UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Thaís Bardini Idalino

UTILIZANDO DISPOSITIVOS MÓVEIS NA GERAÇÃO DE SENHAS DESCARTÁVEIS

Florianópolis

2012

Thaís Bardini Idalino

UTILIZANDO DISPOSITIVOS MÓVEIS NA GERAÇÃO DE SENHAS DESCARTÁVEIS

Trabalho de Conclusão de Curso submetido ao Curso de Ciências da Computação para a obtenção do Grau de Bacharel em Ciências da Computação.
Orientadora: Dayana Pierina Brustolin Spagnuelo
Coorientador: Ricardo Felipe Custódio

Florianópolis

2012

Dedico este trabalho à minha família, cuja educação e apoio me tornaram o que sou hoje. Dedico também aos meus amigos e namorado que sempre acreditaram, me ajudaram e torceram por mim.

AGRADECIMENTOS

Agradeço ao professor Ricardo Felipe Custódio e a toda a equipe do LabSEC, que tornaram possível a realização deste trabalho. À orientadora Dayana, pela ajuda e paciência. Ao Jean, Anderson, César e tantos outros que me acompanharam e ensinaram nesses mais de dois anos aqui no LabSEC.

A mente que se abre a uma nova idéia jamais voltará ao seu tamanho original.

Albert Einstein

RESUMO

A autenticação por meio de senhas é a mais utilizada atualmente em sistemas computacionais. Porém, diversos usuários escolhem senhas muito fáceis e não tomam o devido cuidado com elas e, no caso da Telemedicina, isso pode por em risco informações privadas e até a vida de pessoas. Este trabalho tem como objetivo a implantação de senhas descartáveis no Sistema Catarinense de Telemedicina e Telessaúde, no qual as senhas serão geradas diretamente do *smartphone* do usuário. Isto será feito por meio de uma versão melhorada do aplicativo Google Autenticador, provendo mais segurança à *master secret* e confiança no compartilhamento da mesma. Com isso, tem-se uma autenticação de múltiplos fatores no ambiente de Telemedicina, aliada a um dispositivo gerador de senhas descartáveis com nível de confiabilidade acrescido em 1, de acordo com o manual do *National Institute of Standards and Technology.*

Palavras-chave: Telemedicina, Senhas descartáveis, OTP, Google Autenticador, Autenticação

ABSTRACT

Authentication with passwords is the most used in computer systems. However, many users choose very weak passwords and don't take proper care of them and, in case of Telemedicine, this may compromise private information and even peoples lives . This work aims to deploy One-Time Passwords in the Sistema Catarinense de Telemedicina e Telessaúde, in which passwords are generated directly from the user's smartphone. This will be done through an improved version of the Google Authenticator app, providing more security to the master secret storage and trust in the sharing process. Thus, we achieve a multifactor authentication in the Telemedicine environment, combined with a password generator device which has the reliability level added in 1, according to the manual National Institute of Standards and Technology.

Keywords: Telemedicine, One-Time Passwords, OTP, Google Authenticator, Authentication

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

SUMÁRIO

1 INTRODUÇÃO

A utilização de senhas é algo que está presente no dia-a-dia da maioria das pessoas, indo desde senhas para efetuar transações bancárias, até senhas para acesso à sistemas mais simples ou redes sociais. Elas são a barreira mais utilizada em sistemas que precisam de alguma forma de autenticação e autorização de acesso, pelo fato de serem fáceis de aplicar tanto em nível de implementação quanto de usabilidade.

Este trabalho é parte de um projeto que está sendo desenvolvido no Laboratório de Segurança em Computação (LabSEC) [1] da Universidade Federal de Santa Catarina (UFSC)[2] em parceria com o Laboratório de Informática Médica e Telemedicina (LabTelemed) [3] financiado pela Financiadora de Estudos e Projetos (FINEP)[4].

No ambiente de Telemedicina em que será aplicado, a a autenticação de um médico perante o sistema tem uma importância ainda maior. Ao provar sua identidade, o profissional tem em mãos ferramentas capazes de emitir laudos e analisar exames de pacientes, auxiliando principalmente os que residem em lugares distantes. Tais ferramentas em mãos erradas, podem apresentar risco à saúde dos pacientes pela possibilidade de emissão de laudos falsos. Também é possível expor dados que seriam de interesse apenas dos pacientes e seus respectivos médicos.

O fato de utilizar um *login* e senha para se autenticar no sistema não garante que quem os está inserindo é realmente a pessoa autorizada ao acesso. Atualmente existem diversos tipos de ataques visando obter a senha de acesso de determinado usuário a determinado sistema. Por exemplo o *sniffing* na rede, visando obter a senha nos pacotes transmitidos na rede sem proteção e engenharia social, visando obter a senha do próprio usuário, o elo mais fraco no processo de autenticacão.

Além disso, pessoas tendem a escolher senhas fáceis e/ou utilizar a mesma senha para se autenticar em diferentes locais. Isso as torna vulneráveis, pois são fáceis de adivinhar e uma vez descobertas, podem comprometer uma quantidade imensurável de informações e dados.

Uma das maneiras de contornar esse problema é através da autenticação baseada em múltiplos fatores, que vem se tornando popular e que tem fácil implantação. Neste modelo, não apenas *login* e senha

[1] http://www.labsec.ufsc.br/
[2] http://ufsc.br/
[3] http://www.telemedicina.ufsc.br/joomla/
[4] http://www.finep.gov.br/

são utilizados, mas também a apresentação de um segundo fator de autenticação, que pode ser desde um objeto de posse única do usuário até características biométricas. Dentre os mecanismos de autenticação de múltiplos fatores está o de senhas descartáveis (do inglês *One-Time Passwords*, OTP), apresentado como alternativa principalmente em sistemas bancários, mas também nas contas de usuário do Google[5] e Facebook[6] .

Este trabalho propõe a implantação de senhas descartáveis na autenticação do Sistema Catarinense de Telemedicina e Telessaúde (STT), utilizando como base a implementação já existente do Google, o *Google Authenticator* (Google Inc., 2012). Porém, algumas modificações precisam ser feitas, pois apresenta problemas visíveis de segurança, tanto no compartilhamento da *master secret* quanto no seu armazenamento.

1.1 JUSTIFICATIVA

O STT tem por objetivo facilitar o acesso e emissão de exames e laudos por pacientes e médicos, fazendo uso da Internet. Dessa maneira o paciente pode fazer o exame em sua cidade e enviá-lo pela rede a um médico especializado. O médico, por sua vez, pode analisá-lo e emitir o laudo de qualquer lugar que esteja. Esta tecnologia foi desenvolvida pelo Grupo Cyclops da UFSC[7], e atualmente está disponível em quase 90% do estado (CYCLOPS, 2010).

O STT pode ser caracterizada como um sistema de caráter social, cujas principais vantagens estão na acessibilidade e facilidade de assistência da população em geral. Como apresentado em Magalhães e Santos (MAGALHAES; SANTOS, 2003), os problemas de segurança tornam-se ainda mais evidentes com o uso crescente das Tecnologias da Informação e Comunicação (TIC) e, dentre estes problemas, tem-se destaque o da autenticação. A Telemedicina pode ser considerada uma TIC e este problema é uma questão fundamental, já que o acesso indevido às informações pode ser de grande risco.

Na Telemedicina, este é um risco que não se pode correr. Se um indivíduo mal intencionado conseguir acesso à senhas de pacientes ou médicos, poderia desde visualizar informações particulares dos pacientes, como resultados de exames, até emitir laudos falsos, apresentando um risco à vida deles.

[5] https://www.google.com
[6] http://www.facebook.com/
[7] http://cyclops.telemedicina.ufsc.br/

Através da análise do código fonte, percebe-se que a segurança do sistema de Telemedicina existente está baseada em um *login* e senha que cada usuário escolheu e utiliza para se autenticar perante o sistema. De acordo com Haller e Atkinson (HALLER; ATKINSON, 1994), a utilização de senhas dessa maneira não é apropriada para garantir segurança na autenticação na Internet atualmente. Como citou Silva (SILVA, 2007), grande parte das deficiências na autenticação de sistemas que utilizam apenas *login* e senha como barreira de segurança estão no funcionamento da memória humana. Comenta ainda que se não houvesse a necessidade de memorizar as senhas, estas poderiam ser muito seguras.

Além do mais, para contornar o problema de esquecimento, os usuários costumam anotar senhas em papéis, escolher senhas muito fáceis ou até mesmo utilizar a mesma em vários sistemas.

Os dados contidos no sistema de Telemedicina são extremamente sensíveis, pois se referem a saúde de pessoas, e de maneira alguma podem correr riscos de comprometimento. Acredita-se que a proposta de implantação de senhas descartáveis neste pode aumentar significativamente a sua segurança e confiabilidade.

1.2 OBJETIVOS

Abaixo serão apresentados os objetivos deste trabalho.

1.2.1 Objetivo Geral

A proposta deste trabalho está no estudo de senhas descartáveis, investigando uma forma de aplicá-las no sistema de Telemedicina, em que o usuário as gere em seu próprio dispositivo móvel.

Pretende-se fazer um estudo minucioso de senhas descartáveis e algorítmos existentes para utilizá-los como base para este trabalho. Planeja-se estudar maneiras de aumentar a segurança do processo de configuração e geração destas senhas, de forma que inviabilize o roubo ou clonagem do gerador de senhas do usuário para ser utilizado em outro aparelho. A proposta é de enriquecer a forma de gerar as senhas descartáveis, de modo que se possa garantir que elas vieram única e exclusivamente do dispositivo móvel para o qual foi inicialmente desejado, e que apenas o dono estava de posse dele.

1.2.2 Objetivos Específicos

Pretende-se aumentar a segurança no compartilhamento da *master secret*, assim como o seu armazenamento cifrado no dispositivo com um segredo que apenas o usuário conheça, garantindo que somente ele seja capaz de gerar aquelas senhas descartáveis.

Objetivos de base

Abaixo são apresentados os objetivos de base deste trabalho.

- Cifrar a *master secret* com um segredo que apenas o usuário tenha conhecimento;
- Implementar as modificações propostas para a plataforma Android;
- Testar o aplicativo em diferentes dispositivos móveis da plataforma;
- Fazer a integração com um Módulo de Segurança Criptográfico (do inglês, *Hardware Security Module*, HSM) que será responsável pelo armazenamento das *master secrets* dos usuários e fará o papel de servidor de autenticação.

Objetivos avançados

Abaixo são apresentados os objetivos avançados deste trabalho.

- Implementar as modificações nas plataformas iOS e Blackberry, suportadas pelo *Google Authenticator*;
- Relacionar a senha gerada com características únicas do dispositivo móvel aonde está sendo gerada. Por exemplo a Identificação Internacional de Equipamento Móvel (do inglês, *International Mobile Equipment Identity*, Imei).

1.3 LIMITAÇÕES

Este trabalho é uma proposta de melhoria na autenticação do sistema de Telemedicina, porém, a integração deste sistema com as fer-

ramentas desenvolvidas não será abordada neste trabalho. Adicionalmente, ele entra apenas em questões de autenticação, sem se preocupar com os aspectos de autorização.

1.4 METODOLOGIA

O desenvolvimento deste trabalho se deu inicialmente com um levantamento bibliográfico e estudo do estado da arte. Estes estudos foram direcionados à questões de autenticação, tipos de autenticação de múltiplos fatores, aprofundamento no estudo de senhas descartáveis e os sistemas que as utilizam. Após isso, foi-se estudado o básico de programação em Android para, logo em seguida, entender a implementação do *Google Authenticator*, utilizado como base para este trabalho.

Ao identificar as deficiências do aplicativo *Google Authenticator*, desenvolveu-se um protocolo que visou melhorar a segurança do mesmo. O protocolo também envolve a integração com o servidor e *web service* de autenticação, ambos desenvolvidos no LabSEC. O primeiro se refere a um HSM, que armazenará os dados do usuário necessários para a autenticação através de OTP. Já o segundo é um *web service* que ficará entre o sistema de Telemedicina e o HSM, interfaceando a comunicação entre eles.

A implementação *mobile* deste protocolo foi testada com testes unitários e testes no próprio dispositivo móvel. Os testes unitários foram utilizados para avaliar a corretude das funções fundamentais, já os outros testes foram utilizados para verificar a funcionalidade do aplicativo como um todo, da integração do que foi desenvolvido com o que já existia. Por fim, foram feitos testes de integração com o *web service* e o servidor de autenticação através da simulação do Sistema de Telemedicina.

1.5 ESTADO DA ARTE

De acordo com Haller et al (HALLER et al., 1998), a ideia por trás do OTP foi inicialmente proposta por Leslie Lamport em seu artigo de 1981 (LAMPORT, 1981), que também inspirou a criação do sistema de S/KEY (HALLER, 1995), no qual o OTP foi derivado.

Como citado por Cheng (CHENG, 2011), a implementação do OTP atualmente pode ser feita de duas maneiras: em software ou hardware. Ele afirma que a implementação em hardware tem a vantagem

de ser portátil, porém, não pode ter sua ativação e uso imediatos, pois o usuário necessita adquirir o token e isso pode levar um certo tempo e ser caro, assim como a sua manutenção. Já a em software pode ser ativada até mesmo pela internet, e seu desenvolvimento pode ser mais rápido e fácil que o em hardware, mas a *master secret* é armazenada em um computador ou aparelho móvel, pondo em risco sua segurança, pois pode ser secretamente copiada.

Outro inconveniente da autenticação por OTP é de que cada servidor de autenticação que o usuário utiliza necessita de um dispositivo gerador das senhas. Se ele quiser utilizar este tipo de autenticação em diferentes serviços, ele vai precisar de um gerador de OTP para cada serviço (CHENG, 2011).

Há diversas implementações com propósito bancário, como, por exemplo, o Bradesco, que utiliza um tipo de OTP impresso em papel, contendo 70 senhas descartáveis que devem ser apresentadas juntamente com a senha do usuário para autenticação (Banco Bradesco S.A., 2012). Outro exemplo é o *RSA SecurID Authenticator* (RSA Security Inc, 2010), que utiliza um *token* que pode ser tanto por hardware (como pequenos dispositivos que podem ser carregados no bolso) quanto por software (disponíveis para celulares, para Windows e Mac OSX). Este *token* possui uma senha semente e gera o OTP através de cálculos matemáticos baseados nessa semente e no horário de geração da senha, e a cada 60 segundos uma nova senha é gerada. Os relógios do *token* e do servidor de autenticação devem estar sincronizados, para que eles possam gerar as mesmas senhas. Por fim, para a efetuar a autenticação, o usuário precisa fornecer o OTP juntamente com sua senha e *login*.

Um aplicativo muito utilizado atualmente é o Google Authenticator (Google Inc., 2012). Ele está disponível para as plataformas Android, iOS e Blackberry e é utilizada para uma autenticação mais forte nas contas da empresa. Nesse aplicativo, para que o cliente e o servidor de autenticação possam gerar as mesmas senhas, e de maneira independente, eles precisam passar por um processo de *setup*, onde fazem um acordo de uma semente (conhecida como *master secret*), e a partir dela gerar o os OTPs. Para utilizar este aplicativo, o usuário faz o *download* do mesmo em seu celular e seleciona a opção de cadastrar uma conta. Este cadastro pode ser através da entrada dos seus dados e a *master secret* manualmente, ou obtendo a última através de um QR Code. O acordo desta semente é feito pelo browser, e o usuário a recebe direto na tela do computador, em texto claro caso tenha sido escolhida a primeira opção, e em QR Code para a segunda. Após a configuração da conta no aplicativo, esta *master secret* é salva em claro no banco de

dados.

1.6 ESTRUTURA DO TRABALHO

No capítulo 2 são apresentados conceitos básicos relacionados à autenticação, como os seus fatores, autenticação de múltiplos fatores, senhas descartáveis e o servidor de autenticação utilizado neste trabalho.

No capítulo 3 é abordada a proposta desse trabalho de uma maneira mais geral, apontando as melhorias feitas no Google Autenticador para que fosse possível garantir uma confiabilidade maior no aplicativo gerador das senhas.

No capítulo 4 é detalhada a integração entre o aplicativo do usuário e servidor de autenticação com o ambiente de Telemedicina.

Nos capítulos 5 e 6 são detalhas as tecnologias utilizadas e a implementação, seguidos pelo capítulo 7 com uma análise de confiabilidade do produto final adquirido.

Por fim, no capítulo 8 são apresentadas as considerações finais obtidas através da realização deste trabalho e sugestões de trabalhos futuros.

Nota ao leitor: Parte deste trabalho resultou em um artigo publicado no XII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSeg 2012), intitulado de Senhas descartáveis em dispositivos móveis para ambientes de Telemedicina (IDALINO; SPAGNUELO, 2012). O mesmo recebeu menção honrosa no VI Workshop de Trabalhos de Iniciação Científica e de Graduação do SBSeg 2012.

2 REVISÃO BIBLIOGRÁFICA

2.1 AUTENTICAÇÃO

Segundo Nakamura (NAKAMURA, 2007), o acesso à sistemas e recursos depende da verificação do usuário e essa verificação deve permitir que apenas o usuário legítimo tenha acesso aos recursos requeridos. Dessa forma, a identificação e autenticação são as primeiras linhas de defesa da maioria dos sistemas, prevenindo que pessoas não autorizadas tenham acesso a eles (GUTTMAN; ROBACK, 1995). Portanto, autenticação é, um processo que determina se um usuário é quem diz ser e diferentes sistemas requerem diferentes formas de autenticação (Huntington Ventures Ltd., 2006).

Segundo o FFIEC (Federal Financial Institutions Examination Council, 2005), a autenticação depende apenas de que um usuário forneça uma identificação válida seguida de uma ou mais credenciais de autenticação (fatores), que caso corretos, provam sua identidade perante o sistema. Ainda de acordo com o FFIEC (Federal Financial Institutions Examination Council, 2005), um fator é uma informação secreta ou única, ligada à um usuário.

2.1.1 Fatores de autenticação

Existem três categorias de fatores de autenticação que podem ser usados separadamente ou em conjunto: com base no que o usuário sabe, no que ele possui e no que ele é (GUTTMAN; ROBACK, 1995).

Com base no que o usuário sabe

De acordo com Nakamura (NAKAMURA, 2007), este fator de autenticação é baseado em algum conhecimento do usuário, no qual a técnica mais convencional é a utilização de senhas. Segundo Guttman e Roback (GUTTMAN; ROBACK, 1995), o sistema compara a senha dada pelo usuário com a previamente combinada entre eles, se forem iguais, ele é então autenticado.

As senhas tem sido utilizadas por muito tempo e estão presentes na maioria dos sistemas por serem de fácil implementação e integração. Desta forma, o uso de senha é natural para os usuários, pois já estão familiarizados com este tipo de método.

Entretanto, como apresentado por Kessler (KESSLER, 1996), a senha é considerada uma maneira fraca de autenticação. Isso deve-se a diversos fatores, sendo que o principal deles é o fato de sua escolha e/ou gerenciamento ser diretamente dependente do usuário. A maioria dos usuários não dá o devido cuidado à sua senha, nem as escolhe sabiamente, podendo pôr em risco a segurança das informações e dados presentes no sistema. De acordo com Guttman e Roback (GUTTMAN; ROBACK, 1995), há muitas maneiras de a autenticação por senha ser quebrada, e dentre elas estão:

- Adivinhar ou encontrar senhas: Grande parte das deficiências nos sistemas que exigem autenticação através de senhas estão nas condições da memória humana em memorizar senhas fortes o suficiente para não serem adivinhadas. Um usuário pode possuir ou necessitar de diversas senhas, dessa maneira, é comum o uso da mesma senha para diferentes contas, ou até mesmo anotá-las ou escolher as mais fáceis (SILVA, 2007).

- Engenharia social: O próprio usuário pode ser levado a compartilhar sua senha com um colega por meio da técnica de engenharia social, que é feita através da exploração da confiança das pessoas.

- Monitoramento eletrônico: Senhas podem ser capturadas ao serem transmitidas pela rede. Apesar de cifradas, podem tanto serem descobertas com softwares especializados quanto serem utilizadas cifradas, já que o sistema do outro lado não saberá diferenciar se a senha cifrada veio do usuário ou de alguém mal intencionado.

Com base no que o usuário possui

Esse fator é fundamentado em dispositivos que o usuário possui. Estes dispositivos podem ser divididos em dispositivos de memória e dispositivos inteligentes (NAKAMURA, 2007).

- **Dispositivos de memória:** A função desses dispositivos está apenas em armazenar dados, não em processá-los, e quase sempre são utilizados em conjunto com senhas. Por exemplo, os serviços bancários que necessitam de um cartão e uma senha(NAKAMURA, 2007). De acordo com Guttman e Roback (GUTTMAN; ROBACK, 1995), os cartões de memória utilizados em conjunto com senhas são muito mais seguros do que o uso de senhas por si só e também

não são caros de produzir. Entretanto, requerem um hardware de leitura especializado, aumentando significativamente o custo de uso. Além disso, quando perdidos ou falsificados, apresentam mais dificuldades na recuperação do que simples senhas.

- **Dispositivos inteligentes:** Esses dispositivos são similares aos de memória, com a diferença de que são capazes de processar algumas informações devido a circuitos integrados (NAKAMURA, 2007). Podem ser divididos em três categorias, levando em consideração suas características físicas (em formato de cartão ou outros pequenos objetos portáteis), a interface (manual ou eletrônica) e o protocolo (troca de senhas estáticas, geração dinâmica de senhas ou desafio-resposta) (GUTTMAN; ROBACK, 1995). Pelo fato de apresentar a possibilidade de geração dinâmica de senhas, estes dispositivos podem amenizar o problema de monitoramento de senhas na rede, porém, há o transtorno em relação à perda do dispositivo. Além disso, a necessidade de um hardware de leitura especializado ainda pode representar uma barreira para o processo de autenticação (GUTTMAN; ROBACK, 1995).

Com base no que o usuário é

Biometria no contexto de autenticação refere-se à utilizar características do usuário para identificá-lo perante o sistema (MAGALHAES; SANTOS, 2003). Podem ainda ser separadas em características fisiológicas (como impressões digitais), padrões de retina e características comportamentais (como a voz).

De acordo com Nakamura (NAKAMURA, 2007), a autenticação é feita com base na coleta dos dados do usuário através de dispositivos biométricos. Esses dados são então enviados pela rede, de forma segura, até o servidor de autenticação, onde os dados serão comparados.

Ainda com base em Nakamura (NAKAMURA, 2007), grande parte dos problemas na utilização da autenticação com base no que o usuário sabe e possui giram em torno do esquecimento das senhas, da perda ou falsificação dos cartões e tokens. A biometria, por outro lado, não apresenta nenhum desses problemas, pois trata de características do próprio usuário. Porém, como citou Nakamura (NAKAMURA, 2007), dentre as várias características utilizadas na autenticação dos usuários, apenas duas delas podem ser consideradas únicas: a íris do olho e a impressão digital. O reconhecimento de face, por exemplo, pode ser burlado apenas apresentando uma foto do usuário em questão para a

câmera responsável por capturar a imagem.

Assim como nos outros fatores, este também apresenta um elevado custo por causa dos dispositivos necessários para a captura dessas características. Outro problema é o receio do usuário, tanto em relação ao laser que faz a leitura da retina prejudicar sua saúde, quanto à segurança de seus dados armazenados no servidor de autenticação. Estes receios na verdade existem mais por falta de informação dos usuários, já que este risco não é encontrado na maioria dos sistemas.

2.1.2 Senhas dinâmicas

Leslie Lamport em seu trabalho em 1981 (LAMPORT, 1981) propôs um novo sistema de autenticação através de senhas dinâmicas. Inicialmente ele apontou as seguintes vulnerabilidades no uso das senhas da maneira usual:

1. Possibilidade de acesso às informações armazenadas no próprio sistema, como acesso à arquivos de senhas.

2. Interceptando a comunicação do usuário com o sistema.

3. Divulgação da senha por parte do usuário, ou escolha de uma senha fraca.

Com base nestes problemas, ele sugeriu soluções para os dois primeiros casos. Para resolver o problema de acesso aos arquivos de senhas, basta, ao invés de armazenar uma senha x no servidor, armazenar *F(x)*, sendo F uma função cuja inversa é impraticável de calcular. Dessa maneira, o atacante que tentar ler não será capaz de descobrir x, o que é garantido pela função. Já para autenticar, basta o usuário enviar sua senha x ao sistema e este aplica *F(x)* para então compará-la com o que ele já possui e assim, caso seja a senha correta, autenticar o usuário.

Em adição à técnica anterior, para minimizar o impacto da segunda vulnerabilidade basta, ao invés de o usuário fornecer sempre uma mesma senha, apresentar uma sequência de senhas diferentes a cada autenticação. O usuário escolheria uma "semente"inicial x, e computaria uma série de senhas através de F(x) como mostrado abaixo:

$$Fn - 1(x), ..F(F(x)), F(x), x$$

O processo de autenticação seria realizado da seguinte maneira:

- O usuário enviaria secretamente *Fn(x)* para o servidor;

- Para efetuar a primeira autenticação no sistema, o usuário envia *Fn-1(x)* para o servidor de autenticação pela rede;

- O servidor aplica a função F no que ele recebeu e verifica se o resultado é igual ao valor que ele possui, já que $F(Fn\text{-}1(x)) = Fn(x)$.

- Caso a autenticação seja executada com sucesso, o servidor armazena a senha recém enviada pelo usuário para efetuar as próximas autenticações.

Novamente, pelas propriedades da função F, um atacante não poderia descobrir o x nem as senhas que serão utilizadas posteriormente, garantindo assim a segurança contra um ataque pelo monitoramento da rede do usuário. De acordo com (CHENG, 2011), o conceito de Lamport facilitou o nascimento de diversos novos tipos de autenticação, e dentre eles está o de autenticação de múltiplos fatores.

2.1.3 Autenticação de múltiplos fatores

De acordo com o FFIEC (Federal Financial Institutions Examination Council, 2005), a autenticação de um fator (do inglês, *single-factor authentication*) ocorre quando se utiliza apenas um dos fatores apresentados anteriormente para verificar a identidade de um usuário, sendo mais comum o uso de senha. Já os métodos de autenticação que utilizam mais de um dos fatores são conhecidos como autenticação de múltiplos fatores (do inglês, *multifactor authentication*). Ele afirma também que este último método é mais difícil de ser comprometido, podendo ser utilizado para prover um nível maior de segurança.

O FFIEC (Federal Financial Institutions Examination Council, 2005) ainda cita que a autenticação de dois fatores (do inglês, *Two-Factor Authentication*, 2FA) é constantemente vista em operações bancárias. Num sistema de atendimento automático por exemplo, o usuário apresenta um cartão do banco (o que o usuário possui) e uma senha (o que o usuário sabe) para poder realizar qualquer operação.

De acordo com Cheng (CHENG, 2011), apenas o usuário tem o seu segundo fator de autenticação, e um intruso deve ser incapaz de obtê-lo facilmente. A autenticação de múltiplos fatores apresenta uma maior segurança em comparação com a utilização básica de *login* e senha,

sendo assim, é recomendável utilizá-la em sistemas que requerem um alto nível de segurança.

2.2 ONE-TIME PASSWORDS

Senhas descartáveis (ou do inglês, *One-Time Passwords*, OTP) são senhas que podem ser utilizadas como um segundo fator de autenticação e estão em constante crescimento de utilização. Nesse tipo de autenticação, cada senha é válida apenas uma vez e é utilizada em conjunto com o *login* e a senha estática do usuário. Tais senhas são geradas independentemente pelo cliente e servidor de autenticação, no qual ambos utilizam uma mesma senha semente (ou *master secret*) acordada previamente (CHENG, 2011).

Um grupo de empresas no início dos anos 2000 decidiu criar a *Initiative for Open AuTHentication* (OATH) (Initiative for Open AuTHentication, 2011), cujo objetivo era o de desenvolvimento de uma arquitetura de referência de autenticação forte, utilizando padrões de segurança já existentes. Em dezembro de 2005 a OATH apresentou na RFC4226 (M'RAIHI et al., 2005) o OTP baseado em contador (*HMAC-Based One Time Password*, HOTP). Já o OTP baseado em tempo (*Time-based One Time Password*, TOTP) foi proposto em um *Internet Engineering Task Force* (IETF) (M'RAIHI et al., 2011) em setembro de 2010, tendo sua última atualização em maio de 2011.

2.2.1 HMAC

De acordo com a RFC2104 (KRAWCZYK; BELLARE; CANETTI, 1997), os mecanismos que provêm integridade de mensagens trocadas entre duas partes, sendo estes baseados em uma senha compartilhada, são conhecidos como código de autenticação de mensagem (do inglês, *Message Authentication Code*, MAC). HMAC é, portanto, um algoritmo de código de autenticação de mensagem combinado com qualquer algoritmo de *hash*.

O propósito do HMAC é de que a mensagem trocada entre as partes seja acompanhada de uma verificação, que consiste no *hash* de operações lógicas feitas em cima da mensagem e da senha compartilhada, como pode ser visto na Figura 1. Considerando que apenas os dois lados possuam a senha, ninguém poderá realizar uma alteração válida na mensagem.

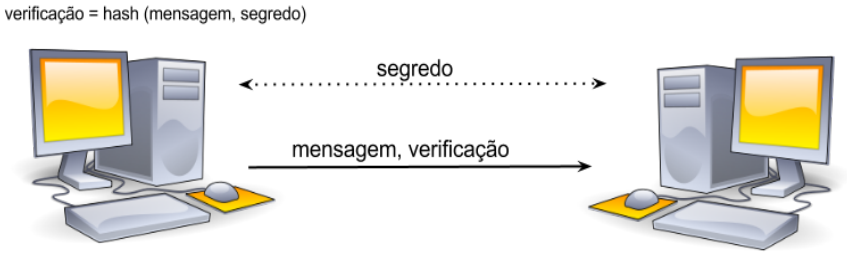

Figura 1: Envio de mensagem usando HMAC

2.2.2 HOTP

De acordo com M' Raihi (M'RAIHI et al., 2005), HOTP é um algoritmo de geração de senhas descartáveis baseado em HMAC que utiliza a função de *hash* SHA-1 e um contador ou sequência para a computação da senha descartável. Este algoritmo é apresentado na fórmula 2.1, onde **K** é um segredo compartilhado entre as partes, também conhecido como *master secret*, **C** representa o contador ou sequência e ***Truncate*** é uma função que vai reduzir aleatoriamente o resultado da função de HMAC em uma senha OTP de seis dígitos.

$$HOTP(K, C) = Truncate(HMAC\_SHA1(K, C)) \qquad (2.1)$$

2.2.2.1 Requisitos do algoritmo de HOTP

A RFC4226 (M'RAIHI et al., 2005) apresenta alguns requisitos para o algoritmo de HOTP:

- Necessita ser baseado em um contador ou sequência;
- Deve haver a possibilidade de utilização tanto em tokens quanto em dispositivos mais sofisticados. Por este motivo deve ser econômico no consumo de energia, na quantidade de botões necessários para sua utilização e no tamanho dos dados a serem mostrados na tela. Adicionalmente é recomendado que tenha valores numéricos, para que a senha possa ser utilizada em aparelhos mais restritivos;

- A senha gerada deve ter um tamanho razoável, que seja fácil de o usuário ler e transcrever. Porém, não é recomendado ter menos de seis dígitos;

- O mecanismo de ressincronização do contador deve ser intuitivo;

- A *master secret* utilizada no algoritmo deve ter no mínimo 128 bits, mas a recomendação é que seja de 160 bits.

2.2.2.2 Validação da senha descartável

Para o processo de validação de senhas HOTP, ambos os lados devem ter feito um acordo da *master secret* previamente. O servidor faz o mesmo processo de geração de OTP que o cliente ao receber uma senha, com a diferença de que ele incrementa o contador apenas depois de uma autenticação realizada com sucesso. O cliente, por sua vez, incrementa o contador a cada senha descartável gerada.

Pelo fato de não haver garantia de sincronização do contador entre os geradores envolvidos, o servidor possiu um protocolo de ressincronização. Esse protocolo possui um parâmetro **S** que representa o tamanho da sequência de senhas (janela) que serão geradas para ressincronização. Dessa maneira, ao receber uma senha inicialmente inválida, o servidor calcula as **S** próximas senhas HOTP e analisa se alguma delas coincide com a recebida. Opcionalmente, o servidor pode requerer ao cliente uma sequência de senhas HOTP para fazer a ressincronização, já que o forjamento de valores consecutivos é extremamente difícil.

2.2.2.3 Considerações de segurança

A RFC4226 (M'RAIHI et al., 2005) afirma que o melhor ataque contra o algoritmo de HOTP é a força bruta. Apresenta também que um atacante que conseguiu obter uma sequência de senhas válidas, com o intuito de construir uma função para gerar as próximas, tem tantas chances de conseguir quanto à tentativa de geração randômica de senhas.

A análise de segurança pode ser aproximada pela função:

$$Sec = \frac{s.v}{10^{Digit}} \tag{2.2}$$

Onde **Sec** é a probabilidade de sucesso de um atacante, **s** é o

tamanho da janela de ressincronização, **v** é o número de tentativas e ***Digit*** é a quantidade de dígitos da senha HOTP. Pode-se alterar os parâmetros para obter uma maior segurança, porém, não podendo esquecer dos requisitos de usabilidade.

2.2.3 TOTP

Este algoritmo de senhas descartáveis é descrito na RFC6238 (M'RAIHI et al., 2011) e trata-se de uma extensão do HOTP, com a diferença de que ao invés de usar um contador na geração da senha, é usado um fator baseado em tempo.

2.2.3.1 Requisitos do algoritmo de TOTP

A RFC6238 (M'RAIHI et al., 2011) apresenta algumas restrições para algoritmos de TOTP, dentre elas estão a de que tanto o provedor (o usuário que gera a senha e quer se autenticar) quanto o verificador (servidor de autenticação) devem ser capazes de obter o *Unix timestamp*, que nada mais é que o tempo em segundos desde janeiro de 1970. Além disso, devem usar como base o HOTP, assim como ter um compartilhamento da *master secret* e o mesmo valor de *timestep* entre provedor e verificador. Deveria haver uma única *master secret* para cada provedor, que deve ser gerada randomicamente ou utilizando algoritmos de derivação de chaves. Além disso, deve ser armazenada de forma seguro e protegida contra acesso e uso não autorizados.

2.2.3.2 Descrição do algoritmo

O algoritmo do TOTP é simplesmente representado por:

$$TOTP = HOTP(K, T) \tag{2.3}$$

Onde **K** é a *master secret* e **T** é um número inteiro que repesenta a quantidade de *timesteps* desde T0 até o *Unix Time* atual, calculado através de:

$$T = \frac{(CurrentUnixTime - T0)}{X} \tag{2.4}$$

Onde **X** representa o *timestep* adotado entre provedor e verifi-

cador, usualmente seu valor é de 30 segundos. Já **T0** é o *Unix Time* inicial, de onde serão contados os *timesteps*. Por padrão esse valor é 0.

A melhor representação dos parâmetros utilizados pode ser observada na Figura 2. O **X** tem por valor 30 segundos, e o **T** recebe o valor 3, já que o *Unix Time* atual se encontra no terceiro *timestep* desde o tempo inicial.

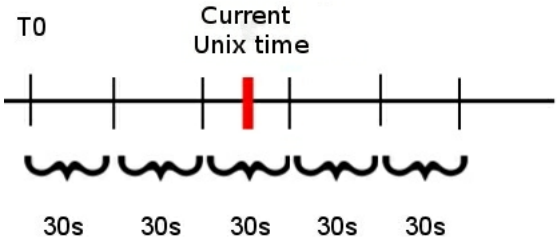

Figura 2: Calculo de T em função de *Timestep*

Como o algoritmo do TOTP é uma adaptação do HOTP, a segurança do primeiro depende do segundo. Assim como no HOTP, o método mais eficaz para ataques é a força bruta.

2.2.3.3 Validação da senha descartável e tamanho do *timestep*

Tanto o provedor quanto o verificador calculam o OTP utilizando o *timestep* atual. Porém, devido aos problemas de latência da rede ou sincronização dos relógios, o fator T (calculado na Equação 2.4) pode divergir entre provedor e verificador. Com o intuito de contornar esse problema, o verificador deve assumir uma política de aceitação, onde, ao receber um OTP, ele não o verifique gerando uma senha apenas no *timestep* de recebimento, mas também em anteriores. A recomendação é de fazer a verificação apenas com o anterior, pois mais do que isso pode gerar brechas para ataques.

Outro ponto a ser discutido é o tamanho do *timestep*, que anteriormente foi citado como valor padrão de 30 segundos. Seu tamanho indica a validade da senha OTP gerada. A cada novo *timestep*, uma nova senha é gerada e a anterior se torna inválida. Um *timestep* grande pode ajudar a não ocorrer o problema de latência na rede, mas pode prover um tempo maior para que um atacante roube e use essa senha OTP. Porém, também não pode ser muito pequeno, pois compromete-

ria a usabilidade. Ao disponibilizar um tempo pequeno, talvez este não seja suficiente para que o usuário leia e digite o valor do OTP. Por estes motivos a RFC6238 (M'RAIHI et al., 2011) recomendou 30 segundos como um valor ideal.

2.2.3.4 Ressincronização

Como o algoritmo utiliza o tempo como parâmetro, junto com ele pode surgir o problema de os relógios do provedor e verificador não estarem sincronizados. Este também pode ser contornado com políticas de aceitação, no qual o provedor calcularia OTPs com uma quantidade fixa de *timesteps* anteriores e posteriores, verificando a senha recebida com todos estes OTPs antes de invalidar a autenticação. Ao perceber uma dessincronização, o verificador se ressincroniza para evitar problemas com futuras senhas daquele provedor.

2.3 ASI-HSM

Um Módulo de Segurança Criptográfico (do inglês, *Hardware Security Module*, HSM) é um dispositivo composto por um hardware, software e *firmware*. De acordo com Sutil(SUTIL, 2011), este dispositivo possui a finalidade de prover proteção e gerência ao ciclo de vida de chaves criptográficas.

O ASI-HSM, de acordo com Souza (SOUZA, 2008), é um módulo de segurança criptográfica com tecnologia brasileira, de código aberto e de menor custo. Foi patrocinado pelo GT ICPEdu II e projetado pela empresa brasileira Kryptus[1] em parceria com a Rede Nacional de Ensino e Pesquisa (RNP)[2] . O software responsável pela gerência do ASI-HSM, o OpenHSMd[3], foi desenvolvido pelo LabSEC.

De acordo com Souza (SOUZA, 2008), o ASI-HSM possui um perímetro criptográfico composto por uma Unidade de Segurança e uma Unidade Gestora. Ele apresenta que a Unidade de Segurança conta com sensores de tensão, temperatura, luminosidade e detecção de intrusão física, cujo objetivo é de detectar qualquer tentativa de violação. Já a Unidade Gestora hospeda o *firmware* responsável pela gerência do ciclo de vida das chaves criptográficas, as ferramentas e

[1] http://kryptus.com.br
[2] http://www.rnp.br
[3] https://projetos.labsec.ufsc.br/openhsmd

bibliotecas necessárias, assim como as próprias chaves.

3 PROPOSTA

De acordo com Cheng (CHENG, 2011), geradores de senhas descartáveis em software armazenam a *master secret* no próprio dispositivo, e se copiada por um indivíduo malicioso, pode comprometer a segurança do sistema. É evidente que o processo de *setup* da *master secret* pelo aplicativo *Google Autenticador* apresenta vulnerabilidades em relação à segurança da mesma. Um usuário mal intencionado pode obter a *master secret* simplesmente observando a tela do usuário, ou até mesmo, de alguma maneira, ter acesso ao celular e extraí-la do banco de dados, tendo em vista que além de armazená-la no *smartphone* do usuário, o mesmo é feito em claro no seu banco de dados. A *master secret* pode ser considerada a base de todo o processo de geração das senhas descartáveis, e apenas o servidor e o usuário devem detê-la, pois qualquer um que possuí-la pode ser capaz de gerar as mesmas senhas e se passar pelo usuário no processo de autenticação.

Tendo em vista estes problemas, sugere-se que a *master secret* seja armazenada de forma cifrada com um PIN (*Personal Identification Number*), sendo este uma senha que apenas o usuário conheça. Dessa maneira, além de ter acesso ao celular do usuário e extrair a *master secret*, o indivíduo mal intencionado deveria também ter conhecimento do PIN para poder decifrar e utilizar a *master secret*. Para evitar que o PIN seja extraído do banco de dados, este não fica armazenado no *smartphone*. Sua validação é dada unicamente pelo sucesso ou fracasso na decifragem da *master secret*, quando o usuário o fornece na aplicação.

CONTEÚDO REMOVIDO

3.1 INTEGRAÇÃO

Em modelos de autenticação baseados em OTP, proteger a *master secret* contra roubos não significa somente proteger o gerador, pois existe uma cópia da mesma no servidor de autenticação. Além disto, é um requisito comum a todos os modelos de autenticação que a máquina responsável pelo armazenamento das credenciais dos usuários seja confiável. Desta forma este trabalho também propõe a troca do servidor de autenticação por um Módulo de Segurança Criptográfico (do inglês, *Hardware Security Module*, HSM), para torná-lo mais resistente a ataques. Propõe também o uso de um *framework* de autenticação

desenvolvido no LabSEC, para que a autenticação de múltiplos fatores seja facilmente aplicável em qualquer ambiente, porém aplicado neste trabalho em ambientes de Telemedicina. Este, além de fornecer uma abstração com a camada do HSM, provê fatores adicionais de autenticação, configurações, entre outros.

3.1.1 Servidor de autenticação

HSMs são dispositivos usados principalmente em aplicações que requerem um maior rigor no armazenamento e processamento de informações sensíveis, como por exemplo em aplicações militares e bancárias. Normalmente são usados para o armazenamento de chaves criptográficas, processamento de autenticação de usuários, execução de funções criptográficas, entre outros (SOUZA, 2008).

Para o desenvolvimento deste trabalho, foi utilizado o ASI-HSM que, por se tratar de um HSM, já possui barreiras contra os principais ataques sendo, portanto, menos vulnerável se comparado a um servidor simples. O ASI-HSM utiliza o OpenHSMd, que é um *firmware* voltado à gerência do ciclo de vida de chaves criptográficas, o qual precisaria ser completamente modificado para atender às necessidades do modelo proposto. Porém uma modificação no OpenHSMd não faria sentido, uma vez que seu propósito inicial pouco tem a ver com autenticações baseadas em OTP. Além do mais, isso aumentaria a interface de interação com o HSM, aumentando assim a vulnerabilidade do mesmo, uma vez que se aumenta a variedade de ataques aplicáveis.

A melhor alternativa encontrada foi a criação de um novo *firmware*, específico para o modelo. Este possui interface somente para ações básicas, como a criação de usuários, da *master secret* e a autenticação.

Para a criação de um usuário, o mesmo solicita o processo ao Sistema de Telemedicina, que se comunica com o HSM informando um identificador único do usuário e o tipo de OTP escolhido por ele (passos 1 e 2 da Figura **??**). O HSM por sua vez retorna a master secret, transformada em qr code pelo sistema de Telemedicina e lido pelo usuário.

IMAGEM E CONTEÚDO REMOVIDOS

Para poder visualizar seu OTP, o usuário precisa fornecer o PIN escolhido na hora da criação de seu usuário. Dessa forma, a sua *master secret* é decifrada do banco de dados e utilizada para calcular o OTP (passos 1 e 2 da Figura 3). Já a autenticação do usuários propriamente

dita se dá pelo fornecimento do identificador do mesmo (um *login*, por exemplo), da senha que ele já utilizava no sistema e do seu OTP recém calculado. O sistema de Telemedicina verifica o identificador e senha do usuário, como já faz hoje em dia, e passa a tarefa de verificar o OTP para o HSM (passos 3, 4 e 5 da Figura 3). Este, por sua vez, gera um OTP de acordo com o tipo definido na hora da criação do usuário e verifica se coincide com o fornecido. Em caso positivo, o usuário se autentica com sucesso no sistema. Se qualquer uma das credenciais fornecidas não estiver de acordo com a verificação, o usuário não se autenticará no sistema (passos 5 e 6 da Figura 3).

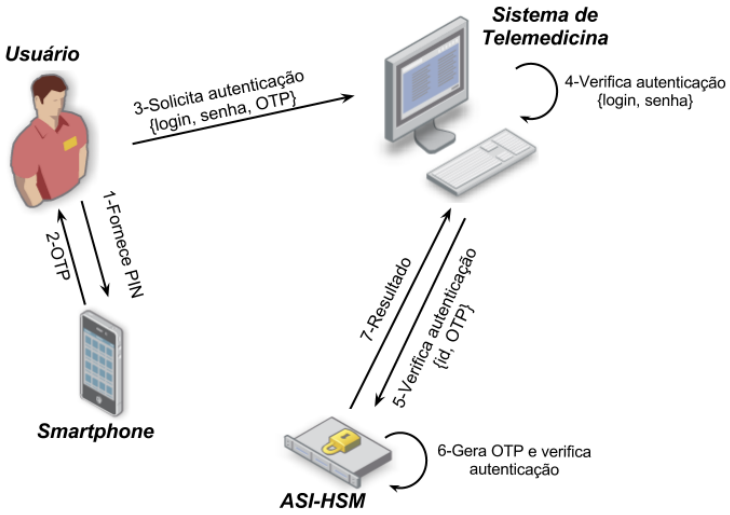

Figura 3: Autenticação de Usuários

O fato de todas as comunicações serem feitas através de um túnel seguro (*Secure Sockets Layer*, SSL) e a interface ser bastante restrita aumenta a dificuldade de um atacante obter sucesso em uma de suas tentativas. Além disto, se ainda assim um atacante conseguir penetrar o HSM, as *master secrets* estão cifradas pela chave do HSM, fazendo com que seja necessário obter-se também a chave privada deste, do contrário somente se obteria texto ilegível.

3.1.2 Framework de autenticação

Para que seja possível a implantação de um sistema mais seguro de autenticação em ambientes de Telemedicina, foi desenvolvido um serviço de autenticação que serve como uma interface entre o HSM e o sistema. Este serviço funciona como um *web service* que tem o propósito de ser genérico, independente de linguagem e abstrair a camada do HSM. O mesmo provê diversos métodos de autenticação que podem ser utilizados como um fator adicional, e dentre estes métodos, está o de senhas descartáveis proposto neste trabalho. Devido ao ambiente em que está sendo proposta a sua aplicação, o *framework* é flexível, de maneira a não impedir que um médico emita um lado caso esqueça seu *smartphone* em casa, por exemplo.

O sistema de Telemedicina é o responsável por prover a interface gráfica e ele se comunica diretamente com o serviço para solicitar autenticações e operações gerenciais. O sistema repassa os dados referentes a autenticação de usuário para o serviço, que os manipula e solicita operações no HSM quando necessário.

A Figura **??** é o complemento da Figura **??** (**FIGURAS REMOVIDAS**), mostrando a interação entre o *web service* com o restante dos componentes para efetuar a criação de um usuário. Toda a parte de criação e autenticação de usuários foi atribuida ao *web service*, que contém um banco de dados com todos os seus usuários e tipos de autenticação de cada um deles. O Sistema de Telemedicina se comunica diretamente com o *web service*, que por sua vez faz a comunicação com o HSM, facilitando dessa maneira a sua integração não só com ambientes médicos, mas com qualquer outro tipo de serviço que necessite de uma autenticação mais forte.

IMAGEM REMOVIDA

Já o processo de autenticação é muito parecido com o da Figura 3 apresentada anteriormente, porém, quem fará a autenticação de *login* e senha é o *web service*, e o mesmo requisitará ao HSM a verificação do OTP. Este processo pode ser visto por completo na Figura 4.

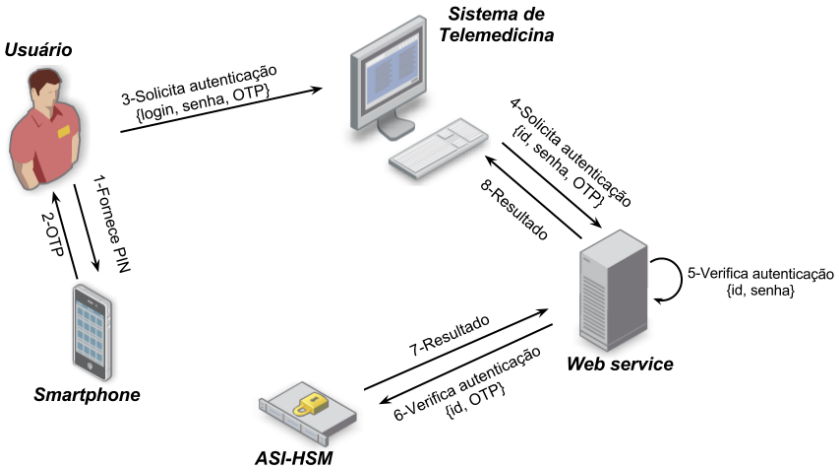

Figura 4: Autenticação de usuário com o Framework de Autenticação

4 TECNOLOGIAS UTILIZADAS

Abaixo serão apresentadas as ferramentas necessárias para o desenvolvimento do aplicativo.

Para a implementação do aplicativo Android foi utilizado o *Android Software Development Kit* (SDK), revisão 17, que fornece as ferramentas necessárias para o desenvolvimento, testes e depuração de aplicativos da plataforma. A aplicação em si foi desenvolvida na IDE Eclipse, na versão Helios Service Release 2, sob o *plugin* Android Development Tools (ADT). Este *plugin* proporciona um ambiente integrado ao Eclipse para o desenvolvimento na plataforma Android, proporcionando facilidades na criação de projetos, *design* da interface, entre outros.

Para a execução e testes da aplicação como um todo, inicialmente foi utilizado o Android Virtual Device (AVD), rodando a plataforma 2.3.3, que simula um dispositivo Android na própria IDE Eclipse. Porém, com o avanço do aplicativo, houve a necessidade da utilização de um dispositivo portátil, portanto, utilizou-se um celular Samsung Galaxy Y para rodar a aplicação a partir de então. Para auxílio no desenvolvimento foi utilizado a ferramenta LogCat, que apresenta o *log* de todo o sistema, servindo também de depurador. Para depurar mais profundamente o código, principalmente no processo de integração com o servidor de autenticação, foi necessário o uso do *plugin* de *debug* do próprio Eclipse.

Para a realização dos testes unitários nas funcionalidades que podem ser consideradas a base do protocolo proposto, utilizou-se o Android JUnit Test, que é uma extensão do JUnit para aplicativos Android. Já para controle de versão, foram utilizados o Subversion (SVN) e o Dropbox. O SVN foi utilizado para manter o controle das versões constantemente, e o Dropbox apenas como um *backup*.

A orientação para download e configuração de um ambiente de desenvolvimento Android pode ser encontrada em
http://developer.android.com/.

5 DESCRIÇÃO DA IMPLEMENTAÇÃO

Uma parte fundamental para a integração dessa proposta ao Sistema de Telemedicina está na modificação do aplicativo Google Authenticator, transformando-o em LabSEC Autenticador [1]. O processo de implementação e as modificações necessárias serão descritos abaixo.

Foram necessárias diversas modificações tanto na parte de interface, quanto nos controles principais do aplicativo. Estas modificações foram esboçadas no diagrama de atividades representado na Figura **??**.

IMAGEM REMOVIDA

Inicialmente, ao receber um QR Code no navegador, provindo do Sistema de Telemedicina (explicado na Figura **??**), o usuário abre o aplicativo e seleciona a opção de "Acordo de chave seguro"(Figura 5). Logo em seguida um leitor de QR Code será disponibilizado, basta o usuário posicioná-lo em cima da figura para que o passo 1 da Figura **??** seja executado. Neste passo, é extraído o texto do QR Code e o mesmo passa por um *parser*. Este *parser* identifica que tipo de QR Code é esse, se é o relacionado com um acordo de chave seguro, esse texto é decifrado com a chave pública do HSM já conhecida pelo aplicativo (passo 2). Após isso, todos os dados provenientes do HSM são identificados (passo 3 do diagrama de atividades, e os dados foram apresentados na Figura **??**).

CONTEUDO REMOVIDO

Todas estas etapas são feitas na classe AuthenticatorActivity, que é a atividade principal do aplicativo, responsável por controlar todo o fluxo de telas e a parte lógica do propósito geral.

A parte de armazenamento de usuário no banco de dados se inicia com a solicitação de um PIN para cifragem da *master secret*. Para isso, uma outra atividade é iniciada, a EnterPinActivity. Esta é responsável apenas por disponibilizar uma tela com um pedido de PIN e de confirmação, como feito em qualquer sistema que cadastra uma senha (Figura 6). Ao obter o PIN do usuário, o fluxo retorna à atividade principal.

CONTEUDO REMOVIDO

Para finalizar, a *master secret* é cifrada utilizando o algoritmo simétrico AES e como chave o PIN anteriormente fornecido (passo 9). Por fim, os dados do novo usuário são salvos no banco de dados e o QR Code que deve ser retornado ao HSM é disponibilizado na tela do celular (passo 11 do diagrama de atividades e Figura **??**).

[1] Disponível em https://play.google.com/store

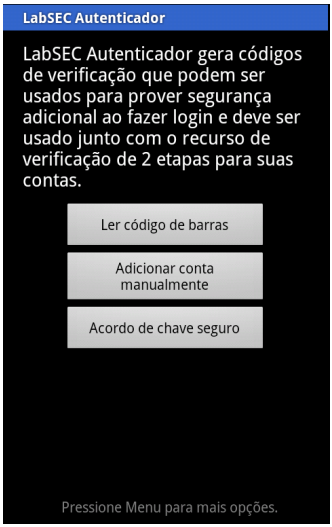

Figura 5: Tela inicial do aplicativo

Logo em seguida, uma tela de listagem dos usuários é mostrada (passo 12 do diagrama de atividades). Esta listagem é gerenciada pela atividade PinListAdapter, que é responsável por capturar os eventos de interface e mostrar os OTPs. Ela trabalha em conjunto com a AuthenticatorActivity, que calcula os OTPs e possui a listagem de usuários cadastrados. Inicialmente o OTP do usuário recém cadastrado não é mostrado na tela, pois sua *master secret* ainda está cifrada (Figura 7). Para tal, o usuário deve clicar no botão de "Ativar", que solicitará o PIN utilizado na cifragem (passo 12.1 do diagrama de atividades e Figura 8). Quando a *master secret* é decifrada (passo 13), a mesma é salva em memória, o OTP é calculado (passo 12.2) e disponibilizado ao usuário (Figura 9). Este processo se repete a cada 30 segundos e, para que o usuário não precise entrar com seu PIN tão frequentemente, é utilizada a *master secret* salva em memória para os cálculos seguintes.

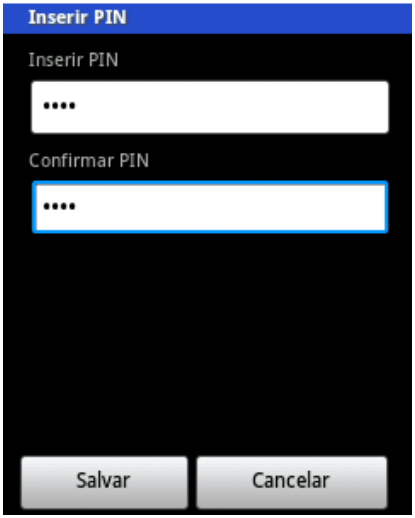

Figura 6: Tela de requisição do PIN

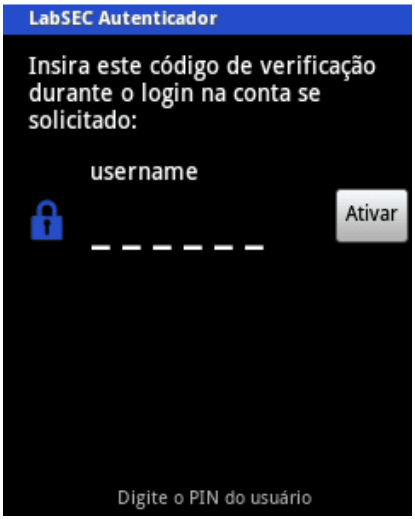

Figura 7: Listagem de usuários

Figura 8: Solicitação de PIN para decifragem da *master secret*

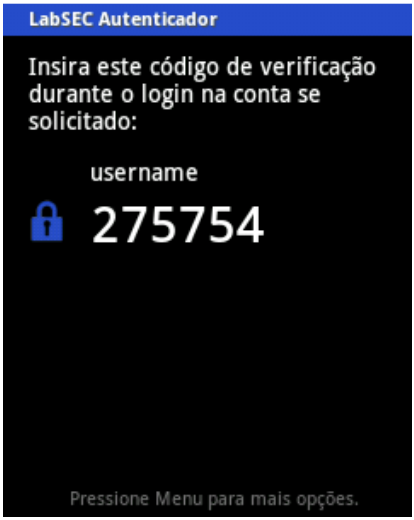

Figura 9: Listagem de usuários e respectivos OTPs calculados

6 ANÁLISE

Esta análise é baseada no *Electronic Authentication Guideline* do *National Institute of Standards and Technology* (NIST) (BURR et al., 2011).

De acordo com esse guia, um *token* possui um segredo que pode ser usado em processos de autenticação. Estes podem existir tanto em hardware e software, quanto na própria memória humana. No caso desse trabalho, o *token* pode ser o dispositivo móvel e a memória, que armazenam os OTPs e PINs respectivamente.

Ainda neste contexto, o dispositivo responsável pelos OTPs pode ser classificado entre *Single-factor One-Time Password Device* (SF OTP Device) e *Multi-factor One-Time Password Device* (MF OTP Device). Um SF OTP Device é um hardware que gera OTPs com o uso de uma *master secret* como semente para a geração. Ao executar a autenticação, é apresentado um OTP que prova ao mesmo tempo a posse e controle do dispositivo. Já um MF OTP Device é um hardware que gera os OTPs, porém requer uma ativação através do uso de um segundo fator de autenticação. Ele é "algo que você possui", e pode ser ativado através de "algo que você sabe"ou "o que você é".

Com isso, observa-se que o Google Autenticador original se enquadra na classificação de SF OTP Device, pelo fato de simplesmente gerar os OTPs e mostrá-los na tela. Já o LabSEC Autenticador pode ser considerado um MF OTP Device, pois antes de poder mostrar os OTPs gerados, é preciso passar por um processo de ativação, inserindo o PIN utilizado na cifragem da *master secret*. Conclui-se então que através do uso do PIN, foi melhorada a confiança no aplicativo, passando-o de *Single-Factor* para *Multi-Factor*.

Além disso, o guia classifica os *tokens* em níveis de confiança. Quando usados separadamente, os *tokens* em memória são classificados como nível 1 e o SF OTP Device em nível 2. Porém, quando usados em conjunto, como foi o caso do LabSEC Autenticador, dentre os 4 níveis possíveis, ele consegue alcançar um nível 3.

Já no ambiente de Telemedicina, transformou-se uma autenticação de um único *token* (ou único fator) em uma autenticação que utiliza múltiplos *tokens* (ou múltiplos fatores). Isso se dá pelo fato de se utilizar *login* e senha (o que o usuário sabe) em conjunto com um OTP, que prova a posse do dispositivo que o gera, sendo considerado "o que o usuário possui".

Com isso, conclui-se que a confiabilidade tanto no dispositivo

gerador de OTPs quanto na própria autenticação do sistema de Telemedicina foi melhorada.

7 CONSIDERAÇÕES FINAIS

Este trabalho teve como objetivo geral propor a implantação de um serviço de autenticação baseado em senhas descartáveis no sistema da Rede Catarinense de Telemedicina. Para tal, foram necessários estudos aprofundados deste tipo de autenticação, assim como os sistemas utilizados atualmente. Dentre estes, foi escolhido o Google Autenticador como base, porém, foram implementadas modificações para o compartilhamento e armazenamento da *master secret*, pois estes apresentavam vulnerabilidades de segurança. O produto final desse trabalho foi o LabSEC Autenticador, que além de suportar o Google Autenticador, tem a nova funcionalidade de armazenamento e acordo de chaves seguro.

Para que fosse possível a implantação deste tipo de autenticação no ambiente de Telemedicina, foram necessários um gerador de senhas, um servidor de autenticação e um *web service*. O aplicativo gerador de senhas descartáveis é o LabSEC Autenticador, que está no *smartphone* em posse do usuário. Um *web service* foi integrado ao Telemedicina para abstrair a camada do servidor de autenticação e essa integração foi descrita no capítulo 3.1. Como servidor de autenticação foi utilizado um HSM e o *firmware* para comunicação externa foi desenvolvido pela equipe responsável pela implementação do HSM do LabSEC. Já o *web service* é um projeto financiado pela FINEP, e o objetivo é o de fornecer fatores de autenticação, dentre eles está o de OTP apresentado neste trabalho.

No capítulo 5 foi detalhada a implementação do aplicativo. Foi apresentado um diagrama de atividade, descrevendo todas as modificações feitas no código, seguindo a ordem em que os métodos são chamados ao utilizar o aplicativo. Foram também apresentados alguns *screen captures* do aplicativo em ambiente de operação. Por fim, no capítulo 6 foi apresentada uma análise com base nas definições feitas pelo guia de autenticação do NIST.

7.1 TRABALHOS FUTUROS

Neste trabalho foram finalizados todos os objetivos de base descritos na introdução, implementando-os na plataforma Android. Como trabalho futuro, propõe-se a implementação dos mesmos nas plataformas iOS e Blackberry, tendo em vista que ambas suportam o aplicativo

Google Autenticador.

Outro trabalho futuro proposto consiste no relacionamento entre *master secret* e dispositivo móvel, como incluir na geração do OTP o IMEI do *smartphone* ou algum dado contido no *sim card*.

REFERÊNCIAS

Banco Bradesco S.A. *Cartão Chave de Segurança Bradesco*. 2012. <http://www.bradescoseguranca.com.br>.

BURR, W. E. et al. *SP 800-63-1. Electronic Authentication Guideline*. Gaithersburg, MD, United States, 2011.

CHENG, F. Security attack safe mobile and cloud-based one-time password tokens using rubbing encryption algorithm. *Mobile Networks and Applications*, p. 304 – 336, 2011.

CYCLOPS. *Sistema Catarinense de Telemedicina e Telessaúde*. 2010. <https://www.telemedicina.ufsc.br/rctm/>.

Federal Financial Institutions Examination Council. *Authentication in an Internet Banking Environment*. Arlington, VA - Estados Unidos da América, Outubro 2005. <http://www.ffiec.gov/press/pr101205.htm>.

Google Inc. *Google Authenticator*. 2012. <http://code.google.com/p/google-authenticator/>.

GUTTMAN, B.; ROBACK, E. A. *An Introduction to Computer Security: The NIST Handbook*. Estados Unidos da América: [s.n.], 1995. 179 – 191 p. <http://csrc.nist.gov/publications/nistpubs/800-12/handbook.pdf>.

HALLER, N. *The S/KEY One-Time Password System*. IETF, fev. 1995. RFC 1760 (Informational). (Request for Comments, 1760). <http://www.ietf.org/rfc/rfc1760.txt>.

HALLER, N.; ATKINSON, R. *On Internet Authentication*. IETF, out. 1994. RFC 1704 (Informational). (Request for Comments, 1704). <http://www.ietf.org/rfc/rfc1704.txt>.

HALLER, N. et al. *A One-Time Password System*. IETF, fev. 1998. RFC 2289 (Standard). (Request for Comments, 2289). <http://www.ietf.org/rfc/rfc2289.txt>.

Huntington Ventures Ltd. *AuthenticationWorld.Com, The business of authentication*. 2006. <http://www.authenticationworld.com/>.

IDALINO, T. B.; SPAGNUELO, D. Senhas descartáveis em dispositivos móveis para ambientes de telemedicina. In: *SBSeg 2012 WTICG ()*. http://sbseg2012.ppgia.pucpr.br/: [s.n.], 2012.

Initiative for Open AuTHentication. *OATH, Initiative for Open Authentication*. 2011. <http://www.openauthentication.org/aboutr>.

KESSLER, G. C. Passwords - strengths and weaknesses. In: . [s.n.], 1996. <http://www.garykessler.net/library/password.html>.

KRAWCZYK, H.; BELLARE, M.; CANETTI, R. *HMAC: Keyed-Hashing for Message Authentication*. IETF, fev. 1997. RFC 2104 (Informational). (Request for Comments, 2104). Updated by RFC 6151. <http://www.ietf.org/rfc/rfc2104.txt>.

LAMPORT, L. Password authentication with insecure communication. *Communications of the ACM*, p. 770 – 772, 1981.

MAGALHAES, P. S.; SANTOS, H. D. Biometria e autenticação. In: *Actas da 4a Conferência da Associação Portuguesa de Sistemas de Informação*. Porto. Portugal: [s.n.], 2003. p. 2 – 5.

M'RAIHI, D. et al. *HOTP: An HMAC-Based One-Time Password Algorithm*. IETF, dez. 2005. RFC 4226 (Informational). (Request for Comments, 4226). <http://www.ietf.org/rfc/rfc4226.txt>.

M'RAIHI, D. et al. *TOTP: Time-Based One-Time Password Algorithm*. IETF, maio 2011. RFC 6238 (Informational). (Request for Comments, 6238). <http://www.ietf.org/rfc/rfc6238.txt>.

NAKAMURA, E. T. *Segurança de redes em ambientes corporativos*. São Paulo, SP - Brasil: Novatec Editora Ltda., 2007. CapÃtulos 6 e 11 p.

RSA Security Inc. *RSA SecurID Two-factor Authentication*. [S.l.], 2010.

SILVA, D. R. P. da. *A memória humana no uso de senhas*. Tese (Doutorado) — Pontifícia Universidade Católica do Rio Grande do Sul, 2007.

SOUZA, T. C. S. de. *Aspectos Técnicos e Teóricos da Gestão do Ciclo de Vida de Chaves Criptográficas no OpenHSM*. Dissertação (Mestrado) — Universidade Federal de Santa Catarina, 2008.

SUTIL, J. M. *Gestão Segura de Múltiplas Instâncias de uma Mesma Chave de Assinatura em Autoridades Certificadoras*. Dissertação (Mestrado) — Universidade Federal de Santa Catarina, 2011.

ANEXO A – Código Fonte

Abaixo é apresentado o código fonte, incluindo o já existente do Google Autenticador. Abaixo de cada classe será especificado o que foi desenvolvido neste trabalho.

```
1  // Copyright (C) 2010 Google Inc.
2
3  package br.ufsc.labsec.authenticator;
4
5  import android.content.ContentValues;
6  import android.content.Context;
7  import android.database.Cursor;
8  import android.database.DatabaseUtils;
9  import android.database.sqlite.SQLiteDatabase;
10
11 /**
12  * A database of email addresses and secret values
13  *
14  * @author sweis@google.com (Steve Weis)
15  */
16 public class AccountDb {
17   public static final Integer DEFAULT_COUNTER = 0;  // for
       hotp type
18   private static final String TABLE_NAME = "accounts";
19   static final String ID_COLUMN = "_id";
20   static final String EMAIL_COLUMN = "email";
21   static final String PIN_COLUMN = "has_pin";
22   static final String SECRET_COLUMN = "secret";
23   static final String COUNTER_COLUMN = "counter";
24   static final String TYPE_COLUMN = "type";
25   private static final String PATH = "databases";
26   private static SQLiteDatabase DATABASE = null;
27
28   /**
29    * Types of secret keys.
30    */
31   public enum OtpType {  // must be the same as in res/
       values/strings.xml:type
32     TOTP (0),  // time based
33     HOTP (1);  // counter based
34
35     public final Integer value;  // value as stored in
       SQLite database
36     OtpType(Integer value) {
37       this.value = value;
38     }
39
40     public static OtpType getEnum(Integer i) {
41       for (OtpType type : OtpType.values()) {
42         if (type.value.equals(i)) {
43           return type;
44         }
45       }
```

```
46
47          return null;
48        }
49
50      }
51
52      private AccountDb() {
53        // Don't new me
54      }
55
56      /*
57       * initialize() must be called before any other AccountDb
          methods can be used.
58       */
59      static void initialize(Context context) {
60        if (DATABASE != null) {
61          return;
62        }
63
64        DATABASE = context.openOrCreateDatabase(PATH, Context.
          MODE_PRIVATE, null);
65        String createTableIfNeeded = String.format(
66            "CREATE TABLE IF NOT EXISTS %s" +
67            " (%s INTEGER PRIMARY KEY, %s TEXT NOT NULL, %s TEXT
           NOT NULL, " +
68            " %s INTEGER DEFAULT %s, %s INTEGER, %s INTEGER)",
69            TABLE_NAME, ID_COLUMN, EMAIL_COLUMN, SECRET_COLUMN,
          COUNTER_COLUMN,
70            DEFAULT_COUNTER, TYPE_COLUMN, PIN_COLUMN);
71        DATABASE.execSQL(createTableIfNeeded);
72      }
73
74      static Cursor getNames() {
75        return DATABASE.query(TABLE_NAME, null, null, null, null
          , null, null, null);
76      }
77
78      static Cursor getAccount(String email) {
79        return DATABASE.query(TABLE_NAME, null, EMAIL_COLUMN +
          "= ?",
80            new String[] {email}, null, null, null, null);
81      }
82
83      static boolean nameExists(String email) {
84        Cursor cursor = getAccount(email);
85        try {
86          return !cursorIsEmpty(cursor);
87        } finally {
88          tryCloseCursor(cursor);
89        }
90      }
91
```

```
92    static String getSecret(String email) {
93      Cursor cursor = getAccount(email);
94      try {
95        if (!cursorIsEmpty(cursor)) {
96          cursor.moveToFirst();
97          return cursor.getString(cursor.getColumnIndex(
      SECRET_COLUMN));
98        }
99      } finally {
100       tryCloseCursor(cursor);
101     }
102     return null;
103   }
104
105   /**
106    * Used to inform if the user is using the secure master
       secret type
107    * @param email
108    * @return integer 1 for true|0 otherwise
109    */
110   static int hasPin(String email) {
111             Cursor cursor = getAccount(email);
112             try {
113               if (!cursorIsEmpty(cursor)) {
114                 cursor.moveToFirst();
115                 return cursor.getInt(cursor.getColumnIndex(
      PIN_COLUMN));
116               }
117             } finally {
118               tryCloseCursor(cursor);
119             }
120             return 0;
121           }
122
123   static Integer getCounter(String email) {
124     Cursor cursor = getAccount(email);
125     try {
126       if (!cursorIsEmpty(cursor)) {
127         cursor.moveToFirst();
128         return cursor.getInt(cursor.getColumnIndex(
      COUNTER_COLUMN));
129       }
130     } finally {
131       tryCloseCursor(cursor);
132     }
133     return null;
134   }
135
136   static void incrementCounter(String email) {
137     ContentValues values = new ContentValues();
138     values.put(EMAIL_COLUMN, email);
139     Integer counter = getCounter(email);
```

```
140       values.put(COUNTER_COLUMN, counter + 1);
141       DATABASE.update(TABLE_NAME, values, whereClause(email),
          null);
142     }
143
144     static OtpType getType(String email) {
145       Cursor cursor = getAccount(email);
146       try {
147         if (!cursorIsEmpty(cursor)) {
148           cursor.moveToFirst();
149           Integer value = cursor.getInt(cursor.getColumnIndex(
          TYPE_COLUMN));
150           return OtpType.getEnum(value);
151         }
152       } finally {
153         tryCloseCursor(cursor);
154       }
155       return null;
156     }
157
158     static void setType(String email, OtpType type) {
159       ContentValues values = new ContentValues();
160       values.put(EMAIL_COLUMN, email);
161       values.put(TYPE_COLUMN, type.value);
162       DATABASE.update(TABLE_NAME, values, whereClause(email),
          null);
163     }
164
165     private static String whereClause(String email) {
166       return EMAIL_COLUMN + " = " + DatabaseUtils.
          sqlEscapeString(email);
167     }
168
169     static void delete(String email) {
170       DATABASE.delete(TABLE_NAME, whereClause(email), null);
171     }
172
173     /**
174      * Save key to database, creating a new user entry if
          necessary.
175      * @param email the user email address. When editing, the
          new user email.
176      * @param secret the secret key.
177      * @param oldEmail If editing, the original user email,
          otherwise null.
178      * @param type hotp vs totp
179      * @param counter only important for the hotp type
180      */
181     static void update(String email, String secret, String
          oldEmail,
182         Integer type, Integer counter, int hasPin) {
183       ContentValues values = new ContentValues();
```

```
184       values.put(EMAIL_COLUMN, email);
185       values.put(SECRET_COLUMN, secret);
186       values.put(TYPE_COLUMN, type);
187       values.put(COUNTER_COLUMN, counter);
188       values.put(PIN_COLUMN, hasPin);
189
190       int updated = DATABASE.update(TABLE_NAME, values,
191                                     whereClause(oldEmail),
          null);
192       if (updated == 0) {
193         DATABASE.insert(TABLE_NAME, null, values);
194       }
195     }
196
197     /**
198      * Returns true if the cursor is null, or contains no rows
          .
199      */
200     public static boolean cursorIsEmpty(Cursor c) {
201       return c == null || c.getCount() == 0;
202     }
203
204     /**
205      * Closes the cursor if it is not null and not closed.
206      */
207     public static void tryCloseCursor(Cursor c) {
208       if (c != null && !c.isClosed()) {
209         c.close();
210       }
211     }
212   }
```

Code A.1: Classe AccountDb

Neste trabalho foi adicionada na tabela AccountDb a coluna has_pin, com o objetivo de identificar quais contas utilizam a *master secret* cifrada no banco de dados. Para a obtenção dessa informação foi criado o método da linha 105 à 121.

```
1  // Copyright (C) 2009 Google Inc.
2
3  package br.ufsc.labsec.authenticator;
4
5  import br.ufsc.labsec.authenticator.R;
6  import br.ufsc.labsec.authenticator.AccountDb.OtpType;
7  import br.ufsc.labsec.authenticator.Base32String.
       DecodingException;
8
9  import android.app.Activity;
10 import android.app.AlertDialog;
11 import android.content.ActivityNotFoundException;
12 import android.content.Context;
```

```
13 import android.content.DialogInterface;
14 import android.content.Intent;
15 import android.content.pm.PackageManager.
       NameNotFoundException;
16 import android.database.Cursor;
17 import android.net.Uri;
18 import android.os.Bundle;
19 import android.os.Handler;
20 import android.os.Vibrator;
21 import android.text.ClipboardManager;
22 import android.text.Html;
23 import android.text.method.LinkMovementMethod;
24 import android.util.Log;
25 import android.view.ContextMenu;
26 import android.view.ContextMenu.ContextMenuInfo;
27 import android.view.Menu;
28 import android.view.MenuInflater;
29 import android.view.MenuItem;
30 import android.view.View;
31 import android.view.View.OnClickListener;
32 import android.view.ViewGroup;
33 import android.widget.AdapterView;
34 import android.widget.AdapterView.AdapterContextMenuInfo;
35 import android.widget.AdapterView.OnItemClickListener;
36 import android.widget.Button;
37 import android.widget.EditText;
38 import android.widget.LinearLayout;
39 import android.widget.ListView;
40 import android.widget.TextView;
41 import android.widget.Toast;
42
43 import java.math.BigInteger;
44 import java.security.GeneralSecurityException;
45
46 import javax.crypto.BadPaddingException;
47 import javax.crypto.Mac;
48 import javax.crypto.spec.SecretKeySpec;
49
50 /**
51  * The main activity that displays usernames and codes
52  *
53  * @author sweis@google.com (Steve Weis)
54  * @author adhintz@google.com (Drew Hintz)
55  * @author thais.idalino@inf.ufsc.br (ThaÃs Bardini Idalino
       )
56  */
57 public class AuthenticatorActivity extends Activity
       implements OnClickListener {
58
59         private static AuthenticatorActivity SINGLETON; //
       used only by saveSecret
60
```

```
61          /** The tag for log messages */
62          static final String TAG = "Authenticator";
63          private static final long VIBRATE_DURATION = 200L;
64
65          private TextView mStatusText;
66          private TextView mEnterPinTextView;
67          private ListView mUserList;
68          private PinListAdapter mUserAdapter;
69          private PinInfo[] mUsers = {};
70          private Button mScanBarcodeButton;
71          private Button mEnterKeyButton;
72          private Button mSecureMSAgreementButton;
73          private LinearLayout mButtonsLayout;
74          private Handler mHandler;
75          private Runnable mRefreshTask;
76          private static Context mContext;
77
78          public static boolean mAccessibilityAvailable;
79          private static String mVersion;
80
81          static {
82                  try {
83                          WrapAccessibilityEvent.
        checkAvailable();
84                          mAccessibilityAvailable = true;
85                  } catch (VerifyError e) {
86                          mAccessibilityAvailable = false;
87                  }
88          }
89
90          private static DiffieHellman DIFFIE_HELLMAN = new
        DiffieHellman();
91          private static boolean LABSEC_AUTH = false;
92          static final String DEFAULT_USER = "Default account
        ";
93          private static final String OTP_SCHEME = "otpauth";
94          private static final String TOTP = "totp"; // time-
        based
95          private static final String HOTP = "hotp"; //
        counter-based
96          private static final String USER_PARAM = "user";
97          private static final String SECRET_PARAM = "secret";
98          private static final String COUNTER_PARAM = "counter
        ";
99          private static final int CHECK_KEY_VALUE_ID = 0;
100         private static final int RENAME_ID = 1;
101         private static final int DELETE_ID = 2;
102         private static final int COPY_TO_CLIPBOARD_ID = 3;
103         private static final int SCAN_REQUEST = 31337;
104         private static final int PIN_REQUEST = 11111;
105         private static final String ZXING_MARKET = "market
        ://search?q=pname:com.google.zxing.client.android";
```

```
106         private static final String ZXING_DIRECT = "https://
    zxing.googlecode.com/files/BarcodeScanner3.1.apk";
107         private static final String OPEN_SOURCE_URI = "http
    ://code.google.com/p/google-authenticator/";
108         private static final String TERMS_URI = "http://www.
    google.com/accounts/TOS";
109         private static final String PRIVACY_URI = "http://
    www.google.com/mobile/privacy.html";
110 
111         // Based on default OTP interval, but set to update
    more frequently to avoid
112         // stale codes.
113         private static final int REFRESH_INTERVAL_SEC =
    PasscodeGenerator.INTERVAL / 6;
114 
115         /** Called when the activity is first created. */
116         @Override
117         public void onCreate(Bundle savedInstanceState) {
118                 super.onCreate(savedInstanceState);
119                 SINGLETON = this;
120                 AccountDb.initialize(this);
121                 HSMDb.initialize(this);
122                 setContentView(R.layout.main);
123 
124                 // restore state on screen rotation
125                 Object savedState =
    getLastNonConfigurationInstance();
126                 if (savedState != null) {
127                         mUsers = (PinInfo[]) savedState;
128                 }
129 
130                 mUserList = (ListView) findViewById(R.id.
    user_list);
131                 mStatusText = (TextView) findViewById(R.id.
    status_text);
132                 mSecureMSAgreementButton = (Button)
    findViewById(R.id.secure_ms_agreement);
133                 mSecureMSAgreementButton.setOnClickListener(
    this);
134                 mScanBarcodeButton = (Button) findViewById(R
    .id.scan_barcode_button);
135                 mScanBarcodeButton.setOnClickListener(this);
136                 mEnterKeyButton = (Button) findViewById(R.id
    .enter_key_button);
137                 mEnterKeyButton.setOnClickListener(this);
138                 mButtonsLayout = (LinearLayout) findViewById
    (R.id.main_buttons);
139                 mButtonsLayout.setVisibility(View.GONE);
140                 mEnterPinTextView = (TextView) findViewById(
    R.id.enter_pin);
141                 mEnterPinTextView.setVisibility(View.GONE);
142                 mContext = getApplicationContext();
```

```
143
144                 mUserList.setVisibility(View.GONE);
145                 if (mAccessibilityAvailable) {
146                         mUserAdapter = new PinListAdapter(
    this, R.layout.user_row, mUsers);
147                 } else {
148                         mUserAdapter = new PinListAdapter(
    this, R.layout.user_row_legacy,
149                                                 mUsers);
150                 }
151                 mUserList.setAdapter(mUserAdapter);
152                 mUserList.setOnItemClickListener(new
    OnItemClickListener() {
153                         public void onItemClick(AdapterView
    <?> arg0, View row, int arg2,
154                                         long arg3) {
155                                 OnButtonClickListener
    clickListener = (OnButtonClickListener) row
156                 .getTag();
157                                 Button nextOtp = (Button)
    row.findViewById(R.id.next_otp);
158                                 if ((clickListener != null)
    && nextOtp.isEnabled()) {
159                                         clickListener.
    onClick(row);
160                                 }
161                                 if (mAccessibilityAvailable)
     {
162                                         WrapAccessibilityEvent
    .sendEvent(mUserList,
163                                                 WrapAccessibilit
    .TYPE_VIEW_SELECTED);
164                                 }
165                         }
166                 });
167
168                 try {
169                         mVersion = getPackageManager().
    getPackageInfo(getPackageName(), 0).versionName;
170                 } catch (NameNotFoundException e) {
171                         mVersion = "Unknown";
172                 }
173
174                 // Create the handler used for processing
    refresh tasks.
175                 mHandler = new Handler();
176         }
177
178         @Override
179         public Object onRetainNonConfigurationInstance() {
180                 return mUsers; // save state of users and
    currently displayed PINs
```

```
181         }
182
183         @Override
184         protected void onResume() {
185                 super.onResume();
186                 Log.i(TAG, "onResume");
187                 Intent intent = getIntent();
188                 Uri uri = intent.getData();
189
190                 // If this activity was started by the user
    clicking on a link, then
191                 // we should fetch the secret key from the
    given URL.
192                 if (uri != null) {
193                         parseSecret(uri);
194                         setIntent(new Intent());
195                 }
196
197                 // Schedule the automatic refresh task.
198                 Runnable task = new Runnable() {
199                         public void run() {
200                                 // This task perpetuates
    itself by scheduling for another round  of
201                                 // execution using the same
    handler on main activity.
202                                 // Before continuing
    execution, we verify that it is still current
203                                 // for the activity -- if the
     activity is stopped/restarted,
204                                 // a task object will
    scheduled and this one allowed to expire.
205                                 if (mRefreshTask == this) {
206                                         AuthenticatorActivity
    .this.refreshUserList();
207                                         mHandler.postDelayed
    (this, REFRESH_INTERVAL_SEC * 1000);
208                                 }
209                         }
210                 };
211
212                 // Kick-start the refresh loop by running
    the task once.
213                 mRefreshTask = task;
214                 mRefreshTask.run();
215         }
216
217         @Override
218         protected void onPause() {
219                 super.onPause(); // Required by Android.
220                 mRefreshTask = null;
221         }
222
```

```
223         /** Display list of user emails and updated pin
    codes. */
224         protected void refreshUserList() {
225                 refreshUserList(false);
226         }
227
228         /**
229          * Display list of user emails and updated pin codes
    .
230          *
231          * @param isAccountModified
232          *            if true, force full refresh
233          */
234         protected void refreshUserList(boolean
    isAccountModified) {
235
236                 // If the users have changed, let the (
    potentially running) widget know
237                 // it needs to be
238                 // updated
239                 Intent intent = new Intent(
240                                 AuthenticatorWidget.
    WidgetReceiver.APPWIDGET_UPDATE);
241                 intent.setClass(this, AuthenticatorWidget.
    WidgetReceiver.class);
242                 sendBroadcast(intent);
243
244                 Cursor cursor = AccountDb.getNames();
245                 try {
246                         if (!AccountDb.cursorIsEmpty(cursor)
    ) {
247                                 int index = cursor.
    getColumnIndex(AccountDb.EMAIL_COLUMN);
248                                 if (isAccountModified ||
    mUsers.length != cursor.getCount()) {
249                                         mUsers = new PinInfo
    [cursor.getCount()];
250                                 }
251                                 for (int i = 0; i < cursor.
    getCount(); i++) {
252                                         cursor.
    moveToPosition(i);
253                                         String user = cursor
    .getString(index);
254                                         Log.i(TAG, "onResume
     user: " + user);
255                                         computeAndDisplayOtp
    (user, i, false, null);
256                                 }
257
258                                 if (mAccessibilityAvailable)
     {
```

```
259                                     mUserAdapter = new
    PinListAdapter(this, R.layout.user_row,
260                                                     mUsers
    );
261                             } else {
262                                     mUserAdapter = new
    PinListAdapter(this,
263                                                     R.
    layout.user_row_legacy, mUsers);
264                             }
265                             mUserList.setAdapter(
    mUserAdapter); // force refresh of display
266
267                             if (mUserList.getVisibility
    () != View.VISIBLE) {
268                                     mEnterPinTextView.
    setText(R.string.enter_pin);
269                                     mEnterPinTextView.
    setVisibility(View.VISIBLE);
270                                     mButtonsLayout.
    setVisibility(View.GONE);
271                                     mUserList.
    setVisibility(View.VISIBLE);
272                                     registerForContextMenu
    (mUserList);
273                             }
274
275                     } else {
276                             // If the user started up
    this app but there is no secret key
277                             // yet,
278                             // then tell the user to
    visit a web page to get the secret key.
279                             mUsers = new PinInfo[0]; //
    clear any existing user PIN state
280                             tellUserToGetSecretKey();
281                     }
282             } finally {
283                     AccountDb.tryCloseCursor(cursor);
284             }
285     }
286
287     /**
288      * Tells the user to visit a web page to get a
    secret key.
289      */
290     private void tellUserToGetSecretKey() {
291             String notInitialized = getString(R.string.
    not_initialized);
292             CharSequence styledNotInitalized = Html.
    fromHtml(notInitialized);
```

```
293                 mEnterPinTextView.setText(
    styledNotInitalized);
294                 mEnterPinTextView.setMovementMethod(
    LinkMovementMethod.getInstance());
295                 mEnterPinTextView.setVisibility(View.VISIBLE
    );
296                 mButtonsLayout.setVisibility(View.VISIBLE);
297                 mUserList.setVisibility(View.GONE);
298         }
299 
300         /**
301          * Computes the PIN and saves it in mUsers. This
    currently runs in the UI
302          * thread so it should not take more than a second
    or so. If necessary, we
303          * can move the computation to a background thread.
304          *
305          * @param user the user email to display with the
    PIN
306          * @param position the index for the screen of this
    user and PIN
307          * @param computeHotp true if we should increment
    counter and display new hotp
308          * @return the entered PIN to decrypt the master
    secret
309          */
310         public String computeAndDisplayOtp(String user, int
    position,
311                                            boolean computeHotp,
    String pin) {
312                 OtpType type = AccountDb.getType(user);
313                 int hasPin = AccountDb.hasPin(user);
314 
315                 PinInfo currentOtp;
316                 if (mUsers[position] != null) {
317                         currentOtp = mUsers[position]; //
    existing PinInfo, so we'll update it
318                 } else {
319                         currentOtp = new PinInfo();
320                         currentOtp.mOtp = getString(R.string
    .empty_pin);
321                 }
322 
323                 if (hasPin == 0) { //if is the old simple
    otp, just get the secret and the otp value
324                         if (currentOtp.mSecret == null) {
325                                 String secret = getSecret(
    user, null);
326                                 currentOtp.mSecret = secret;
327                         }
328                 } else {
329                         currentOtp.mIsLabSEC = true;
```

```
330                         if (pin != null) { //If user just
    entered the pin value and ativated the otp display
331                                 String secret = getSecret(
    user, pin);//return null if the pin is wrong
332                                 currentOtp.mSecret = secret;
333                         }
334                 }
335
336                 if (currentOtp.mSecret != null && currentOtp
    .mSecret != "") { //if there is a secret, calculate the
    otp value
337                         if (type == OtpType.TOTP) {
338                                 currentOtp.mOtp = computeOTP
    (currentOtp.mSecret, null);
339                         } else if (type == OtpType.HOTP) {
340                                 currentOtp.mIsHotp = true;
341                                 if (computeHotp) {
342                                         AccountDb.
    incrementCounter(user);
343                                         Integer counter =
    AccountDb.getCounter(user);
344                                         currentOtp.mOtp =
    AuthenticatorActivity.computeOTP(
345                                                                                 curr
    .mSecret, counter.longValue());
346                                 }
347                         }
348                 } //otherwise, it will be an empty value
349                 currentOtp.mUser = user;
350                 mUsers[position] = currentOtp;
351                 return currentOtp.mOtp;
352
353         }
354
355         /**
356          * Reads the secret key that was saved on the phone.
357          * @param user the user of the secret
358          * @param pin the pin used to encrypt the master
    secret if is secure master secret type|null otherwise
359          * @return the secret key
360          */
361         static String getSecret(String user, String pin) {
362                 String secret = AccountDb.getSecret(user);
363                 int hasPin = AccountDb.hasPin(user);
364                 if (hasPin == 1) {
365                         String decryptedSecret = "";
366                         try {
367                                 decryptedSecret = Crypto.
    decrypt(Crypto.hash(pin), secret);
368                         } catch (BadPaddingException e) {
369                                 Context context = getContext
    ();
```

```
370                                     CharSequence text = "Wrong
    pin";
371                                     int duration = Toast.
    LENGTH_SHORT;
372
373                                     Toast toast = Toast.makeText
    (context, text, duration);
374                                     toast.show();
375                             } catch (Exception e) {
376                                     e.printStackTrace();
377                             }
378                             return decryptedSecret;
379                     }
380                     return secret;
381             }
382
383             /**
384              * Return the activity context, used in static
    methods
385              * @return Context
386              */
387             public static Context getContext()
388             {
389                     return AuthenticatorActivity.mContext;
390             }
391
392             /**
393              * Computes the one-time password given the secret
    key.
394              *
395              * @param secret the secret key
396              * @param counter null if using totp, otherwise
    value of hotp counter
397              * @return the OTP, or if error an error message
398              */
399             public static String computeOTP(String secret, Long
    counter) {
400                     if (secret == null || secret.length() == 0)
    {
401                             return mContext.getString(R.string.
    empty_pin);
402                     }
403                     try {
404                             final byte[] keyBytes = Base32String
    .decode(secret);
405                         Mac mac = Mac.getInstance("HMACSHA1");
406                             mac.init(new SecretKeySpec(keyBytes,
     ""));
407                             PasscodeGenerator pcg = new
    PasscodeGenerator(mac);
408                             if (counter == null) { // time-based
     totp
```

```
409                                 return pcg.
    generateTimeoutCode();
410                         } else { // counter-based hotp
411                                 return pcg.
    generateResponseCode(counter);
412                         }
413                 } catch (GeneralSecurityException e) {
414                         return "General security exception";
415                 } catch (DecodingException e) {
416                         return "Decoding exception";
417                 }
418
419         }
420
421         public static byte[] hexStringToByteArray(String s)
    {
422                 int len = s.length();
423                 byte[] data = new byte[len / 2];
424                 for (int i = 0; i < len; i += 2) {
425                     data[i / 2] = (byte) ((Character.digit(s.
    charAt(i), 16) << 4)
426                                         + Character.digit(s.charAt
    (i+1), 16));
427                 }
428                 return data;
429         }
430
431         /**
432          * Parses a secret value from a URI. The format will
     be:
433          *
434          * https://www.google.com/accounts/KeyProv?user=
    username#secret OR
435          * totp://username@domain#secret
436          * otpauth://totp/user@example.com?secret=FFF...
437          * otpauth://hotp/user@example.com?secret=FFF...&
    counter=123
438          *
439          * Obs.: o tipo Ã© -1 pra
440          * TOTP e o contador pra HOTP
441          *
442          * @param uri
443          *            The URI containing the secret key
444          */
445         private void parseSecret(Uri uri) {
446                 Integer type = AccountDb.OtpType.TOTP.value;
447                 Integer counter = AccountDb.DEFAULT_COUNTER;
448                 String authority = uri.getAuthority();
449                 String user = DEFAULT_USER;
450                 String path = uri.getPath();
451                 if (LABSEC_AUTH) {
452                         if (authority != null) {
```

```
453                                 user = authority;
454                         }
455                         path = path.substring(1);
456                         try {
457 
458                 CODIGO REMOVIDO
459 
460                 // 5 - ask for pin to encrypt the master
    secret and then save all data
461                                 Intent intent = new Intent(
    getApplicationContext(), EnterPinActivity.class);
462                                 intent.putExtra("USER", user
    );
463                                 intent.putExtra("SECRET",
    secretInBase32);
464                                 intent.putExtra("TYPE", type
    .toString());
465                                 intent.putExtra("COUNTER",
    counter.toString());
466 
467                                 startActivityForResult(
    intent, PIN_REQUEST);
468                         } catch (Exception e) {
469                                 e.printStackTrace();
470                         }
471 
472                 } else {
473                         String scheme = uri.getScheme().
    toLowerCase();
474                         String secret;
475 
476                         if (OTP_SCHEME.equals(scheme)) {
477                                 if (authority != null &&
    authority.equals(TOTP)) {
478                                         type = AccountDb.
    OtpType.TOTP.value;
479                                 } else if (authority != null
     && authority.equals(HOTP)) {
480                                         type = AccountDb.
    OtpType.HOTP.value;
481                                         String
    counterParameter = uri
482                                         .getQueryParameter(COUNTER_PARAM);
483                                         if (counterParameter
     != null) {
484                                                 counter =
    Integer.parseInt(counterParameter);
485                                         }
486                                 }
487 
488                                 if (path != null && path.
    length() > 1) {
```

```
489                                     user = path.
    substring(1); // path is "/user", so remove leading /
490                             }
491
492                             secret = uri.
    getQueryParameter(SECRET_PARAM);
493                             // TODO(adhintz) remove TOTP
     scheme here and in AndroidManifest.xml
494                     } else if (TOTP.equals(scheme)) {
495                             if (authority != null) {
496                                     user = authority;
497                             }
498                             secret = uri.getFragment();
499                     } else { // https://www.google.com
    ... URI format
500                             String userParam = uri.
    getQueryParameter(USER_PARAM);
501                             if (userParam != null) {
502                                     user = userParam;
503                             }
504                             secret = uri.getFragment();
505                     }
506
507                     if (secret == null || secret.length
    () == 0) {
508                             Log.e(TAG, "Secret key not
    found in URI");
509                             new AlertDialog.Builder(this
    ).setTitle(R.string.error_title)
510                 .setMessage(R.string.error_uri)
511                 .setIcon(android.R.drawable.ic_dialog_alert)
512                 .setPositiveButton(R.string.ok, null).show()
    ;
513                             return;
514                     }
515
516                     if (!secret.equals(getSecret(user,
    null))
517                 || counter != AccountDb.getCounter(user)
518                 || type != AccountDb.getType(user).value) {
519                             saveSecret(this, user,
    secret, null, type, counter, 0);
520                             mStatusText.setText(R.string
    .secret_saved);
521                     }
522             }
523         }
524
525         /**
526          * Saves the secret key to local storage on the
    phone.
527          *
```

```
528         * @param user the user email address. When editing,
     the new user email.
529         * @param secret the secret key
530         * @param originalUser If editing, the original user
     email, otherwise null.
531         * @param type hotp vs totp
532         * @param counter only important for the hotp type
533         * @param pin only important for the labsec otp type
    . 1 if master secret is encrypted, 0 otherwise
534         */
535        static void saveSecret(Context context, String user,
     String secret,
536                               String originalUser, Integer type
    , Integer counter, int hasPin) {
537                if (originalUser == null) { // new user
    account
538                        originalUser = user;
539                }
540                if (secret != null) {
541                        AccountDb.update(user, secret,
    originalUser, type, counter, hasPin);
542                        ((Vibrator) context.getSystemService
    (Context.VIBRATOR_SERVICE))
543                .vibrate(VIBRATE_DURATION);
544                }
545
546                SINGLETON.refreshUserList(true);
547        }
548
549        /** Converts user list ordinal id to user email */
550        private String idToEmail(long id) {
551                return mUsers[(int) id].mUser;
552        }
553
554        @Override
555        public void onCreateContextMenu(ContextMenu menu,
    View v,
556                                        ContextMenuInfo menuInfo
    ) {
557                super.onCreateContextMenu(menu, v, menuInfo)
    ;
558                AdapterContextMenuInfo info = (
    AdapterContextMenuInfo) menuInfo;
559                String user = idToEmail(info.id);
560                OtpType type = AccountDb.getType(user);
561                menu.setHeaderTitle(user);
562                menu.add(0, COPY_TO_CLIPBOARD_ID, 0, R.
    string.copy_to_clipboard);
563                // Option to display the check-code is only
    available for HOTP accounts.
564                if (type == OtpType.HOTP) {
```

```
565                         menu.add(0, CHECK_KEY_VALUE_ID, 0, R
    .string.check_code_menu_item);
566                 }
567                 menu.add(0, RENAME_ID, 0, R.string.rename);
568                 menu.add(0, DELETE_ID, 0, R.string.delete);
569         }
570
571         @Override
572         public boolean onContextItemSelected(MenuItem item)
    {
573                 AdapterContextMenuInfo info = (
    AdapterContextMenuInfo) item
574         .getMenuInfo();
575                 Intent intent;
576                 final String user = idToEmail(info.id); //
    final so listener can see value
577                 switch (item.getItemId()) {
578             case COPY_TO_CLIPBOARD_ID:
579                 ClipboardManager clipboard = (
    ClipboardManager) getSystemService(CLIPBOARD_SERVICE);
580                 clipboard.setText(mUsers[(int) info.id].mOtp
    );
581                 return true;
582             case CHECK_KEY_VALUE_ID:
583                 intent = new Intent(Intent.ACTION_VIEW);
584                 intent.setClass(this, CheckCodeActivity.
    class);
585                 intent.putExtra("user", user);
586                 startActivity(intent);
587                 return true;
588             case RENAME_ID:
589                 final Context context = this; // final so
    listener can see value
590                 final View frame = getLayoutInflater().
    inflate(R.layout.rename,
591                                                                 (
    ViewGroup) findViewById(R.id.rename_root));
592                 final EditText nameEdit = (EditText) frame
593                 .findViewById(R.id.rename_edittext);
594                 nameEdit.setText(user);
595                 new AlertDialog.Builder(this)
596                 .setTitle(
597                         String.format(getString(R.string.
    rename_message),
598                                                 user))
599                 .setView(frame)
600                 .setPositiveButton(
601                                         R.string.submit,
602                                         this.
    getRenameClickListener(context, user, nameEdit))
603                 .setNegativeButton(R.string.cancel, null).
    show();
```

```
604                 return true;
605             case DELETE_ID:
606                 new AlertDialog.Builder(this)
607                 .setTitle(R.string.delete_message)
608                 .setMessage(user)
609                 .setIcon(android.R.drawable.ic_dialog_alert)
610                 .setPositiveButton(R.string.ok,
611                                     new DialogInterface.
    OnClickListener() {
612                         public void onClick(DialogInterface
    dialog,
613
     whichButton) {
614                             AccountDb.delete(user);
615                             refreshUserList();
616                         }
617                 }).setNegativeButton(R.string.cancel, null).
    show();
618                 return true;
619             default:
620                 return super.onContextItemSelected(item);
621                 }
622         }
623
624         DialogInterface.OnClickListener
    getRenameClickListener(
625                                                                 final
     Context context, final String user, final EditText
    nameEdit) {
626                 return new DialogInterface.OnClickListener()
     {
627                         public void onClick(DialogInterface
    dialog, int whichButton) {
628                                 String newName = nameEdit.
    getText().toString();
629                                 if (newName != user) {
630                                         if (AccountDb.
    nameExists(newName)) {
631                                                 Toast.
    makeText(context, R.string.error_exists, 15)
632                                 .show();
633                                         } else {
634                                                 saveSecret(
    context, newName, getSecret(user, null),
635                                     user, AccountDb.getType(
    user).value,
636                                     AccountDb.getCounter(user
    ), 0); // CHECAR
637                                         }
638                                 }
639                         }
640                 };
```

```
641         }
642
643         @Override
644         public boolean onCreateOptionsMenu(Menu menu) {
645                 // Inflate the menu XML resource.
646                 MenuInflater inflater = getMenuInflater();
647                 inflater.inflate(R.menu.main, menu);
648                 return true;
649         }
650
651         @Override
652         public boolean onPrepareOptionsMenu(Menu menu) {
653                 // Do not display the Refresh button if we
    do not have any keys.
654                 if (mUserList.getVisibility() == View.GONE)
    {
655                         menu.findItem(R.id.refresh).
    setVisible(false);
656                 } else {
657                         menu.findItem(R.id.refresh).
    setVisible(true);
658                 }
659                 return true;
660         }
661
662         @Override
663         public boolean onMenuItemSelected(int featureId,
    MenuItem item) {
664                 Intent intent;
665                 switch (item.getItemId()) {
666             case R.id.secure_ms_agreement:
667                 this.secureMSAgreement();
668                 return true;
669             case R.id.enter_key_item:
670                 this.manuallyEnterKey();
671                 return true;
672             case R.id.scan_barcode:
673                 this.scanBarcode();
674                 return true;
675             case R.id.refresh:
676                 refreshUserList();
677                 return true;
678             case R.id.about:
679                 new AlertDialog.Builder(this)
680                 .setTitle(R.string.about_menu_item)
681                 .setMessage(
682                                                         Html
    .fromHtml(String.format(
683                                                         getString
    (R.string.about_text), mVersion)))
684                 .setPositiveButton(R.string.ok, null).show()
    ;
```

```
685                 return true;
686             case R.id.opensource:
687                 intent = new Intent(Intent.ACTION_VIEW, Uri.
    parse(OPEN_SOURCE_URI));
688                 startActivity(intent);
689                 return true;
690             case R.id.terms:
691                 intent = new Intent(Intent.ACTION_VIEW, Uri.
    parse(TERMS_URI));
692                 startActivity(intent);
693                 return true;
694             case R.id.privacy:
695                 intent = new Intent(Intent.ACTION_VIEW, Uri.
    parse(PRIVACY_URI));
696                 startActivity(intent);
697                 return true;
698                 }
699
700                 return super.onMenuItemSelected(featureId,
    item);
701         }
702
703         @Override
704         public void onActivityResult(int requestCode, int
    resultCode, Intent intent) {
705                 if (requestCode == SCAN_REQUEST &&
    resultCode == Activity.RESULT_OK) {
706                         String contents = intent.
    getStringExtra("SCAN_RESULT");
707                         Log.i("QRCODE", contents);
708                         parseSecret(Uri.parse(contents));
709                 }
710
711                 if (requestCode == PIN_REQUEST && resultCode
     == Activity.RESULT_OK) {
712                         String pin = intent.getStringExtra("
    PIN_RESULT");
713                         String user = intent.getStringExtra
    ("USER");
714                         String secret = intent.
    getStringExtra("SECRET");
715                         String type = intent.getStringExtra
    ("TYPE");
716                         String counter = intent.
    getStringExtra("COUNTER");
717
718             CODIGO REMOVIDO
719
720                         try {
721                                 secret = Crypto.encrypt(
    Crypto.hash(pin), secret);
```

```
722                                 saveSecret(this, user,
    secret, null, Integer.parseInt(type), Integer.parseInt(
    counter), 1);
723                                 mStatusText.setText(R.string
    .secret_saved);
724
725                 CODIGO REMOVIDO
726
727                         } catch (Exception e) {
728                                 e.printStackTrace();
729                         }
730                 }
731         }
732
733
734         public void onClick(View view) {
735                 if (view == mScanBarcodeButton) {
736                         this.scanBarcode();
737                 } else if (view == mEnterKeyButton) {
738                         this.manuallyEnterKey();
739                 } else if (view == mSecureMSAgreementButton)
     {
740                         this.secureMSAgreement();
741                 }
742         }
743
744         private void manuallyEnterKey() {
745                 Intent intent = new Intent(Intent.
    ACTION_VIEW);
746                 intent.setClass(this, EnterKeyActivity.class
    );
747                 startActivity(intent);
748         }
749
750         private void scanBarcode() {
751                 Intent intentScan = new Intent("com.google.
    zxing.client.android.SCAN");
752                 intentScan.putExtra("SCAN_MODE", "
    QR_CODE_MODE");
753                 intentScan.putExtra("SAVE_HISTORY", false);
754                 try {
755                         startActivityForResult(intentScan,
    SCAN_REQUEST);
756                 } catch (ActivityNotFoundException e) {
757                         showDownloadDialog();
758                 }
759         }
760
761         /**
762          * Generate a barcode that will be used in the
    secure master secret
763          * agreement
```

```
764             *
765             * @param String
766             *            msg the message that will form the
        qrcode
767             */
768            private void generateBarcode(String msg) {
769                    Intent intent = new Intent("com.google.zxing
        .client.android.ENCODE");
770                    intent.putExtra("ENCODE_DATA", msg);
771                    intent.putExtra("ENCODE_TYPE", "TEXT_TYPE");
772                    intent.putExtra("ENCODE_SHOW_CONTENTS",
        false);
773
774                    try {
775                            startActivity(intent);
776                    } catch (ActivityNotFoundException e) {
777                            showDownloadDialog();
778                    }
779            }
780
781            /**
782             * Function that starts the secure master secret
        agreement. The barcode to
783             * be scanned is that one displayed by the server.
        After this, the user have
784             * to enter a PIN.
785             */
786            private void secureMSAgreement() {
787                    LABSEC_AUTH = true;
788                    scanBarcode();
789            }
790
791    CODIGO REMOVIDO
792
793            /**
794             * Prompt to download ZXing from Market. If Market
        app is not installed,
795             * such as on a development phone, open the HTTPS
        URI for the ZXing apk.
796             */
797            private void showDownloadDialog() {
798                    new AlertDialog.Builder(this)
799            .setTitle(R.string.install_dialog_title)
800            .setMessage(R.string.install_dialog_message)
801            .setIcon(android.R.drawable.ic_dialog_alert)
802            .setPositiveButton(R.string.install_button,
803                            new DialogInterface.
        OnClickListener() {
804                    public void onClick(DialogInterface dialog,
805                                    int whichButton) {
806                            Intent intent = new Intent(Intent.
        ACTION_VIEW,
```

```
807                                                 Uri.parse(
    ZXING_MARKET));
808                 try {
809                     startActivity(intent);
810                 } catch (ActivityNotFoundException e) { //
    if no Market app
811                     intent = new Intent(Intent.ACTION_VIEW,
    Uri
812                                                 .parse(ZXING_DIRECT)
    );
813                     startActivity(intent);
814                 }
815             }
816         }).setNegativeButton(R.string.cancel, null).show();
817         }
818
819 }
```

Code A.2: Classe AuthenticatorActivity

Nesta classe foram feitas alterações no método da linha 310 para suportar a geração de OTPs quando a *master secret* se encontra cifrada no banco de dados. No método da linha 361 foi adicionado o parâmetro *pin* com o objetivo de retornar a *master secret* em claro, decifrando-a com o *pin* se este for o caso.

O método da linha 445 foi alterado com o objetivo de, ao entrar com os dados de um usuário que terá sua *master secret* cifrada, redirecionar para uma *activity* que obtêm o *pin* para cifragem. Foi adicionado também o item da linha 666 e 739 para responder ao clique no botão de criação deste tipo de usuário na interface, assim como a criação do método da linha 786, que inicia todo o processo. Já a partir da linha 711 foi adicionada a chamada do método que cifra a *master secret* antes da mesma ser salva no banco de dados, junto com os outros dados do usuário.

```
1  package br.ufsc.labsec.authenticator;
2
3  import java.security.InvalidKeyException;
4  import java.security.MessageDigest;
5  import java.security.NoSuchAlgorithmException;
6  import java.security.PublicKey;
7  import javax.crypto.BadPaddingException;
8  import javax.crypto.Cipher;
9  import javax.crypto.IllegalBlockSizeException;
10 import javax.crypto.NoSuchPaddingException;
11 import javax.crypto.spec.SecretKeySpec;
12 import javax.security.cert.CertificateException;
13 import android.util.Base64;
14
```

```
15 public class Crypto {
16         static final String RSA = "RSA/None/PKCS1Padding";
17         static final String AES = "AES";
18         static final String SHA256 = "SHA256";
19 
20         /**
21          * Encrypts a plain text with a given public key
22          *
23          * @param plaintext the plain text to be encrypted
24          * @param publicKey the public key
25          *
26          * @return the text encrypted
27          */
28     public static String encrypt(String plaintext, PublicKey
     publicKey) throws Exception {
29         Cipher cipher = Cipher.getInstance(RSA);
30         cipher.init(Cipher.ENCRYPT_MODE, publicKey);
31         byte[] encrypted = cipher.doFinal(plaintext.getBytes
    ());
32         encrypted = Base64.encode(encrypted, Base64.DEFAULT)
    ;
33         return new String(encrypted);
34     }
35 
36         /**
37          * Decrypt an encrypted text with a given public key
38          *
39          * @param encrypted the encrypted text
40          * @param publicKey the public key
41          *
42          * @return the text decrypted
43          */
44     public static String decrypt(String encrypted, PublicKey
     publicKey) throws Exception {
45         Cipher cipher = Cipher.getInstance(RSA);
46         cipher.init(Cipher.DECRYPT_MODE, publicKey);
47         byte[] decoded = Base64.decode(encrypted, Base64.
    DEFAULT);
48         byte[] decrypted = cipher.doFinal(decoded);
49         return new String(decrypted);
50     }
51 
52     /**
53      * Gets the PublicKey object by the given pem
    certificate
54      *
55      * @param certPem
56      *
57      * @return PublicKey
58      * @throws CertificateException
59      */
```

```
60      public static PublicKey getPublicKeyFromCert (String
        certPem) throws CertificateException
61      {
62          byte[] certData = certPem.getBytes();
63          javax.security.cert.X509Certificate cert = javax.
        security.cert.X509Certificate.getInstance(certData);
64          return cert.getPublicKey();
65      }
66
67      /**
68       * Encrypts a message with the given symmetric password
69       *
70       * @param key the string password
71       * @param message the message to be encrypted
72       *
73       * @return String the message encrypted
74       * @throws NoSuchAlgorithmException
75       * @throws NoSuchPaddingException
76       * @throws InvalidKeyException
77       * @throws IllegalBlockSizeException
78       * @throws BadPaddingException
79       */
80      public static String encrypt (byte[] key, String message
        ) throws NoSuchAlgorithmException,
        NoSuchPaddingException, InvalidKeyException,
        IllegalBlockSizeException, BadPaddingException
81      {
82          String base64;
83          SecretKeySpec skeySpec = new SecretKeySpec(key, AES)
        ;
84          Cipher cipher = Cipher.getInstance(AES);
85          cipher.init(Cipher.ENCRYPT_MODE, skeySpec);
86          byte[] encrypted = cipher.doFinal(message.getBytes()
        );
87          base64 = Base64.encodeToString(encrypted, Base64.
        DEFAULT);
88          return base64;
89      }
90
91      /**
92       * Decrypts a message with the given symmetric password
93       *
94       * @param key the string password
95       * @param message the message encrypted with the key
96       *
97       * @return String the message decrypted
98       * @throws NoSuchAlgorithmException
99       * @throws NoSuchPaddingException
100      * @throws InvalidKeyException
101      * @throws IllegalBlockSizeException
102      * @throws BadPaddingException
103      */
```

```
104     public static String decrypt (byte[] key, String message
        ) throws NoSuchAlgorithmException,
        NoSuchPaddingException, InvalidKeyException,
        IllegalBlockSizeException, BadPaddingException
105     {
106         SecretKeySpec skeySpec = new SecretKeySpec(key, AES)
        ;
107         Cipher cipher = Cipher.getInstance(AES);
108         cipher.init(Cipher.DECRYPT_MODE, skeySpec);
109         byte[] decrypted = cipher.doFinal(Base64.decode(
        message, Base64.DEFAULT));
110         return new String(decrypted);
111     }
112
113     /**
114      * Generates the hash of the given string
115      *
116      * @param pin the pin to be transformed
117      *
118      * @return the hash of the pin in bytes
119      */
120
121     public static byte[] hash(String pin)
122     {
123         MessageDigest md;
124         byte[] pinHash = null;
125         try {
126             md = MessageDigest.getInstance(SHA256);
127             md.update(pin.getBytes());
128             pinHash  = md.digest();
129         } catch (NoSuchAlgorithmException e) {
130                         e.printStackTrace();
131         }
132         return pinHash;
133     }
134 }
```

Code A.3: Classe Crypto

Esta classe foi desenvolvida inteiramente no trabalho.

```
1  package br.ufsc.labsec.authenticator;
2
3  import android.app.Activity;
4  import android.content.Intent;
5  import android.graphics.Color;
6  import android.os.Bundle;
7  import android.view.View;
8  import android.view.View.OnClickListener;
9  import android.widget.Button;
10 import android.widget.EditText;
11 import android.widget.TextView;
```

```
12 import br.ufsc.labsec.authenticator.R;
13 /**
14  * The activity that allows users to enter the pin to
       encrypt their master secrets.
15  *
16  * @author thais.idalino@inf.ufsc.br
17  */
18 public class EnterPinActivity extends Activity implements
       OnClickListener {
19
20     private EditText pin;
21     private EditText pinConfirmation;
22     private Button submitButton;
23     private Button cancelButton;
24     private TextView mStatusText;
25
26     String user;
27     String secret;
28     String type;
29     String counter;
30     String mobileDH;
31     String nounce;
32
33     @Override
34         public void onCreate(Bundle savedInstanceState) {
35             super.onCreate(savedInstanceState);
36             setContentView(R.layout.enter_pin);
37
38             pin = (EditText) findViewById(R.id.pin);
39             pinConfirmation = (EditText) findViewById(R.id.
       pin_confirm);
40             submitButton = (Button) findViewById(R.id.
       submit_button);
41             cancelButton = (Button) findViewById(R.id.
       cancel_button);
42             mStatusText = (TextView) findViewById(R.id.
       status_text);
43
44             submitButton.setOnClickListener(this);
45             cancelButton.setOnClickListener(this);
46
47             Intent it = getIntent();
48             user = it.getStringExtra("USER");
49             secret = it.getStringExtra("SECRET");
50             type = it.getStringExtra("TYPE");
51             counter = it.getStringExtra("COUNTER");
52             mobileDH = it.getStringExtra("MOBILE_DH");
53             nounce = it.getStringExtra("NOUNCE");
54         }
55
56     public void onClick(View view) {
57             if (view == submitButton) {
```

```
58                  if (validatePin()) {
59                          Intent returnIntent = new Intent();
60                          returnIntent.putExtra("PIN_RESULT",
    getEnteredPin());
61                          returnIntent.putExtra("USER", user);
62                          returnIntent.putExtra("SECRET",
    secret);
63                          returnIntent.putExtra("TYPE", type);
64                          returnIntent.putExtra("COUNTER",
    counter);
65                          returnIntent.putExtra("MOBILE_DH",
    mobileDH);
66                          returnIntent.putExtra("NOUNCE",
    nounce);
67                          setResult(RESULT_OK, returnIntent);
68                  finish();
69                  }
70          } else if (view == cancelButton) {
71                  finish();
72          }
73          }
74
75          /**
76           * Returns the entered pin, used to encrypt the
    master secret
77           * @return String
78           */
79          private String getEnteredPin() {
80              String enteredPin = pin.getText().toString();
81              return enteredPin;
82          }
83
84          /**
85           * Returns the confirmation of the pin
86           * @return String
87           */
88          private String getReEnteredPin()
89          {
90                  String reEnteredPin = pinConfirmation.
    getText().toString();
91              return reEnteredPin;
92          }
93
94          /**
95           * Check if both entered pins are equal
96           * @return boolean
97           */
98          private boolean validatePin ()
99          {
100                 try {
101                         if (getEnteredPin().equals(
    getReEnteredPin())) {
```

```
102                                     return true;
103                             } else {
104                                     mStatusText.setText(R.string
    .enter_pin_dont_match);
105                             mStatusText.setTextColor(Color.RED);
106                             return false;
107                             }
108                     } catch (Exception e) {
109                             mStatusText.setText(e.getMessage());
110                         mStatusText.setTextColor(Color.RED);
111                             return false;
112                     }
113
114             }
115     }
```

Code A.4: Classe EnterPinActivity

Esta classe foi desenvolvida inteiramente no trabalho.

```
1   package br.ufsc.labsec.authenticator;
2
3   import android.content.ContentValues;
4   import android.content.Context;
5   import android.database.Cursor;
6   import android.database.DatabaseUtils;
7   import android.database.sqlite.SQLiteDatabase;
8
9   public class HSMDb {
10          private static final String TABLE_NAME = "hsms";
11          static final String ID_COLUMN = "_id";
12          static final String CERTIFICATE_COLUMN = "
    certificate";
13          private static final String PATH = "databases";
14          private static SQLiteDatabase DATABASE = null;
15
16          /**
17           * Initialize the HSM's table creating it if doesn't
     exists
18           *
19           * @param context
20           */
21          static void initialize(Context context) {
22                  if (DATABASE != null) {
23              return;
24              }
25
26              DATABASE = context.openOrCreateDatabase(PATH,
    Context.MODE_PRIVATE, null);
27              String createTableIfNeeded = String.format(
28                                                      "CREATE
    TABLE IF NOT EXISTS %s" +
```

```
29                                                          " (%s
    INTEGER PRIMARY KEY, %s TEXT NOT NULL)",
30                                                          TABLE_NAME
    , ID_COLUMN, CERTIFICATE_COLUMN);
31              DATABASE.execSQL(createTableIfNeeded);
32              insertHSMCertificate();
33          }
34
35          /**
36           * Inserts a default key in db.
37           * In future, just receive by parameter the key.
38           */
39          static void insertHSMCertificate() {
40          ContentValues values = new ContentValues();
41
42          String key = "-----BEGIN CERTIFICATE-----\n" +
43                  "MIIBkTCB+6
    ADAgEBAgEAMA0GCSqGSIb3DQEBCwUAMA8xDTALBgNVBAMTBG5hbWUw\n
    " +
44                  "
    HhcNNzAwMTAxMDAwMDAwWhcNNzIxMDIxMDAwMDAwWjAPMQ0wCwYDVQQDEwRuYW1l
    \n" +
45                  "
    MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDcLSaMcnZjXQ7vZPa/
    e4teMrhw\n" +
46                  "
    WEknP7TlKhKQ2xSrXe5YJXA0XGqPzCbUbSmuqXYx0OB1rOdqC8NAsVWv
    +s0LmeIl\n" +
47                  "/4eyMO93E6N+
    ULw9HfDDX0ZJjhwo2L9MoGs4YulgtNl9uzmM8SbPCEQvoVWnDB6+\n"
    +
48                  "3
    ey4PiolZ8JUiayP4QIDAQABMA0GCSqGSIb3DQEBCwUAA4GBADkBJyVrwtCR8bz3
    \n" +
49                  "mwD7Qx9y/i7/aGGEHWc5bGSDmNUmd/
    TWwnEQHzGHfFgmWLjdr5UblcB7kqzm6LuJ\n" +
50                  "YWImdOOVZoUfO4KJWtLOAOeG5RzrWAR/
    X1x1AEj6hs99uVawUZ8OGpHKSeA/W4Ua\n" +
51                  "8hwjnnqCtMMuBgOf5DqCylKpSANt\n" +
52                  "-----END CERTIFICATE-----";
53
54          values.put(ID_COLUMN, 1);
55          values.put(CERTIFICATE_COLUMN, key);
56
57          int updated = DATABASE.update(TABLE_NAME, values,
58                                        ID_COLUMN + " = " +
    DatabaseUtils.sqlEscapeString("1"), null);
59          if (updated == 0) {
60              DATABASE.insert(TABLE_NAME, null, values);
61          }
62      }
63
```

```
64          /**
65           * Gets the hsm cursor
66           *
67           * @param id the hsm's id
68           *
69           * @return the hsm cursor
70           */
71      static Cursor getHSMCursor(String id) {
72              return DATABASE.query(TABLE_NAME, null,
    ID_COLUMN + "= ?", new String[] {id}, null, null, null);
73      }
74
75      /**
76       * Uses the cursor to get the hsm key
77       *
78       * @param id the hsm's id
79       *
80       * @return the hsm public key
81       */
82      static String getCertificate(int id) {
83              Cursor cursor = getHSMCursor(id+"");
84              try {
85              if (!cursorIsEmpty(cursor)) {
86                  cursor.moveToFirst();
87                  return cursor.getString(cursor.
    getColumnIndex(CERTIFICATE_COLUMN));
88              }
89              } finally {
90              tryCloseCursor(cursor);
91              }
92              return null;
93      }
94
95      /**
96       * Returns true if the cursor is null, or contains no
    rows.
97       */
98      public static boolean cursorIsEmpty(Cursor c) {
99          return c == null || c.getCount() == 0;
100     }
101
102     /**
103      * Closes the cursor if it is not null and not closed.
104      */
105     public static void tryCloseCursor(Cursor c) {
106         if (c != null && !c.isClosed()) {
107             c.close();
108         }
109     }
110 }
```

Code A.5: Classe HSMDb

Esta classe foi desenvolvida inteiramente no trabalho.

```
1  // Copyright (C) 2009 Google Inc.
2
3  package br.ufsc.labsec.authenticator;
4
5  import br.ufsc.labsec.authenticator.R;
6  import br.ufsc.labsec.authenticator.AccountDb.OtpType;
7  import android.app.AlertDialog;
8  import android.content.Context;
9  import android.content.DialogInterface;
10 import android.os.Handler;
11 import android.text.InputType;
12 import android.view.LayoutInflater;
13 import android.view.View;
14 import android.view.View.OnClickListener;
15 import android.view.ViewGroup;
16 import android.widget.ArrayAdapter;
17 import android.widget.Button;
18 import android.widget.EditText;
19 import android.widget.TextView;
20
21 /**
22  * A tuple of user, OTP value, and type, that represents a
       particular user.
23  *
24  * @author adhintz@google.com (Drew Hintz)
25  */
26 class PinInfo {
27         public String mOtp; // calculated OTP, or a
       placeholder if not calculated
28         public String mUser;
29         public String mSecret;
30         public boolean mIsHotp = false; // used to see if
       button "Get code" needs to be displayed
31         public boolean mIsLabSEC = false;// used to see if
       button "Activate" needs to be displayed
32 }
33
34 /**
35  * Displays the list of users and the current OTP values.
36  */
37 public class PinListAdapter extends ArrayAdapter<PinInfo> {
38         public static final float SCALEX_NORMAL = (float)
       1.0;
39         public static final float SCALEX_UNDERSCORE = (float
       ) 0.87;
40         private AuthenticatorActivity mContext;
41
42         public PinListAdapter(Context context, int userRowId
       , PinInfo[] items) {
43                 super(context, userRowId, items);
```

```
44                  mContext = (AuthenticatorActivity) context;
45              }
46
47              /**
48               * Displays the user and OTP for the specified
        position. If HOTP, creates
49               * button for generating the next OTP value.
50               */
51              @Override
52              public View getView(int position, View convertView,
        ViewGroup parent) {
53                  LayoutInflater inflater = mContext.
        getLayoutInflater();
54                  PinInfo currentPin = getItem(position);
55
56                  View row;
57                  if (AuthenticatorActivity.
        mAccessibilityAvailable) {
58                      row = inflater.inflate(R.layout.
        user_row, null);
59                  } else {
60                      row = inflater.inflate(R.layout.
        user_row_legacy, null);
61                  }
62                  TextView pinView = (TextView) row.
        findViewById(R.id.pin_value);
63                  TextView userView = (TextView) row.
        findViewById(R.id.current_user);
64                  Button buttonView = (Button) row.
        findViewById(R.id.next_otp);
65                  Button activate = (Button) row.findViewById(
        R.id.activate_labsec);//used in secure master secret
        accounts
66
67                  if (currentPin.mIsLabSEC && (currentPin.
        mSecret == null || currentPin.mSecret.length() == 0 ||
        currentPin.mOtp.equals(mContext.getString(R.string.
        empty_pin)))) {
68                      buttonView.setVisibility(View.GONE);
69                      activate.setVisibility(View.VISIBLE)
        ;
70                      ((ViewGroup) row)
71              .setDescendantFocusability(ViewGroup.
        FOCUS_BLOCK_DESCENDANTS);
72                      OnButtonClickListener clickListener
        = new OnButtonClickListener(
73
        , row, position, false);
74                      activate.setOnClickListener(
        clickListener);
75                      row.setTag(clickListener);
76                  } else if (currentPin.mIsHotp) {
```

```
77                          boolean hasSecret = false;
78                          if(currentPin.mSecret != null) {
79                                  hasSecret = true;
80                          }
81                          activate.setVisibility(View.GONE);
82                          buttonView.setVisibility(View.
    VISIBLE);
83                          ((ViewGroup) row)
84                  .setDescendantFocusability(ViewGroup.
    FOCUS_BLOCK_DESCENDANTS); // makes long press work
85                          OnButtonClickListener clickListener
    = new OnButtonClickListener(
86
    , row, position, hasSecret);
87                          buttonView.setOnClickListener(
    clickListener);
88                          row.setTag(clickListener);
89                  } else { // TOTP, so no button needed
90                          buttonView.setVisibility(View.GONE);
91                          activate.setVisibility(View.GONE);
92                  }
93
94                  if (mContext.getString(R.string.empty_pin).
    equals(currentPin.mOtp)) {
95                          pinView.setTextScaleX(
    SCALEX_UNDERSCORE); // smaller gap between underscores
96                  } else {
97                          pinView.setTextScaleX(SCALEX_NORMAL)
    ;
98                  }
99
100                 pinView.setText(currentPin.mOtp);
101                 userView.setText(currentPin.mUser);
102
103                 return row;
104         }
105 }
106
107 /**
108  * Listener for the Button that generates the next OTP value
        .
109  */
110 class OnButtonClickListener implements OnClickListener {
111         private static final long NEXT_OTP_TIMEOUT_MS =
    5000;
112         private AuthenticatorActivity mContext;
113         private boolean mHasSecret;
114         private View mRow;
115         private int mPosition;
116         public Handler mHandler = new Handler();
117         private Runnable mEnableButton = new Runnable() {
118     public void run() {
```

```
119     Button nextOtp = (Button) mRow.findViewById(R.id.
        next_otp);
120     nextOtp.setEnabled(true);
121 }
122 };
123
124 OnButtonClickListener(AuthenticatorActivity context, View
        row, int position, boolean hasSecret) {
125 mContext = context;
126 mRow = row;
127 mPosition = position;
128 mHasSecret = hasSecret;
129 }
130
131 /**
132  * {@inheritDoc}
133  */
134 public void onClick(View v) {
135 final TextView userView = (TextView) mRow
136 .findViewById(R.id.current_user);
137 final TextView pinView = (TextView) mRow.findViewById(R.id.
        pin_value);
138 Button nextOtp = (Button) mRow.findViewById(R.id.next_otp);
139 nextOtp.setEnabled(false);
140 mHandler.postDelayed(mEnableButton, NEXT_OTP_TIMEOUT_MS);
141 String user = (String) userView.getText();
142 final OtpType type = AccountDb.getType(user);;
143
144 if (AccountDb.hasPin(user) == 1 && mHasSecret == false) {
145 AlertDialog.Builder alert = new AlertDialog.Builder(mContext
        );
146 alert.setMessage("Enter Pin :");
147
148 // Set an EditText view to get user input
149 final EditText input = new EditText(mContext);
150 input.setInputType(InputType.TYPE_CLASS_TEXT | InputType.
        TYPE_TEXT_VARIATION_PASSWORD);
151 alert.setView(input);
152
153 alert.setPositiveButton("Ok",
154 new DialogInterface.OnClickListener() {
155 public void onClick(DialogInterface dialog,
156 int whichButton) {
157
158 String pin = input.getText().toString();
159 String otp = null;
160 if(type == OtpType.HOTP)
161 otp = mContext.computeAndDisplayOtp((String) userView.
        getText(), mPosition, true, pin);
162 else
163 otp = mContext.computeAndDisplayOtp((String) userView.
        getText(), mPosition, false, pin);
```

```
164
165 pinView.setText(otp);
166 pinView.setTextScaleX(PinListAdapter.SCALEX_NORMAL);
167
168 return;
169 }
170 });
171 alert.setNegativeButton("Cancel", new
172 DialogInterface.OnClickListener() {
173 public void onClick(DialogInterface dialog, int which) {
174 return;
175 }
176 });
177 alert.show();
178 } else {
179 String otp = null;
180 if (type == OtpType.HOTP) {
181 otp = mContext.computeAndDisplayOtp(user, mPosition, true,
        null);
182 } else {
183 otp = mContext.computeAndDisplayOtp(user, mPosition, false,
        null);
184 }
185 pinView.setText(otp);
186 pinView.setTextScaleX(PinListAdapter.SCALEX_NORMAL); //
        adjust to display numbers
187 }
188 }
189
190 }
```

Code A.6: Classe PinListAdapter

Nesta classe foram feitas algumas alterações, a primeira delas foi a adição do atributo da linha 31, que informa se o usuário listado possui a *master secret* cifrada. Caso seja, é mostrado um botão de ativação para entrada do *pin*, criado e exibido à partir da linha 65, até o final do método. Da linha 144 até o final do bloco acontece o tratamento do botão de ativação, que mostra um campo para a entrada do *pin* e a chamada dos métodos responsáveis pela geração do OTP.

ANEXO B – Artigo

Senhas descartáveis em dispositivos móveis para ambientes de Telemedicina

Thaís Bardini Idalino[1] **and Dayana Pierina Brustolin Spagnuelo**[1]

[1] Universidade Federal de Santa Catarina
Departamento de Informática e de Estatística
Campus Universitário - Florianópolis - SC - Brazil
{thais.idalino,dayspagnuelo}@inf.ufsc.br

Resumo. *Este trabalho é parte de um projeto que está sendo desenvolvido no Laboratório de Segurança em Computação [LabSEC] da Universidade Federal de Santa Catarina [UFSC] em parceria com o Laboratório de Informática Médica e Telemedicina [LabTelemed] financiado pela Financiadora de Estudos e Projetos [FINEP]. Tem como objetivo implantar o uso de senhas descartáveis na Rede Catarinense de Telemedicina, através de uma versão melhorada do Google Authenticator e do uso do ASI-HSM como servidor de autenticação. Para a implantação na Telemedicina, propõe-se o uso de um serviço de autenticação desenvolvido também no LabSEC para abstrair a camada do HSM.*

Abstract. *This paper is part of a project that is being developed at the Laboratory of Computer Security (LabSEC) of the Federal University of Santa Catarina (UFSC) in association with the Laboratory of Medical Informatics and Telemedicine (LabTelemed) and sponsored by the Financier of Studies and Projects (FINEP). It aims to deploy the use of One-Time Passwords in the Santa Catarina Telemedicine Network, through an improved version of Google Authenticator and the use of the ASI-HSM as an authentication server. For the deployment in the Telemedicine environment, we propose the use of a authentication service developed at LabSEC to abstract the HSM's layer.*

1. Introdução

A utilização de senhas é algo que está presente no dia-a-dia da maioria das pessoas, desde senhas para efetuar transações bancárias, até senhas para acesso à sistemas mais simples ou redes sociais. Elas são a barreira mais utilizada em sistemas que precisam de alguma forma de autenticação e autorização de acesso, pelo fato de serem fáceis de aplicar tanto em nível de implementação quanto de usabilidade.

No ambiente de Telemedicina, a autenticação de um médico perante o sistema tem uma importância ainda maior. Ao provar sua identidade, o profissional tem em mãos ferramentas capazes de emitir laudos e analisar exames de pacientes, auxiliando principalmente os que residem em lugares distantes. Tais ferramentas em mãos erradas, podem apresentar risco à saúde dos pacientes pela possibilidade de emissão de

laudos falsos. Também é possível expor dados que seriam de interesse apenas dos pacientes e seus respectivos médicos.

O fato de utilizar um *login* e senha para se autenticar no sistema não garante que quem os está inserindo é realmente a pessoa autorizada ao acesso. Diversos tipos de ameaças existem atualmente, dentre eles estão os ataques capazes de capturar as senhas que estão transitando pela rede, ou, de acordo com [da Silva 2007], até mesmo o elo mais fraco nesse mecanismo de autenticação: o próprio usuário. As pessoas tendem a escolher senhas fáceis e/ou utilizar a mesma para se autenticar em diferentes locais. Isso as torna vulneráveis, pois são fáceis de adivinhar e uma vez descobertas, podem comprometer uma quantidade imensurável de informações e dados.

Uma das maneiras de contornar esse problema é através da autenticação baseada em múltiplos fatores, que vem se tornando popular e que tem fácil implantação. Neste modelo, não apenas *login* e senha são utilizados, mas também a apresentação de um segundo fator de autenticação que pode ser desde um objeto de posse única do usuário, até características biométricas. Entre os mecanismos de autenticação de múltiplos fatores está o de senhas descartáveis (do inglês *One-Time Passwords*, OTP), apresentado como alternativa principalmente em sistemas bancários, mas também nas contas de usuário do Google e Facebook.

Este trabalho propõe a implantação de senhas descartáveis na autenticação do sistema da Rede Catarinense de Telemedicina (RCTM), utilizando como base a implementação já existente do Google, o Google Authenticator [Google Inc. 2012]. O restante do trabalho está organizado da seguinte maneira: Uma breve discussão da importância da segurança na autenticação em ambientes de telemedicina é discutida na seção 2. Na seção 3 é apresentada a proposta do trabalho, com foco no gerador de senhas descartáveis na subseção 3.1 e no servidor de autenticação na 3.2. Na seção 4 é apresentada a integração entre o sistema de telemedicina e a proposta sugerida. Por fim, na seção 5 são apresentadas as considerações finais e proposta de trabalhos futuros.

2. Segurança em ambientes de Telemedicina

A RCTM tem por finalidade facilitar o acesso a serviços de saúde por pacientes e médicos, como o acesso e emissão de exames e laudos, fazendo o uso da Internet. Dessa maneira, o paciente pode fazer um exame em sua cidade e enviá-lo pela rede a um médico especializado, este por sua vez pode analisá-lo e emitir o laudo de qualquer lugar que esteja [Cyclops 2010].

A RCTM pode ser caracterizada como um sistema de caráter social, cujas principais vantagens estão na acessibilidade e facilidade de assistência da população em geral. Segundo [Nakamura 2007], com a evolução constante e surgimento de novas tecnologias, as pessoas ganham em acessibilidade, produtividade e diversas facilidades, mas também os riscos devem ser avaliados e minimizados. Como apresentado em [Magalhães and Santos 2003], os problemas de segurança tornam-se ainda mais evidentes com o uso cada vez maior dessas tecnologias e, dentre estes problemas, tem-se

como destaque o da autenticação.

Ainda de acordo com [Magalhães and Santos 2003], a segurança no processo de autenticação é uma questão fundamental, já que o acesso indevido à informações pode ser de grande risco. De acordo com [FFIEC 2005] o roubo de identidade é frequentemente encontrado como o resultado de uma autenticação utilizando um único fator (utilizando apenas um *login* e senha por exemplo), afirmando que esse modo de autenticação é inadequado. No ambiente de Telemedicina, o impacto de um ataque desta natureza é ainda maior, pela sensibilidade dos dados envolvidos. Se um indivíduo mal intencionado conseguir acesso às senhas de pacientes ou médicos, poderia desde visualizar informações particulares, como resultados de exames, até emitir laudos falsos, apresentando um risco à vida dos pacientes.

Tendo em vista estes ataques, acredita-se que a adoção de um processo de autenticação mais seguro neste ambiente pode ser uma boa solução para o problema. Através da garantia de identidade dos usuários que acessam o sistema, pode-se reduzir as chances de sucesso em ataques desta natureza, ou até extinguí-las.

2.1. Trabalhos Correlatos

De acordo com Haller et al [Haller et al. 1998], a ideia por trás do OTP foi inicialmente proposta por Leslie Lamport em seu artigo de 1981 [Lamport 1981], que também inspirou a criação do sistema de S/KEY [Haller 1995], no qual o OTP foi derivado.

Hoje em dia sua utilização está em grande evidência em ambientes bancários, como o Bradesco, por exemplo, que utiliza um tipo de OTP impresso em papel, contendo 70 senhas descartáveis que devem ser apresentadas juntamente com a senha do usuário para autenticação [Banco Bradesco S.A. 2012].

O *RSA SecurID Authenticator* [RSA Security Inc 2010] é outro exemplo. Ele utiliza um token que pode ser tanto por hardware (como pequenos dispositivos que podem ser carregados no bolso) quanto por software (disponíveis para celulares, para Windows e Mac OSX). Este token possui uma senha semente e gera o OTP através de cálculos matemáticos baseados nessa semente e num *timestamp* sincronizado com o servidor de autenticação. A cada 60 segundos uma nova senha é gerada, e para a autenticação, o usuário precisa entrar com seu *login*, sua senha e o OTP concatenado à ela.

Diversos trabalhos sobre a utilização de múltiplos fatores de autenticação em ambientes de Telemedicina foram encontrados, porém nenhum contribui de forma significativa para este trabalho.

3. Proposta

Com base na importância dos dados armazenados na RCTM, propõe-se a adoção de um modelo de autenticação que apresente maior segurança. Como citado em [FFIEC 2005], métodos de autenticação baseados em múltiplos fatores são mais difíceis de comprometer e mais confiáveis. Neste trabalho, será utilizado como segundo fator de autenticação as senhas descartáveis (do inglês, *One-Time Passwords*,

OTP), onde a apresentação destas prova que o usuário além de possuir seu *login* e senha, também tem posse do dispositivo que as gera. O aplicativo de geração de senhas descartáveis utilizado nesse trabalho será baseado na implementação do Google Authenticator [Google Inc. 2012]. Nas subseções seguintes descreve-se as principais tecnologias envolvidas e o desenvolvimento da proposta.

3.1. Gerador de senhas

O gerador de senhas descartáveis Google Authenticator [Google Inc. 2012] está disponível para as plataformas Android, iOS e Blackberry e é utilizada para uma autenticação mais forte nas contas da empresa. Nesta implementação, para que o cliente e o servidor de autenticação possam gerar as mesmas senhas e de maneira independente, eles precisam passar por um processo de inicialização. Neste processo faz-se um acordo de uma senha inicial (conhecida como *master secret*) entre eles, e a partir dela são capazes de gerar as mesmas senhas descartáveis isoladamente.

O aplicativo suporta a implementação de OTP com parâmetros baseados em contador (*HMAC-Based One Time Password*, HOTP) [M'Raihi et al. 2005] e em tempo (*Time-based One Time Password*, TOTP) [M'Raihi et al. 2011]. Ela está em conformidade com as respectivas RFCs, qualquer alteração nas mesmas por avanços na criptoanálise por exemplo, pode também ser alterada no aplicativo. Ao se utilizar o parâmetro de tempo, o relógio do dispositivo deve ser sincronizado com o do servidor. Já com um parâmetro baseado em contador, o mesmo recebe um valor inicial, geralmente zero. Este contador é incrementado à cada geração de senha no dispositivo do usuário e à cada autenticação realizada com sucesso no servidor. Em ambos os casos há a possibilidade de cliente e servidor estarem dessincronizados, neste caso é previsto que o servidor adote uma política que possua uma janela de ressincronização, gerando uma sequência de OTPs anteriores e posteriores, se certificando assim em que posição o cliente se encontra para sincronizar-se à ele.

Para utilizar o aplicativo, o usuário instala o mesmo em seu celular e realiza o cadastro de uma conta, inserindo seus dados e a *master secret*. Esta pode ser fornecida manualmente, ou obtida através da leitura de um código de barras em 2D, conhecido como *QR Code*. O acordo desta senha é feito pelo browser, e o usuário a recebe direto na tela do computador, em texto claro caso tenha sido escolhida a primeira opção, e em forma de imagem para a segunda.

De acordo com [Cheng 2011], geradores de senhas descartáveis em software armazenam a *master secret* no próprio dispositivo, e se copiada por um indivíduo malicioso, pode comprometer a segurança do sistema. Com base na análise do código do aplicativo Google Authenticator, pode-se observar uma possível vulnerabilidade proveniente do processo de armazenamento da *master secret*, tendo em vista que além de armazená-la no *smartphone* do usuário, o mesmo é feito em claro no seu banco de dados. A *master secret* pode ser considerada a base de todo o processo de geração das senhas descartáveis e apenas o servidor e o usuário devem detê-la. Entretanto, se um indivíduo mal intencionado obter acesso ao *smartphone* do usuário original e extraí-la

do seu banco de dados, ele pode ser capaz de gerar as mesmas senhas e se passar por esse usuário no processo de autenticação.

Tendo em vista este problema, uma solução sugerida é a de armazenar essa *master secret* cifrada com um PIN (*Personal Identification Number*), sendo este uma senha que apenas o usuário conheça. Dessa maneira, além de ter acesso ao celular do usuário e extrair a *master secret*, o indivíduo mal intencionado deveria também ter conhecimento do PIN, caso contrário de nada adiantaria tê-la. Para evitar que o PIN seja extraído do banco de dados, este não fica armazenado no *smartphone*. Sua validação é dada unicamente pelo sucesso ou fracasso na decifragem da *master secret*, quando o usuário o fornece na aplicação.

Outra alternativa proposta visa garantir que a senha descartável foi gerada única e exclusivamente no *smartphone* para o qual foi destinada. Para isto, propõe-se a modificação do algoritmo de geração de OTPs para levar em consideração não só a *master secret* e um dos parâmetros (de tempo ou contador), mas também incluir em sua geração a Identificação Internacional de Equipamento Móvel (do inglês, *International Mobile Equipment Identity*, IMEI), que é um valor único que identifica o dispositivo. Com isso, de nada adiantaria um indivíduo mal intencionado obter a *master secret* do usuário, já que o dispositivo em que ele geraria os OTPs não possuiria o mesmo IMEI. Esta alternativa também impede a clonagem do aplicativo, uma vez que a obtenção do IMEI é diretamente dependente dos métodos disponibilizados pela plataforma do dispositivo. No entanto, se um indivíduo mal intencionado conseguir obtê-lo por meio de outras ferramentas ou aplicativos, ainda seria necessário obter o PIN para decifrar a *master secret*.

3.2. Servidor de autenticação

Em modelos de autenticação baseados em OTP, proteger a *master secret* contra roubos não significa somente proteger o gerador, pois existe uma cópia da mesma no servidor de autenticação. Além disto, é um requisito comum a todos os modelos de autenticação que a máquina responsável pelo armazenamento das credenciais dos usuários seja confiável. Desta forma este trabalho também propõe a troca do servidor de autenticação por um Módulo de Segurança Criptográfico (do inglês, *Hardware Security Module*, HSM), para torná-lo mais resistente a ataques.

HSMs são dispositivos usados principalmente em aplicações que requerem um maior rigor no armazenamento e processamento de informações sensíveis, como por exemplo em aplicações militares e bancárias. Normalmente são usados para o armazenamento de chaves criptográficas, processamento de autenticação de usuários, execução de funções criptográficas, entre outros [de Souza 2008].

Para o desenvolvimento deste trabalho, foi utilizado o ASI-HSM. De acordo com [de Souza 2008], ele é um módulo de segurança criptográfica com tecnologia brasileira, de código aberto e de menor custo. Foi patrocinado pelo GT ICPEDU II e projetado pela empresa brasileira Kryptus [Kryptus] em parceria com a Rede Nacional de Ensino e Pesquisa [RNP]. O software responsável pela gerência do ASI-HSM, o

OpenHSMd [OpenHSMd], foi desenvolvido pelo LabSEC.

Por se tratar de um HSM, este já possui barreiras contra os principais ataques sendo, portanto, menos vulnerável se comparado a um servidor simples. O ASI-HSM utiliza o OpenHSMd, que é um *firmware* voltado à gerência do ciclo de vida de chaves criptográficas, o qual precisaria ser completamente modificado para atender às necessidades do modelo proposto. Porém uma modificação no OpenHSMd não faria sentido, uma vez que seu propósito inicial pouco tem a ver com autenticações baseadas em OTP. Além do mais, isso aumentaria a interface de interação com o HSM, aumentando assim a vulnerabilidade do mesmo, uma vez que se aumenta a variedade de ataques aplicáveis.

A melhor alternativa encontrada foi a criação de um novo *firmware* específico para o modelo. Este possui interface somente para duas ações: criação de usuários e autenticação. Para a criação de usuários, utiliza-se um identificador único do usuário, o IMEI do seu *smartphone* e o tipo de OTP escolhido por ele e retorna a *master secret* já computada. A autenticação de usuários se dá pelo fornecimento do identificador do usuário e do OTP do mesmo e retorna somente se este é um OTP válido ou não. Desta forma o novo firmware possui uma interface bastante restrita.

O fato de todas as comunicações serem feitas através de um túnel seguro (*Secure Sockets Layer*, SSL) e a interface ser bastante restrita aumenta a dificuldade de um atacante obter sucesso em uma de suas tentativas. Além disto, se ainda assim um atacante conseguir penetrar o HSM, as *master secrets* estão cifradas pela chave do HSM, fazendo com que seja necessário obter-se também a chave privada deste, do contrário somente se obteria texto ilegível.

4. Integração com o Framework de Autenticação

Para que seja possível a implantação de um sistema mais seguro de autenticação na RCTM, foi desenvolvido um serviço de autenticação que serve como uma interface entre o HSM e o sistema. Este serviço funciona como um *web service* que tem o propósito de ser genérico, independente de linguagem e abstrair a camada do HSM. O mesmo provê diversos métodos de autenticação que podem ser utilizados como um fator adicional, e dentre estes métodos, está o de senhas descartáveis proposto neste trabalho. O framework é flexível, de maneira a não impedir que um médico emita um lado caso esqueça seu *smartphone* em casa, por exemplo.

O sistema de Telemedicina é o responsável por prover a interface gráfica e ele se comunica diretamente com o serviço para solicitar a autenticação e operações gerenciais. O sistema repassa os dados provenientes do usuário ou dele próprio para o serviço, que os manipula e solicita operações no HSM quando necessário.

Para o cadastro de um usuário que deseja utilizar OTP como um segundo fator de autenticação, o mesmo precisa entrar com seus dados no sistema de Telemedicina, escolhendo o tipo de OTP (contador ou tempo) que deseja utilizar. A Telemedicina por sua vez solicitará ao serviço a vinculação desse usuário com este tipo de

autenticação. O serviço irá solicitar ao HSM a criação de um usuário de OTP, passando como parâmetro um identificador do usuário, o IMEI fornecido pelo mesmo e o tipo de OTP. O HSM calcula a *master secret* e a retorna ao serviço. Este finaliza o processo de cadastro do usuário enviando a *master secret* juntamente com o tipo de OTP escolhido pelo usuário e o *username* que ele já utiliza para o sistema de Telemedicina, codificados em um QR Code. A Telemedicina será a responsável por disponibilizá-lo ao usuário pelo *browser*, que através do aplicativo em seu celular pode ler o QR Code e obter todos os dados necessário para a criação de seu usuário no aplicativo, sem precisar entrar com dados, como no modelo original. Por fim, antes de a *master secret* ser salva no banco de dados do celular, é solicitado um PIN ao usuário para cifrá-la. Este processo pode ser visto na Figura 1.

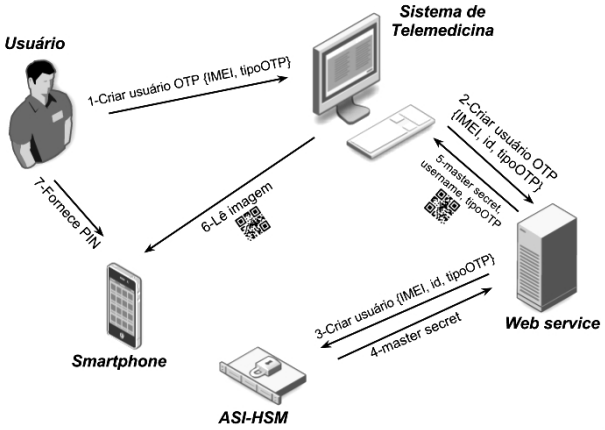

Figura 1. Processo de criação de usuário OTP

Já o processo de autenticação se dá pelo fornecimento do *login*, senha e OTP do usuário através da interface gráfica do serviço de Telemedicina. Para que o usuário obtenha seu OTP, primeiro precisa fornecer o PIN com o qual a *master secret* foi cifrada. A Telemedicina por sua vez repassa os dados do usuário ao serviço de autenticação, que faz a verificação de usuário e senha. Este solicita ao HSM a verificação do usuário e OTP apresentados. Caso todas retornem com sucesso, o usuário é autenticado no sistema. Este processo pode ser visto na Figura 2.

5. Considerações Finais

Neste trabalho foi apresentado um modelo de segundo fator de autenticação baseado em senhas descartáveis e sua respectiva implementação. Esse modelo tem como propósito aumentar a confiabilidade na autenticação do sistema de Telemedicina através do uso de um aplicativo derivado do Google Authenticator. Propõe também o uso do

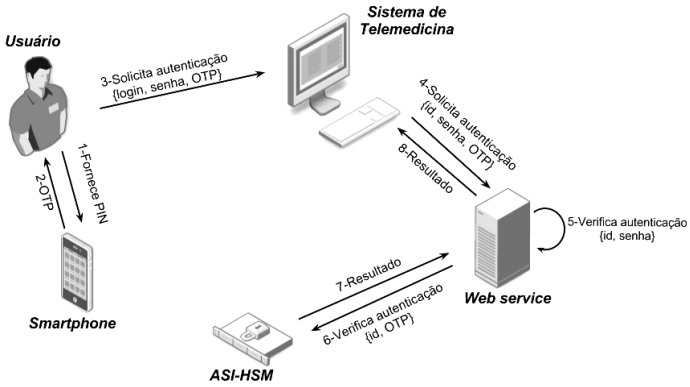

Figura 2. Processo de autenticação de usuário OTP

ASI-HSM com um novo firmware para fazer o papel do servidor de autenticação, e um *web service* que fará a comunicação entre o sistema de Telemedicina e o HSM, além de prover outros meios de autenticação.

Como foi mencionado, o uso de autenticação de apenas um fator compromete o sistema como um todo, principalmente pelos descuidos do usuário na hora de escolher e manipular suas senhas. A autenticação de múltiplos fatores utilizando senhas descartáveis se mostrou uma solução adequada, pelo fato de serem geradas por um aplicativo e de serem de natureza aleatória, não sendo passíveis de ataques de dicionário ou engenharia social. Tendo em vista que cada senha é utilizada apenas uma vez, a interceptação da mesma através da rede deixa de ser um problema. Um atacante que possui uma sequência de senhas com o intuito de construir uma função para geração das próximas, tem tantas chances de obter sucesso quanto um que não a possui, dado que o melhor ataque contra o algoritmo é a força bruta [M'Raihi et al. 2005].

Ainda de acordo com [M'Raihi et al. 2005], análise de segurança pode ser aproximada pela função:

$$Sec = \frac{s.v}{10^{Digit}}$$

Onde **Sec** é a probabilidade de sucesso de um atacante, **s** é o tamanho da janela de ressincronização prevista pelo modelo, **v** é o número de tentativas de autenticação antes do bloqueio do usuário e **Digit** é a quantidade de dígitos da senha OTP.

A implementação apresentada utiliza como gerador de senhas descartáveis o próprio *smartphone* do usuário. Pelo fato de este ser um dispositivo de uso pessoal, os usuários estão habituados com ele e não terão grandes problemas para entender o funcionamento do aplicativo. Além do mais, um ataque através do roubo do dispositivo seria facilmente percebido.

A integração através de um *web service* de autenticação trouxe muitas vantagens para o sistema de Telemedicina, pois abstraiu completamente o uso do HSM, separou o processo de autenticação do de autorização e é independente de linguagem. Este último é muito importante pois torna possível a integração do *web service* com outros sistemas.

Como melhoria do trabalho proposto, sugere-se a utilização de outras características do dispositivo móvel do usuário na geração das senhas descartáveis. Uma alternativa seria a utilização dos dados contidos no seu *smartcard*, aumentando ainda mais a relação entre a senha gerada e o *smartphone*. Outra melhoria consiste em modificar o processo de *setup* da *master secret*, pois da maneira como é implementado hoje, a mesma é mostrada diretamente na tela do computador do usuário. Qualquer um que conseguir obtê-la no período em que ela está exposta pode utilizá-la para gerar as mesmas senhas do usuário original.

Referências

[Banco Bradesco S.A. 2012] Banco Bradesco S.A. (2012). Cartão chave de segurança bradesco. http://www.bradescoseguranca.com.br.

[Cheng 2011] Cheng, F. (2011). Security attack safe mobile and cloud-based one-time password tokens using rubbing encryption algorithm. *Mob. Netw. Appl.*, 16(3):304–336.

[Cyclops 2010] Cyclops (2010). Sistema catarinense de telemedicina e telessaúde. https://www.telemedicina.ufsc.br/rctm.

[da Silva 2007] da Silva, D. R. P. (2007). *A memória humana no uso de senhas*. PhD thesis, Pontifícia Universidade Católica do Rio Grande do Sul.

[de Souza 2008] de Souza, T. C. S. (2008). Aspectos técnicos e teóricos da gestão do ciclo de vida de chaves criptográficas no openhsm. Master's thesis, Universidade Federal de Santa Catarina.

[FFIEC 2005] FFIEC (2005). Authentication in an internet banking environment. http://www.ffiec.gov/press/pr101205.htm.

[FINEP] FINEP. Financiadora de Estudos e Projetos. http://www.finep.gov.br/.

[Google Inc. 2012] Google Inc. (2012). Google authenticator. http://code.google.com/p/google-authenticator.

[Haller 1995] Haller, N. (1995). The S/KEY One-Time Password System. RFC 1760 (Informational).

[Haller et al. 1998] Haller, N., Metz, C., Nesser, P., and Straw, M. (1998). A One-Time Password System. RFC 2289 (Standard).

[Kryptus] Kryptus. Kryptus. http://kryptus.com.br/.

[LabSEC] LabSEC. Laboratório de Segurança em Computação. http://www.labsec.ufsc.br/.

[LabTelemed] LabTelemed. Laboratório de Informática Médica e Telemedicina. http://www.telemedicina.ufsc.br/joomla/.

[Lamport 1981] Lamport, L. (1981). Password authentication with insecure communication. *Commun. ACM*, 24(11):770–772.

[Magalhães and Santos 2003] Magalhães, P. S. and Santos, H. D. (2003). Biometria e autenticação. In *Actas da 4a Conferência da Associação Portuguesa de Sistemas de Informação*, pages 2 – 5, Porto. Portugal.

[M'Raihi et al. 2005] M'Raihi, D., Bellare, M., Hoornaert, F., Naccache, D., and Ranen, O. (2005). HOTP: An HMAC-Based One-Time Password Algorithm. RFC 4226 (Informational).

[M'Raihi et al. 2011] M'Raihi, D., Machani, S., Pei, M., and Rydell, J. (2011). TOTP: Time-Based One-Time Password Algorithm. RFC 6238 (Informational).

[Nakamura 2007] Nakamura, E. T. (2007). *Segurança de redes em ambientes corporativos*. Novatec Editora Ltda., São Paulo, SP - Brasil.

[OpenHSMd] OpenHSMd. OpenHSMd. https://projetos.labsec.ufsc.br/openhsmd.

[RNP] RNP. Rede Nacional de Ensino e Pesquisa. http://www.rnp.br/.

[RSA Security Inc 2010] RSA Security Inc (2010). *RSA SecurID Two-factor Authentication*. RSA Security Inc.

[UFSC] UFSC. Universidade Federal de Santa Catarina. http://ufsc.br/.