UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA'

Marcos Schead dos Santos

PROPOSTA DE MIDDLEWARE WEB PARA AUTENTICAÇÃO CONTÍNUA MULTI-FATOR

Florianópolis

2018

Marcos Schead dos Santos

PROPOSTA DE MIDDLEWARE WEB PARA AUTENTICAÇÃO CONTÍNUA MULTI-FATOR

Trabalho de Conclusão de Curso submetido ao Programa de Graduação em Ciências da Computação para a obtenção do Grau de Bacharel em Ciências da Computação.

Florianópolis

2018

Catalogação na fonte elaborada pela biblioteca da
Universidade Federal de Santa Catarina

A ficha catalográfica é confeccionada pela Biblioteca Central.

Tamanho: 7cm x 12 cm

Fonte: Times New Roman 9,5

Maiores informações em:
http://www.bu.ufsc.br/design/Catalogacao.html

Marcos Schead dos Santos

PROPOSTA DE MIDDLEWARE WEB PARA AUTENTICAÇÃO CONTÍNUA MULTI-FATOR

Este Trabalho de Conclusão de Curso foi julgado aprovado para a obtenção do Título de "Bacharel em Ciências da Computação", e aprovado em sua forma final pelo Programa de Graduação em Ciências da Computação.

Florianópolis, 21 de novembro 2018.

Prof. Dr. Renato Cislaghi
Coordenador de Projetos

Banca Examinadora:

Prof. Dr. Jean Everson Martina

Prof. Dr. Martín Vigil

Prof. Dr. Douglas Silva

Dedico este trabalho à minha família, que sempre esteve ao meu lado nas mais diversas nuances desta vida

AGRADECIMENTOS

Agradeço ao meu orientador e membros da banca por me guiar no processo de desenvolvimento deste trabalho. Um agradecimento mais que especial ao Vinicius Weisheimer pela ajuda na criação da infra-estrutura utilizada.

À todos os amigos e colaboradores do Laboratório Bridge que proporcionaram um ambiente de crescimento e amizade durante boa parte da minha graduação.

You can have it all but how much do you want it?

Noel Gallagher

RESUMO

A autenticação é o processo de verificar a identidade de uma pessoa, de modo que seu acesso ao sistema seja permitido ou negado. Um usuário nunca usa um sistema com a finalidade de autenticar-se, mas sim para exercer as funcionalidades inerentes à aplicação. Embora necessária, torna-se um incômodo comum, pois quebra o fluxo principal das aplicações. Existem mecanismos que autenticam o usuário continuamente gerando um valor que quantifica a certeza da identidade do usuário. Eles podem ser obtidos, por exemplo, usando reconhecimento facial ou padrões comportamentais de digitação. No entanto, uma questão em aberto é como utilizar mais de um valor contínuo simultaneamente para melhorar a autenticação. A proposta desse trabalho é apresentar e implementar uma arquitetura, onde um ou mais dispositivos possam gerar distintos valores periodicamente, e associá-los a um usuário. Os serviços que estiverem interessados nesses valores de autenticação contínuos daquele indivíduo, serão notificados sempre que ocorrer uma mudança neles. Assim, os serviços podem tomar suas próprias decisões de negócio, com base nos novos valores obtidos. Por meio de um aplicativo de celular que representa um serviço interessado foi feita a validação dos artefatos criados.

Palavras-chave: autenticação, contínua, multi-fator, middleware

ABSTRACT

Authentication is the process of verifying a person's identity, allowing or not its access. When using an application, an user wants to produce value for his business, not authenticate itself. Although necessary, it becomes uncomfortable because it breaks the main flow usage. There are mechanisms to authenticate the user continuously, generating reliability identity values. These values can be obtained by face recognition or keystroke dynamics for instance. However, an open issue is how to use more than one value at the same time to improve the authentication. The main purpose of this work is present and implement an architecture, that one or more devices are able to generate authentication values time by time and associate it to an user. The services that want to get these authentication values will be notified when something changes. Then all services could make their domain decisions upon these new values. Through a smart phone app acting as a service all the artifacts validation were done.

Keywords: authentication, continuous , multi-factor, middleware

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

SUMÁRIO

1 INTRODUÇÃO

1.1 MOTIVAÇÃO

A autenticação não é uma funcionalidade com o intuito de produzir algo. O uso do sistema tem valor para o usuário na criação do seu domínio de trabalho, independente do tipo de software que se está operando. É algo que precisa ser feito, pois existem pessoas com más intenções que podem beneficiar-se de forma indevida. Assim, esse processo existe para garantir que o sistema está nas mãos de alguém confiável.

Entretanto, ninguém gosta de ser incomodado com uma pergunta recorrente confirmando as credenciais de acesso. Se existisse apenas um sistema, isso não seria problema. Porém, no contexto atual em que diversos serviços aparecem para o dia, é inviável gerenciar muitas senhas diferentes e difíceis de serem quebradas. Principalmente com a expansão dos smartphones e tablets, que criaram um vasto mercado de aplicativos, armazenando informações cada vez mais específicas. Na prática, duas ou três credenciais novas são criadas e seu uso é distribuído entre os vários aplicativos, sites e entre outros. Isso abre uma vulnerabilidade grande em suas informações, caso qualquer uma das senhas sejam descobertas.

Há casos em que a autenticação é extremamente necessária e inclusive, pode necessitar de diversos níveis para garantir a segurança da informação. Em situações que existem documentos sigilosos e de cunho militar, por exemplo, deve-se ter senhas fortes para evitar o acesso não autorizado. No cotidiano, o que os usuários desejam é usufruir os serviços e serem importunados o mínimo possível. Em poucos momentos, de preferência com um intervalo grande, uma senha pode ser exigida, para garantir a identidade ou para evitar o acesso não-autorizado à certos recursos.

Existem formas de validar a identidade do usuário. A seguir é apresentado cinco categorias de autenticação. O uso de login e senha é apenas uma delas, que está relacionado ao que o usuário sabe. Pode ser uma palavra chave ou uma informação, essa é uma das categorias de autenticação. A segunda forma está vinculada ao que ele tem: um cartão, um pendrive, algo físico necessário para o reconhecimento da identidade. A terceira é por algo que ele é: uma digital, reconhecimento de íris, algum padrão de comportamento, ou seja, algo que possa identificar o indivíduo. A quarta categoria envolve o local do indivíduo. Por

exemplo, um serviço oferecido pela universidade, só pode ser acessível dentro dela, qualquer acesso externo não é permitido. Isso pode ser estendido para o tempo, que é a quinta forma. Um serviço só pode ser acessado em um certo horário pré-estabelecido. A última forma seria por testemunha, em que um terceiro garante a identidade de uma pessoa para o sistema, permitindo então o acesso dela.

Algumas dessas categorias podem conter serviços de autenticação contínua. Um autenticador contínuo verifica num intervalo curto de tempo, a identidade do usuário. Em alguns casos, pode ser retornado um valor, representando uma porcentagem de certeza, ou um simplismente um valor de sim e não. Um dispositivo verificar a cada 1 minuto a íris do usuário, pode gerar um valor desse tipo, enquanto que uma autenticação temporal, apenas informa sim ou não.

Conhecendo as categorias citadas acima, cada uma com suas peculiaridades, pode não ser interessante utilizar apenas e somente uma. Em alguns casos, pode ser interessante avaliar mais de uma e tomar decisões com base num conjunto delas. Isso garante uma confiabilidade maior, visto que fontes diferentes são consultadas.

Com isso, vem as seguintes perguntas: é possível agregar diferentes formas de autenticação contínua que perturbe o mínimo possível o usuário e melhore a confiança no sistema? É possível ter uma arquitetura que forneça isso à diversos sistemas diferentes?

1.2 JUSTIFICATIVAS

Visto que a autenticação contínua é uma abordagem que possui vários estudos na literatura, verificar-se-á como pode ser agregado diferentes autenticadores. Juntos podem ser vistos como um terceiro fator de autenticação. Quando um serviço for utilizado por alguém, essa pessoa possui um conjunto de dispositivos. Cada dispositivo gera um valor de autenticação, denominado NDC (Nível de confiabilidade). O NDC informa a certeza de que o usuário do serviço, é realmente quem diz ser. Agregando os níveis, é possível fortalecer o acesso correto.

Alguns dos trabalhos pesquisados geram de alguma forma, uma variável própria similar ao NDC. No entanto, essas autenticações são todas locais ao dispositivo, ou seja, somente o computador utilizado detém o nível. A proposta, é que seja possível criar um middleware, em que vários serviços diferentes, possam consumir o NDC. Além disso, pensando num fluxo arquitetural mais complexo, permitir que qualquer serviço de informação possa fornecer *feedback* de decisão para outros

serviços.

Este trabalho tentará responder as perguntas feitas na seção anterior. Os serviços do usuário verificarão de forma contínua sua identidade, por meio de várias fontes. Somente será necessário confirmar as credenciais, se nenhum dos autenticadores fornecerem medidas adequadas. Inclusive, pode-se identificar uma mudança de comportamento suspeita, sobre quem opera a aplicação.

É importante que haja um *middleware* que associe diversas fontes de autenticação contínua. Em um mundo que o número de dispositivos e informações aumenta constantemente, ter uma segurança maior torna-se essencial. Desse modo, possuir mais de uma forma de garantir a autenticidade do usuário é fundamental. Se novas fontes de autenticação contínua puderem ser adicionadas de um modo simples, as aplicações terão um grau de confiança maior.

A criação e implementação desse modelo irá ampliar o estudo e uso de autenticadores contínuos concomitantes. Cada nível gerado é independente no envio do valor, e cada serviço define a sua política de agregação.

1.3 OBJETIVOS GERAIS

Este trabalho tem por objetivo propor uma arquitetura para o uso de uma ou mais variáveis de autenticação contínua. Os serviços que tiverem interesse, receberão essas variáveis em tempo real, assim que geradas. Junto disso, será feito sua implementação em uma linguagem de propósito geral, com verificação através de testes unitários.

Com isso pronto, será feito uma validação simples, por meio de uma aplicação. Esta representará um serviço interessado nos NDCs do usuário e fará uso deles para mantê-lo autenticado ou não.

O escopo do trabalho não abrange como serão gerados os níveis de confiabilidade. A geração deles, utilizada nos experimentos, tem apenas o propósito de auxiliar na validação do modelo proposto.

Não será realizado um experimento elaborado com usuários. Além da limitação de tempo, existem muitas variações possíveis, que por si só, podem elaborar um estudo mais aprofundado.

1.4 OBJETIVOS ESPECÍFICOS

Para atingir os objetivos gerais, alguns pontos específicos devem ser definidos. Cada um deles são fundamentais para a construção deste trabalho num todo.

- Revisar na literatura os métodos existentes de autenticação.
- Definir uma arquitetura que permita a comunicação segura entre as entidades.
- Desenvolver modelos simples que representem serviços e dispositivos para teste empírico.
- Criar um servidor central, para fazer o gerenciamento dos níveis que foram gerados pelos dispositivos.
- Integrar as trocas de mensagens entre as partes que compõem a arquitetura proposta.
- Gerar testes unitários para verificar os fluxos existentes.
- Validar o middleware com um aplicativo de celular.

1.5 METODOLOGIA CIENTÍFICA

A elaboração deste trabalho aconteceu em várias etapas. Na primeira parte, foi feita uma pesquisa de alguns trabalhos relacionados, utilizando as palavras chave (colocar aqui as palavras chave usadas). Com eles, foi possível descobrir o atual estado da arte em autenticação.

Em seguida, construiu-se o fluxo de informações, para entender como ocorrem as trocas de mensagens. Depois uma modelagem em alto nível da arquitetura, para avaliar o melhor meio de desenvolver o sistema. Com base nela, a codificação do middleware e seus respectivos testes foram implementados. Por fim, a validação foi feito por meio de alguns cenários de uso, para garantir a funcionalidade do modelo.

2 TRABALHOS RELACIONADOS

Ao pesquisar os trabalhos publicados de autenticação, verificou-se a existência de diversos tipos. Alguns são mais comuns no cotidiano e outros em situações mais específicas.

Alguns deles usam autenticação continua, ou seja, o processo ocorre o tempo todo em que o usuário usa o sistema. Outros a verificação é feita somente uma vez, de modo estático, no momento que antecede o uso da aplicação.

Além disso, é possível moldar uma autenticação tanto com um fator, como com vários diferentes. As seções seguintes detalharão os trabalhos existentes de cada um.

2.1 AUTENTICAÇÃO COM UMA FONTE

Nesse tipo de autenticação, apenas uma fonte é utilizada para verificar a identidade do usuário. É importante estudar alguns tipos de fontes individuais, para entender como a combinação delas pode ser feita da melhor forma possível.

As próximas subseções descrevem os tipos mais comuns. Alguns deles são combinados e aparecem em trabalhos da próxima seção.

2.1.1 Autenticação por Segredo

Usado por sistemas hoje, tem o seu maior exemplo com o uso de login e senha. Em alguns casos, é feito uma autenticação maior, usando além da senha, uma frase que responde alguma pergunta secreta, garantindo um duplo fator de autenticação.

Embora seja um mecanismo de fácil criação, conforme apontado por Stajano (2011), o nível de segurança exigido por ele não é nada prático. A recomendação é que uma senha, além de possuir um número grande de caracteres, não possua palavras comuns do cotidiano. Além disso, não é recomendável que ela seja anotada, e apenas o usuário lembre dela.

Adicionado a isso, as senhas devem ser mudadas de forma regular. Em uma realidade, que cada pessoa tem pelo menos 50 serviços diferentes, sejam aplicativos ou sistemas web, é algo completamente inviável.

2.1.2 Autenticação por Token

Nesse caso, seria o uso de algum objeto, acessório ou dispositivo, em que somente com ele, seja possível acessar o sistema. Uma bastante comum seria o cartão de crédito / débito para efetuar pagamentos de compras em geral, ou a chave física usada em residências.

Há diversos outros dispositivos, que seguem o mesmo principio. Um grupo de pesquisadores desenvolveu um bracelete denominado KBID (*Kerberos Bracelet Identification*) (CARRIGAN; MARTIN; RUSHANAN, 2016) em que é possível realizar a autenticação em um serviço, através desse acessório.

Outro tipo de abordagem, é a o projeto *Pico* da Universidade de Cambridge (STAJANO, 2011). Foi desenvolvido um dispositivo confiável, que permite autenticar uma pessoa, assim como o bracelete acima, para substituir o sistema de senhas, de modo seguro.

Seus objetivos mínimos eram: não precisar decorar as senhas, ser escalável, de modo que várias aplicações possam usá-lo, e ser tão seguro quanto usar senha, pelo menos. Além disso, há outros benefícios prometidos, como por exemplo, a autenticação ser contínua durante a sessão inteira e não haver a possibilidade de reuso de credenciais, em aplicações diferentes.

2.1.3 Autenticação Biométrica

A autenticação é baseada em algo que o usuário é, e pode ser tanto no sentido fisiológico, como no seu comportamento. Enquanto a primeira se baseia nas digitais da mão, geometria da mão, ou região facial, a comportamental busca achar padrões de fala, digitação e caminhada.

O *TouchAlytics* é um exemplo de comportamental (FRANK et al., 2013), em que foi feito uma análise de usuários fazendo atividades básicas no smartphone, como leitura de um texto, e verificado padrões de uso: pressão exercída na tela, movimento e intervalo de tempo ao deslizar a tela. A conclusão mostrou que é possível autenticar dessa forma, dentro do cenário em que foram realizados os experimentos.

Já no trabalho de Yap et al. (2008), que envolve o uso de autenticação sign-on e continua foram usados dois dispositivos para capturar informações inerentes ao usuário: o rosto da pessoa e as suas digitais. Ambas são informações biométricas psicológicas. Assim que o usuário faz uma primeira autenticação com suas credenciais, um processo de

tempos em tempos captura suas digitais e seu rosto. Para cada um, é gerado uma pontuação, e depois outra, através da combinação delas. Se essa combinação for menor que um limite, os processos são congelados e o usuário deixa de estar autenticado.

Uma outra abordagem (NIINUMA; JAIN, 2010) concebida foi o uso de reconhecimento facial, junto com um padrão de cor das roupas do usuário, utilizado na hora do login inicial. Tanto o reconhecimento facial, como das cores, é feito de modo contínuo. O trabalho evidencia que em certas posturas as quais uma pessoa possa fazer, como virar o rosto, ou ler alguma informação em um livro, não são suficientes para manter o usuário autenticado. Logo, é proposto um padrão de cores, na hora em que é feito o login inicial no sistema, e usado junto com o reconhecimento facial, para evitar que quando o usuário realize padrões corporais comuns, ele precise sempre se autenticar novamente. Apesar de ocorer algumas situações de *False Reject*, quando a iluminação do ambiente muda, ou *False Accept*, quando a cor do corpo é muito parecida com a do ambiente no fundo, o trabalho mostrou resultados bem interessantes.

2.1.4 Autenticação por Localização

Utiliza o local em que se encontra o computador que foi usado para se autenticar como informação para validar o acesso do usuário. Esse tipo de autenticação, normalmente é usada com alguma das outras três acima, e nunca de forma isolada. Ainda que uma empresa, decidisse adotar esse metodo, de forma que apenas na rede interna dela, seja possível usar um sistema, alguém que não é funcionário, poderia usar de engenharia social, para usar o computador de forma não autorizada.

O trabalho de Denning e MacDoran (1996) mostra que essa forma já era pensada faz tempo. Ela permite que a autenticação seja feita de forma contínua, de forma semelhante ao que se deseja neste trabalho, e é possível usá-la como assinatura em documento, como prova de que tal documento existiu nesse local em um certo instante de tempo.

Com o uso da tecnologia móvel em grande escala neste momento, Zhang, Kondoro e Muftic (2012) propuseram um modelo de autenticação, que leva em conta a localização do cliente, graças ao dispositivo de GPS embutido. Conforme dito por eles, diferentes das outras abordagens, não há a necessidade de um aparelho especial, para detectar a localização, e sim só usar o que já está disponível no celular. Considerando também, que é possível forjar um local, tanto em nível de

hardware, de sistema operacional, como da aplicação, foi proposta um mecanismo, com três níveis, para garantir a autenticidade da localização: dois conjuntos de coordenadas de fontes diferentes, o endereço IP do smartphone e o endereço MAC de um ponto de acesso próximo com sinal forte.

2.2 AUTENTICAÇÃO COM MÚLTIPLAS FONTES

Para aumentar a segurança e a confiabilidade, mais de uma fonte é utilizada com o intuito de reforçar a identidade do usuário.

Este trabalho propõe a construção de um modelo deste tipo, de modo que os valores contínuos gerados combinam-se para gerar um valor aplicável. Diferente dos trabalhos a seguir, esses valores são coletados de forma independente. Assim, cabe a aplicação interessada definir como usará os valores de cada fonte.

Nas subseções seguintes, são descritos alguns exemplos encontrados. A primeira combina fatores biométricos, enquanto a segunda faz um uso maior de credenciais com token.

2.2.1 Biométrico multimodal

Conforme já discutido, um mecanismo de autenticação biométrica usa um fator inerente ao comportamento ou a sua fisiologia. Serão mostrados dois trabalhos que obtiveram sucesso unindo mais de um fator biométrico na autenticação, aumentando a precisão do processo.

O primeiro trabalho proposto por Azzini et al. (2008) usa dois fatores biométricos: a digital e o rosto do usuário. Após uma autenticação inicial, com o uso de credenciais, o sistema gera um valor entre 0 e 100, em um intervalo de tempo. Este valor representa o quão bem o rosto do usuário corresponde ao template armazenado. É feita sua comparação com um *threshold*: se estiver abaixo, o sistema uma amostra da digital, para verificar se mantém a sessão aberta ou não.

Para garantir uma maleabilidade no uso desses valores, foi usado um controlador *fuzzy* com regras Mandani. Os valores gerados pelo rosto e pela digital alimentam um mecanismo de inferência *fuzzy*. Ele por sua vez obtem um valor, que expressa o nível de confiança que os mecanismos tem na identidade do usuário. Através de um conjunto de regras, esses valores geram um valor de confiança do sistema em si. E para obter um valor contínuo aproximado, o mecanismo *defuzzifier* usa

a técnica padrão da área do centroide, com esse valor de confiança do sistema.

Neste outro trabalho, Brömme e Al-Zubi (2003) propuseram uma forma de autenticar com o uso de uma caneta eletrônica, usada em *tablets* e derivados. O primeiro fator é desenhar um esboço composto pela junção de formas ou figuras combinadas entre si. E como segundo fator acrescentado por eles, está o conhecimento do usuário sobre a forma adequada desse esboço. O acordo de como o esboço deve ser desenhado é feito quando o usuário registra-se no sistema.

Esse diferencial, do acordo entre o sistema e o usuário, gerou resultados bem interessantes. No teste realizado por eles, com 4 formas diferentes, à medida que ficavam mais complexas, a taxa de erro de reconhecimento diminua. No caso mais complexo, com o conjunto de validação do experimento, não houve erro.

2.2.2 Multi-fator geral

No geral, autenticações desse tipo fazem uso de um ou mais fatores de independentes, como parte das credenciais do usuário. (SABZEVAR; STAVROU, 2008). Esse fator a mais pode ser: uma senha enviada por SMS, ou um cartão magnético. O uso desse último é recorrente em aplicações bancárias, junto com um PIN (Personal Identification Number). Dois trabalhos desse tipo serão descritos.

O primeiro desenvolvido por Tiwari et al. (2011) trata de um sistema com três fatores. As credenciais de login e senha são um desses fatores. O segundo fator usa um código denominado TIC (Transation Identification Code). O TIC permite verificar tanto o usuário, quanto a transação que está vingente. É um número gerado pela instituição financeira, com 8 ou 16 bits aleatórios, formado por uma sequencia numérica ou uma combinação de digitos alfanuméricos.

O outro fator é a autenticação via SMS (Short Message Service). Para que isso seja viável, a instituição financeira armazena o número de cada usuário. Foi assumido que os usuários carregam o celular regularmente no cotidiano. Desse jeito, apenas usuários válidos receberão a mensagem. A validação é feita através de uma resposta *YES* para uma transação válida e *NO*, caso contrário.

Já nesse trabalho desenvolvido por Karapanos et al. (2015), nomeado como *Sound-Proof*, houve a adição de um fator pouco convencional, dentro dos trabalhos procurados. Da mesma forma que o anterior, foi assumido que o usuário carrega o celular frequentemente. Além das

credenciais, o segundo fator utilizado a proximidade dos dispositivo de login com o celular. Isso é realizado por meio de um comparativo entre o som captado pelo microfone do celular e o outro dispositivo em que se realiza o login.

Esse fator é transparente ao usuário, tornando a experiência de uso, como se tivesse apenas um fator. Foram feitos estudos comparativos desse modelo com o Google 2-Step-Verification como experimento. Os resultados foram estatisticamente bem mais favoráveis ao modelo proposto. Além disso, a maioria dos participantes afirmou que usaria o *Sound-Proof* mesmo que esse duplo fator de autenticação fosse opcional.

3 PROJETO

Antes de desenvolver foi necessário identificar as possíveis trocas de mensagens. Este capítulo apresenta duas soluções possíveis, e os detalhes daquela que foi escolhida na implementação.

Com base nela, algumas questões de segurança foram pensadas para garantir o melhor modo de comunicação. Assim, a definição do protocolo HTTPS (Hyper Text Transfer Protocol Secure) e a especificação JWT são aqui discorridas.

Feito isso, pode-se implementar no capítulo seguinte: uma arquitetura com um fluxo de informação adequado e estrutura segura.

3.1 FLUXO GERAL

Em poucas palavras, este trabalho quer associar diferentes dispositivos de autenticação contínua a um usuário. Os serviços de software em geral, seja uma aplicação *Web* ou *Mobile*, podem então consumir os valores gerados por esses dispositivos. Essa informação pode ser vista como um terceiro fator de autenticação.

Dentro dessa ideia, existem diferentes fluxos possíveis de comunicação, e cada um deles pode ser usado, dependendo dos requisitos necessários. A Figura 1 descreve esta forma. Em um certo intervalo de tempo, independente para cada um, o dispositivo gera um novo NDC associado ao usuário e envia ao servidor essa nova informação.

Após o tratamento devido, o servidor persiste o dado. Os serviços que estão interessados, são notificados que um novo valor foi gerado, e recebem-no para utilizá-los dentro da sua aplicação. Dentro desse fluxo, os serviços seguem o padrão de projeto *Observer*. Através dele, é possível manter os serviços bem desacoplados do servidor, de modo que eles só recebem a informação. No caso, para se obter sempre o nível atualizado, os serviços inscrevem-se num canal onde chegam os novos valores.

Repare que nesse modelo, existe uma comunicação bidirecional entre o serviço e o servidor. Isso permite um envio eficiente das mensagens entre as duas entidades. Dessa forma, se ao consumir um NDC que foge os padrões dos últimos valores, o serviço pode pedir ao servidor, que para aquele dispositivo, algum tipo de providência seja tomada.

Outra forma possível, seria retirar o padrão *Observer* entre o serviço e o servidor. O serviço assume a responsabilidade para consultar

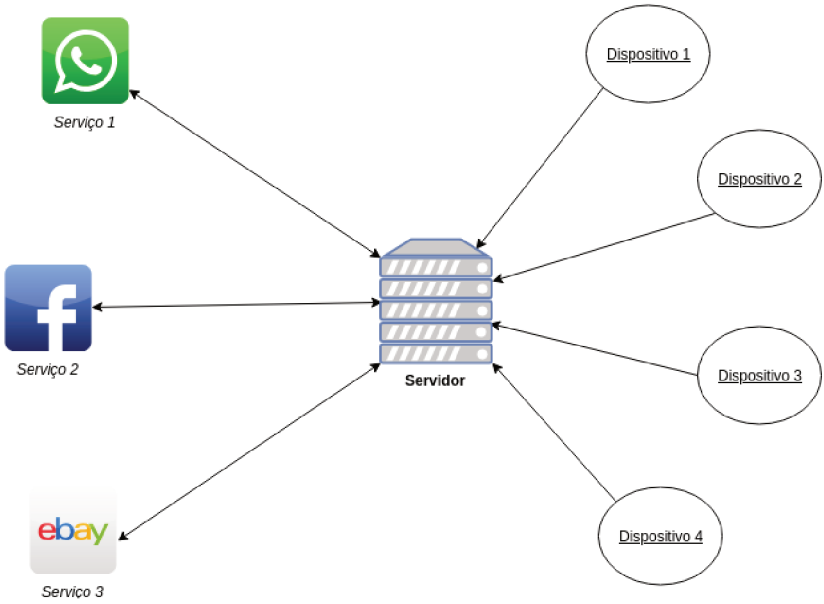

Figura 1 Serviços escutam novos valores

as atualizações do NDC de cada dispositivo, conforme a Figura 2.

Dessa forma, o dispositivo continuará gerando novos NDCs e eles continuarão sendo armazenados. No entanto, cabe ao serviço pedir o último valor gerado ao servidor, conforme a sua necessidade.

Essa solução, caso o serviço queira receber as atualizações em tempo real, não é muito adequada. O serviço pode consultar o servidor alguns microsegundos antes de um novo dado ser inserido, e perder essa atualização nova.

Mesmo que o serviço não queira receber as atualizações em tempo real, fazê-lo realizar *pooling* no servidor tornaria-se contraditório. O objetivo da autenticação contínua, é gerar valores periodicamente em um curto espaço de tempo. Logo, é mais adequado que os valores sejam gerados e transmitidos em tempo real.

Para a construção deste trabalho, será utilizado o primeiro tipo de fluxo de informação. Assim, pode-se garantir que os serviços sempre receberão o último valor atualizado.

Ao receber o NDC, a aplicação decide se mantém o usuário autenticado. Esse seria o fluxo geral, proposto por essa arquitetura. Usando essa forma, é garantido que caso a implementação no servidor mude,

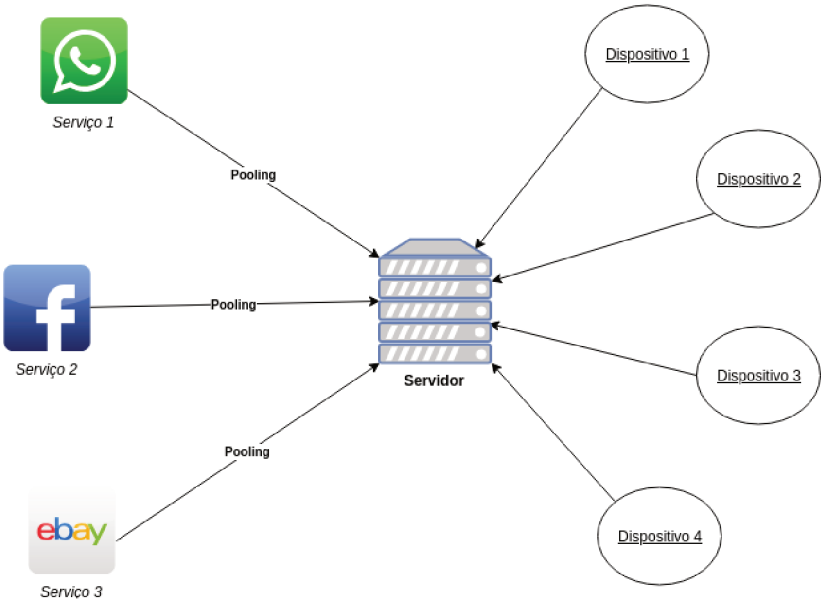

Figura 2 Serviços com pooling periódico

as aplicações ainda continuarão recebendo o valor corretamente.

3.2 ARQUITETURA PROPOSTA

Com o fluxo da aplicação bem definido, pode-se construir uma arquitetura que consiga atender aos requisitos necessários. Os componentes da arquitetura do sistema podem ser descritos de forma sucinta na Figura 3, com as interações entre cada um dos componentes. Antes de apresentar numa forma mais detalhada o processo de comunicação entre as entidades, é necessário apresentar cada uma delas, para entender seu papel dentro da solução proposta.

O primeiro componente é o servidor. Ele é responsável por manter todas as atualizações feitas do NDC com o passar do tempo. Os dispositivos comunicam-se via requisições do protocolo HTTP (Hyper Text Transfer Protocol), enviando os dados no formato JSON (JavaScript Object Notation). E além disso, deve suportar o protocolo de *WebSocket* o qual realiza comunicação *Full-duplex* entre o servidor e um cliente.

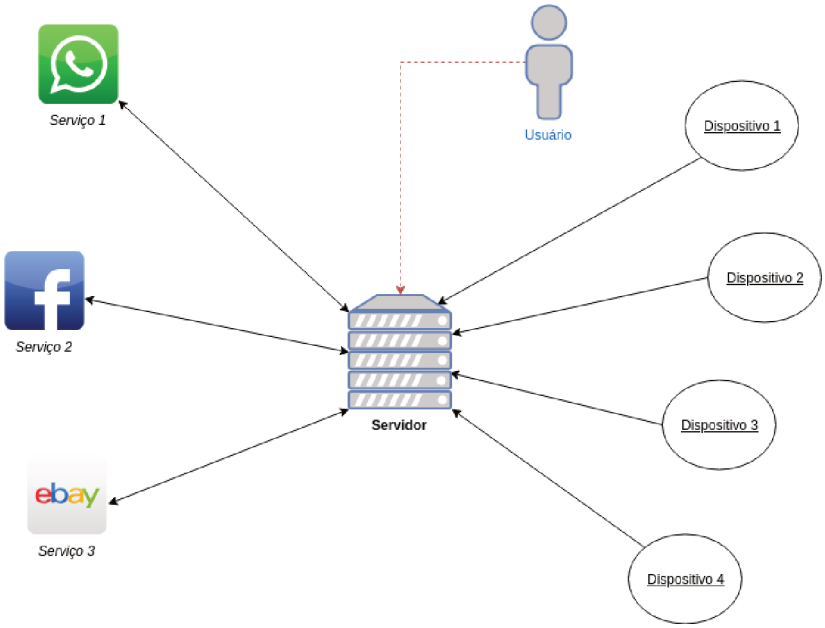

Figura 3 Arquitetura a ser implementada

A segunda entidade fundamental é o usuário. Somente ele pode adicionar os dispositivos e serviços autorizados a estabelecer interações no servidor. Não existem restrições de quais dispositivos um serviço pode escutar atualizações. A partir do momento que o usuário cadastra um serviço, este pode escolher dentre os dispositivos específicos que deseja receber suas atualizações.

O terceiro componente é o dispositivo, capaz de gerar um NDC referente ao usuário. O seu objetivo principal é gerar de tempos em tempos um valor atualizado do NDC. Pode ser qualquer máquina, independente do porte, que consiga gerar um valor contínuo, numa escala de 0% a 100%. Este é o grau de certeza sobre o usuário ser quem ele afirma que é. Para os fins deste trabalho, foram usados dispositivos que geram valores quaisquer nos testes. No caso, os dispositivos que podem atualizar o NDC devem ter as credenciais necessárias.

Depois de gerado esse valor, ele é enviado ao servidor. Este valor enviado pelo dispositivo não é tratado em momento algum. Ele é salvo e retransmitido no formato em que veio. Nada impede que seja usado algum modelo matemático, para melhorar a sua semântica. Seu

timestamp também é gerado, para fins estatísticos futuros.

Em seguida é disparada uma notificação para todos os serviços interessados. Um novo nível de autenticação é gerado, de modo que eles possam manter o usuário logado ou não em suas aplicações, de acordo com suas regras.

Por fim, o quarto componente é um serviço, seja ele qual for, que esteja interessado em receber o NDC de forma periódica. Por serviço, entende-se qualquer aplicação *Web* ou *Mobile*. Ela usará esse valor para a tomada de decisão com respeito a autenticação do usuário.

Para que o serviço possa consumir esses dados, ele deve estar em conformidade com o protocolo *WebSocket* usado. No caso, seria escutar um evento *newValue*. Todo NDC criado, será transmitido por esse meio.

Existe a emissão dos eventos adequados, para a escolha de quais dispositivos deseja-se escutar. Para receber atualizações, basta emitir um evento *join* com o identificador daquele dispositivo. E para não obter mais novas atualizações, deve-se emitir um evento *leave*, também com seu identificador.

3.3 REQUISITOS DE SEGURANÇA

Existem três entidades diferentes nessa arquitetura: o usuário, o dispositivo e o serviço. Cada um deles usa uma forma de autenticação diferente, conforme o tipo de conexão feita.

Para realizar qualquer ação, tanto o usuário, quanto os dispositivos e os serviços devem estar autenticados. Os dois primeiros autenticam-se através do envio de um *token* pelo cabeçalho da requisição, que nesse caso é JWT (Json Web Token), que será explicado logo em seguida.

No caso do serviço, como ele realiza uma conexão persistente com *WebSocket*, é feita apenas uma autenticação inicial, com credenciais cadastradas no servidor.

Além disso, foi usado o protocolo HTTPS, para garantir a criptografia dos dados. Desse modo, os NDCs não podem ser lidos por qualquer outra entidade.

3.3.1 Json Web Token

É um padrão aberto, definido no RFC 7519 por Bradley, Sakimura e Jones (2015), em que consiste uma forma compacta e segura para transmissão de informação entre duas partes em um objeto do tipo JSON. Pode ser verificada, pois é assinada digitalmente, ou usando algum segredo com um algoritmo HMAC (Hash-based Message Authentication Code) ou com um par de chaves assimétricas RSA.

Além de possuir um tamanho pequeno, permitindo sua transmissão em requisições HTTP no *header* ou na URL (Uniform Resource Locator), o *token* possui informações do usuário. Isso evita a necessidade de certas consultas no banco de dados, após a autenticação.

Sua estrutura é uma cadeia de caracteres, dividida em três partes, separadas por um ponto (.):

- *Header*
- *Payload*
- *Signature*

O *header* consiste em um objeto com duas informações: o tipo de *token*, que no caso é JWT e o algoritmo usado, que pode ser um HMAC 256 ou RSA. Esse objeto é codificado em Base64Url, gerando a primeira parte do *token*. Neste trabalho, foi usado o HMAC 256.

O *payload* contém um pedaço de informação sobre algum assunto, que pode ser por exemplo, o identificador de um usuário. Ele nunca deve conter informação sensível em *tokens* que são somente assinados, pois pode ser lida por qualquer um. Da mesma forma, essa segunda parte também é codificada em Base64Url.

E por último, o *signature* é formado pela concatenação do *header* codificado em Base64Url, com o *payload* também codificado, mais o segredo que é definido pelo servidor.

O JWT pode ser usado tanto para autenticação, como para garantir o não-repudio de mensagens. No segundo caso, basta trocar o algoritmo HMAC pelo RSA. Para este projeto, ele foi utilizado apenas para autenticação, pois a aplicação está hospedada em um domínio do LABSEC (Laboratório de Segurança em Computação). Além de garantir a transmissão criptografada dos dados, existe uma CA (Certificate Authority) que assegura a origem do servidor, que será explicado na seção a seguir.

3.3.2 HTTPS

Ao usar o protocolo HTTP padrão, para realizar uma comunicação do cliente com o servidor, toda informação encontra-se em *plain text*. Qualquer entidade pode interceptar a comunicação, e ler o que está sendo enviado entre as partes.

Dessa forma, o HTTPS fornece uma conexão criptografada entre o cliente e o servidor. Sua implementação usa o SSL ou o TLS para cifrar a comunicação. O princípio do protocolo é moldado em cima da criptografia de chave assimétrica.

Essa criptografia utiliza dois tipos de chave: uma privada, e uma pública. Ao cifrar a mensagem com uma das chaves, só é possível decifrá-la com a outra. A chave privada permanece no servidor, devidamente protegida, enquanto que a chave pública, é acessível por qualquer pessoa, que queira enviar uma mensagem ao servidor.

No entanto, da maneira que foi apresentado, ainda é possível forjar um novo par de chaves, e induzir o usuário que ele está comunicando com o servidor correto. Esse tipo de ataque é conhecido como *man in the middle*. Para isso, deve haver uma forma de identificar de modo seguro, que uma chave pública está associada àquele servidor em questão.

Isso é feito por uma entidade denominada autoridade certificadora. Dentre várias funcionalidades, ela é responsável por emitir, distribuir, renovar, revogar e gerenciar certificados digitais. Um certificado digital é um meio de identificar de modo seguro, que uma chave pública pertence ao servidor em questão.

Observando um exemplo simples, em que o navegador acessa um site, o servidor envia o certificado, e verifica se a autoridade que assinou o certificado, encontra-se em uma lista de autoridades certificadoras. Caso esteja, a comunicação pode ser estabelecida.

Com o intuito de realizar este trabalho, o LABSEC forneceu um servidor com o domínio labsec.ufsc.br, para a que a aplicação fosse hospedada. Junto disso, foi gerado um par de chaves RSA, de modo que a chave privada encontra-se no servidor com acesso restrito ao root.

O uso de HTTPS mostra-se importante em dois aspectos. Sob o nível de privacidade, pois seria possível ler qualquer geração de NDCs dos dispositivos ao servidor, assim como ler os valores que os serviços recebem efetivamente. E outra questão, é reforçar a autenticação das partes. Como é utilizado um *token* nessa parte, seria possível, mesmo que por um período curto de tempo, um atacante obter o *token* e se passar pelo usuário, afim de obter acesso ao sistema. Criptografando a

conexão, é possível evitar esses tipos de situação.

4 IMPLEMENTAÇÃO

Nesta parte do trabalho será descrita detalhes de como a arquitetura proposta foi implementada. Primeiro será mostrado alguns *frameworks* e tecnologias utilizadas, que facilitaram muito o processo de desenvolvimento, e depois como o servidor foi construído.

Tanto no servidor central, como nos serviços que consomem esses valores, foi utilizada a linguagem *JavaScript*. Ela possui várias bibliotecas que facilitam o desenvolvimento, e que recebem *feedbacks* constantes pela comunidade. Além disso, é pouco verbosa, gerando menos código e facilitando a manutenabilidade do mesmo para trabalhos futuros.

Já nos dispositivos foi escolhida a plataforma Android. Sua escolha permitiu mostrar de forma clara, uma aplicação simples do middleware. Ela também possui uma comunidade grande de desenvolvedores, que ajudam a mantê-lo com qualidade.

4.0.1 Android

O Android[1] é um sistema operacional usado em diversos dispositivos, desde celulares e tablets, até relógios e televisores. Teve um forte crescimento quando os smartphones tornaram-se bastante populares ao redor do mundo. Possui uma ótima documentação para construção dos mais diversos tipos de aplicativos.

A IDE Android Studio e sua SDK foram criadas para o desenvolvimento rápido e eficiente das aplicações. Neste trabalho foi feita uma aplicação para usufruir da autenticação do middleware. Ela é responsável em controlar o bloqueio de tela, caso os níveis de confiabilidade não estejam de acordo, com um threshold pré-estabelecido.

4.0.2 Node.js

O Node[2] é um *runtime environment* que permite rodar código Javascript, como uma linguagem de propósito geral. Embora seja possível construir uma aplicação local, criando scripts como em Ruby ou Python, seu uso foi popularizado na criação de servidores Web.

Sua arquitetura é baseada em eventos, com um modelo de I/O

[1] https://developer.android.com/
[2] https://nodejs.org/en/

não bloqueante, permitindo um grande volume de requisições. O modelo tradicional aloca uma nova *thread*, para cada cliente conectado, consumindo mais recurso computacional. No caso do Node, assim que uma requisição é feita, a *thread* é liberada no momento que ocorrer um pedido de I/O. Isso permite que novas requisições sejam atendidas mais rápido. Quando o I/O acabar, um novo evento é gerado, contendo o resultado desejado e então a requisição retornar o dado.

No desenvolvimento deste projeto, o Node possui um ecossistema grande de bibliotecas, que são gerenciadas pelo NPM (Node Package Module). Através dele, foram baixados módulos prontos, que permitiram estabelecer um servidor REST (Representational State Transfer), conexão com um banco de dados e o uso de *WebSocket*. Junto disso, foi incluído um módulo de testes, para automatizar as verificações dos fluxos mais comuns do projeto.

4.0.3 Express

O Express[3] é um *framework* desenvolvido para construir aplicações WEB com poucas linhas de código. O Node possui módulos internos para se comunicar com o sistema operacional, como leitura e escrita de arquivos, e protocolo HTTP para troca de dados. O *Express* simplesmente encapsula um conjunto de comportamentos desses módulos, que permitem criar *endpoints* simples, com um conjunto de opções pré-definidas.

Através dele, será montada a API (Application Programming Interface) do servidor, para que algumas ações possam acontecer. Dentre elas tem a autenticação do usuário, registro de um dispositivo para gerar NDCs novos, assim como registro de um serviço novo, para que ele possa receber os NDCs dos dispositivos que lhe interessam.

4.0.4 Socket.io

O Socket.io[4] é uma implementação baseada na tecnologia de *WebSocket*, em que é estabelecida uma conexão bidirecional em um canal *Full-duplex* sobre um *socket* TCP. Esse modelo é apropriado para situações orientadas a eventos, em que o cliente deseja notificar o servidor caso algo aconteça, ou vice-versa.

[3] http://expressjs.com/
[4] https://socket.io/

Todos os serviços que desejam ser notificados sobre a mudança de valores em algum dispositivo, conectam-se ao servidor via *WebSocket*. No momento que estão autenticados, eles enviam um evento ao servidor, informando quais dispositivos deseja ouvir, ou por exemplo, quais não deseja mais escutar novas atualizações.

4.0.5 MongoDB

Devido a carência dos banco de dados relacionais, foram propostos novos modelos voltados a escalabilidade horizontal. Enquanto que nos primeiros as melhorias de software/hardware ocorrem no sistema, nesse novo a expansão ocorre por meio da adição de novas infraestruturas. Esses modelos são conhecidos como NoSQL.

O MongoDB[5] é um banco de dados orientado a documento, que pertence a essa classe de novos modelos. É organizado em um conjunto de coleções, que podem ser entendidas como tabelas do modelo relacional. Cada coleção é formada por diversos documentos, com cada documento possuindo sua própria estrutura de atributos e valores. A representação de um documento possui um formato parecido com a notação de um objeto JSON.

A integração do MongoDB com o Node.js, além da escabilidade, e mapeamento do modelo de documento para os objetos na aplicação foram fatores que ajudaram na escolha. Visto que à medida que novos usuários podem ser adicionados em trabalhos futuros, mais dados são gerados ao longo do tempo.

4.0.6 Mongoose

É uma biblioteca[6] que simplifica a escrita de código de manipulação do banco MongoDB. Em um cenário comum, para se comunicar com o banco, basta ter o drive instalado, para fazer a comunicação e assim realizar as operações nele. Isso tanto quanto num banco de dados relacional ou não-relacional.

Porém, isso pode gerar *boilerplate code* na hora de criar regras de validação e de negócio. O Mongoose vem como uma solução, para evitar isso (*DRY principle*) e deixar o código mais limpo e permitir localizar trechos específicos.

[5] https://www.mongodb.com/
[6] http://mongoosejs.com/

4.0.7 Jest

Para poder realizar os testes unitários, foi utilizado o *framework* Jest[7], o qual é mantido pelo Facebook. Junto com ele, vem diversas funções prontas, que facilitam a escrita de asserções. Possui também suporte para testes assíncronos, que ocorrem em requisições na API REST ou quando forem acessar o banco de dados.

Com ela, foi possível escrever os mais variados casos de forma simples e descritiva. Isso permite uma futura refatoração do código, protegida contra a inserção de possíveis novos bugs.

4.0.8 Supertest

Como a maioria dos casos de teste necessitam a chamada de requisições para a API, o Supertest[8] permite que isso seja simulado via código. Possui também uma boa integração o Jest na hora da validação.

O uso dele também torna o código menos verboso e legível, na hora de verificar como cada teste foi implementado.

4.0.9 Jsonwebtoken

O Jsonwebtoken[9] permite a criação de um JWT, seguindo a especificação detalhada no capítulo anterior. É possível escolher qual dos dois algoritmos utilizar e definir o tempo de expiração do *token*.

O usuário e o dispositivo possuem segredos diferentes na hora de gerar o *token*. Caso ocorra o vazamento de um, isso não compromete a segurança do outro.

4.0.10 Bcrypt

Com o intuito de armazenar as senhas das entidades de forma segura, o Bcrypt[10] possui funções de *hash*, inclusve de forma assíncrona.

É possível também fornecer um parâmetro, denominado *salt*. É um dado aleatório usado como entrada adicional junto à senha, para

[7] https://facebook.github.io/jest/
[8] https://github.com/visionmedia/supertest
[9] https://github.com/auth0/node-jsonwebtoken
[10] https://github.com/dcodeIO/bcrypt.js

gerar o *hash*. Dessa forma, evita-se um ataque de dicionário ou um ataque pre-computado, como o *rainbow table*.

4.1 DESENVOLVIMENTO

4.1.1 Dispositivos

O principal, se não o único, objetivo do dispositivo, é gerar o NDC, de tempos em tempos, e enviá-lo ao servidor. Este fará o tratamento adequado, independente do tipo de dispositivo que for.

No entanto, para que isso seja possível, o dispositivo precisa estar em conformidade com estes três pontos:

- O usuário já o cadastrou no sistema
- Deve estar autenticado via JWT
- Para enviar os dados, deve chamar a requisição HTTP adequada

Assumindo que o usuário esteja autenticado, ele pode cadastrar um novo dispositivo, através de uma requisição *POST* na *url* /device. O corpo da requisição deve conter os valores *name* e *password*. O valor *name* referencia unicamente o dispositivo no servidor para o usuário em questão.

Com o dispositivo cadastrado, basta ele se conectar fazendo uma requisição, conforme descrito no trecho de código:

```
1
2   // Aqui vem as credenciais para realizar o login
3   var data = {
4     name: name,
5     password: password
6   };
7
8   $.ajax({
9     url: '/device/login',
10    type: 'post',
11    data: JSON.stringify(data),
12    headers: {
13      'Content-Type': 'application/json'
14    },
15    success: function (data) {
16      alert("Success in login!");
17    },
18    error: function (data){
19      alert("Error sending data!");
```

```
20      }
21  });
```

Listing 4.1 – Login do dispositivo

Essa requisição, seguindo o padrão JWT, irá retornar um *token*, que deve ser usado nas requisições futuras. O próximo passo, para enviar um novo NDC, é fazer uma requisição conforme o outro trecho a seguir.

```
1   var data = {
2     value: value
3   };
4
5   $.ajax({
6     url: '/device/send',
7     type: 'post',
8     data: JSON.stringify(data),
9     headers: {
10      'Content-Type': 'application/json',
11      'x-auth': token
12    },
13    success: function (data) {
14      alert("Data has been sent!");
15    },
16    error: function (data) {
17      alert("Error sending data: " + data);
18    }
19
20  });
```

Listing 4.2 – Requisição para enviar NDC novo

Note que o *token* aparece ali no cabeçalho da requisição. E para enviar o valor apropriado, basta o dispositivo gerá-lo, de acordo com o método de autenticação.

Como forma de teste manual, foi criado uma página html estática, que pode ser acessada em /device.html. Ela contém dois campos para realizar o login e um outro para enviar um novo NDC referente ao dispositivo.

4.1.2 Serviços

O serviço funciona de forma semelhante ao dispositivo. Para que ele funcione, deve estar em conformidade novamente com alguns pontos:

- O usuário já o cadastrou no sistema

- Deve estar autenticado via *WebSocket*
- Para enviar os dados, deve emitir um evento adequado

Diferente do dispositivo, toda comunicação ocorre apenas via *WebSocket*. Assim é garantido que os serviços interessados em um dispositivo qualquer, obtenham o novo valor gerado em tempo real.

Assume-se que o usuário tenha já cadastrado o serviço, através de uma requisição POST na *URL* /service, com o *body* possuindo os valores *name* e *password*. Assim como o dispositivo, *name* referencia o service unicamente para aquele usuário.

Nesse ponto, tem uma diferença importante entre os dois. Como no caso de *WebSocket* o canal de comunicação fica aberto, a autenticação só ocorre uma vez, conforme o código abaixo.

```
1
2   // Definir a URL do servidor
3   _socket = io.connect('localhost:8080');
4   _socket.on('connect', function() {
5
6     _socket.emit('authentication', {username: username,
7     password: password });
8
9     _socket.on('authenticated', function() {
10      alert("Connected!");
11
12      _socket.on('newValue', (data) => {
13        console.log(`Got ${data} from device!`);
14      });
15
16    });
17
18    _socket.on('unauthorized', function(err){
19      alert("There was an error with the authentication!");
20    });
21
22  });
```

Listing 4.3 – Login do serviço

Para estabelecer um *WebSocket* entre duas partes, é necessário realizar uma requisição HTTP, com as credenciais adequadas. O RFC não define nenhum tipo de *header*, em que as credenciais possam ser armazenadas.

Além disso, o uso de dados sensíveis em URLs deve ser evitado. Muitos endereços podem ser armazenados em logs do servidor com a senha em *plain text*. Dessa forma, uma solução encontrada foi a seguinte:

- Assim que o serviço conectar-se ao socket, deve emitir um evento 'authentication' contendo as credenciais. Se não o fizer, ele será desconectado por timeout, cujo padrão definido é 1 segundo.
- Se as credenciais estiverem corretas, um evento 'authenticated' será emitido pelo servidor.
- Caso contrário, o evento *unauthorized* será emitido, contendo o erro em questão.

Todos os eventos que forem de interesse, devem ser escutados dentro do 'authenticated', que é o caso do evento 'newValue', onde o serviço receberá um dado contendo o último valor gerado de um dispositivo específico.

Com essa implementação, as credenciais ficam no corpo da requisição, todavia, o servidor usa TLS/HTTPS para criptografar os dados durante a transmissão, garantindo o sigilo deles.

4.1.3 Servidor Central

A partir desse ponto, entra a explicação mais detalhada da aplicação, visto que o servidor é quem controla boa parte do tráfego de informações.

O primeiro ponto a ser descrito é sobre a transmissão dos dados. No arquivo de config.js, além de configurações básicas do servidor, aqui é inserido o protocolo HTTPS. No código é mostrado o carregamento da chave privada e do certificado do LABSEC, junto com a cadeia de certificados intermediários.

```
1
2  let server;
3  if (process.env.NODE_ENV === 'production') {
4    const fs = require('fs');
5
6    const options = {};
7    options.key = fs.readFileSync('private-key.pem');
8    options.cert = fs.readFileSync('LABSEC.crt');
9    options.ca = fs.readFileSync('cert-chain.crt');
10
11   server = require('https').createServer(options, app);
12 } else {
13   server = require('http').createServer(app);
14 }
```

Listing 4.4 – Configuração de ambiente

Por questões práticas, o protocolo só é ativado, quando o servidor encontra-se em execução na máquina do LABSEC.

Outra ponto a considerar: só existe apenas um usuário. A senha foi salva no banco, através do REPL (Read–Eval–Print Loop) do Node, usando o método *save()* do *Mongoose*.

```
1
2  bcrypt.genSalt(10, (err, salt) => {
3    bcrypt.hash(PASSWORD_NOT_HASHED, salt, (err, hash) => {
4      user.password = hash
5    });
6  });
```

Listing 4.5 – Método de hash

Uma senha padrão foi persistida, depois de aplicado um algoritmo de *hash*, com um *salt* de 10 caracteres, para prover uma segurança maior. Quando o usuário realiza login no servidor, a biblioteca *bcrypt* verifica a senha, não esquecendo de utilizar o salt gerado.

Para isso, uma requisição HTTP deve ser feita no *endpoint* /user/login, com o corpo da requisição contendo as credenciais.

```
1
2  app.post('/user/login', async (req, res) => {
3    try {
4      const user = await User.findByCredentials(req.body.
       username, req.body.password);
5      const token = await user.generateAuthToken();
6      res.header('x-auth', token).send();
7    } catch (e) {
8      res.status(400).send(e);
9    }
10 });
```

Listing 4.6 – Login do usuário

A função *findByCredentials* retorna uma *promise* contendo o JWT gerado, caso as credenciais estejam corretas. Ele então é retornado no cabeçalho da requisição de resposta. Se as credenciais estiverem incorretas, a requisição de resposta retorna um erro de *bad request*.

```
1
2  UserSchema.methods.generateAuthToken = function() {
3    var user = this;
4    var access = 'auth';
5    var token = jwt.sign({ _id: user._id.toHexString(),
6                           access },
7                           JWT_SECRET_USER,
```

```
8                         { expiresIn: '12h'}).toString();
9
10    user.tokens = user.tokens.concat([{ access, token }]);
11
12    return user.save().then(() => {
13      return token;
14    });
15  };
```

Listing 4.7 – Geração do JWT

Para gerar o JWT, o *payload* informado é um objeto contendo o id do usuário e uma string contendo o tipo de *token*. O segredo é gerado no servidor a partir do momento em que é iniciado. No caso do *logout*, o objeto *token* 'auth' do usuário é apagado no banco de dados, usando uma requisição *post* no *endpoint* /user/logout.

É necessário para esta e outras requisições, que o usuário esteja autenticado, ou seja, com o *token* gerado em seu poder. O processo de verificação é feito conforme o código demonstrado. O dispositivo utiliza a mesma ideia que o usuário. No express, os métodos são *middlewares*, associados ao *endpoint*, de forma que sua execução sempre ocorre antes do código da requisição.

```
1  const authenticateUser = (req, res, next) => {
2    const token = req.header('x-auth');
3
4    User.findByToken(token).then((user) => {
5      if (!user) {
6        return Promise.reject(`No JWT found.`);
7      }
8
9      req.user = user;
10     req.token = token;
11     next();
12   }).catch((e) => {
13     res.status(401).send(e);
14   });
15 };
```

Listing 4.8 – Middleware de autenticação do usuário

Embora não fosse necessário um método para encontrar o *token*, sabendo que existe apenas um usuário, esta camada a mais permite que essa parte do código cresça. Não há necessidade em saber como é feita a implementação de busca do *token*, ficando mais fácil codificar para que novos usuários sejam inseridos ao servidor.

```
1  UserSchema.statics.findByToken = function (token) {
2    var User = this;
```

```
  var decoded;

  try {
    decoded = jwt.verify(token, JWT_SECRET_USER);
  } catch (e) {
    return Promise.reject('Invalid JWT.');
  }

  return User.findOne({
    '_id': decoded._id,
    'tokens.token': token,
    'tokens.access': 'auth'
  })
};
```

Listing 4.9 – Verificação do JWT

Dentro da *promise* de retorno, é feita uma pesquisa ao banco, com o id obtido ao decodificar o JWT, caso o segredo esteja correto.

Assim que estiver autenticado, o usuário pode adicionar um dispositivo ou um serviço. Para isso, deve fornecer suas credenciais, enviadas via POST em /device ou /service. O princípio de funcionamento do código é semelhante em ambos.

```
app.post('/device', authenticateUser, (req, res) => {
  const name = req.body.name;
  const password = req.body.password;

  const device = new Device({ name, password });
  device.save().then(() => {
    res.send('Success!');
  }).catch((e) => {
    res.status(400).send(e);
  });
});
```

Listing 4.10 – Adicionar dispositivo novo

Depois de autenticar o *token*, as credenciais são obtidas, e usadas para construir um objeto *device*, que é então persistido.

O service possui um mecanismo um pouco diferente: como sua comunicação com o servidor é via *WebSocket*, a autenticação ocorre apenas uma vez, no momento em que é estabelecida.

```
require('socketio-auth')(io, {
  authenticate: function (socket, data, callback) {
    Service.findByCredentials(data.username, data.password)
    .then((service) => {
      socket.name = service.name;
      service.devices.forEach(
```

```
7           (deviceName) => socket.join(deviceName)
8         );
9         return callback(null, true);
10      }).catch((e) => {
11        return callback(new Error("User not found"));
12      });
13    }
14  });
```

Listing 4.11 – Autenticação do serviço

O código acima é executado, assim que o cliente emite o evento 'authenticated' conforme já fora descrito. Após a validação, caso ela esteja correta, o socket associa um atributo seu ao nome do serviço, e manda escutar todos os dispositivos que ele estiver interessado. Os dispositivos de interesse do serviço sempre são armazenados no banco de dados. Dessa forma, toda vez que o serviço reconectar, suas preferências já estarão estabelecidas.

Assumindo que o serviço está conectado, ele tem duas escolhas: pedir para receber atualizações do dispositivo ou parar de recebê-las.

```
1  io.on('connection', (socket) => {
2
3    socket.on('leave', (params, callback) => {
4      Service.removeDeviceFromService(socket.name, params.name
       )
5      .then((service) => {
6        socket.leave(params.name);
7        callback();
8      }).catch((e) => {
9        callback(e.message);
10     });
11   });
12
13 });
```

Listing 4.12 – Pedir para não escutar mais novos NDCs de um dispositivo

Acima, o serviço fornece o nome do dispositivo, que não se deseja receber mais atualizações. No caso bem sucedido, o socket para de escutar por novos envios, em nível de protocolo.

```
1  ServiceSchema.statics.removeDeviceFromService = function (
       nameService, nameDevice) {
2    var Service = this;
3
4    return Service.findOne({ 'name': nameService }).then((
       service) => {
5      if (!service) {
```

```
6         throw new Error('No service found');
7       }
8
9       return Device.findOne({ 'name': nameDevice }).then((
        device) => {
10        if (!device) {
11          throw new Error('No device found');
12        }
13
14        service.devices = service.devices.filter(
15          (name) => name !== nameDevice);
16        service.save().then((service) => {
17          return service;
18        });
19      });
20    });
21  };
```

Listing 4.13 – Validação para não obter novos NDCs

O método do serviço acima, tem como objetivo validar a existência dos dois, através dos métodos *findOne*, e garantir a persistência dessa configuração no MongoDB. Isto é feito filtrando a lista atual do serviço, e persistindo ela novamente, entre as linhas 14 e 18. O método para receber novas atualizações, como possui um comportamento semelhante, não é necessário ser mostrado.

Um dos endpoints chaves da aplicação é o envio de um NDC por um dispositivo. Assim que for verificada sua identidade, é feito o envio de um valor atualizado para todos os serviços interessados.

```
1  app.post('/device/send', authenticateDevice, (req, res) => {
2    const device = req.device;
3    const value = req.body.value
4
5    const NDCData = {
6      device: device.name,
7      value: value
8    }
9
10   const newNDC = new NDC(NDCData);
11   newNDC.save(() => {
12     io.to(req.device.name).emit('newValue', NDCData);
13     res.status(200).send('Success!');
14   });
15 });
```

Listing 4.14 – Envio de NDC ao servidor

No corpo da requisição, o dispositivo envia seu identificador e o NDC atualizado. Em futuras atualizações, é possível adicionar novas

variáveis, para ajudar o servidor no refinamento do processo. Assim que obtido o valor, um registro contendo seu timestamp é salvo. Quando persistido, todos os serviços que estiverem escutando pelo dispositivo, receberão o valor novo para usarem em suas aplicações.

5 VERIFICAÇÃO

Esta parte do trabalho tem por objetivo verificar o fluxo da aplicação, através do uso de testes unitários. A criação deles foi guiada pelos *endpoints* da API REST, assim como pelas emissões de mensagens via *WebSocket* do serviço.

Os testes foram separados sob o ponto de vista das entidades já específicadas, que são capazes de alterar o estado do sistema:

- Usuário
- Dispositivo
- Serviço

Para cada uma, existem um conjunto de fluxos possíveis. Os testes foram implementados para cobrirem todos eles. As próximas seções desse capítulo irão descrever cada fluxo existente, usando algum trecho de código para ilustrar pelo menos um deles.

5.1 TESTES DO USUÁRIO

Dentro do contexto dessa aplicação, um usuário pode adicionar dispositivos e serviços de sua preferência. No entanto, não se pode esquecer de outras ações básicas, como por exemplo, realizar login, para evitar acesso não autorizado.

A ação do login é a primeira que será descrita. Dentro do login, podem acontecer as três situações abaixo:

- O usuário conseguiu autenticar-se e recebe um JWT para ser usado nas próximas requisições.
- A autenticação falhou, pois um *token* inválido foi enviado.
- Não houve sequer autenticação, pois o usuário requisitado não existe.

Os passos do login são apresentados no teste abaixo. É feita uma requisição POST em /user/login, com as credenciais provenientes do conjunto de testes. Após a requisição, espera-se as seguintes consequências: uma requisição de resposta com *status* 200 deve ser retornada; um JWT deve vir no no cabeçalho indexado por x-auth; o

JWT deve ser salvo no banco de dados, assim que for criado. Essa última verificação é feita entre as linhas 19-22.

```
1  describe('POST /user/login', () => {
2    it('should login user and return auth token', (done) => {
3      request(app)
4        .post('/user/login')
5        .send({
6          username: userData.username,
7          password: userData.password
8        })
9        .expect(200)
10       .expect((res) => {
11         expect(res.headers['x-auth']).toBeTruthy();
12       })
13       .end((err, res) => {
14         if (err) {
15           return done(err);
16         }
17
18         User.findById(userData._id).then((user) => {
19           expect(user.toObject().tokens[1]).toMatchObject({
20             access: 'auth',
21             token: res.headers['x-auth']
22           });
23           done();
24         }).catch((e) => done(e));
25       });
26   });
27 };
```

Listing 5.1 – Teste de login com credenciais válidas

Os outros dois testes possuem um código parecido, com apenas algumas variações para se adequar a cada caso.

No segundo teste de falha, espera-se que a requisição retorne um status 400, informando que a senha está incorreta. Também, nenhum *token* deve vir no *header* e nada é inserido no array de *tokens* do usuário.

No terceiro teste, um *status* 400 também é retornado, mas com uma mensagem informando que o usuário não existe. Além disso, não deve haver nenhum JWT no *header* da requisição. Como o usuário não existe, não faz sentido pesquisar se um *token* foi gerado no banco.

O logout ilustrado a seguir, possui um fluxo mais simples: uma requisição DELETE é feita no *endpoint* /user/logout, fornecendo o JWT obtido anteriormente no login.

```
1  describe('DELETE /user/logout', () => {
2    it('should logout user and remove the auth token', (done)
       => {
3      request(app)
```

```
4         .delete('/user/logout')
5         .set('x-auth', userData.tokens[0].token)
6         .send({})
7         .expect(200)
8         .expect((res) => {
9           expect(res.headers['x-auth']).toBeFalsy();
10        })
11        .end((err, res) => {
12          if (err) {
13            return done(err);
14          }
15
16          User.findById(userData._id).then((user) => {
17            expect(user.toObject().tokens.find((token) =>
      token.access === 'auth')).toBeFalsy();
18            done();
19          }).catch((e) => done(e));
20        });
21    });
22
23  });
24  };
```

Listing 5.2 – Teste de logout com token válido

Se tudo estiver de acordo, uma requisição com *status* 200 será retornada, sem o *token* definido no *header*. Entre as linhas 18-24 é verificado se o JWT foi efetivamente removido do banco de dados.

O próximo teste foi a alteração de senha. O sistema foi implementado para vir com uma senha padrão, que pode e deve ser modificada pelo usuário. Nesse teste, uma resposta contendo *Password updated!* deve ser retornada. Além disso, o usuário deve ser buscado usando a senha modificada. Se isso acontecer, então o fluxo estará correto.

```
1   it('should not update user password', (done) => {
2     request(app)
3       .post('/user/newpassword')
4       .set('x-auth', userData.tokens[0].token)
5       .send({ newPassword: 'newPassword'})
6       .expect(200)
7       .expect((res) => {
8         expect(res.text).toBe(('Password updated'));
9       })
10      .end((err, res) => {
11        if (err) {
12          return done(err);
13        }
14
15        User.findByCredentials(userData.username, '
      newPassword').then((user) => {
```

```
16            if (!user) {
17              done('Password was not updated!')
18            }
19            done();
20          }).catch((e) => done(e));
21
22        });
23    });
24  };
```

Listing 5.3 – Teste de alteração da senha do usuário

Como um usuário pode adicionar um dispositivo, também foi feito um teste para contemplar isso. Adicionar um dispositivo segue o mesmo princípio de adicionar um serviço. Dessa forma, seus testes também são parecidos e podem ser explicados de forma conjunta. Para ilustrar, será usado o código de teste do dispositivo.

Para executar o teste, é criado um novo dispositivo com suas credenciais, conforme mostra abaixo.

```
1   it('should create a device for user', (done) => {
2     const newDeviceData = {
3       name: 'device_002',
4       password: 'fjsdfksjoirehr'
5     };
6
7     request(app)
8       .post('/device')
9       .set('x-auth', userData.tokens[0].token)
10      .send(newDeviceData)
11      .expect(200)
12      .expect((res) => {
13        expect(res.text).toBe('Success!');
14      })
15      .end((err, res) => {
16        if (err) {
17          return done(err);
18        }
19
20          Device.findByCredentials(newDeviceData.name,
    newDeviceData.password).then((device) => {
21            if (!device) {
22              done('Device does not exist!')
23            }
24            done();
25          }).catch((e) => done(e));
26        });
27    });
28  };
```

Listing 5.4 – Teste de criação de dispositivo

Antes de fazer a requisição, é incluído no *header* o *token* de um usuário do conjunto de dados do teste. Então é feita a requisição com as credenciais do dispositivo que deve ser criado. Espera-se uma resposta *Success!* e é verificado se o novo dispositivo foi adicionado ao banco de dados.

Um outro fluxo possível, seria tentar adicionar um dispositivo com o mesmo nome. Nesse caso, um status 400 é enviado, informando que a operação não pode ser feita.

Como pôde ser visto, um conjunto de testes foi implementado para cobrir vários casos possíveis sob o ponto de vista do usuário.

5.2 TESTES DO SERVIÇO

Do mesmo modo que o usuário, um serviço também deve estar autenticado para interagir com o servidor. Isso é feito de uma forma diferente, já que ele se conecta por um canal persistente.

As situações de login para o serviço são as mesmas que foram descritas no começo da seção do usuário. No entanto, o teste é um pouco diferente.

```
1   it('should connect', (done) => {
2     const _socket = ioClient.connect('http://localhost:8080'
      );
3
4     _socket.on('connect', function(data){
5       _socket.emit('authentication', {
6         username: serviceData.name,
7         password: serviceData.password });
8
9       _socket.on('authenticated', function() {
10        done();
11        _socket.disconnect();
12      });
13
14    });
15  });
16 };
```

Listing 5.5 – Teste de autenticação do serviço

Uma conexão *WebSocket* é iniciada e a autenticação dela é feita logo em seguida. Caso o servidor confirme as credenciais, um evento é disparado. O socket recebe esse evento, determinando que o serviço está autenticado.

No caso em que qualquer uma das credenciais estão erradas, o

socket recebe o evento *unauthorized* descrito na implementação e a conexão não acontece.

Além do login, um serviço pode fazer outras duas funcionalidades: pedir para receber os novos NDCs de um dispositivo ou também pedir para não receber mais novos NDCs. Abaixo está o teste, em que um serviço pede para escutar as atualizações de um dispositivo do conjunto de teste.

```
1   it('should listen to device', (done) => {
2     const _socket = ioClient.connect('http://localhost:8080'
      );
3
4     _socket.on('connect', function(data){
5       _socket.emit('authentication', {
6         username: serviceData.name,
7         password: serviceData.password });
8
9       _socket.on('authenticated', function() {
10        _socket.emit('join', { name: deviceData.name },
11        function (e) {
12          if (e) {
13            done('Join was not successful');
14            _socket.disconnect();
15          } else {
16            done();
17            _socket.disconnect();
18          }
19
20        });
21      });
22
23    });
24  });
25 };
```

Listing 5.6 – Teste para escutar um dispositivo

Quando a função done() é chamada sem parâmetro, o teste de fato passou. Se tiver parâmetro, a string fornecida indica o erro gerado.

Outro fluxo, seria o serviço querer escutar valores de um dispositivo que não existe. Nesse caso, a única diferença é que o done() não é mais chamado no else e sim quando vem um erro.

O teste para a funcionalidade de deixar de escutar um dispositivo, funciona de modo semelhante. A única diferença está na emissão do evento: ao invés de *join*, é emitido o evento *leave*.

5.3 TESTES DO DISPOSITIVO

Não muito diferente do usuário e do serviço, o *login* funciona da mesma forma já explicado anteriormente: possui os três fluxos possíveis. Abaixo está o teste com o fluxo de falha, em que não existe um dispositivo no banco de dados.

```
1   it('should fail due no device in database', (done) => {
2     request(app)
3       .post('/device/login')
4       .send({
5         name: 'fakedevice',
6         password: 'jf384jgerkv'
7       })
8       .expect(400)
9       .expect((res) => {
10        expect(res.text).toBe('Device does not exists');
11        expect(res.headers['x-auth']).toBeFalsy();
12      })
13      .end((err, res) => {
14        if (err) {
15          return done(err);
16        }
17        done();
18      });
19  });
20 };
```

Listing 5.7 – Teste de falha ao logar com um dispositivo não existente

A resposta deve ser um texto, informando que o dispositivo não existe, e nenhum JWT deve estar no cabeçalho da requisição de retorno.

O teste de logout é semelhante ao do usuário. Logo, não há necessidade de mostrar seus detalhes.

O último teste, embora seja do domínio do dispositivo, ele afeta diretamente os serviços que estão relacionados a ele. Quando a requisição for feita, os serviços recebem os valores enviados pelo dispositivo em questão. Logo, esse teste valida a integração dos dispositivos com os serviços.

```
1   it('should send data to service', (done) => {
2       const _socket = ioClient.connect('http://localhost
      :8080');
3
4       _socket.on('connect', function(data) {
5
6         _socket.emit('authentication', { username:
      serviceData.name,
```

```
7                                                password:
    serviceData.password });
8
9           _socket.on('authenticated', function() {
10              _socket.emit('join', { name: deviceData.name },
    function (e) {
11                  if (e) {
12                    done(e);
13                  } else {
14                    request(app)
15                      .post('/device/send')
16                      .set('x-auth', deviceData.tokens[0].
    token)
17                      .send({ value: '30' })
18                      .expect(200)
19                      .expect((res) => {
20                        expect(res.text).toBe('Success!');
21                      })
22                      .end((err, res) => {
23                        if (err) {
24                          return done(err);
25                        }
26                      });
27                  }
28                });
29
30              _socket.on('newValue', (data) => {
31                if (data.value === '30' && data.device ===
    deviceData.name) {
32                    done();
33                  } else {
34                    done('Incorrect data received!');
35                  }
36                });
37          });
38        });
39
40    });
41  };
```

Listing 5.8 – Teste de envio do NDC

Esse teste pode ser visto como uma composição de testes anteriores, com passos bem definidos. Os passos são detalhados a seguir:

- A conexão com o *WebSocket* é iniciada
- Um evento de autenticação é emitido com as credenciais corretas
- Quando estiver autenticado, o serviço emite outro evento, pedindo para escutar novos NDCs gerados pelo dispositivo do conjunto de teste

- Assim que o evento tiver sucesso, uma requisição de geração de NDC do dispositivo é simulada ao servidor, com o valor 30.
- O *socket* do serviço recebe um valor, através do evento *newValue*, junto com o nome do dispositivo.
- O valor recebido pelo *socket*, deve ser o mesmo fornecido no corpo da requisição.

Com esses testes, o essencial das funcionalidades da aplicação foram cobertos. Foram feitos 21 testes no total. Uma descrição resumida, gerada pelo Jest, pode ser vista a seguir.

```
POST /device/login
  ✓ should login device and return auth token (110ms)
  ✓ should fail device user auth (83ms)
  ✓ should fail due no device in database

DELETE /device/logout
  ✓ should logout device and remove the auth token

POST /device/send
  ✓ should send data to service (110ms)

Login service via websocket
  ✓ should connect (84ms)
  ✓ should fail service user auth (83ms)
  ✓ should fail due no service in database

Listening to devices
  ✓ should listen to device (89ms)
  ✓ should fail listening due no device in database (89ms)

Stop listening to devices
  ✓ should stop listen to device (89ms)
  ✓ should fail stop listening due no device in database (85ms)

POST /user/login
  ✓ should login user and return auth token (85ms)
  ✓ should fail login user auth (82ms)
  ✓ should fail due no user in database

DELETE /user/logout
  ✓ should logout user and remove the auth token

POST /device
  ✓ should create a device for user (160ms)
  ✓ should not create a device due duplicated name (81ms)

POST /service
  ✓ should create a service for user (158ms)
  ✓ should not create a service due duplicated name (81ms)

POST /user/newpassword
  ✓ should not update user password (159ms)


21 passing (13s)
```

Figura 4 Testes executados com sucesso

6 VALIDAÇÃO

Por fim, esse capítulo mostra um exemplo simples, de como um serviço pode usar o middleware implementado.

O aplicativo de celular criado consome níveis de confiabilidade, gerados por três dispositivos. Assim que é aberto, ele executa em *background* e deriva um novo valor, à medida que os dispositivos enviam novos valores.

O valor usado pela aplicação é derivado pela seguinte fórmula, uma média ponderada:

$$NDC_{atual} = \frac{NDC_1 * 5 + NDC_2 * 3 + NDC_3 * 2}{10}$$

Cada NDC é obtido em um tempo diferente. Dessa forma, o valor adapta-se de acordo com o peso daquele dispositivo. Os NDCs são aplicações *Web* com uma simples página HTML, conforme a figura 5. Elas possuem apenas a função de login, e envio de um NDC novo.

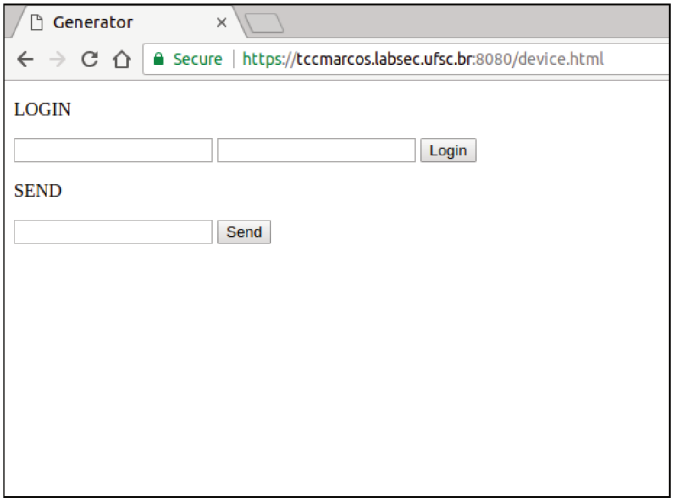

Figura 5 Tela de um dispositivo

Assim que o valor for calculado, ele é comparado a um *threshold* empírico, usado apenas para validação. Se estiver abaixo, a tela é bloqueada e o usuário deve desbloqueá-lo, usando o próprio sistema do

celular. O valor 60% foi definido como padrão.

A figura 6 mostra a tela inicial. Seu uso é apenas informativo, para detectar que o processo está rodando.No momento que a aplicação abriu, o *WebSocket* é estabelecido com o servidor do LABSEC.

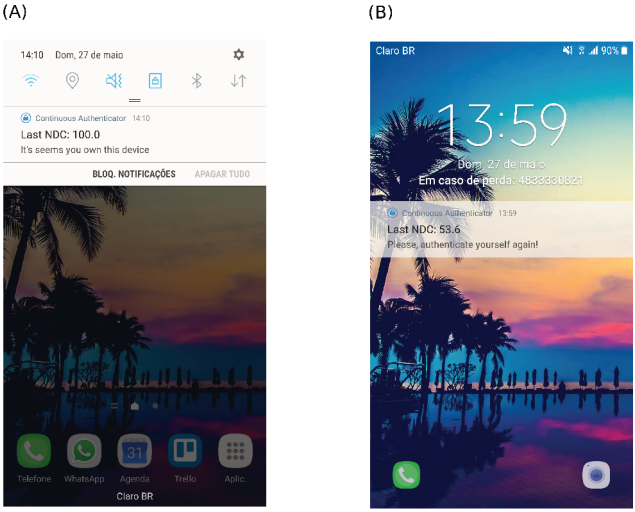

Figura 6 Tela inicial do aplicativo (A) e escolha de dispositivos (B)

Clicando no ícone do canto direito inferior, é possível decidir quais dispositivos deseja-se obter o NDC atualizado. A figura 6 também mostra essa tela, e para escolher, basta marcar o checkbox específico.

Toda vez que um valor for gerado, a barra de status é atualizada com ele. Em (A) na figura 6 mostra como que o aplicativo apresenta essa informação.

Nas seções seguintes são mostrados três casos: dois deles, a autenticação é mantida, e no outro a tela é bloqueada. Cada etapa será descrita, de modo que a validação possa ser reproduzida. Será assumido que o NDC começa em 100% para facilitar a visualização dos cálculos.

6.1 PRIMEIRO CASO

Nessa primeira situação, o NDC3 irá receber apenas 10%. Não é necessário saber os valores do NDC1 nem do NDC2, mas se sabe que o valor combinado deles é o atual, logo:

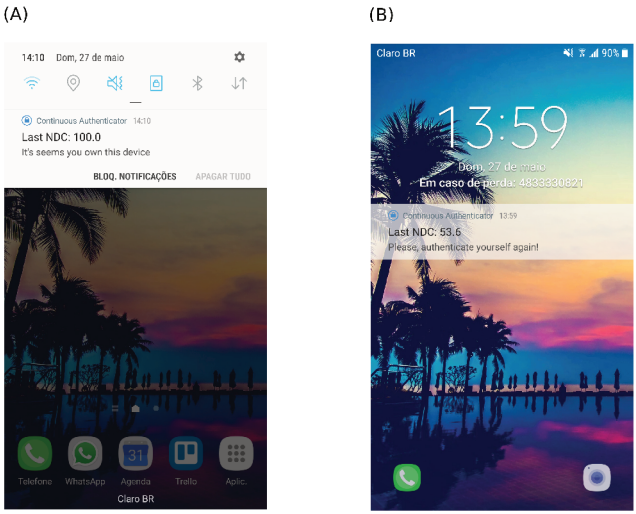

Figura 7 Status bar com NDC (A) e tela bloqueada com threshold abaixo (B)

$$NDC_{atual} = NDC_1 * 5 + NDC_2 * 3 = 100 * 8$$

Com isso, para gerar o novo NDC, basta fazer:

$$NDC_{novo} = \frac{100 * 8 + 10 * 2}{10}$$

O novo valor obtido é 82%. Como é maior que o *threshold* definido, o celular mantem-se autenticado.)

6.2 SEGUNDO CASO

Nesse próximo caso, o NDC2 irá receber apenas 15%. Aplicando o mesmo princípio usado no caso anterior:

$$NDC_{atual} = NDC_1 * 5 + NDC_2 * 2 = 100 * 7$$

Com isso, para gerar o novo NDC, basta fazer:

$$NDC_{novo} = \frac{100 * 7 + 15 * 3}{10}$$

O novo valor obtido é 74,5%. Do mesmo modo, o celular mantem-se autenticado.

6.3 TERCEIRO CASO

Nesse próximo caso, o NDC1 irá receber apenas 12%. Aplicando o mesmo princípio usado nos outros casos:

$$NDC_{atual} = NDC_2 * 3 + NDC_2 * 2 = 100 * 5$$

Com isso, para gerar o novo NDC, basta fazer:

$$NDC_{novo} = \frac{100 * 5 + 12 * 3}{10}$$

O novo valor obtido é 53,6%. Nesta situação, o celular é bloqueado. Conforme mostra a Figura 7, aparece uma mensagem informando que um novo desbloqueio pelo celular deve ser feito.

7 CONCLUSÃO

A autenticação é algo cuja unica constante em seu estudo é a mudança. Novos trabalhos são propostos a todo momento, com o intuito de aumentar a segurança e a usabilidade do usuário. Nos trabalhos pesquisados, pode-se perceber os seus diversos tipos, e como sua combinação pode melhorar esse processo.

A elaboração da arquitetura e sua implementação foram feitas com sucesso. Com a verificação através de testes unitários é garantido uma manutenibilidade segura do código. Junto disso, fazendo uma validação com um simples aplicativo, pode-se observar como seria o uso do resultado final.

Dessa forma, os objetivos finais foram alcançados e assim um novo modelo multi-fator proposto. Diferente dos outros, ele permite que qualquer autenticador contínuo seja usado. No entanto, ele deve estar de acordo com a especificação da API descrita no texto.

Ainda assim, há diversos pontos que podem ser melhorados. Em relação aos trabalhos futuros, existem algumas alternativas a serem exploradas:

- Fazer um experimento com uma amostra de usuários adequada, usando dispositivos de autenticação contínua reais, para obter um feedback mais aprimorado da proposta.
- Melhorar questões de implementação do middleware: permitir mais de um usuário, deixar que o usuário gerencie quais dispositivos podem ser usados pelos serviços, desenvolver uma interface de controle para o usuário, etc.
- Procurar possíveis brechas na arquitetura proposta, assim como na implementação feita, e propor soluções para as mesmas.
- Propor uma análise matemática em cima dos valores gerados: será que a aritmética ponderada é a melhor forma? Verificar diferentes formas de criar o NDC mais atual. Feito isso, discutir qual delas é mais apropriada sob o ponto de visto de usabilidade e segurança.

REFERÊNCIAS

AZZINI, A. et al. A fuzzy approach to multimodal biometric continuous authentication. *Fuzzy Optimization and Decision Making*, Springer, v. 7, n. 3, p. 243, 2008.

BRADLEY, J.; SAKIMURA, N.; JONES, M. Json web token (jwt). 2015.

BRÖMME, A.; AL-ZUBI, S. Multifactor biometric sketch authentication. In: *BIOSIG*. [S.l.: s.n.], 2003. p. 81–90.

CARRIGAN, J.; MARTIN, P.; RUSHANAN, M. Kbid: Kerberos bracelet identification (short paper). In: SPRINGER. *International Conference on Financial Cryptography and Data Security*. [S.l.], 2016. p. 544–551.

DENNING, D. E.; MACDORAN, P. F. Location-based authentication: Grounding cyberspace for better security. *Computer Fraud & Security*, Elsevier, v. 1996, n. 2, p. 12–16, 1996.

FRANK, M. et al. Touchalytics: On the applicability of touchscreen input as a behavioral biometric for continuous authentication. *IEEE transactions on information forensics and security*, IEEE, v. 8, n. 1, p. 136–148, 2013.

KARAPANOS, N. et al. Sound-proof: Usable two-factor authentication based on ambient sound. In: *USENIX Security Symposium*. [S.l.: s.n.], 2015. p. 483–498.

NIINUMA, K.; JAIN, A. K. Continuous user authentication using temporal information. In: *Proc. SPIE*. [S.l.: s.n.], 2010. v. 7667, n. 1, p. 76670L.

SABZEVAR, A. P.; STAVROU, A. Universal multi-factor authentication using graphical passwords. In: IEEE. *Signal Image Technology and Internet Based Systems, 2008. SITIS'08. IEEE International Conference on*. [S.l.], 2008. p. 625–632.

STAJANO, F. Pico: No more passwords! In: SPRINGER. *Security Protocols Workshop*. [S.l.], 2011. v. 7114, p. 49–81.

TIWARI, A. et al. A multi-factor security protocol for wireless payment-secure web authentication using mobile devices. *arXiv preprint arXiv:1111.3010*, 2011.

YAP, R. H. et al. Physical access protection using continuous authentication. In: IEEE. *Technologies for Homeland Security, 2008 IEEE Conference on*. [S.l.], 2008. p. 510–512.

ZHANG, F.; KONDORO, A.; MUFTIC, S. Location-based authentication and authorization using smart phones. In: IEEE. *Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on*. [S.l.], 2012. p. 1285–1292.

APÊNDICE A – Código Fonte

```
    {
      "name": "auth-app",
      "version": "1.0.0",
      "main": "index.js",
      "license": "MIT",
      "scripts": {
        "start": "node server/app.js",
        "test": "mocha server/**/*.test.js"
      },
      "dependencies": {
        "bcryptjs": "^2.4.3",
        "body-parser": "^1.18.2",
        "express": "^4.16.2",
        "jsonwebtoken": "^8.2.0",
        "mongoose": "^4.5.9",
        "nodemon": "^1.17.1",
        "socket.io": "^2.0.4",
        "socketio-auth": "^0.1.0"
      },
      "devDependencies": {
        "eslint": "^4.19.1",
        "jest": "^22.4.3",
        "mocha": "^3.0.2",
        "supertest": "^2.0.0"
      }
    }
};
```

Listing A.1 – package.json

```
    const mongoose = require('mongoose');
    const bcrypt = require('bcryptjs');
    const crypto = require('crypto');
    const jwt = require('jsonwebtoken');

    function generateRandomNumber(length) {
      return crypto.randomBytes(Math.ceil(length/2)).
    toString('hex').slice(0, length);
    }

    const JWT_SECRET_SIZE = 1024;
    const JWT_SECRET_TIME = 24 * 60 * 1000;
    let JWT_SECRET_USER = generateRandomNumber(
    JWT_SECRET_SIZE);
    setInterval(() => {
      JWT_SECRET_USER = crypto.randomBytes(Math.ceil(
    JWT_SECRET_SIZE/2)).toString('hex').slice(0,
    JWT_SECRET_SIZE);
    }, JWT_SECRET_TIME);

    var UserSchema = new mongoose.Schema({
```

```
18        username: {
19          type: String,
20          required: true,
21          trim: true,
22          minlength: 1,
23          unique: true
24        },
25        password: {
26          type: String,
27          require: true,
28          minlength: 6
29        },
30        tokens: [{
31          access: {
32            type: String,
33            require: true
34          },
35          token: {
36            type: String,
37            require: true
38          }
39        }]
40      });
41
42      UserSchema.methods.generateAuthToken = function() {
43        var user = this;
44        var access = 'auth';
45        var token = jwt.sign({ _id: user._id.toHexString(),
        access },
46                             JWT_SECRET_USER,
47                             { expiresIn: '12h'}).toString();
48
49        user.tokens = user.tokens.concat([{ access, token }]);
50
51        return user.save().then(() => {
52          return token;
53        });
54      };
55
56      UserSchema.statics.findByCredentials = function (
        username, password) {
57        var User = this;
58
59        return User.findOne({ username }).then((user) => {
60          if (!user) {
61            return Promise.reject('User doesnt exists');
62          }
63
64          return new Promise((resolve, reject) => {
65            bcrypt.compare(password, user.password, function (
        err, res) {
66              if (res) {
```

```
67                resolve(user);
68              } else {
69                reject('Password incorrect');
70              }
71            });
72          });
73        });
74      };
75
76      UserSchema.statics.findByToken = function (token) {
77        var User = this;
78        var decoded;
79
80        try {
81          decoded = jwt.verify(token, JWT_SECRET_USER);
82        } catch (e) {
83          return Promise.reject(`Invalid JWT.`);
84        }
85
86        return User.findOne({
87          '_id': decoded._id,
88          'tokens.token': token,
89          'tokens.access': 'auth'
90        })
91      };
92
93      UserSchema.methods.removeToken = function (token) {
94        var user = this;
95
96        return user.update({
97          $pull: {
98            tokens: {
99              token
100           }
101         }
102       });
103     };
104
105     UserSchema.methods.updatePassword = function (
        newPassword) {
106       var user = this;
107       user.set({ password: newPassword });
108       return user.save();
109     };
110
111     UserSchema.pre('save', function (next) {
112       const user = this;
113       if (user.isModified('password')) {
114         bcrypt.genSalt(10, (err, salt) => {
115           bcrypt.hash(user.password, salt, (err, hash) => {
116             user.password = hash;
117             next();
```

```
118             });
119           });
120         } else {
121           next();
122         }
123       });
124
125       var User = mongoose.model('User', UserSchema);
126
127       module.exports = { User, JWT_SECRET_USER };
128   };
```

Listing A.2 – user.js

```
1   const bcrypt = require('bcryptjs');
2   const mongoose = require('mongoose');
3
4   const { Device } = require('./device');
5
6   const ServiceSchema = new mongoose.Schema({
7     name: {
8       type: String,
9       required: true,
10      trim: true,
11      minlength: 5,
12      unique: true
13    },
14    password: {
15      type: String,
16      require: true,
17      minlength: 6
18    },
19    devices: {
20      type: [String]
21    }
22  });
23
24
25  ServiceSchema.pre('save', function (next) {
26    const service = this;
27
28    if (service.isModified('password')) {
29      bcrypt.genSalt(10, (err, salt) => {
30        bcrypt.hash(service.password, salt, (err, hash) => {
31          service.password = hash;
32          next();
33        });
34      });
35    } else {
36      next();
37    }
38  });
```

```
39
40 ServiceSchema.statics.findByCredentials = function (name,
       password) {
41   var Service = this;
42
43   return Service.findOne({ name }).then((service) => {
44     if (!service) {
45       return new Promise.reject();
46     }
47
48     return new Promise((resolve, reject) => {
49       bcrypt.compare(password, service.password, function (
       err, res) {
50         if (res) {
51           resolve(service);
52         } else {
53           reject();
54         }
55       });
56     });
57   }).catch(() => {
58     throw new Error('Error trying to login!');
59   });
60 };
61
62 ServiceSchema.statics.addDeviceToService = function (
       nameService, nameDevice) {
63   var Service = this;
64
65   return Service.findOne({ 'name': nameService }).then((
       service) => {
66     if (!service) {
67       throw new Error('No service found');
68     }
69     return Device.findOne({ 'name': nameDevice }).then((
       device) => {
70       if (!device) {
71         throw new Error('No device found');
72       }
73
74       service.devices = service.devices.concat(nameDevice);
75       service.save().then((service) => {
76         return service;
77       });
78     });
79   });
80 };
81
82 ServiceSchema.statics.removeDeviceFromService = function (
       nameService, nameDevice) {
83   var Service = this;
84
```

```
85    return Service.findOne({ 'name': nameService }).then((
        service) => {
86      if (!service) {
87        throw new Error('No service found');
88      }
89
90      return Device.findOne({ 'name': nameDevice }).then((
        device) => {
91        if (!device) {
92          throw new Error('No device found');
93        }
94
95        service.devices = service.devices.filter((name) =>
      name !== nameDevice);
96        service.save().then((service) => {
97          return service;
98        });
99      });
100   });
101 };
102
103 const Service = mongoose.model('Service', ServiceSchema);
104
105 module.exports = { Service };
106 };
```

Listing A.3 – service.js

```
1  const mongoose = require('mongoose');
2
3  const NDCSchema = new mongoose.Schema({
4    deviceName: {
5      type: String,
6      required: true,
7      trim: true,
8      minlength: 5
9    },
10   moment: {
11     type: Date,
12     default: Date.now,
13     require: true,
14   },
15   data: {
16     type: String
17   }
18 });
19
20 const NDC = mongoose.model('NDC', NDCSchema);
21
22 module.exports = { NDC };
23 };
```

Listing A.4 – ndc.js

```
var mongoose = require('mongoose');

mongoose.Promise = global.Promise;
mongoose.connect(process.env.MONGODB_URI);

module.exports = { mongoose };
};
```

Listing A.5 – mongoose.js

```
const { User } = require('./user');
const { Device } = require('./device');

const authenticateUser = (req, res, next) => {
  const token = req.header('x-auth');

  User.findByToken(token).then((user) => {
    if (!user) {
      return Promise.reject(`No JWT found.`);
    }

    req.user = user;
    req.token = token;
    next();
  }).catch((e) => {
    res.status(401).send(e);
  });
};

const authenticateDevice = (req, res, next) => {
  const token = req.header('x-auth');
  Device.findByToken(token).then((device) => {
    if (!device) {
      return Promise.reject();
    }

    req.device = device;
    req.token = token;
    next();
  }).catch((e) => {
    res.status(401).send(e);
  });
}

module.exports = { authenticateUser, authenticateDevice };
};
```

Listing A.6 – middleware.js

```
const bcrypt = require('bcryptjs');
const crypto = require('crypto');
const jwt = require('jsonwebtoken');
const mongoose = require('mongoose');

const generateRandomNumber = (length) => {
  return crypto.randomBytes(Math.ceil(length/2)).toString('
    hex').slice(0, length);
}

const JWT_SECRET_SIZE = 1024;
const JWT_SECRET_TIME = 24 * 60 * 1000;
let JWT_SECRET_DEVICE = generateRandomNumber(JWT_SECRET_SIZE
    );
setInterval(() => {
  JWT_SECRET_DEVICE = crypto.randomBytes(Math.ceil(
    JWT_SECRET_SIZE/2)).toString('hex').slice(0,
    JWT_SECRET_SIZE);
}, JWT_SECRET_TIME);

const DeviceSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
    trim: true,
    minlength: 5,
    unique: true
  },
  password: {
    type: String,
    require: true,
    minlength: 6
  },
  tokens: [{
    access: {
      type: String,
      require: true
    },
    token: {
      type: String,
      require: true
    }
  }]
});


DeviceSchema.pre('save', function (next) {
  const device = this;

  if (device.isModified('password')) {
    bcrypt.genSalt(10, (err, salt) => {
```

```
48        bcrypt.hash(device.password, salt, (err, hash) => {
49          device.password = hash;
50          next();
51        });
52      });
53    } else {
54      next();
55    }
56  });
57
58  DeviceSchema.statics.findByCredentials = function (name,
      password) {
59    var Device = this;
60
61    return Device.findOne({ name }).then((device) => {
62      if (!device) {
63        return Promise.reject(‘Device doesnt exists‘);
64      }
65
66      return new Promise((resolve, reject) => {
67        bcrypt.compare(password, device.password, function (
      err, res) {
68          if (res) {
69            resolve(device);
70          } else {
71            reject('Password incorrect');
72          }
73        });
74      });
75    });
76  };
77
78  DeviceSchema.methods.generateAuthToken = function () {
79    var device = this;
80
81    const access = 'auth';
82    const token = jwt.sign({ _id: device._id.toHexString(),
      access },
83                           JWT_SECRET_DEVICE,
84                           { expiresIn: '12h'}).toString();
85
86    device.tokens = device.tokens.concat([{ access, token }]);
87
88    return device.save().then(() => {
89      return token;
90    });
91  };
92
93  DeviceSchema.statics.findByToken = function (token) {
94    var Device = this;
95    var decoded;
96
```

```
  try {
    decoded = jwt.verify(token, JWT_SECRET_DEVICE);
  } catch (e) {
    return Promise.reject();
  }

  return Device.findOne({
    '_id': decoded._id,
    'tokens.token': token,
    'tokens.access': 'auth'
  });
};

DeviceSchema.methods.removeToken = function (token) {
  const device = this;

  return device.update({
    $pull: {
      tokens: {
        token
      }
    }
  });
};

const Device = mongoose.model('Device', DeviceSchema);

module.exports = { Device, JWT_SECRET_DEVICE };
};
```

Listing A.7 – device.js

```
const path = require('path');
const express = require('express');
const socketIO = require('socket.io');
const bodyParser = require('body-parser');

const publicPath = path.join(__dirname, '../public');

const app = express();
app.use(bodyParser.json());
app.use(express.static(publicPath));

let server;
if (process.env.NODE_ENV === 'production') {
  const fs = require('fs');

  const options = {};
  options.key = fs.readFileSync('private-key.pem');
  options.cert = fs.readFileSync('labsec.crt');
  options.ca = fs.readFileSync('cert-chain.crt');

```

```
  server = require('https').createServer(options, app);
} else {
  server = require('http').createServer(app);
}

const io = socketIO(server);

module.exports = { app, io, server };
};
```

Listing A.8 – config.js

```
require('./mongoose');
const { app, io, server } = require('./config');

const { authenticateUser, authenticateDevice } = require('./
    middleware');
const { User } = require('./user');

const { Device } = require('./device');
const { Service } = require('./service');
const { NDC } = require('./ndc');

app.post('/user/login', async (req, res) => {
  try {
    const user = await User.findByCredentials(req.body.
    username, req.body.password);
    const token = await user.generateAuthToken();
    res.header('x-auth', token).send();
  } catch (e) {
    res.status(400).send(e);
  }

});

app.post('/user/newpassword', authenticateUser, async (req,
    res) => {
  try {
    await req.user.updatePassword(req.body.newPassword);
    res.status(200).send('Password updated!');
  } catch (e) {
    res.status(400).send(e);
  }
});

app.delete('/user/logout', authenticateUser, async (req, res
    ) => {
  try {
    await req.user.removeToken(req.token);
    res.status(200).send();
  } catch (e) {
    res.status(400).send();
```

```
37    }
38  });
39
40  app.post('/device', authenticateUser, (req, res) => {
41    const name = req.body.name;
42    const password = req.body.password;
43
44    const device = new Device({ name, password });
45    device.save().then(() => {
46      res.send('Success!');
47    }).catch((e) => {
48      res.status(400).send(e);
49    });
50
51  });
52
53  app.post('/service', authenticateUser, (req, res) => {
54    const name = req.body.name;
55    const password = req.body.password;
56
57    const service = new Service({ name, password });
58    service.save().then(() => {
59      res.send('Success!');
60    }).catch((e) => {
61      res.status(400).send(e);
62    });
63
64  });
65
66  app.post('/device/login', async (req, res) => {
67    try {
68      const device = await Device.findByCredentials(req.body.
        name, req.body.password);
69      const token = await device.generateAuthToken();
70      res.header('x-auth', token).send('Success!');
71    } catch (e) {
72      res.status(400).send(e);
73    }
74  });
75
76  app.delete('/device/logout', authenticateDevice, async (req,
        res) => {
77    try {
78      await req.device.removeToken(req.token);
79      res.status(200).send();
80    } catch (e) {
81      res.status(400).send();
82    }
83  });
84
85  require('socketio-auth')(io, {
86
```

```
87    authenticate: function (socket, data, callback) {
88      Service.findByCredentials(data.username, data.password).
        then((service) => {
89        socket.name = service.name;
90        service.devices.forEach((deviceName) => socket.join(
        deviceName));
91        return callback(null, true);
92      }).catch(() => {
93        return callback(new Error("User not found"));
94      });
95    }
96  });
97
98  io.on('connection', (socket) => {
99
100   socket.on('join', (params, callback) => {
101     Service.addDeviceToService(socket.name, params.name).
        then(() => {
102       socket.join(params.name);
103       callback();
104     }).catch((e) => {
105       callback(e.message);
106     });
107   });
108
109   socket.on('leave', (params, callback) => {
110     Service.removeDeviceFromService(socket.name, params.name
        ).then(() => {
111       socket.leave(params.name);
112       callback();
113     }).catch((e) => {
114       callback(e.message);
115     });
116   });
117
118 });
119
120
121 app.post('/device/send', authenticateDevice, (req, res) => {
122   const device = req.device;
123   const value = req.body.value
124
125   const NDCData = {
126     device: device.name,
127     value: value
128   }
129
130   const newNDC = new NDC(NDCData);
131   newNDC.save(() => {
132     io.to(req.device.name).emit('newValue', NDCData);
133     res.status(200).send('Success!');
134   });
```

```
135
136 });
137
138 server.listen(process.env.PORT);
139
140 module.exports = { app };
141
142 };
```

Listing A.9 – app.js

```
1  const expect = require('expect');
2  const request = require('supertest');
3
4  const { User } = require('./../user');
5  const { Device } = require('./../device');
6  const { Service } = require('./../service');
7  const { app } = require('./../app');
8  const { createDefaultUser,
9          createDefaultDevice,
10         createDefaultService,
11         userData,
12         deviceData,
13         serviceData } = require('./seed');
14
15 beforeEach(createDefaultUser);
16 beforeEach(createDefaultDevice);
17 beforeEach(createDefaultService);
18
19 describe('POST /user/login', () => {
20   it('should login user and return auth token', (done) => {
21     request(app)
22       .post('/user/login')
23       .send({
24         username: userData.username,
25         password: userData.password
26       })
27       .expect(200)
28       .expect((res) => {
29         expect(res.headers['x-auth']).toBeTruthy();
30       })
31       .end((err, res) => {
32         if (err) {
33           return done(err);
34         }
35
36         User.findById(userData._id).then((user) => {
37           expect(user.toObject().tokens[1]).toMatchObject({
38             access: 'auth',
39             token: res.headers['x-auth']
40           });
41           done();
```

```
42          }).catch((e) => done(e));
43        });
44    });
45
46    it('should fail login user auth', (done) => {
47      request(app)
48        .post('/user/login')
49        .send({
50          username: userData.username,
51          password: 'fklsdjfosjfosjof'
52        })
53        .expect(400)
54        .expect((res) => {
55          expect(res.text).toBe(`Password incorrect`);
56          expect(res.headers['x-auth']).toBeFalsy();
57        })
58        .end((err, res) => {
59          if (err) {
60            return done(err);
61          }
62
63          User.findById(userData._id).then((user) => {
64            expect(user.toObject().tokens).toMatchObject(
      userData.tokens);
65            done();
66          }).catch((e) => done(e));
67        });
68    });
69
70    it('should fail due no user in database', (done) => {
71      request(app)
72        .post('/user/login')
73        .send({
74          username: 'fakeuser',
75          password: 'fklsdjfosjfosjof'
76        })
77        .expect(400)
78        .expect((res) => {
79          expect(res.text).toBe(`User doesnt exists`);
80          expect(res.headers['x-auth']).toBeFalsy();
81        })
82        .end((err, res) => {
83          if (err) {
84            return done(err);
85          }
86          done();
87        });
88    });
89
90  });
91
92  describe('DELETE /user/logout', () => {
```

```
93    it('should logout user and remove the auth token', (done)
        => {
94      request(app)
95        .delete('/user/logout')
96        .set('x-auth', userData.tokens[0].token)
97        .send({})
98        .expect(200)
99        .expect((res) => {
100         expect(res.headers['x-auth']).toBeFalsy();
101       })
102       .end((err, res) => {
103         if (err) {
104           return done(err);
105         }
106
107         User.findById(userData._id).then((user) => {
108           expect(user.toObject().tokens.find((token) =>
      token.access === 'auth')).toBeFalsy();
109           done();
110         }).catch((e) => done(e));
111       });
112   });
113
114 });
115
116 describe('POST /device', () => {
117   it('should create a device for user', (done) => {
118     const newDeviceData = {
119       name: 'device_002',
120       password: 'fjsdfksjoirehr'
121     };
122
123     request(app)
124       .post('/device')
125       .set('x-auth', userData.tokens[0].token)
126       .send(newDeviceData)
127       .expect(200)
128       .expect((res) => {
129         expect(res.text).toBe('Success!');
130       })
131       .end((err, res) => {
132         if (err) {
133           return done(err);
134         }
135
136         Device.findByCredentials(newDeviceData.name,
      newDeviceData.password).then((device) => {
137           if (!device) {
138             done(`Device doesnt exist!`)
139           }
140           done();
141         }).catch((e) => done(e));
```

```
142         });
143     });
144
145     it('should not create a device due duplicated name', (done
        ) => {
146       request(app)
147         .post('/device')
148         .set('x-auth', userData.tokens[0].token)
149         .send(deviceData)
150         .expect(400)
151         .end(done)
152     });
153
154 });
155
156 describe('POST /service', () => {
157     it('should create a service for user', (done) => {
158       const newServiceData = {
159         name: 'service_002',
160         password: 'fjsdfksjoirehr'
161       };
162
163       request(app)
164         .post('/service')
165         .set('x-auth', userData.tokens[0].token)
166         .send(newServiceData)
167         .expect(200)
168         .expect((res) => {
169           expect(res.text).toBe('Success!');
170         })
171         .end((err, res) => {
172           if (err) {
173             return done(err);
174           }
175
176           Service.findByCredentials(newServiceData.name,
        newServiceData.password).then((service) => {
177             if (!service) {
178               done(`Service doesnt exist!`)
179             }
180             done();
181           }).catch((e) => done(e));
182         });
183     });
184
185     it('should not create a service due duplicated name', (
        done) => {
186       request(app)
187         .post('/service')
188         .set('x-auth', userData.tokens[0].token)
189         .send(serviceData)
190         .expect(400)
```

```
191         .end(done)
192     });
193
194 });
195
196 describe('POST /user/newpassword', () => {
197
198     it('should not update user password', (done) => {
199       request(app)
200         .post('/user/newpassword')
201         .set('x-auth', userData.tokens[0].token)
202         .send({ newPassword: 'newPassword'})
203         .expect(200)
204         .end((err, res) => {
205           if (err) {
206             return done(err);
207           }
208
209           User.findByCredentials(userData.username, '
      newPassword').then((user) => {
210             if (!user) {
211               done(`Password was not updated!`)
212             }
213             done();
214           }).catch((e) => done(e));
215
216         });
217     });
218
219 });
220 };
```

Listing A.10 – user.test.js

```
1  const expect = require('expect');
2  const request = require('supertest');
3
4  const { Device } = require('./../device');
5  const { app, server } = require('./../app');
6
7  const { createDefaultDevice,
8          createDefaultService,
9          deviceData,
10         serviceData } = require('./seed');
11
12 const ioClient = require('socket.io-client');
13
14 beforeEach(createDefaultDevice);
15 beforeEach(createDefaultService);
16
17 describe('Login service via websocket', () => {
18   it('should connect', (done) => {
```

```
19      const _socket = ioClient.connect('http://localhost:8080'
        );
20
21      _socket.on('connect', function(data){
22        _socket.emit('authentication', { username: serviceData
        .name,
23                                         password: serviceData.
        password });
24
25        _socket.on('authenticated', function() {
26          done();
27          _socket.disconnect();
28        });
29
30      });
31    });
32
33    it('should fail service user auth', (done) => {
34      const _socket = ioClient.connect('http://localhost:8080'
        );
35
36      _socket.on('connect', function(data){
37        _socket.emit('authentication', { username: serviceData
        .name,
38                                         password: '
        fakepassword' });
39
40        _socket.on('unauthorized', function(err){
41          done();
42          _socket.disconnect();
43        });
44
45      });
46    });
47
48    it('should fail due no service in database', (done) => {
49      const _socket = ioClient.connect('http://localhost:8080'
        );
50
51      _socket.on('connect', function(data){
52        _socket.emit('authentication', { username: '
        fakeservice',
53                                         password: serviceData.
        password });
54
55        _socket.on('unauthorized', function(err){
56          done();
57          _socket.disconnect();
58        });
59
60      });
61    });
```

```
62
63 });
64
65 describe('Listening to devices', () => {
66   it('should listen to device', (done) => {
67     const _socket = ioClient.connect('http://localhost:8080'
       );
68
69     _socket.on('connect', function(data){
70       _socket.emit('authentication', { username: serviceData
       .name,
71           password: serviceData.password });
72
73       _socket.on('authenticated', function() {
74         _socket.emit('join', { name: deviceData.name },
       function (e) {
75           if (e) {
76             done(`Join wasnt successful`);
77             _socket.disconnect();
78           } else {
79             done();
80             _socket.disconnect();
81           }
82
83         });
84       });
85
86     });
87   });
88
89   it('should fail listening due no device in database', (
       done) => {
90     const _socket = ioClient.connect('http://localhost:8080'
       );
91
92     _socket.on('connect', function(data){
93       _socket.emit('authentication', { username: serviceData
       .name,
94           password: serviceData.password });
95
96         _socket.on('authenticated', function() {
97           _socket.emit('join', 'fakedevice', function (e) {
98             if (e) {
99               done();
100              _socket.disconnect();
101            } else {
102              done(`Join was successful`);
103              _socket.disconnect();
104            }
105          });
106      });
107
```

```
108       });
109     });
110
111   });
112
113   describe('Stop listening to devices', () => {
114     it('should stop listen to device', (done) => {
115       const _socket = ioClient.connect('http://localhost:8080'
          );
116
117       _socket.on('connect', function(data){
118         _socket.emit('authentication', { username: serviceData
          .name,
119             password: serviceData.password });
120
121         _socket.on('authenticated', function() {
122           _socket.emit('leave', { name: deviceData.name },
          function (e) {
123             if (e) {
124               done(`Leave wasnt successful`);
125               _socket.disconnect();
126             } else {
127               done();
128               _socket.disconnect();
129             }
130
131           });
132         });
133
134       });
135     });
136
137     it('should fail stop listening due no device in database',
           (done) => {
138       const _socket = ioClient.connect('http://localhost:8080'
          );
139
140       _socket.on('connect', function(data) {
141         _socket.emit('authentication', { username: serviceData
          .name,
142             password: serviceData.password });
143
144           _socket.on('authenticated', function() {
145             _socket.emit('join', 'fakedevice', function (e) {
146               if (e) {
147                 done();
148                 _socket.disconnect();
149               } else {
150                 done(`Join was successful`);
151                 _socket.disconnect();
152               }
153             });
```

```
154         });
155
156       });
157     });
158
159 });
160 };
```

Listing A.11 – service.test.js

```
1  const jwt = require('jsonwebtoken');
2
3  const { User, JWT_SECRET_USER } = require('./../user');
4  const { Device, JWT_SECRET_DEVICE } = require('./../device')
       ;
5  const { Service } = require('./../service');
6
7  const { ObjectID } = require('mongodb');
8
9  const userId = new ObjectID();
10 const userData = {
11   _id: userId,
12   username: 'usertest',
13   password: 'lkdf59083589',
14   tokens: [{
15     access: 'auth',
16     token: jwt.sign({ _id: userId, access: 'auth'},
       JWT_SECRET_USER).toString()
17   }]
18 };
19
20 const deviceId = new ObjectID();
21 const deviceData = {
22   _id: deviceId,
23   name: 'device_001',
24   password: 'jfsdkiw9843fjk',
25   tokens: [{
26     access: 'auth',
27     token: jwt.sign({ _id: deviceId, access: 'auth'},
       JWT_SECRET_DEVICE).toString()
28   }]
29 }
30
31 const serviceId = new ObjectID();
32 const serviceData = {
33   _id: serviceId,
34   name: 'service_001',
35   password: 'fkjsfk9839832'
36 }
37
38 const createDefaultUser = (done) => {
39   User.remove({}).then(() => {
```

```
    const user = new User(userData);
    return user.save();
  }).then(() => done());
};

const createDefaultDevice = (done) => {
  Device.remove({}).then(() => {
    const device = new Device(deviceData);
    return device.save();
  }).then(() => done());
};

const createDefaultService = (done) => {
  Service.remove({}).then(() => {
    const service = new Service(serviceData);
    return service.save();
  }).then(() => done());
};

module.exports = { createDefaultUser, createDefaultDevice,
  createDefaultService, userData, deviceData, serviceData };
};
```

Listing A.12 – seed.js

```
const expect = require('expect');
const request = require('supertest');

const { Device } = require('./../device');
const { app } = require('./../app');

const { createDefaultDevice,
        createDefaultService,
        deviceData,
        serviceData } = require('./seed');

const ioClient = require('socket.io-client');

beforeEach(createDefaultDevice);
beforeEach(createDefaultService);

describe('POST /device/login', () => {
  it('should login device and return auth token', (done) =>
    {
    request(app)
      .post('/device/login')
      .send({
        name: deviceData.name,
        password: deviceData.password
      })
      .expect(200)
      .expect((res) => {
```

```
27          expect(res.headers['x-auth']).toBeTruthy();
28        })
29        .end((err, res) => {
30          if (err) {
31            return done(err);
32          }
33
34          Device.findOne({ name: deviceData.name }).then((
      device) => {
35            expect(device.toObject().tokens[1]).toMatchObject
      ({
36              access: 'auth',
37              token: res.headers['x-auth']
38            });
39            done();
40          }).catch((e) => done(e));
41        });
42    });
43
44     it('should fail device user auth', (done) => {
45      request(app)
46        .post('/device/login')
47        .send({
48          name: deviceData.name,
49          password: 'fklsdjfosjfosjof'
50        })
51        .expect(400)
52        .expect((res) => {
53          expect(res.text).toBe(`Password incorrect`);
54          expect(res.headers['x-auth']).toBeFalsy();
55        })
56        .end((err, res) => {
57          if (err) {
58            return done(err);
59          }
60
61          Device.findById(deviceData._id).then((device) => {
62            expect(device.toObject().tokens).toMatchObject(
      deviceData.tokens);
63            done();
64          }).catch((e) => done(e));
65        });
66    });
67
68    it('should fail due no device in database', (done) => {
69      request(app)
70        .post('/device/login')
71        .send({
72          name: 'fakedevice',
73          password: 'jf384jgerkv'
74        })
75        .expect(400)
```

```
76        .expect((res) => {
77          expect(res.text).toBe(`Device doesnt exists`);
78          expect(res.headers['x-auth']).toBeFalsy();
79        })
80        .end((err, res) => {
81          if (err) {
82            return done(err);
83          }
84          done();
85        });
86    });
87
88  });
89
90  describe('DELETE /device/logout', () => {
91    it('should logout device and remove the auth token', (done
        ) => {
92      request(app)
93        .delete('/device/logout')
94        .set('x-auth', deviceData.tokens[0].token)
95        .send({})
96        .expect(200)
97        .expect((res) => {
98          expect(res.headers['x-auth']).toBeFalsy();
99        })
100       .end((err, res) => {
101         if (err) {
102           return done(err);
103         }
104
105         Device.findById(deviceData._id).then((user) => {
106           expect(user.toObject().tokens.find((token) =>
        token.access === 'auth')).toBeFalsy();
107           done();
108         }).catch((e) => done(e));
109       });
110   });
111
112 });
113
114 describe('POST /device/send', () => {
115
116   it('should send data to service', (done) => {
117       const _socket = ioClient.connect('http://localhost
        :8080');
118
119       _socket.on('connect', function(data) {
120
121         _socket.emit('authentication', { username:
        serviceData.name,
122                                          password:
        serviceData.password });
```

```
123
124         _socket.on('authenticated', function() {
125             _socket.emit('join', { name: deviceData.name },
    function (e) {
126                 if (e) {
127                   done(e);
128                 } else {
129                   request(app)
130                     .post('/device/send')
131                     .set('x-auth', deviceData.tokens[0].
    token)
132                     .send({ value: '30' })
133                     .expect(200)
134                     .expect((res) => {
135                       expect(res.text).toBe('Success!');
136                     })
137                     .end((err, res) => {
138                       if (err) {
139                         return done(err);
140                       }
141                     });
142                 }
143               });
144
145             _socket.on('newValue', (data) => {
146               if (data.value === '30' && data.device ===
    deviceData.name) {
147                 done();
148               } else {
149                 done('Incorrect data received!');
150               }
151             });
152         });
153       });
154
155   });
156
157 });
158 };
```

Listing A.13 – device.test.js

```
1  <!DOCTYPE html>
2  <html>
3
4    <head>
5      <meta charset="utf-8">
6      <title>Generator</title>
7    </head>
8
9    <body>
10     <div>
```

```
11        <p>LOGIN</p>
12        <input id="username" />
13        <input id="password" />
14        <button id="login">Login</button>
15      </div>
16
17      <div>
18        <p>JOIN</p>
19        <input id="deviceName" />
20        <button id="join">Join</button>
21        <button id="leave">Leave</button>
22      </div>
23
24      <script src="https://cdnjs.cloudflare.com/ajax/libs/
        socket.io/2.0.4/socket.io.js"></script>
25      <script src="/js/jquery-3.3.1.min.js"></script>
26      <script src="/js/service.js"></script>
27    </body>
28
29  </html>
30  };
```

Listing A.14 – service.html

```
1   <!DOCTYPE html>
2   <html>
3
4     <head>
5       <meta charset="utf-8">
6       <title>Generator</title>
7     </head>
8
9     <body>
10      <div>
11        <p>LOGIN</p>
12        <input id="username" />
13        <input id="password" />
14        <button id="login">Login</button>
15      </div>
16
17      <div>
18        <p>SEND</p>
19        <input id="deviceValue" />
20        <button id="send">Send</button>
21      </div>
22
23      <script src="/js/jquery-3.3.1.min.js"></script>
24      <script src="/js/device.js"></script>
25    </body>
26
27  </html>
28  };
```

Listing A.15 – device.html

```
var _socket;

jQuery('#login').on('click', function (e) {
  alert("Connecting to socket");

  _socket = io.connect('localhost:8080');
  _socket.on('connect', function(){

    _socket.emit('authentication', {username: jQuery('#
    username').val(), password: jQuery('#password').val()});

    _socket.on('authenticated', function() {
      alert("Connected!");

      _socket.on('newValue', (data) => {
        alert(`Got ${data} from device!`);
      });

    });

    _socket.on('unauthorized', function(err){
      alert("There was an error with the authentication!");
    });

  });
});


jQuery('#join').on('click', function (e) {
  var device = { name: jQuery('#deviceName').val() }

  _socket.emit('join', device, function (e) {
    if (e) {
      alert(e);
    } else {
      alert("Join successful!")
    }
  });
});

jQuery('#leave').on('click', function (e) {
  var device = { name: jQuery('#deviceName').val() }

  _socket.emit('leave', device, function (e) {
    if (e) {
      alert(e);
    } else {
      alert("Exit successful!");
```

```
48      }
49    });
50  });
51  };
```

Listing A.16 – service.js

```
1   var _token = 'NO_VALUE';
2
3   $('#login').on('click', function (e) {
4     alert("Connecting to server ...");
5
6     var name = $('#username').val();
7     var password = $('#password').val();
8
9     var data = {
10      name: name,
11      password: password
12    };
13
14    $.ajax({
15      url: '/device/login',
16      type: 'post',
17      data: JSON.stringify(data),
18      headers: {
19        'Content-Type': 'application/json'
20      },
21      success: function (data, status, xhr) {
22        _token = xhr.getResponseHeader('x-auth');
23        alert("Success in login!");
24      },
25      error: function (data){
26        alert("Error trying to login!");
27      }
28    });
29
30  });
31
32  $('#send').on('click', function (e) {
33    alert("Sending data ...");
34
35    var data = {
36      value: $('#deviceValue').val()
37    };
38
39    $.ajax({
40      url: '/device/send',
41      type: 'post',
42      data: JSON.stringify(data),
43      headers: {
44        'Content-Type': 'application/json',
45        'x-auth': _token
```

```
46      },
47      success: function (data) {
48        alert("Data has been sent!");
49      },
50      error: function (data) {
51        alert("Error sending data: " + data);
52      }
53
54    });
55  });
56  };
```

Listing A.17 – device.js

APÊNDICE B – Artigo

Proposta de Middleware Web para Autenticação Contínua Multi-fator

Marcos Schead dos Santos[1]

[1]Departamento de Informática e Estatística
Universidade Federal de Santa Catarina (UFSC) – Florianópolis, SC – Brasil

m.santos@ufsc.br

Abstract. *Authentication is the process of verifying a person's identity, allowing or not its access. When using an application, an user wants to produce value for his business, not authenticate itself. Although necessary, it becomes uncomfortable because it breaks the main flow usage. There are mechanisms to authenticate the user continuously, generating reliability identity values. These values can be obtained by face recognition or keystroke dynamics for instance. However, an open issue is how to use more than one value at the same time to improve the authentication. The main purpose of this work is present and implement an architecture, that one or more devices are able to generate authentication values time by time and associate it to an user. The services that want to get these authentication values will be notified when something changes. Then all services could make their domain decisions upon these new values. Through a smart phone app acting as a service all the artifacts validation were done.*

Resumo. *A autenticação é o processo de verificar a identidade de uma pessoa, de modo que seu acesso ao sistema seja permitido ou negado. Um usuário nunca usa um sistema com a finalidade de autenticar-se, mas sim para exercer as funcionalidades inerentes à aplicação. Embora necessária, torna-se um incômodo comum, pois quebra o fluxo principal das aplicações. Existem mecanismos que autenticam o usuário continuamente gerando um valor que quantifica a certeza da identidade do usuário. Eles podem ser obtidos, por exemplo, usando reconhecimento facial ou padrões comportamentais de digitação. No entanto, uma questão em aberto é como utilizar mais de um valor contínuo simultaneamente para melhorar a autenticação. A proposta desse trabalho é apresentar e implementar uma arquitetura, onde um ou mais dispositivos possam gerar distintos valores periodicamente, e associá-los a um usuário. Os serviços que estiverem interessados nesses valores de autenticação contínuos daquele indivíduo, serão notificados sempre que ocorrer uma mudança neles. Assim, os serviços podem tomar suas próprias decisões de negócio, com base nos novos valores obtidos. Por meio de um aplicativo de celular que representa um serviço interessado foi feita a validação dos artefatos criados.*

1. Introdução

A autenticação não é uma funcionalidade com o intuito de produzir algo. O uso do sistema tem valor para o usuário na criação do seu domínio de trabalho, independente do tipo de software que se está operando. É algo que precisa ser feito, pois existem pessoas com más intenções que podem beneficar-se de forma indevida. Assim, esse processo existe para garantir que o sistema está nas mãos de alguém confiável.

Entretanto, ninguém gosta de ser incomodado com uma pergunta recorrente confirmando as credenciais de acesso. Se existisse apenas um sistema, isso não seria problema. Porém, no contexto atual em que diversos serviços aparecem para o dia, é inviável gerenciar muitas senhas diferentes e difíceis de serem quebradas. Principalmente com a expansão dos smartphones e tablets, que criaram um vasto mercado de aplicativos, armazenando informações cada vez mais específicas. Na prática, duas ou três credenciais novas são criadas e seu uso é distribuído entre os vários aplicativos, sites e entre outros. Isso abre uma vulnerabilidade grande em suas informações, caso qualquer uma das senhas sejam descobertas.

Há casos em que a autenticação é extremamente necessária e inclusive, pode necessitar de diversos níveis para garantir a segurança da informação. Em situações que existem documentos sigilosos e de cunho militar, por exemplo, deve-se ter senhas fortes para evitar o acesso não autorizado. No cotidiano, o que os usuários desejam é usufruir os serviços e serem importunados o mínimo possível. Em poucos momentos, de preferência com um intervalo grande, uma senha pode ser exigida, para garantir a identidade ou para evitar o acesso não-autorizado à certos recursos.

Existem formas de validar a identidade do usuário. A seguir é apresentado cinco categorias de autenticação. O uso de login e senha é apenas uma delas, que está relacionado ao que o usuário sabe. Pode ser uma palavra chave ou uma informação, essa é uma das categorias de autenticação. A segunda forma está vinculada ao que ele tem: um cartão, um pendrive, algo físico necessário para o reconhecimento da identidade. A terceira é por algo que ele é: uma digital, reconhecimento de íris, algum padrão de comportamento, ou seja, algo que possa identificar o indivíduo. A quarta categoria envolve o local do indivíduo. Por exemplo, um serviço oferecido pela universidade, só pode ser acessível dentro dela, qualquer acesso externo não é permitido. Isso pode ser estendido para o tempo, que é a quinta forma. Um serviço só pode ser acessado em um certo horário pré-estabelecido. A última forma seria por testemunha, em que um terceiro garante a identidade de uma pessoa para o sistema, permitindo então o acesso dela.

Algumas dessas categorias podem conter serviços de autenticação contínua. Um autenticador contínuo verifica num intervalo de tempo, a identidade do usuário. Em alguns casos, pode ser retornado um valor, representando uma porcentagem de certeza, ou um simplesmente um valor de sim e não. Um dispositivo verificar a cada 1 minuto a íris do usuário, pode gerar um valor desse tipo, enquanto que uma autenticação temporal, apenas informa sim ou não.

Conhecendo as categorias citadas acima, cada uma com suas peculiaridades, pode não ser interessante utilizar apenas e somente uma. Em alguns casos, pode ser interessante avaliar mais de uma e tomar decisões com base num conjunto delas. Isso garante uma confiabilidade maior, visto que fontes diferentes são consultadas.

Com isso, vem as seguintes perguntas: é possível agregar diferentes formas de autenticação contínua que perturbe o mínimo possível o usuário e melhore a confiança no sistema? É possível ter uma arquitetura que forneça isso à diversos sistemas diferentes?

2. Objetivos

Este trabalho tem por objetivo propor uma arquitetura para o uso de uma ou mais variáveis de autenticação contínua. Os serviços que tiverem interesse, receberão essas variáveis em tempo real, assim que geradas. Junto disso, será feito sua implementação em uma linguagem de propósito geral, com verificação através de testes unitários.

Com isso pronto, será feito uma validação simples, por meio de uma aplicação. Esta representará um serviço interessado nos NDCs do usuário e fará uso deles para mantê-lo autenticado ou não.

O escopo do trabalho não abrange como serão gerados os níveis de confiabilidade. A geração deles, utilizada nos experimentos, tem apenas o propósito de auxiliar na validação do modelo proposto.

Não será realizado um experimento elaborado com usuários. Além da limitação de tempo, existem muitas variações possíveis, que por si só, podem elaborar um estudo mais aprofundado.

3. Projeto

Em poucas palavras, este trabalho quer associar diferentes dispositivos de autenticação contínua a um usuário. Os serviços de software em geral, seja uma aplicação *Web* ou *Mobile*, podem então consumir os valores gerados por esses dispositivos. Essa informação pode ser vista como um terceiro fator de autenticação.

Dentro dessa ideia, existem diferentes fluxos possíveis de comunicação, e cada um deles pode ser usado, dependendo dos requisitos necessários. A Figura 1 descreve esta forma. Em um certo intervalo de tempo, independente para cada um, o dispositivo gera um novo NDC associado ao usuário e envia ao servidor essa nova informação.

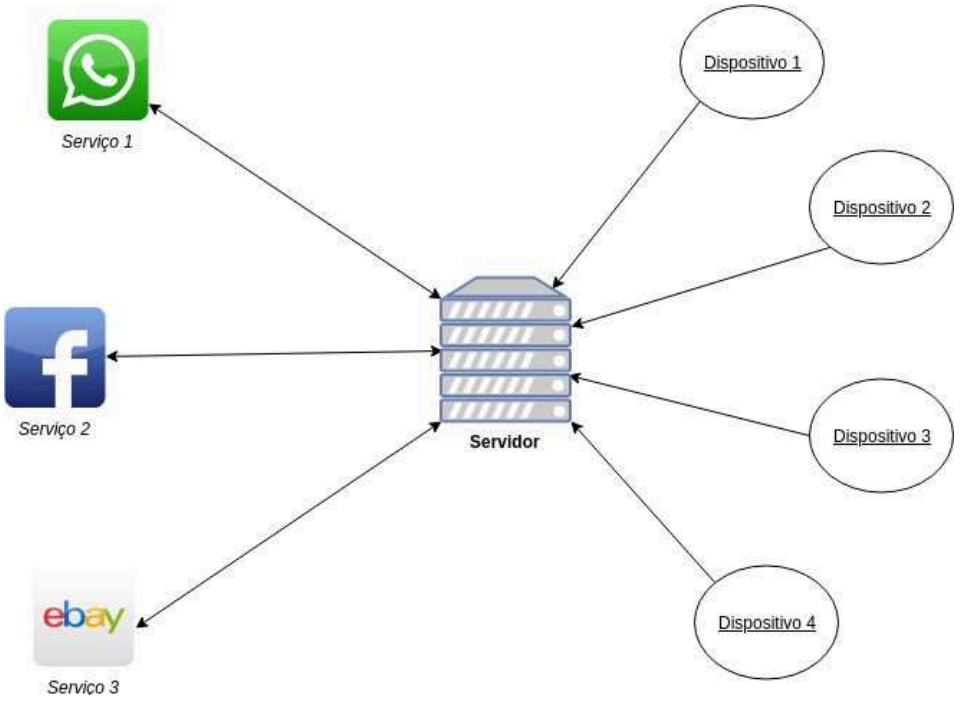

Figura 1. Arquitetura proposta

Após o tratamento devido, o servidor persiste o dado. Os serviços que estão interessados, são notificados que um novo valor foi gerado, e recebem-no para utilizá-los dentro da sua aplicação. Dentro desse fluxo, os serviços seguem o padrão de projeto *Observer*. Através dele, é possível manter os serviços bem desacoplados do servidor, de modo que eles só recebem a informação. No caso, para se obter sempre o nível atualizado, os serviços inscrevem-se num canal onde chegam os novos valores.

O primeiro componente é o servidor. Ele é responsável por manter todas as atualizações feitas do NDC com o passar do tempo. Os dispositivos comunicam-se via requisições do protocolo HTTP (Hyper Text Transfer Protocol), enviando os dados no formato JSON (JavaScript Object Notation). E além disso, deve suportar o protocolo de *WebSocket* o qual realiza comunicação *Full-duplex* entre o servidor e um cliente.

A segunda entidade fundamental é o usuário. Somente ele pode adicionar os dispositivos e serviços autorizados a estabelecer interações no servidor. Não existem restrições de quais dispositivos um serviço pode escutar atualizações. A partir do momento que o usuário cadastra um serviço, este pode escolher dentre os dispositivos específicos que deseja receber suas atualizações.

O terceiro componente é o dispositivo, capaz de gerar um NDC referente ao usuário. O seu objetivo principal é gerar de tempos em tempos um valor atualizado do NDC. Pode ser qualquer máquina, independente do porte, que consiga gerar um valor contínuo, numa escala de 0% a 100%. Este é o grau de certeza sobre o usuário ser quem ele afirma que é. Para os fins deste trabalho, foram usados dispositivos que geram valores quaisquer nos testes. No caso, os dispositivos que podem atualizar o NDC devem ter as credenciais necessárias.

Depois de gerado esse valor, ele é enviado ao servidor. Este valor enviado pelo dispositivo não é tratado em momento algum. Ele é salvo e retransmitido no formato em que veio. Nada impede que seja usado algum modelo matemático, para melhorar a sua semântica. Seu *timestamp* também é gerado, para fins estatísticos futuros.

Em seguida é disparada uma notificação para todos os serviços interessados. Um novo nível de autenticação é gerado, de modo que eles possam manter o usuário logado ou não em suas aplicações, de acordo com suas regras.

Por fim, o quarto componente é um serviço, seja ele qual for, que esteja interessado em receber o NDC de forma periódica. Por serviço, entende-se qualquer aplicação *Web* ou *Mobile*. Ela usará esse valor para a tomada de decisão com respeito a autenticação do usuário.

Para que o serviço possa consumir esses dados, ele deve estar em conformidade com o protocolo *WebSocket* usado. No caso, seria escutar um evento *newValue*. Todo NDC criado, será transmitido por esse meio.

Existe a emissão dos eventos adequados, para a escolha de quais dispositivos deseja-se escutar. Para receber atualizações, basta emitir um evento *join* com o identificador daquele dispositivo. E para não obter mais novas atualizações, deve-se emitir um evento *leave*, também com seu identificador.

4. Implementação

Nesta parte do trabalho será descrita detalhes de como a arquitetura proposta foi implementada.

Tanto no servidor central, como nos serviços que consomem esses valores, foi utilizada a linguagem *JavaScript*. Ela possui várias bibliotecas que facilitam o desenvolvimento, e que recebem *feedbacks* constantes pela comunidade. Além disso, é pouco verbosa, gerando menos código e facilitando a manutenibilidade do mesmo para trabalhos futuros.

Já nos dispositivos foi escolhida a plataforma Android. Sua escolha permitiu mostrar de forma clara, uma aplicação simples do middleware. Ela também possui uma comunidade grande de desenvolvedores, que ajudam a mantê-lo com qualidade.

4.1. Dispositivos

O principal, se não o único, objetivo do dispositivo, é gerar o NDC, de tempos em tempos, e enviá-lo ao servidor. Este fará o tratamento adequado, independente do tipo de dispositivo que for.

No entanto, para que isso seja possível, o dispositivo precisa estar em conformidade com estes três pontos:

- O usuário já o cadastrou no sistema
- Deve estar autenticado via JWT
- Para enviar os dados, deve chamar a requisição HTTP adequada

Assumindo que o usuário esteja autenticado, ele pode cadastrar um novo dispositivo, através de uma requisição *POST* na *url* /device. O corpo da requisição deve

conter os valores *name* e *password*. O valor *name* referencia unicamente o dispositivo no servidor para o usuário em questão. Com o dispositivo cadastrado, basta ele se conectar fazendo uma requisição.

Essa requisição, seguindo o padrão JWT, irá retornar um *token*, que deve ser usado nas requisições futuras.

Note que o *token* aparece ali no cabeçalho da requisição. E para enviar o valor apropriado, basta o dispositivo gerá-lo, de acordo com o método de autenticação.

Como forma de teste manual, foi criado uma página html estática, que pode ser acessada em /device.html. Ela contém dois campos para realizar o login e um outro para enviar um novo NDC referente ao dispositivo.

4.2. Serviços

O serviço funciona de forma semelhante ao dispositivo. Para que ele funcione, deve estar em conformidade novamente com alguns pontos:

- O usuário já cadastrou no sistema
- Deve estar autenticado via WebSocket
- Para enviar os dados deve emitir um evento adequado

Diferente do dispositivo, toda comunicação ocorre apenas via *WebSocket*. Assim é garantido que os serviços interessados em um dispositivo qualquer, obtenham o novo valor gerado em tempo real.

Assume-se que o usuário tenha já cadastrado o serviço, através de uma requisição POST na *URL* /service, com o *body* possuindo os valores *name* e *password*. Assim como o dispositivo, *name* referencia o service unicamente para aquele usuário.

Nesse ponto, tem uma diferença importante entre os dois. Como no caso de *WebSocket* o canal de comunicação fica aberto, a autenticação só ocorre uma vez.

Para estabelecer um *WebSocket* entre duas partes, é necessário realizar uma requisição HTTP, com as credenciais adequadas. O RFC não define nenhum tipo de *header*, em que as credenciais possam ser armazenadas.

Além disso, o uso de dados sensíveis em URLs deve ser evitado. Muitos endereços podem ser armazenados em logs do servidor com a senha em *plain text*. Dessa forma, uma solução encontrada foi a seguinte:

- Assim que o serviço conectar-se ao socket, deve emitir um evento 'authentication' contendo as credenciais. Se não o fizer, ele será desconectado por timeout, cujo padrão definido é 1 segundo.
- Se as credenciais estiverem corretas, um evento 'authenticated' será emitido pelo servidor.
- Caso contrário, o evento *unauthorized* será emitido, contendo o erro em questão.

Todos os eventos que forem de interesse, devem ser escutados dentro do 'authenticated', que é o caso do evento 'newValue', onde o serviço receberá um dado contendo o último valor gerado de um dispositivo específico.

Com essa implementação, as credenciais ficam no corpo da requisição, todavia, o servidor usa TLS/HTTPS para criptografar os dados durante a transmissão, garantindo o sigilo deles.

4.2. Servidor Central

A partir desse ponto, entra a explicação mais detalhada da aplicação, visto que o servidor é quem controla boa parte do tráfego de informações.

O primeiro ponto a ser descrito é sobre a transmissão dos dados. Dentro de um arquivo de configuração, além de questões básicas do servidor, foi inserido o protocolo HTTPS. Nele encontra-se o carregamento da chave privada e do certificado do LABSEC, junto com a cadeia de certificados intermediários.

Por questões práticas, o protocolo só é ativado, quando o servidor encontra-se em execução na máquina do LABSEC.

Outra ponto a considerar: só existe apenas um usuário. A senha foi salva no banco, através do REPL (Read–Eval–Print Loop) do Node, usando a API fornecida pelo *Mongoose*.

Uma senha padrão foi persistida, depois de aplicado um algoritmo de *hash*, com um *salt* de 10 caracteres, para prover uma segurança maior. Quando o usuário realiza login no servidor, a biblioteca bcrypt em *Javascript* verifica a senha, não esquecendo de utilizar o salt gerado.

Para isso, uma requisição HTTP deve ser feita no *endpoint* /user/login, com o corpo da requisição contendo as credenciais.

Foi feita uma função que retorna uma *promise* contendo o JWT gerado, caso as credenciais estejam corretas. Ele então é retornado no cabeçalho da requisição de resposta. Se as credenciais estiverem incorretas, a requisição de resposta retorna um erro de *bad request*.

Para gerar o JWT, o *payload* informado é um objeto contendo o id do usuário e uma string contendo o tipo de *token*. O segredo é gerado no servidor a partir do momento em que é iniciado. No caso do *logout*, o objeto *token auth* do usuário é apagado no banco de dados, usando uma requisição *post* no *endpoint* /user/logout.

É necessário para esta e outras requisições, que o usuário esteja autenticado, ou seja, com o *token* gerado em seu poder. O dispositivo utiliza a mesma ideia que o usuário. No express, os métodos são *middlewares*, associados ao *endpoint*, de forma que sua execução sempre ocorre antes do código da requisição.

Embora não fosse necessário um método para encontrar o *token*, sabendo que existe apenas um usuário, esta camada a mais permite que essa parte do código cresça. Não há necessidade em saber como é feita a implementação de busca do *token*, ficando mais fácil codificar para que novos usuários sejam inseridos ao servidor.

Dentro da *promise* de retorno, é feita uma pesquisa ao banco, com o id obtido ao decodificar o JWT, caso o segredo esteja correto.

Assim que estiver autenticado, o usuário pode adicionar um dispositivo ou um serviço. Para isso, deve fornecer suas credenciais, enviadas via POST em /device ou /service. O princípio de funcionamento do código é semelhante em ambos.

Depois de autenticar o *token*, as credenciais são obtidas, e usadas para construir um objeto *device*, que é então persistido.

O service possui um mecanismo um pouco diferente: como sua comunicação com o servidor é via *WebSocket*, a autenticação ocorre apenas uma vez, no momento em que é estabelecida.

O código acima é executado, assim que o cliente emite o evento '*authenticated*' conforme já fora descrito. Após a validação, caso ela esteja correta, o socket associa um atributo seu ao nome do serviço, e manda escutar todos os dispositivos que ele estiver interessado. Os dispositivos de interesse do serviço sempre são armazenados no banco de dados. Dessa forma, toda vez que o serviço reconectar, suas preferências já estarão estabelecidas.

Assumindo que o serviço está conectado, ele tem duas escolhas: pedir para receber atualizações do dispositivo ou parar de recebê-las.

Um dos endpoints chaves da aplicação é o envio de um NDC por um dispositivo. Assim que for verificada sua identidade, é feito o envio de um valor atualizado para todos os serviços interessados.

No corpo da requisição, o dispositivo envia seu identificador e o NDC atualizado. Em futuras atualizações, é possível adicionar novas variáveis, para ajudar o servidor no refinamento do processo. Assim que obtido o valor, um registro contendo seu timestamp é salvo. Quando persistido, todos os serviços que estiverem escutando pelo dispositivo, receberão o valor novo para usarem em suas aplicações.

5. Validação

Por fim, esse capítulo mostra um exemplo simples, de como um serviço pode usar o middleware implementado.

O aplicativo de celular criado consome níveis de confiabilidade, gerados por três dispositivos. Assim que é aberto, ele executa em *background* e deriva um novo valor, à medida que os dispositivos enviam novos valores.

O valor usado pela aplicação é derivado pela seguinte fórmula, uma média ponderada:

$$NDC_{atual} = NDC_1 * 5 + NDC_2 * 3 + NDC_3 * 2 / 10$$

Cada NDC é obtido em um tempo diferente. Dessa forma, o valor adapta-se de acordo com o peso daquele dispositivo. Os NDCs são aplicações *Web* com uma simples página HTML. Elas possuem apenas a função de login, e envio de um NDC novo.

Assim que o valor for calculado, ele é comparado a um *threshold* empírico, usado apenas para validação. Se estiver abaixo, a tela é bloqueada e o usuário deve desbloqueá-lo, usando o próprio sistema do celular. O valor 60% foi definido como padrão.

No momento que a aplicação abriu, o *WebSocket* é estabelecido com o servidor do LABSEC. A Figura 2 mostra a barra de notificação quando está autenticado (A) e quando a autenticação foi negada (B).

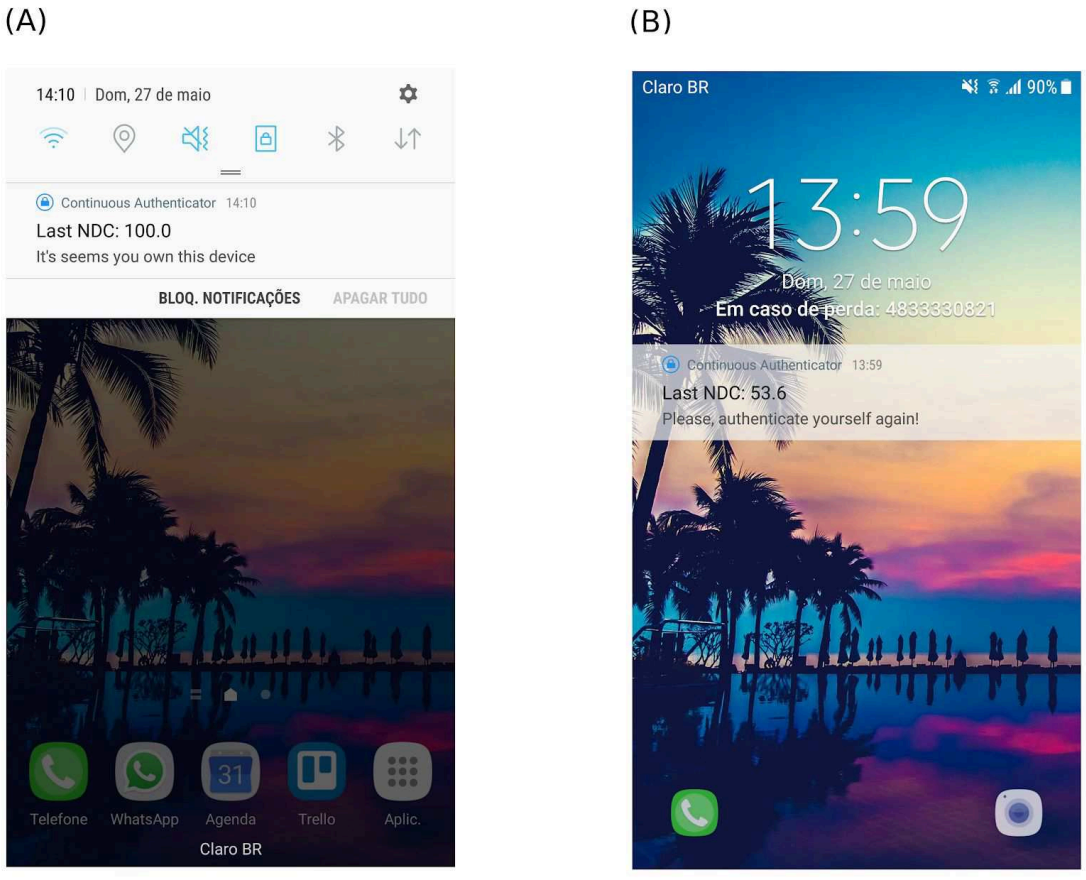

Figura 2. Telas do serviço

6. Conclusão

A autenticação é algo cuja unica constante em seu estudo é a mudança. Novos trabalhos são propostos a todo momento, com o intuito de aumentar a segurança e a usabilidade do usuário. Nos trabalhos pesquisados, pode-se perceber os seus diversos tipos, e como sua combinação pode melhorar esse processo.

A elaboração da arquitetura e sua implementação foram feitas com sucesso. Com a verificação através de testes unitários é garantido uma manutenibilidade segura do código. Junto disso, fazendo uma validação com um simples aplicativo, pode-se observar como seria o uso do resultado final.

Dessa forma, os objetivos finais foram alcançados e assim um novo modelo multi-fator proposto. Diferente dos outros, ele permite que qualquer autenticador contínuo seja usado. No entanto, ele deve estar de acordo com a especificação da API descrita no texto.

Ainda assim, há diversos pontos que podem ser melhorados. Em relação aos trabalhos futuros, existem algumas alternativas a serem exploradas:

- Fazer um experimento com uma amostra de usuários adequada, usando dispositivos de autenticação contínua reais, para obter um feedback mais aprimorado da proposta.
- Melhorar questões de implementação do middleware: permitir mais de um usuário, deixar que o usuário gerencie quais dispositivos podem ser usados pelos serviços, desenvolver uma interface de controle para o usuário, etc.
- Procurar possíveis brechas na arquitetura proposta, assim como na implementação feita, e propor soluções para as mesmas.
- Propor uma análise matemática em cima dos valores gerados: será que a aritmética ponderada é a melhor forma? Verificar diferentes formas de criar o NDC mais atual. Feito isso, discutir qual delas é mais apropriada sob o ponto de visto de usabilidade e segurança.

7. References

AZZINI, A. et al. A fuzzy approach to multimodal biometriccontinuous authentication.Fuzzy Optimization and Decision Making,Springer, v. 7, n. 3, p. 243, 2008

BRADLEY, J.; SAKIMURA, N.; JONES, M. Json web token (jwt).2015.

BRÖMME, A.; AL-ZUBI, S. Multifactor biometric sketchauthentication. In:BIOSIG. [S.l.: s.n.], 2003. p. 81–90.

Knuth, CARRIGAN, J.; MARTIN, P.; RUSHANAN, M. Kbid: Kerberosbracelet identification (short paper). In: SPRINGER.InternationalConference on Financial Cryptography and Data Security. [S.l.], 2016.p. 544–551.

DENNING, D. E.; MACDORAN, P. F. Location-based authentication:Grounding cyberspace for better security.Computer Fraud & Security,Elsevier, v. 1996, n. 2, p. 12–16, 1996.

FRANK, M. et al. Touchalytics: On the applicability of touchscreeninput as a behavioral biometric for continuous authentication.IEEEtransactions on information forensics and security, IEEE, v. 8, n. 1,p. 136–148, 2013.

KARAPANOS, N. et al. Sound-proof: Usable two-factor authenticationbased on ambient sound. In:USENIX Security Symposium. [S.l.: s.n.],2015. p. 483–498

NIINUMA, K.; JAIN, A. K. Continuous user authentication usingtemporal information. In:Proc. SPIE. [S.l.: s.n.], 2010. v. 7667, n. 1,p. 76670L

SABZEVAR, A. P.; STAVROU, A. Universal multi-factorauthentication using graphical passwords. In: IEEE.Signal ImageTechnology and Internet Based Systems, 2008. SITIS'08. IEEEInternational Conference on. [S.l.], 2008. p. 625–632.

STAJANO, F. Pico: No more passwords! In: SPRINGER.SecurityProtocols Workshop. [S.l.], 2011. v. 7114, p. 49–81.

TIWARI, A. et al. A multi-factor security protocol for wirelesspayment-secure web authentication using mobile devices.arXivpreprint arXiv:1111.3010, 2011.

YAP, R. H. et al. Physical access protection using continuousauthentication. In: IEEE.Technologies for Homeland Security, 2008IEEE Conference on. [S.l.], 2008. p. 510–512

ZHANG, F.; KONDORO, A.; MUFTIC, S. Location-basedauthentication and authorization using smart phones. In: IEEE.Trust, Security and Privacy in Computing and Communications(TrustCom), 2012 IEEE 11th International Conference on. [S.l.],2012. p. 1285–1292.