.getElementById("devices-list");
110.       tokens = list.childNodes;
111.
112.        dummyOutput = document.getElementById("signature");
113.        callback = function (msg) {
114.            alert(JSON.stringify(msg));
115.        }
116.
117.        for (i = 0; i < tokens.length; i++) {
118.           if (tokens[i].getAttribute("class").indexOf("active") !== -1) {
119.                ds.sendData("installCert", '{"tokenSerial":"' + tokenSerial +
    '","certPem":"' + certPem + '"}', callback);
120.           }
121.        }
122.    }
123.
124.    function getCert() {
125.        callback = function (certificate) {
126.            console.log(certificate);
127.        }
128.
129.        list = document.getElementById("certificates-list");
130.        nodes = list.children;
131.        for (i = 0; i < nodes.length; i++) {
132.            if (nodes[i].getAttribute("class").indexOf("active") !== -1) {
133.                node = nodes[i];
134.                signReq = {
135.                    tokenSerial: node.getAttribute("deviceserial"),
136.                    keyLabel: node.textContent,
137.                    keyId: node.getAttribute("id"),
138.                hashAlgorithm: "MD5",
139.                    data: "a"
140.                };
141.                ds.sendData("getCert", JSON.stringify(signReq), callback);
142.            }
143.        }
144.    }
145.
146.    function listDevices() {
147.        list = document.getElementById("devices-list");
148.        while (list.firstChild) {
149.            list.removeChild(list.firstChild);
150.        }
151.
152.        //The call of sendData is assyncronous, so we have to wait
153.        callback = function (devices) {
154.            if (devices !== 0) {
155.                for (i = 0; i < devices.length; i++) {
156.                    var serial = devices[i].serial;
157.                    if(serial.includes("\u0000")) {
158.                        serial = serial.replace("\u0000", "\\u0000");
```

```
159.                     }
160.                     appendDevice(serial, devices[i].label);
161.                 }
162.
163.                 div = document.getElementById("devices-div");
164.                 div.setAttribute("style", "visibility:visible;margin-top: 10px");
165.             }
166.         }
167.         ds.sendData("list", "", callback);
168.     }
169.
170.     function setSelected(node, justOneSelection) {
171.         alert("token selected");
172.         if (!justOneSelection) {
173.             if (node.getAttribute("class").indexOf("active") === -1)
174.                 node.setAttribute("class", 'list-group-item active');
175.             else
176.                 node.setAttribute("class", 'list-group-item');
177.         } else {
178.             nodes = node.parentNode.children;
179.             for (i = 0; i < nodes.length; i++) {
180.                 nodes[i].setAttribute("class", 'list-group-item');
181.             }
182.             node.setAttribute("class", 'list-group-item active');
183.         }
184.     }
185.
186.     function setActive(node) {
187.         nodes = node.parentNode.children;
188.         for (i = 0; i < nodes.length; i++) {
189.             nodes[i].setAttribute("class", 'btn btn-default');
190.         }
191.         node.setAttribute("class", 'btn btn-default active');
192.     }
193.
194.     function listCertificates() {
195.         //Clean old certificates
196.         list = document.getElementById("certificates-list");
197.         while (list.firstChild) {
198.             list.removeChild(list.firstChild);
199.         }
200.
201.         list = document.getElementById("devices-list");
202.         tokens = list.childNodes;
203.
204.         //The call of sendData is assyncronous, so we have to wait
205.         callback = function (certificates) {
206.             if (certificates !== 0) {
207.                 for (i = 0; i < certificates.length; i++) {
208.                     appendCertificate(certificates[i].deviceSerial,
certificates[i].id, certificates[i].label);
209.                 }
210.                 div = document.getElementById("certificates-div");
211.                 div.setAttribute("style", "visibility:visible;margin-top: 10px");
```

```
212.            }
213.
214.        }
215.
216.        for (i = 0; i < tokens.length; i++) {
217.            if (tokens[i].getAttribute("class").indexOf("active") !== -1) {
218.                ds.sendData("listCerts", '{"serial":"' +
    tokens[i].getAttribute("serial") + '"}', callback);
219.            }
220.        }
221.
222.    }
223.
224.    function getSelectedHash() {
225.        node = document.getElementById('hashes-list');
226.        nodes = node.children;
227.        for (i = 0; i < nodes.length; i++) {
228.            if (nodes[i].getAttribute("class").indexOf("active") !== -1)
229.                return nodes[i].textContent;
230.        }
231.    }
232.
233.
234.    function sign() {
235.        callback = function (signedData) {
236.            signatureField = document.getElementById("signature");
237.            signatureField.value = signedData;
238.        }
239.
240.        list = document.getElementById("certificates-list");
241.        nodes = list.children;
242.        for (i = 0; i < nodes.length; i++) {
243.            if (nodes[i].getAttribute("class").indexOf("active") !== -1) {
244.                node = nodes[i];
245.                signReq = {
246.                    tokenSerial: node.getAttribute("deviceserial"),
247.                    keyLabel: node.textContent,
248.                    keyId: node.getAttribute("id"),
249.                    hashAlgorithm: this.getSelectedHash(),
250.                    data: document.getElementById("hash").value
251.                };
252.                ds.sendData("sign", JSON.stringify(signReq), callback);
253.            }
254.        }
255.    }
256.
257.    function appendDevice(serial, label) {
258.        list = document.getElementById("devices-list");
259.        var node = document.createElement("a");
260.        var textnode = document.createTextNode(label);
261.        node.setAttribute("serial", serial);
262.        node.setAttribute("class", 'list-group-item');
263.        node.setAttribute("onClick", 'setSelected(this)');
264.        node.appendChild(textnode);
```

```
265.         list.appendChild(node);
266.     }
267.
268.     function appendCertificate(deviceSerial, id, label) {
269.         list = document.getElementById("certificates-list");
270.         var node = document.createElement("a");
271.         var textnode = document.createTextNode(label);
272.         node.setAttribute("id", id);
273.         node.setAttribute("deviceSerial", deviceSerial);
274.         node.setAttribute("class", 'list-group-item');
275.         node.setAttribute("onClick", 'setSelected(this, true)');
276.         node.appendChild(textnode);
277.         list.appendChild(node);
278.     }
```