Controle de Acesso a Dados com Propagação de Identidade em Aplicações Web baseadas em Serviços

Felipe Leão[1,2], Leonardo Guerreiro Azevedo[1,2,3], Talles Santana[2], Fernanda Baião[1,2], Claudia Cappelli[1,2]

[1]Programa de Pós-Graduação em Informática (PPGI)
Universidade Federal do Estado do Rio de Janeiro (UNIRIO)
Av. Pasteur, 458 - Urca - Rio de Janeiro – Brasil - 22290-240

[2]Universidade Federal do Estado do Rio de Janeiro (UNIRIO)
Av. Pasteur, 458 - Urca - Rio de Janeiro – Brasil - 22290-240

[3]IBM Research - Brasil
Av. Pasteur, 138 - Botafogo - Rio de Janeiro – Brasil - 22290-240

{felipe.leao, azevedo, talles.santana, fernanda.baiao, claudia.cappelli}@uniriotec.br, LGA@br.ibm.com

Abstract. *Information Security is one of the main issues in current organizations. In this context, role-based data access control and user identity propagation in n-tier architectures are foremost concerns. This work proposes a pragmatic architecture to handle identity propagation and authorization rules application on web services oriented systems. The implementation is based on the SOAP protocol and interceptor handlers. Experimental tests results demonstrated that the proposal is low intrusive related to changes on existing applications required to start its use. It provides isolation to concurrent data access. Besides, performance tests demonstrated that there is a small overhead on its use, which is 14% in average.*

Resumo. *Segurança de informação é uma das principais questões em organizações atuais. Neste contexto, destacam-se especialmente a necessidade de controle de acesso aos dados por perfil de usuário e a propagação da identidade do usuário em arquiteturas de múltiplas camadas. Este artigo propõe uma arquitetura para sistemas desenvolvidos empregando serviços web para propagação de identidade e aplicação de regras de autorização. A implementação desta arquitetura está baseada no uso do protocolo SOAP e de handlers. Resultados de testes experimentais demonstraram que a solução é pouco intrusiva em relação a mudanças necessárias nas aplicações existentes. A solução permite isolamento durante acesso concorrente aos dados. Além disso, o teste de desempenho demonstrou que o overhead no seu uso é muito pequeno (média de 14%).*

1 Introdução

Segurança de informação – mais especificamente, o controle de acesso a dados – é uma das principais preocupações nas organizações atuais. Neste universo, as maiores questões se encontram nos mecanismos de controle (ou autorização) de acesso para garantia da integridade dos dados [Sandhu *et al*. 1996; Yang 2009; Calì e Martinenghi

LEÃO, F.; AZEVEDO, L. G.; SANTANA, T.; BAIÃO, F.; CAPPELLI, C.
Controle de Acesso a Dados com Propagação de Identidade em Aplicações Web baseadas em Serviços
iSys - Revista Brasileira de Sistemas de Informação, Rio de Janeiro, vol. 7, No. 1, p. 48-65, 2014.

2008; Murthy e Sedlar 2007]. Estes mecanismos fazem com que regras de autorização de acesso a dados sejam garantidas. Uma **regra de autorização** (ou "assertiva de ação de autorização") restringe a **quem** é permitido realizar uma **ação** na organização sobre quais **informações** [BRG 2009].

A propagação de identidade em sistemas de múltiplas camadas tem o objetivo de transmitir a identidade do usuário desde a camada cliente até o Sistema de Gerenciamento de Banco de Dados (SGBD). Este, por sua vez, utilizará a identidade do usuário recebida para aplicar regras de autorização no acesso aos dados a fim de retornar para o usuário apenas os dados que ele tem direito.

Arquiteturas de aplicação para acesso a bancos de dados podem ser divididas em quatro tipos [Leão *et al.*, 2011a]:

(i) Aplicação cliente se conecta diretamente ao SGBD utilizando o usuário do Sistema Operacional onde a aplicação cliente está sendo executada. Neste caso, em geral, o SGBD consegue identificar o usuário a partir de variáveis de ambiente.

(ii) Aplicação cliente se conecta diretamente ao SGBD, mas utilizando usuário diferente do usuário do Sistema Operacional, ou seja, o usuário do sistema operacional é *x* enquanto que o usuário da aplicação é *y*. Neste caso, o SGBD consegue empregar suas variáveis de ambiente para identificar *x.* No entanto, ele não consegue reconhecer o usuário *y* da aplicação cliente.

(iii) Aplicação cliente se conecta a um servidor de aplicação que acessa o banco de dados. Neste caso, existem 3 camadas de acesso e o SGBD não consegue identificar o usuário da aplicação cliente.

(iv) Aplicação cliente se conecta a um servidor de aplicação que invoca serviços web para acessar o banco de dados. Neste caso, existem várias camadas de indireção de modo que o SGBD também não é capaz de identificar o usuário da aplicação cliente.

Para que o SGBD possa aplicar as devidas regras de autorização, ele deve reconhecer o real usuário que pretende acessar os dados. No cenário (i), o SGBD consegue identificar o usuário diretamente a partir de variáveis de ambiente que armazenam informações do Sistema Operacional onde a aplicação cliente está sendo executa. Uma destas informações é o usuário do sistema operacional que, no cenário (i), é o próprio usuário da aplicação cliente. Logo, para desenvolver mecanismos de autorização de acesso empregando a identidade do usuário nenhuma alteração precisa ser realizada na aplicação cliente para se obter o usuário da mesma.

Por outro lado, nos cenários (ii), (iii) e (iv), o banco de dados não consegue identificar o usuário da aplicação cliente através de suas variáveis de ambiente. Logo, é necessário tratar a recuperação e propagação do usuário a cada camada sendo acessada durante o tratamento de sua solicitação. Para isto, é necessário alterar as implementações das diferentes camadas para tratar a propagação. Isto é muito intrusivo uma vez que requer grandes mudanças nas aplicações.

Este artigo propõe uma arquitetura para aplicações em múltiplas camadas que fazem uso de serviços web que leva a intrusão mínima na propagação de dados e

execução de regras de autorização. Esta arquitetura contribui como uma solução para a propagação de identidade nos cenários (ii), (iii) e (iv) podendo ser também aplicada ao cenário (i).

A implementação da arquitetura está baseada no uso do protocolo SOAP (*Simple Object Access Protocol*) [Gudgin *et al.* 2007] e de *handlers* [Oracle 2008] para propagar a identidade do usuário. Já para a execução de regras de autorização, a proposta baseia-se no *framework* FARBAC proposto por [Azevedo *et al.* 2010] cujos testes de implementação apresentados por [Puntar *et al.* 2011] demonstraram seu desempenho e flexibilidade em relação à solução tradicional.

Testes experimentais foram realizados para avaliar a arquitetura proposta em ambientes de alta concorrência de acesso a dados através do *framework* para testes TestNG [Beus e Suleiman 2007] acessando dados gerados de acordo com o TPC-H *Benchmark* [TPC Council, 2008] de apoio a decisão. Os resultados demonstraram eficácia e eficiência na propagação de identidade e execução de regras de autorização, além de isolamento em cenários com execuções concorrentes.

Este trabalho está organizado em 6 seções. A Seção 2 conceitua os mecanismos de controle de acesso, serviços web, o protocolo SOAP e SOAP Handlers e propagação de identidade. A Seção 3 apresenta o padrão WS-Security e contextualiza o seu uso como alternativa para armazenamento dos dados a serem propagados na mensagem. A Seção 4 detalha a arquitetura proposta, avaliada em testes experimentais relatados na Seção 5. A Seção 6 conclui o trabalho e aponta para trabalhos futuros.

2 Conceitos Relacionados

A solução proposta neste trabalho realiza o controle de acesso a dados empregando o *framework* FARBAC. A proposta contempla o uso de serviços web e a propagação de identidade é realizada através do protocolo SOAP.

2.1 Mecanismos para Controle de Acesso e o FARBAC

Mecanismos para controle de acesso têm por objetivo restringir os dados que podem ser visualizados por cada usuário ou por cada grupo de usuários. Os SGBDs disponíveis no mercado possuem mecanismos para tratar autorização de acesso [Jeloka *et al.*, 2008][Elmasri e Navathe, 2010] que podem ser classificados em DAC (*Discretionary Access Control*), MAC (*Mandatory Access Control*), ambos propostos por DoD [1983], ou RBAC (*Role-Based Access Control*) [Ferraiolo e Khun, 1992].

Enquanto DAC restringe o acesso e as operações realizáveis por cada usuário ou grupos em determinados objetos inteiros do banco de dados (tais como tabelas, procedimentos e visões), políticas MAC (*label-security*) se baseiam na definição de regras sobre cada registro [Yang 2009]. Já para o RBAC, o controle de acesso se dá pela união de funções e informações. Neste caso, o interesse maior está em proteger a integridade da informação: "quem é que pode realizar quais ações sobre quais informações" em um nível de granularidade mais fino, tal como valores de células de tabelas [Ferraiolo e Khun 1992]. RBAC é por natureza mais flexível que DAC e MAC. Entretanto, definir e aplicar regras RBAC em cenários reais é uma tarefa complexa e utilizar as abordagens padrão de SGBDs atuais requer grande esforço e conhecimento dos responsáveis por criar e dar manutenção às regras.

Para facilitar a definição e aplicação de políticas RBAC, foi proposto o *framework* FARBAC (*Flexible Approach for Role-Based Access Control*) [Azevedo *et al.,* 2010] - mecanismo flexível e de fácil utilização para auxiliar administradores de bancos de dados a definir e administrar políticas RBAC a fim de implementar controle de acesso aos dados. O *framework* foi implementado com base no mecanismo VPD (*Virtual Private Database*) disponibilizado pelo SGBD Oracle [Jeloka *et al.,* 2008]. As regras de autorização são armazenadas segundo um modelo de dados que permite a geração de cláusulas WHERE (predicados) de forma dinâmica em tempo de execução. Estas cláusulas são utilizadas em comandos enviados por aplicações ao SGBD: durante a execução de uma consulta as regras de autorização agem como filtros, removendo tuplas às quais o usuário não tem acesso. É importante observar que para o FARBAC aplicar regras de autorização é necessário conhecer o usuário que está utilizando a aplicação cliente, o que depende de mecanismos de propagação de identidade.

2.2 Serviços web, SOAP e SOAP Handlers

Um serviço web é um sistema de software projetado para suportar troca de mensagens de maneira interoperável entre computadores na rede. Ele corresponde a uma interface descrita em um formato capaz de ser processado por máquina, como, por exemplo, empregando o padrão WSDL (*Web Service Description Language*[1]). Outros sistemas interagem com o serviço web de acordo com sua especificação usando mensagens SOAP, tipicamente transmitida usando HTTP com uma serialização XML em conjunto com outros padrões web [W3C, 2004]. SOAP é um protocolo cujo objetivo é viabilizar a troca de informação estruturada em um ambiente distribuído e descentralizado [Gudgin *et al.* 2007].

JAX-WS [Kotamraju 2009] é a especificação do Java EE 5 que define um modelo de programação para construção de serviços web e seus clientes, provendo facilidades para utilização de serviços web fracamente acoplados e SOAP *handlers* [Oracle, 2008]. *Handlers* (ou manipuladores) permitem a interceptação de mensagens trafegadas entre consumidores de serviços e provedores de serviços para sua manipulação. A solução proposta neste trabalho foi implementada utilizando *handlers* da especificação JAX-WS 2.0. No entanto, a solução arquitetural pode ser adaptada para quaisquer linguagens e tecnologias que fornecerem funcionalidades similares.

2.3 Protocolo SOAP e a Propagação de Identidade

O protocolo SOAP provê um modelo de processamento distribuído que assume que uma mensagem SOAP é originada em um remetente e tem como objetivo alcançar um destinatário final, passando por nenhum ou vários intermediários [Gudgin *et al.*, 2007], que são os destinatários não terminais ou nós ao longo da rota do remetente para o destinatário final. Um intermediário pode inspecionar e até mesmo manipular uma mensagem SOAP de entrada antes de enviar a mensagem em direção ao destinatário final.

Ainda segundo Gudgin *et al.* [2007], uma mensagem SOAP tem um corpo obrigatório, que pode ser vazio, e um cabeçalho opcional. O cabeçalho destina-se a

[1] www.w3.org/TR/wsdl

carregar meta-informações, sendo adequado à inserção de informações extras que devem ser vistas tanto pelo destinatário final como por intermediários. É possível incluir a assinatura digital do remetente, um voucher ou um *timestamp* que indica quando a informação no corpo da mensagem se torna obsoleta, por exemplo.

Propagação de identidade permite que uma identidade distribuída seja preservada, independente de onde a identidade foi criada, para uso durante o processo de autorização e para propósitos de auditoria [IBM 2011]. A manipulação do cabeçalho de mensagens SOAP possibilita a inserção de informações, como a identidade de um usuário, entre os dados a serem entregues ao destinatário de forma não intrusiva, dado que evita a alteração da funcionalidade exposta pelo serviço e o modo de invocação de tal funcionalidade por parte do remetente.

3 WS-Security

De acordo com a especificação da OASIS [WS-Security, 2006], o *Web Services Security* (WS-Security) descreve um conjunto de extensões para o aperfeiçoamento de mensagens SOAP, a fim de proporcionar proteção através de integridade, confidencialidade e autenticação das mensagens. Os mecanismos do WS-Security protegem os conteúdos das mensagens durante o seu transporte e processamento por intermediários. Além disso, extensões adicionais implementam controle de autenticação e autorização, que protegem provedores de serviços de requisições maliciosas [Erl, 2007]. O WS-Security é projetado para trabalhar em conjunto com outras especificações de serviços web, como mostra a Figura 1.

O WS-Security é flexível e projetado para ser usado como base para aplicação de segurança em serviços web em uma ampla variedade de modelos de segurança, incluindo PKI, Kerberos e SSL. Além disso, a sua especificação provê suporte a múltiplos formatos de *token* de segurança, domínios confiáveis, formatos de assinatura e tecnologias de criptografia. A sua flexibilidade e extensibilidade permitem a acomodação de uma variedade de mecanismos de autenticação e autorização. Por exemplo, um solicitante deve fornecer prova de identidade e um pedido assinado que eles tenham um certificado particular de negócio. Um serviço Web, ao receber tal mensagem poderia então determinar que tipo de confiança depositar no pedido.

O WS-Security provê três principais mecanismos: (i) habilidade de enviar *tokens* de segurança como parte da mensagem; (ii) integridade da mensagem; e, (iii) confidencialidade da mensagem. Esses mecanismos por si só não provêem uma solução de segurança completa para serviços web, contudo, podem ser utilizados em conjunto com outras extensões de serviços web e outros protocolos específicos para acomodar uma maior variedade de modelos de segurança e tecnologias de segurança.

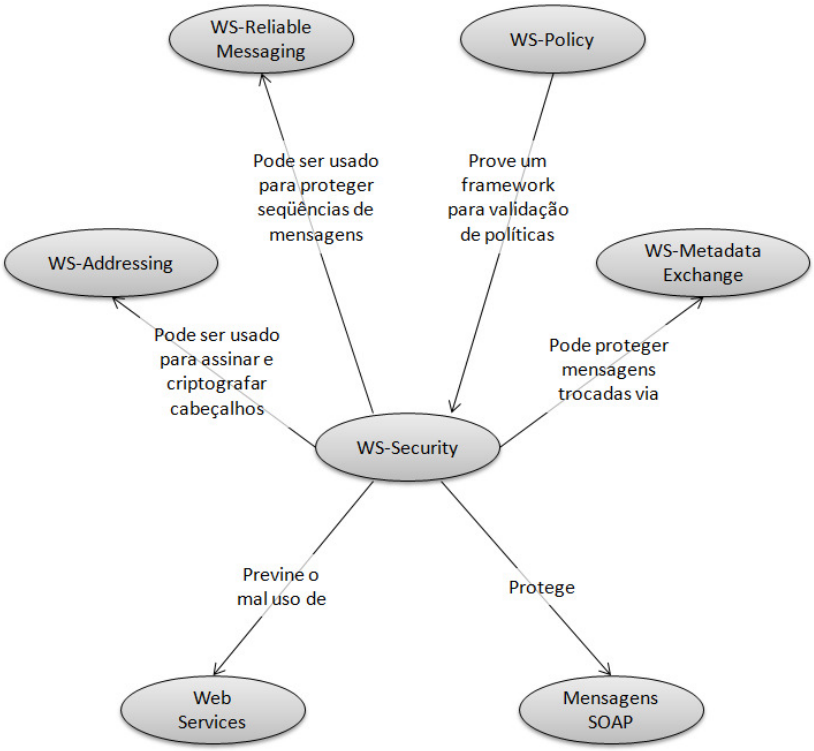

Figura 1 - Relação do WS-Security com outras especificações de *web services* [Adaptado de Erl, 2005]

Tokens de segurança podem ser codificados e anexados às mensagens SOAP. O WS-Security provê *profiles* que descrevem como codificar *Username* Tokens, Tokens X.509, Tokens SAML, REL Tokens e Tokens Kerberos, bem como formas de incluir chaves criptografadas como uma amostra de diferentes tipos de *tokens* binários. O domínio desses mecanismos pode ser estendido para levar informações de autenticação em solicitações de serviços web.

A integridade da mensagem é fornecida através da assinatura XML e *tokens* de segurança para garantir que as mensagens se originaram do remetente adequado e não foram modificadas durante o caminho. Da mesma forma, a confidencialidade da mensagem aproveita XML-Encryption e *tokens* de segurança para manter partes de uma mensagem SOAP confidenciais.

O principal foco deste trabalho está na implementação de mecanismos para extração e inserção de informações na mensagem SOAP para propagação de identidade. Isto foi feito através da implementação de *handlers* que incluem a identidade do usuário em toda mensagem SOAP de saída e procuram por esta informação em toda mensagem SOAP de entrada a fim de realizar a propagação de identidade. Devido a este foco do trabalho, foi decidido que o dado a ser propagado deveria ser armazenado em um atributo *username* do cabeçalho da mensagem SOAP. No entanto, o uso do padrão WS-Security poderia ser utilizado a fim de possibilitar a evolução do trabalho para considerar outras características de segurança. Este uso é apontado como trabalho futuro.

4 Solução Proposta

A proposta de solução deste trabalho segue a arquitetura apresentada na Figura 2. Toma-se como premissa o usuário ter sido devidamente autenticado pela aplicação cliente e ter suas informações armazenadas em um objeto que pode ser acessado de forma isolada. Autenticação do usuário corresponde à verificação mútua da identidade das partes que estão se comunicando [Needham e Schroeder, 1978], ou seja, consideramos que o usuário executando a aplicação cliente é um usuário válido para as camadas servidoras.

Quando a aplicação cliente faz uma requisição a um serviço web executando em um servidor de aplicação, um *handler* intercepta a invocação do serviço e inclui a chave do usuário na mensagem SOAP de requisição (ponto "A" da Figura 2). Quando a mensagem de requisição chega no servidor de aplicação, antes do serviço ser executado, a chave do usuário é extraída da mensagem e armazenada em um objeto cujo escopo é o do tratamento da requisição (ponto "B" da Figura 2). Quando o serviço acessa o banco de dados, a chave do usuário é obtida deste objeto e é injetada no contexto do SGBD (ponto "C" da Figura 2). O banco de dados retorna para o serviço apenas os dados autorizados, o serviço então os retorna para a aplicação cliente. A Figura 2 exemplifica um cenário em que a aplicação cliente invoca diretamente o serviço web. No entanto, cenários semelhantes podem incluir a chamada de mais de um serviço neste fluxo (i.e., aplicação cliente invoca serviço web que invoca outro serviço web que acessa o banco de dados).

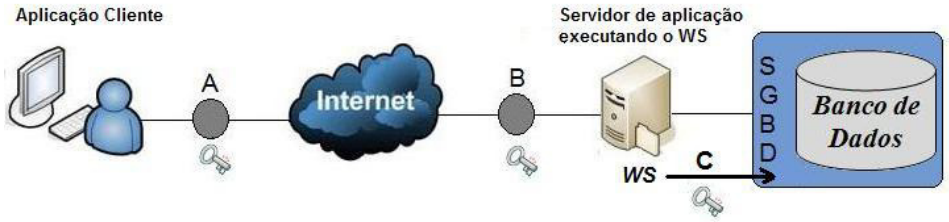

Figura 2 - Arquitetura da Solução Proposta

A implementação da arquitetura proposta consiste na injeção da chave do usuário no cabeçalho da mensagem SOAP [Gudgin *et al.* 2007] enviada para o serviço web. Considera-se um cenário de aplicação onde dois serviços web são utilizados, além de uma aplicação cliente e um SGBD (Figura 3). Os serviços foram divididos segundo a arquitetura proposta por Josuttis [2007]: (i) serviço de dados, responsável por executar operações CRUD – *Create*, *Retrieve*, *Update* e *Delete* – no banco de dados, encapsulando o acesso a dados; e (ii) serviço de lógica, responsável por executar a lógica do negócio. A aplicação cliente acessa o serviço de lógica que acessa o serviço de dados que acessa a banco de dados. O acesso ao banco de dados é feito através do *framework* de mapeamento objeto-relacional Hibernate, o qual injeta os dados do usuário no contexto do SGBD. No lado do SGBD se encontra o *framework* FARBAC, que armazena e aplica as regras de autorização com base no usuário informado.

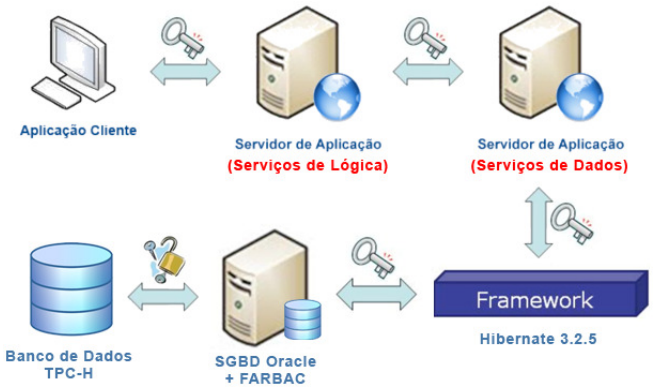

Figura 3 - Cenário para implementação da Solução Proposta

As subseções a seguir detalham as alterações realizadas nos serviços web e na aplicação cliente a fim de viabilizar a propagação da identidade do usuário até o SGBD. Como poderá ser observado, um módulo extra é introduzido, um projeto compartilhado entre os serviços e a aplicação cliente, comportando artefatos em comum.

4.1 Criação do *handler* e projeto compartilhado

Um projeto compartilhado foi criado para comportar todos os artefatos que podem ser utilizados por mais de uma camada (por exemplo, serviço ou cliente). Dentre os artefatos está a classe *handler* que tem duas funções: (i) inserir os dados do usuário a serem propagados no cabeçalho das mensagens SOAP sempre que um requisitante invocar um serviço; e (ii) procurar por dados do usuário no cabeçalho da mensagem SOAP sempre que uma requisição for recebida.

Um *handler* pode ser utilizado em um ambiente JAX-WS considerando dois passos apresentados por Kalin [2009]:

1. Criar uma classe *handler* que implemente a interface *Handler* do pacote `javax.xml.ws.handler`. JAX-WS provê duas subinterfaces de *Handler*, *LogicalHandler* e *SOAPHandler*. O *LogicalHandler* permite acesso apenas ao *payload* (corpo) da mensagem SOAP, enquanto *SOAPHandler* permite acessar a mensagem como um todo.
2. Colocar o *handler* dentro de uma *handler chain*. A *handler chain* especifica quais *handlers* devem ser utilizados pela aplicação e em qual ordem. Isto é feito tipicamente através de um arquivo de configuração, embora *handlers* também possam ser gerenciados através de código.

Uma vez injetado no *handler framework*, o *handler* age como um interceptador que tem acesso à mensagem de chegada e saída. Considerando a visão de uma aplicação cliente, uma mensagem sendo enviada é interceptada, e manipulada, logo após ser disparada (ponto "A" da Figura 2), e uma mensagem de resposta é interceptada antes de ser entregue ao objeto de destino (ponto "B" da Figura 2).

Dependendo do cenário, podem existir mais de um interceptador. Portanto, é necessário determinar quais *handlers* serão utilizados e em qual ordem deverão ser invocados. Neste trabalho, foi utilizada a configuração de *handler* por arquivo de configuração, ao invés de realizar gerenciamento da cadeia via código. Esta escolha se

deve à flexibilidade e facilidade de manutenção, além de ser menos intrusiva no código legado, ocasionando apenas a inclusão de algumas anotações em determinadas classes. O *handler* criado recebeu o nome de "UsernameHandler" e implementa a interface SOAPHandler. Ele inclui a identidade do usuário em toda mensagem SOAP de saída e procura por esta informação em toda mensagem SOAP de entrada. Tais funcionalidades foram implementadas no método *handleMessage()* da classe. Foram definidas três variáveis estáticas no escopo da classe *handler* (Figura 4): NAMESPACE, HEADER_ELEMENT_NAME e *_username*. Esta última foi declarada como pública, para que a informação do usuário nela armazenada pudesse ser utilizada pela aplicação.

```
private final static String NAMESPACE = "http://br.unirio.webservice";
private final static String HEADER_ELEMENT_NAME = "username";
public static final ThreadLocal<String> _username = new ThreadLocal<String>();
```

Figura 4 - Definição das variáveis estáticas do *handler*

A variável "_username" é do tipo *ThreadLocal*, sendo esta classe utilizada para garantir o isolamento do dado em situações de concorrência. Quando um serviço recebe requisições, uma *thread* é criada para atender cada uma destas requisições feitas ao servidor. Desta forma, cada requisição irá possuir sua própria variável _username. O método *handleMessage()* armazena nesta variável o valor extraído do cabeçalho da mensagem SOAP, garantindo que em situações onde múltiplas requisições são atendidas simultaneamente, cada requisição tenha conhecimento do usuário injetado no cabeçalho da mensagem SOAP que originou a requisição em si.

A Figura 5 mostra o método *handleMessage()*, que inicialmente verifica se a mensagem SOAP interceptada é *outbound* ou *inbound*, isto é, se está sendo enviada, ou recebida. Caso a mensagem seja *outbound* (linhas 20 a 31), o método insere as informações do usuário como um novo elemento no cabeçalho da mensagem SOAP através do comando *addTextNode()*. Caso a mensagem seja *inbound* (linhas 7 a 19), o método verifica a existência no cabeçalho SOAP de um elemento com o mesmo *namespace* definido na variável NAMESPACE e com o mesmo nome definido pela variável HEADER_ELEMENT_NAME, ou seja, verifica se no cabeçalho desta mensagem SOAP foi injetado um usuário. Caso exista tal elemento, o valor nele armazenado é extraído e atribuído ao objeto "_username".

```
1  public boolean handleMessage(SOAPMessageContext context) {
2      try {
3          boolean isOutboundDirection =
4                  ((Boolean) context.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY))
5                  .booleanValue();
6
7          if (!isOutboundDirection) {
8              SOAPHeader soapHeader = context.getMessage().getSOAPHeader();
9              NodeList headerElements = soapHeader.getElementsByTagNameNS(
10                 NAMESPACE, HEADER_ELEMENT_NAME);
11             String value = null;
12
13             if ((headerElements != null) && (headerElements.getLength() > 0)) {
14                 Node node = headerElements.item(0);
15                 if (node != null) {
16                     value = node.getFirstChild().getNodeValue();
17                     _username.set(value);
18                 }
19             }
20         }else{
21             SOAPMessage msg = context.getMessage();
22             SOAPEnvelope env = msg.getSOAPPart().getEnvelope();
23             SOAPHeader soapHeader = env.getHeader();
24             if (soapHeader == null){
25                 soapHeader = env.addHeader();
26             }
27             QName qname = new QName(NAMESPACE, HEADER_ELEMENT_NAME);
28             SOAPHeaderElement headerElement = soapHeader.addHeaderElement(qname);
29             headerElement.addTextNode(_username.get().toString());
30             msg.saveChanges();
31         }
32     }catch(SOAPException e){
33         System.out.println(e);
34     }
35     return true;
36 }
```

Figura 5 – Método *handleMessage()*

O projeto compartilhado é então construído, a fim de gerar um arquivo “.JAR“ que possa ser utilizado pelo cliente e pelos dois serviços.

4.2 Inclusão do *handler* no Projeto Cliente e nos Serviços de Lógica e Dados

Para que o usuário possa ser injetado no cabeçalho da mensagem SOAP de forma não intrusiva e para que este mesmo dado possa ser lido por quem receber a mensagem, emissor e receptor devem fazer uso do *handler* disponibilizado pelo módulo compartilhado. A seguir são explicadas as alterações necessárias na aplicação cliente para que esta injete os dados do usuário na mensagem SOAP.

Para facilitar a explicação consideramos que a aplicação cliente consiste em um método “*main()*” que invoca uma funcionalidade do serviço de lógica (Figura 6). Autenticação do usuário está fora do escopo deste trabalho. A identidade do usuário é armazenada no atributo “_username” da classe UsernameHandler (linha 4).

```
3  public static void main(String[] args) throws IOException{
4      UsernameHandler._username.set("Y2R7");
5
6      LogicServiceService service = new LogicServiceService();
7      LogicService port = service.getLogicServicePort();
8
9      String result = port.checkMostCommonPriority("1992-01-01","3");
10     System.out.println("Tipo mais comum de Pedido: "+result);
11 }
```

Figura 6 - Método *main()* da Aplicação Cliente

Apesar de o dado já ter sido injetado na classe UsernameHandler, ainda não foi definido que esta deve ser utilizada pela aplicação como um *handler*. Para que o projeto

reconheça a classe como um *handler* deve-se ir até o pacote de *stubs* (classes) gerados para acessar o serviço web e criar o arquivo "handlerchain.xml" (como ilustrado na Figura 7), na raiz do projeto, onde especifica-se quais classes devem ser utilizadas como *handlers* e em qual ordem. Para gerar o pacote de *stubs* para acessar o serviço foi utilizado o WSIMPORT [ORACLE 2011], um utilitário que é parte integrante do *core* da versão 6 do Java.

```
<?xml version="1.0" encoding="UTF-8"?>
<handler-chains xmlns="http://java.sun.com/xml/ns/javaee">
    <handler-chain>
        <handler>
            <handler-class>br.uniriotec.propid.shared.UsernameHandler</handler-class>
        </handler>
    </handler-chain>
</handler-chains>
```

Figura 7 - Arquivo "handlerchain.xml"

Para colocar o *handlerchain* em uso deve-se alterar a classe que representa o serviço sendo invocado para referenciar o *handlerchain*. No protótipo implementado para exemplificar a solução, a aplicação cliente acessa um serviço de lógica e a classe que representa este serviço é a LogicServiceService, gerado pelo WSIMPORT. Nesta classe, logo antes do seu nome é inserida a anotação @HandlerChain (Figura 8 – linha 4) que informa qual arquivo de configuração deve ser utilizado como *handlerchain*.

```
@WebServiceClient(name = "LogicServiceService",
    targetNamespace = "http://services.logica.c4.propid.uniriotec.br/",
    wsdlLocation = "http://localhost:7001/C4ServicoLogica/LogicServiceService?wsdl")
@HandlerChain(file="handlerchain.xml")
public class LogicServiceService extends Service{

        /* conteúdo da classe omitido */

}
```

Figura 8 - Uso da anotação @HandlerChain

Portanto, na aplicação cliente, para usar o *handlerchain* foi necessário apenas criar o arquivo "handlerchain.xml" no pacote onde estão os *stubs* para acessar o serviço (Figura 7) e incluir a anotação @HandlerChain no serviço sendo invocado (Figura 8).

O serviço de lógica invoca o serviço de dados e este injeta as informações do usuário no banco de dados. Logo, o serviço de lógica deve extrair os dados injetados no cabeçalho da mensagem SOAP e repassar para o serviço de dados, mesmo que não chegue a usar estas informações, como é o caso do protótipo deste trabalho. Enfim, é necessário que todo e qualquer serviço intermediário repasse a informação adiante.

Da mesma forma que na aplicação cliente, no projeto do serviço de lógica foi necessário criar o arquivo de configuração "*handlerchain.xml*" - no pacote onde estão os *stubs* para acessar o serviço de dados - que recebeu a mesma configuração utilizada na aplicação cliente (Figura 7). O serviço de lógica acessa o serviço de dados. O conjunto de *stubs* necessários para o acesso ao serviço de dados foi gerado através do utilitário WSIMPORT. Para determinar que ocorra a injeção do dado do usuário nas mensagens de saída com destino ao serviço de dados, da mesma forma que ocorreu com o uso do serviço de lógica pela aplicação cliente, deve ser feita a inclusão da anotação @HandlerChain (linha 6 da Figura 9) na classe *stub* que especifica o acesso ao serviço de dados (no caso, a classe DataServiceService, gerada pelo utilitário WSIMPORT).

```
3  @WebServiceClient(name = "DataServiceService",
4          targetNamespace = "http://services.c4.propid.uniriotec.br/",
5          wsdlLocation = "http://localhost:7001/C4ServicoDados/DataServiceService?wsdl")
6  @HandlerChain(file="handlerchain.xml")
7  public class DataServiceService  extends Service{
8      /** Código interno da classe omitido */
9  }
```

Figura 9 - Inserção da anotação @HandlerChain no acesso ao serviço de dados

Como o *handler* atua tanto nas mensagens de entrada como nas mensagens de saída do serviço de lógica, foi necessário também referenciar o arquivo "*handlerchain.*xml" com a anotação @HandlerChain no próprio serviço de lógica, como apresentado pela linha 3 da Figura 10. Isto faz com que o *handler* atue na mensagem de entrada deste serviço.

```
2  @WebService
3  @HandlerChain(file="handlerchain.xml")
4  public class LogicService {
5
6      @WebMethod
7      public String checkMostCommonPriority(String dataInicio, String intervalo){
8          String result = "";
9
10         DataServiceService service = new DataServiceService();
11         DataService port = service.getDataServicePort();
12
13         PriorityList lista = port.orderPriorityCheck(dataInicio, intervalo);
14
15         int priorityCount = -1;
16         for(OrderPriorityItem item : lista.getLista()){
17             if(priorityCount < item.getQuantidade()){
18                 result = item.getPrioridade();
19                 priorityCount = item.getQuantidade();
20             }
21         }
22
23         return result;
24     }
25 }
```

Figura 10 - Inserção da anotação @HandlerChain no Serviço de Lógica

O serviço de dados sofre algumas alterações adicionais, não devido à recepção do dado injetado, mas à necessidade de injetar o dado recebido em uma conexão com o banco de dados. Para tal foram utilizadas técnicas similares às propostas em Leão *et al.* [2011a], onde uma fábrica de sessões alternativa é utilizada, em detrimento da fábrica fornecida pelo *framework* Hibernate.

O serviço de dados requer a inclusão da anotação @HandlerChain e da referência ao arquivo *handlerchain*, que assim como no serviço de lógica (Figura 10), permite que o serviço de dados recupere o usuário injetado no cabeçalho da mensagem SOAP. O arquivo *handlerchain.xml* é o mesmo apresentado na Figura 7.

O serviço de dados utiliza um objeto DAO (*Data Access Object*) [Sun Microsystems 2002] para acessar o banco de dados via Hibernate. Neste trabalho, propomos o uso de uma fábrica de sessões (Figura 11), diferente da fornecida pelo *framework* Hibernate. Esta nova fábrica realiza operações adicionais para propagação da identidade para o contexto do banco de dados, antes de retornar a sessão de acesso ao banco a quem a requisitou. Esta nova fábrica faz uso da fábrica de sessões do Hibernate para gerar uma nova sessão e realiza as operações adicionais sobre esta sessão. Ou seja,

ao invés de acessar diretamente a fábrica de sessões do Hibernate, o serviço acessa a nova fábrica e esta acessa a fábrica de sessões original do Hibernate.

```
13      public Session openSession(){
14          Session s = HibernateUtil.getSessionFactory().openSession();
15
16          if(UsernameHandler._username.get() != null){
17              Connection con = s.connection();
18              try{
19                  CallableStatement st = con.prepareCall(
20                          "{call dbms_application_info.set_client_info(?)}");
21                  st.setString(1, UsernameHandler._username.get().toString());
22                  st.executeUpdate();
23              }catch(SQLException e){
24                  return null;
25              }
26          }
27          return s;
28      }
```

Figura 11 – Método *openSession()* da nova fábrica de sessões

A fábrica de sessões do Hibernate é acessada através do método *getSessionFactory()* da classe HibernateUtil (Figura 11 – linha 14), que configura o *framework* Hibernate. Com a referência para a fábrica em mãos, é utilizado o método *openSession()*, que devolve uma sessão com o banco de dados. A partir deste ponto entra o código responsável por realizar a injeção do usuário na sessão. Verifica-se se o artefato UsernameHandler possui o atributo "*_username*" ajustado. O fato de o atributo armazenar algum valor significa que o serviço recebeu, através do cabeçalho da mensagem SOAP que carregou a requisição, a informação de qual usuário está utilizando a aplicação cliente. No caso de o atributo "*_username*" possuir valor diferente de NULL, é realizada a chamada ao procedimento DBMS_APPLICATION_INFO.SET_CLIENT_INFO(), informando como parâmetro o usuário armazenado pelo atributo. A procedure citada faz parte do *namespace* USERENV do banco de dados Oracle [Jeloka *et al.*, 2008], sendo utilizada para ajustar no contexto da sessão o usuário que está de fato fazendo uso daquela sessão. Sendo o usuário reconhecido é possível utilizar o *framework* FARBAC para aplicar regras de autorização de acesso aos dados armazenados.

Para realizar a troca de fábricas basta alterar no objeto DAO responsável por realizar as operações com o banco de dados a linha que define qual classe deve ser utilizada como fábrica de sessões, passando a invocar a nova fábrica de sessões.

5 Testes Experimentais

A fim de avaliar a proposta de solução em um ambiente de alta concorrência, a aplicação cliente e os serviços descritos na seção anterior (de dados e de lógica) foram implementados utilizando tecnologia Java (Java SE para aplicação cliente e JAX-WS para os serviços). Um banco de dados foi carregado seguindo o esquema do *benchmark* TPC-H [TPC Council 2008] sendo configurado com o *framework* FARBAC. Para disponibilizar os serviços, foi utilizado o Oracle Weblogic Server (10g R3) como servidor de aplicação e o Oracle 10g como SGBD. Para realizar os testes concorrentes utilizou-se o *framework* para testes de unidade TestNG [Beust e Suleiman 2007], uma vez que este fornece por natureza as ferramentas necessárias para execução de testes de paralelismo de forma simples e eficaz.

O *benchmark* TPC-H simula o cenário de um sistema de apoio a decisão, provendo um contexto realista que representa a atividade de vendas de uma indústria que administra suas vendas para todo o mundo. Para este teste foi utilizada a consulta *OrderPriorityCheck*, proposta pelos desenvolvedores do *benchmark*, que visa obter a lista com todos os níveis de prioridade dos pedidos e a quantidade de pedidos para cada nível realizados em um determinado espaço de tempo. Esta consulta é invocada através do serviço de dados. O serviço de lógica tem como objetivo retornar o nível de prioridade mais comum entre os pedidos realizados, para isto recupera-se a lista com todos os níveis de prioridade fornecidos pelo serviço de dados e filtra-se aquele que possui a maior quantidade de pedidos registrada.

A seguinte regra de autorização foi implementada no *framework* FARBAC: "Um gerente da América do Norte e Ásia deve somente acessar pedidos de clientes localizados no hemisfério norte e que se encontram nas regiões da Ásia e América". Considerando a consulta, as informações a serem protegidas se encontram nas tabelas *ORDERS*, que armazena os pedidos realizados, e *LINEITEM*, que armazena os itens de cada pedido. As tabelas possuem aproximadamente 3 milhões e 12 milhões de registros, respectivamente. A Figura 12 apresenta os predicados retornados pelo FARBAC.

```
(O_CUSTKEY in (
select C_CUSTKEY
from TPCH.CUSTOMER
inner join TPCH.NATION on N_NATIONKEY = C_NATIONKEY
inner join TPCH.REGION on R_REGIONKEY = N_REGIONKEY
where R_NAME IN ('AMERICA' , 'ASIA') AND
N_HEMISPHERE IN ('NORTH')))
```

```
(L_ORDERKEY in (
select O_ORDERKEY
from TPCH.ORDERS))
```

Figura 12 - Predicados FARBAC para as tabelas *ORDERS* (esquerda) e *LINEITEM* (direita)

Dois usuários foram criados: *Bob*, como gerente da América do Norte e Ásia; e *Alice*, a presidente da empresa, que recebeu o privilégio *EXEMPT ACCESS POLICY*, fazendo com que todas as políticas sejam ignoradas.

A aplicação cliente foi simulada através de um caso de teste implementado com o auxílio do *framework* TestNG (Figura 13). Como a proposta de solução não considera a autenticação de usuários, a passagem destes ao *handler* (para que este realize a injeção na mensagem SOAP) é feita dentro do próprio caso de teste. A utilização do serviço de lógica pode ser observada nas linhas 21, 22 e 24, onde é invocado o método *checkMostCommonPriority()*, que retorna um objeto *String* informando o nível mais comum de prioridade entre os pedidos realizados no intervalo de 3 meses a partir da data 02/07/1992. Espera-se que o usuário *Alice* receba como resposta o nível "1-URGENT", enquanto o usuário *Bob* deve receber o resultado "5-LOW".

O caso de teste foi configurado para ser executado 100 vezes, permitindo o paralelismo de até 5 *threads* por vez. De forma intercalada, foram realizadas invocações com os usuários Bob e Alice. Foi possível observar que a regra de autorização implementada foi de fato aplicada de forma isolada, dado que os valores retornados foram iguais aos declarados como esperados para cada usuário e houve intercalação de nas respostas recebidas. Portanto, foi possível observar que mesmo com diferentes usuários realizando requisições simultâneas a proposta de solução é aplicável em ambientes de alta concorrência utilizados por múltiplos usuários, garantindo que cada usuário só visualize os dados aos quais tem permissão de acesso.

```
public class TesteConcorrenciaTPCH {
    private Map<String, String> mapUsuarios;
    private int flag = 0;

    @BeforeTest
    public void beforeTest() {
        mapUsuarios = new HashMap<String, String>();
            mapUsuarios.put("ALICE", "1-URGENT       ");
            mapUsuarios.put("BOB", "5-LOW          ");
    }

    @Test(threadPoolSize = 5, invocationCount = 100)
    public void testConcorrenciaConsulta() {
        String[] usuarios = mapUsuarios.keySet().toArray(new String[0]);
        String usuario = usuarios[flag];

        UsernameHandler._username.set(usuario);

        if(flag == 1){flag = 0;}else{flag = 1;}

        LogicServiceService service = new LogicServiceService();
        LogicService port = service.getLogicServicePort();

        String result = port.checkMostCommonPriority("1992-07-02","3");
        Assert.assertEquals(result, mapUsuarios.get(usuario));
    }
}
```

Figura 13 - Caso de Teste para simular a aplicação Cliente

A seguir foi realizado teste de desempenho com o objetivo de observar o custo de processamento adicionado pelo uso de *handlers* para realizar a propagação de identidade. Testes de unidade foram executados, primeiramente com serviços que consideravam a propagação através de *handlers* e posteriormente com serviços que não faziam uso dos interceptadores. Em ambos os casos utilizou-se um *Mock Object* para realizar o papel de objeto DAO, objetivando a retirada do banco de dados do cenário de testes para que o processamento deste não interferisse na medição. Assim como no teste de concorrência o cenário contou com um cliente e dois serviços web, sendo um de lógica e um de dados. Os testes de unidade foram executados pela mesma máquina onde se encontravam disponíveis serviços, de modo que o desempenho da rede também não interferisse nos resultados, sendo configuradas 5000 requisições permitindo o paralelismo de até 5 *threads* por vez. Como resultado observou-se que no cenário sem propagação de identidade as 5000 requisições foram realizadas em 3 minutos e 30 segundos (média de 0,042 segundos por requisição), enquanto no cenário considerando a propagação as mesmas 5000 requisições foram executadas em 3 minutos e 59 segundos (média de 0,0478 segundos por requisição), ou seja, 14% a mais de tempo de processamento, em média. Considerando um cenário onde é requisito acesso seguro a informações, este overhead é mínimo, demonstrando que a solução proposta não gera acréscimo de processamento que impacte de forma considerável o ambiente ou o tempo de resposta da arquitetura.

6 Conclusões

Segurança da Informação é um importante desafio para a maioria das organizações. De forma geral, problemas nesta área são solucionados através da implementação de mecanismos para controle de acesso e de regras de autorização nas próprias aplicações. Entretanto, quando tais regras se alteram, é necessário atualizar todos os sistemas que possuem tais regras implementadas, tarefa de grande complexidade, especialmente em cenários compostos por aplicações legadas e um grande número de regras de autorização. Logo, centralizar as regras no banco de dados garantindo uniformidade no acesso a dados é uma solução importante e vem sendo estudada por vários trabalhos da

literatura [DoD, 1983; Ferraiolo e Khun, 1992; Sandhu *et al.*, 1996; Jeloka *et al.*, 2008; Azevedo *et al.*, 2010]. No entanto, para que o banco de dados possa aplicar as regras adequadamente, é necessário que as informações do usuário da aplicação sejam propagadas até o banco de dados de forma adequada.

Este trabalho apresentou uma solução para controle de acesso em uma arquitetura que faz uso de serviços web, com foco em propagação de identidade e aplicação de regras de autorização. As principais contribuições deste trabalho são: (i) a proposta e implementação de arquitetura através do uso de SOAP *handlers* para interceptar requisições a serviços e injetar a identidade real do usuário acessando a aplicação cliente na mensagem SOAP de modo a propagar tal identidade até o SGBD; (ii) integrar a propagação de identidade com a execução de regras de autorização permitindo assim que mecanismos de controle de acesso no SGBD possam ser aplicados efetivamente.

Testes experimentais foram executados para testar o isolamento da solução e desempenho da proposta. Em relação ao isolamento, foi avaliado se o usuário executando a aplicação cliente é propagado até o banco de dados, e se apenas os dados que este usuário tem acesso são disponibilizados, sem interferência de outras conexões concorrentes. Os resultados dos testes confirmaram isolamento. Além disso, observou-se baixíssimo impacto a sistemas que utilizam serviços web, facilitando a tarefa de mantê-los atualizados. Para realizar a propagação da identidade entre as camadas de aplicação, quem consome o serviço (envia a identidade do usuário) precisa configurar um arquivo com a cadeia de *handlers* (p.e., "*handlerchain.xml*" - Figura 7) e incluir no *stub* que representa o serviço invocado uma anotação para considerar a cadeia (anotação @HandlerChain - Figura 8). No serviço que atende à requisição (recebe a identidade do usuário) também é necessário configurar o arquivo da cadeia de *handlers* (p.e., "*handlerchain.xml*" - Figura 7) e indicar que o *handler* deve ser utilizado quando a mensagem de entrada for recebida (inserção da anotação @HandlerChain - Figura 10). Já para propagar a identidade para o contexto do SGBD, propomos o uso de uma nova fábrica de sessões para injeção do usuário (uma proposta de método para criação da conexão é apresentado na Figura 11). Em relação ao teste de desempenho, os resultados demonstraram que o *overhead* é muito pequeno (média de 14%).

A principal premissa deste trabalho é o uso de serviços web implementados utilizando tecnologia Java. Como trabalho futuro temos a implementação da solução empregando outras tecnologias como, por exemplo, .Net e o próprio uso da solução em ambientes em que ambas as tecnologias aparecem. Além disso, propomos expandir a solução para armazenar os dados a serem propagados empregando o padrão WS-Security além de tratar outras questões de segurança como, por exemplo, autenticação e confidencialidade.

Referências Bibliográficas

Azevedo, L.G. et al., (2010). "A Flexible Framework for Applying Data Access Authorization Business Rules". *In Proceedings of the 12th International Conference on Enterprise Information Systems (ICEIS 2010)*. Funchal, Madeira, Portugal , pp. 275-280.

Beust, C., Suleiman, H., (2007) "*Next Generation Java Testing: TestNG and Advanced Concepts*", Publicado por Addison-Wesley Professional , ISBN-10: 0321503104.

BRG, (2000). "Defining Business Rules ~ What Are They Really?", Rev. 1.3, 2000,

http://www.businessrulesgroup.org/first_paper/BRG-whatisBR_3ed.pdf.
Calì, A. & Martinenghi, D., (2008). “Querying Data under Access Limitations”. *In IEEE 24th International Conference on Data Engineering (ICDE 2008)*, Cancun, Mexico, pp. 50-59.
Department of Defense - DoD, (1983). “Trusted computer security evaluation criteria”. http://csrc.nist.gov/publications/history/dod85.pdf.
Elmasri, R., Navathe, S. (2010). “Fundamentals of Database Systems”, Addison Wesley.
ERL, T. (2007). SOA: Principles of Service Design. Prentice Hall, Crawfordsville: Indiana, 608 p.
Ferraiolo, D.A. e Kuhn, D.R., (1992). “Role-Based Access Control”. In National Computer Security Conference. Baltimore, MD, pp. 554-563.
Gudgin, M., Hadley, M., Mendelsohn, N. et al. (2007). “SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)”. World Wide Web Consortium, April 2007. http://www.w3.org/TR/2007/REC-soap12-part1-20070427/.
IBM (2011), “Identity propagation and distributed security”. http://publib.boulder.ibm.com/infocenter/cicsts/v4r1/topic/com.ibm.cics.ts.doc/dfht5/topics/idprop_intro.html.
Jeloka, S., Mulagund, G. e Lewis, N., (2008). “Oracle Database Security Guide”. *Oracle RDBMS 10gR2*. http://download.oracle.com/docs/cd/B19306_01/network.102/b14266.pdf.
Josuttis, N. (2007). “SOA in practice: The Art of Distributed System Design”. O’Reilly, 352 p.
Kalin, M. (2009). “Java Web Services: Up and Running”. Editora O’Reilly, 2009.
Kotamraju, J. (2009), “The Java API for XML-Based Web Services (JAX-WS) 2.2”. http://download.oracle.com/otn-pub/jcp/jaxws-2.2-mrel3-evalu-oth-JSpec/jaxws-2_2-mrel3-spec.pdf.
Leão, F., Puntar, S., Azevedo, L. G., Cappelli, C., Baião, F. (2011a). "Controle de Acesso a Dados em Sistemas de Informação através de Mecanismos de Propagação de Identidade e Execução de Regras de Autorização". In: VII Simpósio Brasileiro de Sistemas de Informação (SBSI 2011), Salvador, Brasil.
Leão, F., Azevedo, L. G., Baião, F., Cappelli, C. (2011b). "Enforcing Authorization Rules in Information Systems". In: IADIS International Conference Applied Computing 2011, Rio de Janeiro, Brasil.
Murthy, R. & Sedlar, E., (2007). Flexible and Efficient Access Control in Oracle. In: *2007 ACM SIGMOD International Conference on Management of Data*. Beijing, p. 973.
Needham, R. M., Schroeder, M.D. (1978) “Using Encryption for Authentication in Large Networks Computers”. Communications of the ACM 21(12).
ORACLE, (2008). “Programming Advanced Features of WebLogic Web Services Using JAX-WS, 10g Release 3”. http://download.oracle.com/docs/cd/E12840_01/wls/docs103/pdf/webserv_adv.pdf.
ORACLE, (2011). “WSimport – Java TM API for XML Web Services (JAX-WS) 2.0”. http://download.oracle.com/javase/6/docs/technotes/tools/share/wsimport.html.
Puntar, S., Azevedo, L.G., Baião, F., Cappelli, C. (2011). “Implementing Flexible and Efficient Authorization Business Rules in Information Systems”. In: IADIS International Conference Applied Computing 2011, Rio de Janeiro, Brasil.
Sandhu, R.S., Coyne, E.J., Feinstein, H.L., Youman, C.E. (1996). “Role-based access control models”. *Computer*. Vol. 29, No. 2, pp. 38-47.
SUN MICROSYSTEMS, Inc. (2002) “Core J2EE Patterns - Data Access Object”. http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html.
TPC Council, (2008). TPC Benchmark H Standard Specification Revision 2.8.0. *Transaction Processing Performance Council*. Available at: http://www.tpc.org/tpch/spec/tpch2.8.0.pdf.
W3C (2004). Web Services Architecture. http://www.w3.org/TR/ws-arch/. Acessado em Dezembro de 2011.

WS-Security (2006). Web Services Security: SOAP Message Security, OASIS Standard Specification, version 1.1. Disponível em http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf. Acessado em 25 de Junho de 2014.

Yang, L., (2009). "Teaching database security and auditing". *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, Vol. 41, No. 1, pp. 241-245.