

LIGA DE ENSINO DO RIO GRANDE DO NORTE
CENTRO UNIVERSITÁRIO DO RIO GRANDE DO NORTE
ESPECIALIZAÇÃO EM DESENVOLVIMENTO DE SISTEMAS CORPORATIVOS

GEWTON JHAMES DE SENA TEIXEIRA

**APLICAÇÕES DE TÉCNICAS DE SEGURANÇA EM SISTEMAS CORPORATIVOS
COM SPRING SECURITY**

NATAL-RN
2012

GEWTON JHAMES DE SENA TEIXEIRA

**APLICAÇÕES DE TÉCNICAS DE SEGURANÇA EM SISTEMAS
CORPORATIVOS COM SPRING SECURITY**

Monografia apresentada ao curso de Pós-Graduação de Desenvolvimento de Sistemas Corporativos do Centro Universitário do Rio Grande do Norte como parte dos requisitos para a obtenção do título de Especialista em Desenvolvimento de Sistemas Corporativos.

Orientador: Msc. David Ricardo do Vale Pereira.

NATAL-RN
2012

GEWTON JHAMES DE SENA TEIXEIRA

**APLICAÇÕES DE TÉCNICAS DE SEGURANÇA EM SISTEMAS CORPORATIVOS
COM SPRING SECURITY**

Monografia apresentada ao Curso de Pós-Graduação de Desenvolvimento de Sistemas Corporativos do Centro Universitário do Rio Grande do Norte como parte dos requisitos para a obtenção do título de Especialista em Desenvolvimento de Sistemas Corporativos.

Aprovada em: 31/10/2012

BANCA EXAMINADORA

Prof. Msc. David Ricardo do Vale Pereira
Orientador

Prof. Msc. Gleydson de Azevedo Ferreira Lima
Membro

Catálogo na Publicação – Biblioteca da UNI-RN
Setor de Processos Técnicos

Teixeira, Gewton Jhames de Sena.

Aplicações de técnicas de segurança em sistemas corporativos com
spring security / Gewton Jhames de Sena Teixeira. – Natal, 2012.
75 f.

Orientador: David Ricardo do Vale Pereira (Ms.)

Monografia (Especialização em Desenvolvimento de Sistemas
Corporativos) – Centro Universitário do Rio Grande do Norte.

1. Aplicações – Monografia. 2. Segurança – Monografia. 3. Sistemas –
Monografia. 4. Spring Security – Monografia. I. Título.

RN/UNI-RN/BC

CDU 004.43

RESUMO

Com a evolução e distribuição remota dos sistemas corporativos, surgiu a necessidade da criação de tecnologias e artifícios que possam garantir a segurança desses sistemas. Esse novo ramo de estudo deve prever ameaças e garantir que sistemas seguros sejam construídos com o uso de boas práticas e mecanismos que começam em nível de *software* e se estendem até as camadas mais baixas de infraestrutura. Nesse contexto, esforços são feitos para a construção de *frameworks* e ferramentas de apoio à construção de sistemas confiáveis. O Spring Security é um *framework* para segurança de sistemas escritos em linguagem de programação Java e que oferece suporte aos mais populares mecanismos de segurança. É uma solução altamente extensível e não-invasiva, oferecendo assim um impacto mínimo aos principais objetivos da aplicação. A grande vantagem competitiva que o *framework* Spring Security possui em relação às demais soluções para segurança de sistemas em Java é sua extensão e integração com a maioria dos mecanismos de segurança atuais, inclusive com outros *frameworks* de segurança para sistemas em Java. Este trabalho descreve o processo de implementação de alguns desses mecanismos de segurança utilizando o *framework* Spring Security.

Palavras-chave: Spring Security. Java. Segurança.

ABSTRACT

The evolution and remotely distribution of corporate systems brought the need to create technologies and schemes to ensure security on those systems. This new area of research must prevent threats and ensure that secure software is built upon good practices and mechanisms that extend from software to lower infrastructure levels. In this context, efforts are made to build frameworks and tools to support the construction of trustworthy software systems. Spring Security is a security framework for Java which supports the most popular security mechanisms. It is also a highly extensible and non-invasive solution, providing a minimum impact to the main objectives of the application. The great competitive advantage that it has when compared to other Java security solutions is its extensibility and integration with most of the current security mechanisms, including other Java security frameworks. This paper describes the process of implementing some of those security mechanisms with Spring Security.

Keywords: Spring Security. Java. Security.

LISTA DE ILUSTRAÇÕES

Figura 1 – <i>Access Control List</i>	17
Figura 2 – Consulta SQL com a variável a ser substituída.	24
Figura 3 – Consulta SQL pronta para execução.....	24
Figura 4 – Consulta SQL alterada por <i>SQL Injection</i>	24
Figura 5 – Navegador exibindo informações sobre a conexão segura estabelecida com o site paypal.com.....	30
Figura 6 – Fluxo de autenticação OpenID.	32
Figura 7 – Exemplo de papéis no modelo OAuth.	33
Figura 8 – Configuração de uma restrição de segurança na plataforma JEE.	37
Figura 9 – Requisição e resposta na plataforma JEE.....	40
Figura 10 – <i>Filter DelegatingFilterProxy</i> , e <i>listener ContextLoaderListener</i>	42
Figura 11 – Configuração automática de segurança HTTP com Spring Security...	42
Figura 12 – Visão geral dos módulos do Spring Framework.	45
Figura 13 – Criação de um <i>UserDetailsService</i> em memória.....	50
Figura 14 – Referenciando um componente <i>UserDetailsService</i> que possui o identificador <i>uds</i>	50
Figura 15 – Modelo padrão de usuários, papéis e grupos do Spring Security.	51
Figura 16 – Declaração de um <i>JdbcDaoImpl</i> , alterando duas de suas consultas padrão.	52
Figura 17 – Configuração de controle de acesso com <i>pointcuts</i> AOP.	53
Figura 18 – Ativando o suporte à <i>annotation @Secured</i>	53
Figura 19 – Exemplos de uso da <i>annotation @Secured</i>	54
Figura 20 – Ativando o suporte as <i>annotations @PreAuthorize</i> , <i>@PreFilter</i> , <i>@PostAuthorize</i> e <i>@PostFilter</i>	54
Figura 21 – Exemplos de uso das <i>annotations @PreAuthorize</i> e <i>@PreFilter</i>	55
Figura 22 – Expressões comuns para controle de acesso.	55
Figura 23 – Habilitando autenticação via OpenID no Spring Security.	56
Figura 24 – Formulário de <i>login</i> com usuário e senha (acima) e com OpenID (abaixo), após a configuração da diretiva <i>openid-login</i>	56
Figura 25 – Exemplo de formulário de autenticação com OpenID.	57
Figura 26 – Definindo um usuário pelo seu OpenID <i>identifier</i> , com o papel <i>ROLE_USER</i>	57

Figura 27 – Configurando um <i>AuthenticationFailureHandler</i> como responsável por lidar com falhas de autenticação.	58
Figura 28 – Configurando um servidor LDAP em memória e carregando os dados do arquivo <i>users.ldif</i>	59
Figura 29 – Configurando um servidor LDAP através de um serviço já existente acessado pela URL <i>springframework.org</i> , com o elemento raiz <i>com.security</i>	59
Figura 30 – Configurando um <i>AuthenticationProvider</i> que utiliza o servidor LDAP para efetuar controle de acesso.	59
Figura 31 – Componentes criados para prover controle de acesso com LDAP.	60
Figura 32 – Configurando autenticação via LDAP e autorização via SGBD.....	62
Figura 33 – Declaração de um <i>CasAuthenticationEntryPoint</i> e de um <i>ServiceProperties</i>	64
Figura 34 – Informando que o novo <i>entry-point</i> será <i>casAuthEntryPoint</i> (instância de <i>CasAuthenticationEntryPoint</i>).....	65
Figura 35 – Inserção de um <i>filter</i> na posição <i>CAS_FILTER</i>	65
Figura 36 – Criação do <i>filter CasAuthenticationFilter</i> , populando sua propriedade <i>authenticationManager</i>	65
Figura 37 – Configurando um <i>CasAuthenticationProvider</i> como um dos <i>authentication providers</i> da aplicação.	66
Figura 38 – Declaração de um componente <i>CasAuthenticationProvider</i>	67

LISTA DE ABREVIATURAS E SIGLAS

ACL	–	<i>Access Control List</i>
API	–	<i>Application Programming Interface</i>
AOP	–	<i>Aspect-Oriented Programming</i>
CA	–	<i>Certificate Authority</i>
CAS	–	<i>Central Authentication Service</i>
DI	–	<i>Dependency Injection</i>
DIT	–	<i>Directory Information Tree</i>
HTTP	–	<i>Hypertext Transfer Protocol</i>
IOC	–	<i>Inversion of Control</i>
JEE	–	<i>Java Enterprise Edition</i>
LDAP	–	<i>Lightweight Directory Access Protocol</i>
MVC	–	<i>Model-View-Controller</i>
OTP	–	<i>One-Time Password</i>
SQL	–	<i>Structured Query Language</i>
SSL	–	<i>Secure Sockets Layer</i>
SSO	–	<i>Single Sign On</i>
TLS	–	<i>Transport Layer Security</i>
URI	–	<i>Uniform Resource Identifier</i>
URL	–	<i>Uniform Resource Locator</i>
XML	–	<i>Extensible Markup Language</i>

SUMÁRIO

1	INTRODUÇÃO	10
1.1	ESTRUTURA DO TRABALHO	12
2	SEGURANÇA DE SISTEMAS CORPORATIVOS	14
2.1	FUNDAMENTOS DE SEGURANÇA	14
2.1.1	Autenticação	15
2.1.1.1	Algo que o usuário sabe	15
2.1.1.2	Algo que o usuário possui	15
2.1.1.3	Algo que o usuário é	16
2.1.2	Autorização	16
2.1.2.1	<i>Access Control Lists</i>	17
2.1.3	Autenticidade	18
2.1.4	Confidencialidade	18
2.1.5	Integridade	19
2.1.6	Disponibilidade	20
2.1.7	Não-repúdio	20
2.1.8	Ativo	21
2.1.9	Ameaça	21
2.1.10	Vulnerabilidade	21
2.2	AMEAÇAS À SEGURANÇA DE SISTEMAS CORPORATIVOS	22
2.2.1	Principais vulnerabilidades em <i>software</i>	22
2.2.1.1	<i>Buffer overflow</i>	23
2.2.1.2	Ataques de validação de entrada	23
2.2.1.3	Ataques de autenticação	26
2.2.1.4	Ataques de autorização	26
2.3	TECNOLOGIAS DE SEGURANÇA EM SOFTWARE	26
2.3.1	Criptografia	27
2.3.1.1	Certificados	28
2.3.1.2	TLS	29
2.3.2	OpenID	30
2.3.3	OAuth	32
2.3.4	Serviço de diretório	34
2.3.5	<i>Single Sign On</i>	35

2.3.6	Segurança JEE	36
2.4	SPRING SECURITY.....	38
2.4.1	Spring Framework	43
3	APLICAÇÕES DE SEGURANÇA COM SPRING SECURITY	48
3.1	CONFIGURAÇÃO DA BASE DE DADOS DE CREDENCIAIS.....	49
3.2	IMPLEMENTAÇÃO DE SEGURANÇA NA <i>SERVICE LAYER</i>	52
3.2.1	Controle de acesso com <i>pointcuts</i>	53
3.2.2	Controle de acesso com <i>annotations</i>	53
3.3	AUTENTICAÇÃO COM OPENID	56
3.4	CONTROLE DE ACESSO COM LDAP	58
3.5	AUTENTICAÇÃO EM UM SERVIÇO <i>SINGLE SIGN ON</i> COM <i>CENTRAL AUTHENTICATION SERVICE</i>	63
3.5.1	Criação do <i>AuthenticationEntryPoint</i>.....	64
3.5.2	Declaração do <i>filter</i> e do <i>AuthenticationProvider</i>	65
3.6	UTILIZANDO OS CÓDIGOS DE EXEMPLO	68
4	CONSIDERAÇÕES FINAIS	69
	REFERÊNCIAS.....	71
	APÊNDICE A	75

1 INTRODUÇÃO

Os problemas atuais de falhas de segurança em *software* estão cada vez mais expostos e explorados. Segundo McGraw et al (2004, tradução nossa), essas falhas custam bilhões por ano à indústria e governo dos EUA. Esses fatores de risco tornam a segurança uma das principais preocupações de desenvolvedores de *software*. (KALALI, 2010, tradução nossa). Os riscos aumentam consideravelmente em ambientes distribuídos, pois os ataques podem vir de um número desconhecido de fontes. Os riscos em sistemas cuja segurança é comprometida vão desde prejuízos financeiros por fraudes ou de perda de informações estratégicas até dados de clientes, prejudicando a reputação de uma empresa perante seu mercado-alvo.

Em abril de 2011, a multinacional Sony foi forçada a indisponibilizar seu serviço de jogos online, a PlayStation Network, devido a ataques realizados em seus sistemas onde informações pessoais de 77 milhões de clientes foram roubadas (SONY COMPUTER ENTERTAINMENT, 2011, tradução nossa), sendo um dos maiores casos de violação de segurança da história até o momento (CBC NEWS, 2011, tradução nossa). O serviço PlayStation Network ficou indisponível por várias semanas, gerando prejuízos financeiros de 171 milhões de dólares (PC MAGAZINE, 2011, tradução nossa). A empresa também foi questionada pelo governo sobre as precauções tomadas para a proteção das informações dos usuários, sob pena de multa (EUROGAMER, 2011, tradução nossa). Sony também sofre ações judiciais por parte de clientes que argumentam sobre sua falha em proteger os dados roubados (JUSTIA, 2011, tradução nossa). Devido ao ataque e às ações tomadas pela Sony para lidar com o mesmo, a reputação da empresa sofreu forte queda após o ocorrido (FINANCIAL TIMES, 2011, tradução nossa).

Com a evolução da internet e da capacidade de processamento dos computadores, há um rápido crescimento na complexidade dos sistemas de informação computadorizados. Tais sistemas, ainda em suas fases iniciais, já são formados por inúmeros componentes interligados e interdependentes, gerando complexidade nas tarefas de planejamento, desenvolvimento e manutenção, que comportam o ciclo de vida dos projetos de *software*. No intuito de gerenciar essa complexidade, foram desenvolvidas práticas de arquitetura de *software* que classificam componentes de um sistema, cada um possuindo uma funcionalidade

específica (WALLS, 2011, tradução nossa). Alguns componentes como auditoria, gerenciamento de transações e segurança são interesses que transcendem os demais objetivos de um sistema, tornando-os distinguíveis das demais funcionalidades do *software* (WALLS, 2011, tradução nossa).

Nesse cenário, surgem *frameworks* que implementam tecnologias de apoio a segurança e confiabilidade de *software*, no intuito de serem utilizados por sistemas finais. Dentre essas tecnologias implementadas pelos *frameworks*, podemos citar: criptografia, que é, segundo Daswani, Kern e Kesavan (2007, tradução nossa), o estudo de como codificar e decodificar mensagens; OAuth, que provê uma forma de clientes acessarem recursos em nome de seus proprietários (HAMMER-LAHAV, 2010, tradução nossa); *Single Sign-On*, uma tecnologia que provê controle de acesso centralizado e autenticação a recursos baseados na web [...] (MULARIEN, 2011, tradução nossa); ACL (*Access Control Lists*), que provê um modelo de autorização de objetos de domínio a nível de instância (MULARIEN, 2011, tradução nossa); SSL (*Secure Sockets Layer*), um protocolo usado para fornecer segurança a nível de transporte HTTP na internet (MULARIEN, 2011, tradução nossa), entre vários outros mecanismos e técnicas.

O Spring Security (<http://static.springsource.org/spring-security>) é um *framework* poderoso e altamente extensível de autenticação e controle de acesso (SPRING SOURCE, 2008, tradução nossa). É escrito em Java e licenciado com a *apache license v2.0*. O *framework* é utilizado hoje em dia em vários ambientes incluindo agências governamentais, aplicações militares e bancos (SPRING SOURCE, 2008, tradução nossa). O Spring Security [...] oferece [...] integração com os mais comuns sistemas de autenticação corporativos; sendo adaptável à maioria das situações com pouco esforço por parte do desenvolvedor (MULARIEN, 2011, tradução nossa).

Poucos são os *frameworks* que oferecem suporte a todos os recursos de segurança em nível de *software* necessários a uma aplicação, obrigando os desenvolvedores a lançarem mão de múltiplos *frameworks* para que a demanda de segurança de um projeto de *software* possa ser atendida, aumentando os riscos e os custos de integração e manutenção. Spring Security é uma ferramenta que integra, dá suporte e é distribuído com “tudo que você precisa para implementar uma

solução de segurança [...] em uma forma concisa e sensata.” (MULARIEN, 2011, tradução nossa).

Este trabalho busca realizar um estudo sobre como Spring Security implementa alguns dos principais mecanismos atuais de segurança de *software* corporativos, e como essas implementações podem ser usadas para fornecer segurança em todas as camadas de projetos de *software* corporativos.

1.1 ESTRUTURA DO TRABALHO

Este trabalho foi estruturado de forma a apresentar os principais conceitos de segurança de *software*, como também detalhar algumas das principais técnicas de controle de acesso e, em seguida, apresentar as implementações de mecanismos que suportem essas técnicas, utilizando Spring Security.

A seção 2 descreve a base teórica necessária à realização deste trabalho. Inicia-se com a definição de segurança assumida para este trabalho, e suas classificações. Em 2.1 são apresentados os elementos do estudo da segurança, a saber: autenticação (ver 2.1.1), autorização (ver 2.1.2), autenticidade (ver 2.1.3), confidencialidade (ver 2.1.4), integridade (ver 2.1.5), disponibilidade (ver 2.1.6), não-repúdio (ver 2.1.7), ativo (ver 2.1.8), ameaça (ver 2.1.9) e vulnerabilidade (ver 2.1.10). Em 2.2 são expostos os tipos de ataques e as principais vulnerabilidades em *software* (2.2.1). Em 2.3 são listadas algumas das principais tecnologias que proporcionam segurança em *software*. A seção 2.3.1 expõe os conceitos de criptografia. Em seguida, em 2.3.2, é apresentada a tecnologia OpenID. Em 2.3.3 é feito um comentário sobre o protocolo de autorização OAuth. Dando continuidade, a seção 2.3.4 faz um breve comentário sobre Serviços de Diretório e a tecnologia LDAP. Logo após, em 2.3.5, temos uma introdução a *Single Sign On* e o protocolo CAS. Em seguida (2.3.6) é feita uma breve explanação de como a tecnologia JEE provê segurança de forma nativa. Finalmente, a seção 2.4 traz uma introdução a tecnologia Spring Security, expondo seus principais recursos, componentes, e demonstrando um cenário de uso básico; finalizando com a visão geral do Spring Framework, suas características e objetivos (ver 2.4.1).

Em seguida, a seção 3 tem como objetivo descrever as etapas realizadas para a implementação de alguns recursos de segurança em *software* com a

tecnologia Spring Security, descrevendo o propósito dos componentes envolvidos. A seção tem início com a justificativa das escolhas dos mecanismos de segurança elencados para este trabalho. 3.1 explica o processo de configuração de uma base de dados de credenciais. Em seguida, em 3.2 temos um detalhamento de como prover segurança à *Service Layer* com Spring Security. Após isso, em 3.3, explica-se a configuração de um sistema para prover segurança por um *provider* OpenID, através do Spring Security. Já 3.4 ilustra como o *framework* Spring Security pode ser utilizado para prover controle de acesso com LDAP. Por fim, detalha-se o funcionamento de alto nível da tecnologia *Central Authentication Service* e como utilizá-la como serviço externo de autenticação em sistemas que utilizam Spring Security (3.5).

A seção 4 possui considerações finais, propostas de trabalhos futuros e resultados alcançados por este trabalho.

2 SEGURANÇA DE SISTEMAS CORPORATIVOS

McGraw (2006, tradução nossa), define segurança em *software* como “a idéia de construir *software* com uma engenharia que o permita continuar a funcionar corretamente sob ataques maliciosos”. No âmbito da engenharia de segurança em sistemas, Anderson (2004, tradução nossa) define:

Engenharia de segurança é construir sistemas para que permaneçam confiáveis em face de erros [...]. Como disciplina, foca nas ferramentas, processos e métodos necessários para projetar, implementar e testar sistemas, e na adaptação de sistemas existentes conforme seus ambientes evoluem. (ANDERSON, 2004, p. 3, tradução nossa).

Entende-se que a definição de McGraw (2006, tradução nossa) aborda o que diz respeito a *software* na engenharia de segurança. Esta é a definição que será assumida neste trabalho.

Segundo Daswani, Kern e Kesavan (2007, tradução nossa), podemos classificar a segurança de sistemas em três aspectos: segurança física, segurança tecnológica e boas práticas e princípios. Segurança física inclui proteção e monitoramento físico de *data centers* e uso de mecanismos de identificação dos administradores. Segurança tecnológica engloba os princípios de segurança na aplicação, na rede e no sistema operacional. Finalmente, as pessoas que interagem com os sistemas devem ser orientados a obedecerem a políticas de segurança, como a não transferência de senhas e de papéis. Em aplicações corporativas, as três áreas devem ser exploradas de forma holística para que se obtenham sistemas seguros. Aplicar princípios de segurança somente em nível de *software*, por exemplo, não garante uma aplicação segura.

2.1 FUNDAMENTOS DE SEGURANÇA

Meier et al (2003, tradução nossa) define seis elementos fundamentais no estudo da segurança: autenticação, autorização, autenticidade, confidencialidade, integridade e disponibilidade. Daswani, Kern e Kesavan (2007, tradução nossa) ainda acrescentam o elemento não-repúdio. Sistemas devem ser estruturados proativamente, de forma a atingirem esses objetivos.

2.1.1 Autenticação

Autenticação é o ato de verificar a identidade de alguém (DASWANI; KERN; KESAVAN, 2007, tradução nossa). Dessa forma, identificando de forma única um agente que interage em uma aplicação, podendo ser, em termos de *software*, “usuários finais, outros serviços, processos ou computadores” (MEIER et al 2003, p. 4, tradução nossa).

A autenticação é realizada comumente por uma combinação de três métodos: a verificação de algo que o usuário é, possui ou sabe.

2.1.1.1 Algo que o usuário sabe

A forma comum e elementar de atingir esse método é com um segredo que somente o usuário saiba, como uma senha. Esse método pode ser considerado como o mais simples de implementar, em comparação com os demais.

Daswani, Kern e Kesavan (2007, tradução nossa) falam sobre o sistema de *one-time password* (OTP), o qual força o usuário a fornecer diferentes senhas em uma lista pré-determinada. A desvantagem desse sistema é a dificuldade de memorização por parte de seres humanos.

2.1.1.2 Algo que o usuário possui

Métodos baseados em algo que o usuário possui são mais difíceis de implementar que o anterior, porém podem oferecer maior garantia de segurança. O uso de *OTP Cards* é uma técnica comum de uso deste método. Consiste em um aparelho que gera uma combinação de caracteres baseada em um algoritmo conhecido pela fonte. Quando o usuário se autentica com sua senha padrão, ele é então solicitado a inserir o número gerado pelo aparelho. Variações desse sistema incluem comunicação direta do dispositivo com o servidor de autenticação, onde o dispositivo sincroniza o número gerado com o servidor, fazendo este só aceitar esse número no processo de autenticação.

Daswani, Kern e Kesavan (2007, tradução nossa) também citam *Smart Cards* como mecanismo de autenticação baseado em algo que o usuário possui. Um *Smart*

Card possui microprocessador e memória, permitindo o processamento da autenticação seja realizado pelo próprio cartão, através de um dispositivo comunicador (um leitor de cartões) que possa receber entrada de usuário (sua senha, por exemplo).

2.1.1.3 Algo que o usuário é

O terceiro e último método de autenticação consiste em identificar unicamente um usuário por uma característica única dele. Segundo Daswani, Kern e Kesavan (2007, tradução nossa), a maioria das técnicas de autenticação que se classificam nessa categoria são técnicas biométricas. Reid (2003, tradução nossa) define biometria como uma tecnologia que usa um traço físico ou psicológico para identificação e/ou autenticação. Técnicas como análise da mão ou da íris, verificação de impressões digitais e reconhecimento de voz para identificar unicamente uma pessoa são exemplos de mecanismos de autenticação baseados no que o usuário é.

Existem barreiras de aceitação sociais com relação a essas técnicas. Segundo Daswani, Kern e Kesavan (2007, tradução nossa), técnicas mais simples como informar uma senha são mais aceitas pelos usuários do que se submeterem a análise manual, por exemplo.

2.1.2 Autorização

Enquanto autenticação foca em determinar a identidade de alguém, autorização é o processo de “verificar se um usuário possui permissão de realizar determinada ação” (DASWANI; KERN; KESAVAN 2007, p. 12, tradução nossa), determinando assim sua autoridade. Segundo Meier et al (2003, tradução nossa), um sistema usa autorização para definir o conjunto de recursos e operações nos quais um usuário possui acesso. Recursos podem ser arquivos, banco de dados, informações, entradas de registros, entre outros.

Normalmente quando um agente requisita um recurso em um sistema, primeiramente é feita a operação de autenticação. Caso essa operação ocorra com sucesso, significa que este agente foi reconhecido pelo sistema como um usuário

existente e válido. Em seguida é feito o processo de autorização, verificado se este usuário possui privilégios suficientes para proceder com sua requisição ao recurso.

2.1.2.1 Access Control Lists

Uma técnica conhecida de aplicação do conceito de autorização são as *Access Control Lists* (ACLs), que representam um conjunto de usuários e um conjunto de recursos correspondentes nos quais eles têm autoridade de acesso (DASWANI; KERN; KESAVAN 2007, tradução nossa). Segundo Andress (2011, tradução nossa), ACLs são construídas em cima de um recurso específico, contendo os identificadores do grupo que possui permissão de acessar o recurso em questão e o que o grupo tem permissão de fazer com o recurso. Essa técnica de autorização é amplamente utilizada em acesso a sistemas de arquivos. Um exemplo de uma ACL pode ser verificado na Figura 1:

Figura 1 – *Access Control List*.

Usuário	Recurso	Privilégios
Luke	file1.txt	Ler, Escrever
Solo	file1.txt	Ler

Fonte: Elaborado pelo autor (2012)

No esquema de autorizações exemplificado na Figura 1, o usuário “Luke” pode executar as ações “Ler” e “Escrever” no recurso “file1.txt” (neste exemplo, o recurso é um arquivo), já o usuário “Solo” somente pode executar, neste recurso, a ação “Ler”.

ACLs são utilizadas para implementar três modelos de controles de acesso, como define Andress (2011, tradução nossa):

- a) *Mandatory Access Control* (MAC): onde o sistema decide pelos privilégios dos recursos, não importa por qual usuário este recurso foi criado. Neste modelo, mesmo que um recurso seja criado por determinado usuário, o sistema pode definir que este recurso não pode ser acessado pelo usuário criador;
- b) *Discretionary Access Control* (DAC): este modelo é semelhante ao MAC, com a exceção de que os usuários são autorizados a determinar quais usuários podem acessar os recursos criados por estes. Este é, segundo

Daswani, Kern e Kesavan (2007, tradução nossa), o modelo utilizado para gerenciar arquivos pelos sistemas operacionais baseados em UNIX, onde os arquivos representam os recursos, e os privilégios são *Ler*, *Escrever* e *Executar*;

- c) *Role-Based Access Control* (RBAC): o modelo RBAC também é semelhante ao modelo MAC, porém o sistema se baseia nos papéis dos usuários para realizar as decisões de autorização. Neste modelo, as autorizações não são concedidas a usuários individuais, mas sim a grupos de usuários que possuem um papel específico. Por exemplo, os usuários administradores de um sistema podem ter acesso de leitura aos relatórios gerenciais do mesmo, enquanto que usuários sem o papel de administrador podem não possuir tal privilégio.

2.1.3 Autenticidade

Autenticidade é a habilidade de identificar quem realizou determinada ação em um sistema, seja um invasor ou um usuário comum que efetuando operações normais (DASWANI; KERN; KESAVAN, 2007, tradução nossa). Um sistema que possui autenticidade garante que os eventos que criaram ou transformaram os dados possuem donos ou responsáveis.

Daswani, Kern e Kesavan (2007, tradução nossa) ainda afirmam que a maioria dos sistemas garante autenticidade através de autenticação e auditoria. Este último refere-se ao processo de gravar informações sobre os eventos ocorridos nos sistemas, como operações de autenticação ou ações realizadas pelos usuários, como a efetivação de uma compra em um sistema *e-commerce*. As informações gravadas pelo mecanismo de auditoria não devem ser modificadas por agentes externos, e devem possuir de forma precisa o instante exato que determinado evento ocorreu.

2.1.4 Confidencialidade

Confidencialidade é o processo de garantir que dados permaneçam privados e confidenciais, e que não possam ser visualizados por usuários sem autorização

(MEIER et al 2003, tradução nossa), tendo como objetivo manter os dados temporários e persistentes em segredo. De acordo com Andress (2011, tradução nossa), confidencialidade é um conceito que pode ser implementado em vários níveis de um processo. Ou seja, em segurança de sistemas corporativos, podemos aplicar confidencialidade nos processos de transferência de dados na rede, onde um agente interceptador pode capturar o fluxo de dados e ter acesso à informação; também se pode aplicar confidencialidade nas informações persistentes armazenadas em servidores, no intuito de proteger essas informações em caso de acesso ilícito.

Segundo Daswani, Kern e Kesavan (2007, tradução nossa), uma das formas mais comuns de se alcançar confidencialidade em sistemas é através de criptografia: em um processo de transferência de dados na rede, as mensagens são embaralhadas com base em uma chave onde somente os sistemas remetente e receptor das mensagens conhecem. Somente com a chave é possível reordenar os dados das mensagens para que se transformem novamente nas informações a serem trocadas. Se as mensagens criptografadas forem interceptadas durante o processo de transferência, o interceptador não será capaz de entender seu conteúdo. O mesmo conceito se aplica para dados persistentes gravados em disco. Se os dados forem capturados e estiverem criptografados, o sistema ou usuário que capturou estes dados não serão capazes de extrair as informações destes.

2.1.5 Integridade

Integridade se refere à prevenção de alterações de dados sem autorização, ou de forma inesperada. Um agente que quebra a confidencialidade de um sistema pode também ter adquirido privilégios de modificação destes dados, assim como um usuário perfeitamente autorizado, pode realizar uma operação indesejada, alterando de forma errônea ou destruindo dados (ANDRESS, 2011, tradução nossa). Da mesma forma, eventos inesperados como corrupção de dados por mau funcionamento das unidades de armazenamento ou causados por acidentes naturais são fatores que contribuem para a quebra de integridade. De posse disso, entende-se que para manter a integridade da informação, além da necessidade de se criar

meios de proteger os dados de acessos não autorizados, também é necessário garantir a capacidade de reverter eventuais mudanças indesejadas nos mesmos.

Daswani, Kern e Kesavan (2007, tradução nossa) introduzem os testes de integridade para identificar alterações de informações em sistemas. Um deles diz respeito à adição de informações redundantes aos dados. Através da redundância, aumentam-se as chances de identificar que determinados dados sendo manipulados foram alterados de forma inesperada. A principal desvantagem desse método é o aumento do fluxo de dados transmitidos e armazenados, devido a adição das informações de redundância.

2.1.6 Disponibilidade

Trata-se de manter o sistema disponível para seus usuários. Sistemas corporativos disponibilizam e guardam informações onde o funcionamento das organizações depende da manipulação dessas. Sistemas distribuídos estão a mercê de ataques externos que tem por objetivo torná-los indisponíveis a seus usuários legítimos. Daswani, Kern e Kesavan (2007, tradução nossa) citam a dificuldade de prover sistemas que sejam ao mesmo tempo altamente seguros e disponíveis todo o tempo. O exemplo mais clássico disso ilustra um computador desconectado da internet e guardado fisicamente em um local seguro onde ninguém tem acesso a ele. Apesar do aumento considerável de segurança, este computador não estará disponível para uso. Em alguns sistemas, como os bancários, disponibilidade e segurança são fatores críticos e determinantes.

Uma das formas de se garantir disponibilidade em sistemas corporativos distribuídos é a introdução de redundância desses sistemas em diferentes *data centers*. Caso um ponto de conexão se torne indisponível, os usuários são automaticamente direcionados para o próximo ponto de reserva disponível, sem impacto considerável à disponibilidade da informação.

2.1.7 Não-repúdio

Daswani, Kern e Kesavan (2007, tradução nossa) definem que o objetivo do não-repúdio é garantir a incontestabilidade de uma transação por todas as partes

envolvidas. O uso de terceiros confiáveis no intuito de auxiliar na condução de transações é uma das formas de atingir não-repúdio. Nessa abordagem, as transações são certificadas por um agente terceiro, fazendo parte do processo transacional. Esse agente pode garantir, posteriormente, se determinado evento ocorreu e em quais condições, garantindo o não-repúdio por todas as partes envolvidas. Em situações reais, são utilizados protocolos de não-repúdio durante troca de informações, agindo como terceiros confiáveis.

2.1.8 Ativo

Ativos, no âmbito da segurança de sistemas corporativos, são recursos de uma organização que devem ser protegidos. De acordo com Meier et al (2003, tradução nossa), ativos podem ser itens tangíveis, como uma página web ou um banco de dados, ou podem ser menos tangíveis, como a reputação da organização. Podemos considerar um dos ativos mais importantes dos sistemas corporativos como sendo a própria informação que ele gera e manipula.

2.1.9 Ameaça

Ameaça é algo que tem potencial de causar danos aos ativos (ANDRESS, 2011, tradução nossa). Normalmente ameaças são direcionadas a tipos específicos de ativos, e muitas vezes só existem em determinados ambientes. Por exemplo, uma ameaça a um determinado algoritmo de criptografia pode não ter o mínimo efeito em outro algoritmo. Daswani, Kern e Kesavan (2007, tradução nossa) lembram que o entendimento das ameaças reais é importante na determinação dos objetivos de segurança dos sistemas.

2.1.10 Vulnerabilidade

Vulnerabilidade nos sistemas corporativos são propriedades do *software* exploradas pelas ameaças para causar algum dano nos ativos (ANDERSON 2004, tradução nossa). A falta da aplicação de técnicas de segurança é um dos principais fatores da existência de vulnerabilidades. Outras vulnerabilidades fogem total ou

parcialmente ao controle da segurança em *software*, como fatores naturais e vulnerabilidades de *hardware* ou rede (ANDRESS 2011, tradução nossa).

Ye (2008, tradução nossa) explica a relação entre ativo, ameaça e vulnerabilidade na geração de risco ao *software*. O risco é formado quando existe a possibilidade de uma ameaça explorar uma vulnerabilidade de um ativo. Se uma mudança nas propriedades do sistema elimina ou modifica as características de uma vulnerabilidade, uma ameaça pode passar a não mais causar riscos ao ativo.

2.2 AMEAÇAS À SEGURANÇA DE SISTEMAS CORPORATIVOS

Andress (2011, tradução nossa) divide os tipos de ataques a sistemas em quatro categorias:

- a) Interceptação: são ataques onde usuários não autorizados possuem acesso a dados e informações, independente do meio utilizado. Ataques desse tipo são considerados ataques contra confidencialidade;
- b) Interrupção: ataques de interrupção tornam recursos indisponíveis de forma temporária ou permanente, causando danos à disponibilidade, podendo se estender a integridade, caso a interrupção cause corrupção nos dados;
- c) Modificação: envolve adulteração em ativos, causando danos diretamente à integridade dos dados. Ataques deste tipo podem também ameaçar a disponibilidade de um sistema corporativo, dependendo da forma que os dados sejam alterados;
- d) Fabricação: esses ataques também são ameaças à integridade. Consistem na geração e propagação de dados falsos na rede, na forma de informação, recursos ou serviços. Ataques desse tipo exploram vulnerabilidades em controle de acesso, autenticação e autorização.

2.2.1 Principais vulnerabilidades em *software*

Essa sessão procura descrever algumas vulnerabilidades de alto nível comuns na construção de *softwares* que podem levar a riscos de segurança em sistemas corporativos.

2.2.1.1 Buffer overflow

Um *buffer* é uma área de memória utilizada para armazenar entradas de dados da aplicação (DASWANI, KERN e KESAVAN 2007, tradução nossa). Normalmente *buffers* possuem uma capacidade máxima de dados definida.

Buffer overflow ocorre quando não se leva em conta a quantidade de dados que podem ser inseridos em um *buffer*, permitindo que usuários insiram mais dados que sua capacidade. Esses dados extras serão então inseridos em áreas de memória não previstas pelo projeto do *software*. De acordo com Anderson (2004, tradução nossa), esses dados que excedem o tamanho do *buffer* podem desviar o fluxo de execução da aplicação, ou até mesmo executar código malicioso.

Daswani, Kern e Kesavan (2007, tradução nossa) ilustram um exemplo de ataque de *buffer overflow* para burlar a pilha de fluxo execução de um programa. No exemplo, uma função *A* executa uma função *B*. *B* espera entrada de dados do usuário e guarda essa entrada em *buffer* representado por um *array*. Dependendo do retorno de *B*, *A* irá executar uma terceira função: *C*, que deve ser protegida. Se o usuário inserir mais dados que a capacidade do *array*, os dados excedentes irão sobrescrever o endereço de retorno da função *B* (função *A*), podendo inclusive desviar a execução diretamente para *C*, independente do que *B* retornou para *A*.

Para evitar a escrita de código que permita ataques de *buffer overflow*, devem ser feitas verificações dos limites dos *buffers*, não permitindo que dados excedentes sejam escritos em áreas de memória. De acordo com Andress (2011, tradução nossa), algumas linguagens automaticamente fazem essa checagem.

2.2.1.2 Ataques de validação de entrada

São ataques onde dados de entrada são inseridos no sistema no intuito de induzi-lo a realizar um comportamento diferente do esperado. Um dos ataques mais comuns dessa categoria trata-se de ataques de *SQL Injection*.

SQL (Structured Query Language) é a linguagem padrão de manipulação de sistema de banco de dados relacionais comerciais (ELMASRI e NAVATHE 2011, tradução nossa). Ataques de *SQL injection* usam entradas de dados em sistemas que utilizam essas entradas para executarem comandos em um banco de dados de

forma não confiável, sem efetuar validações nas mesmas (DASWANI, KERN e KESAVAN 2007, tradução nossa).

Em sistemas corporativos que manipulam informações em forma de dados guardados em banco de dados relacionais, as ações que os usuários realizam nesses sistemas geralmente são convertidas em comandos SQL enviados aos bancos de dados.

Imagina-se uma situação onde o usuário deva inserir um código conhecido de um contrato como entrada de dados em um sistema, e o mesmo deve retornar informações sigilosas do referido contrato. O sistema então monta uma consulta SQL baseado no código que o usuário inseriu (Figura 2):

Figura 2 – Consulta SQL com a variável a ser substituída.

```
"SELECT informacoes
  FROM contrato
 WHERE codigo = codigo_entrada"
```

Fonte: Elaborado pelo autor (2012)

Neste exemplo o sistema substitui *codigo_entrada* pelo valor que o usuário inseriu. Se o usuário inseriu o valor 2398473, temos então (Figura 3):

Figura 3 – Consulta SQL pronta para execução.

```
"SELECT informacoes
  FROM contrato
 WHERE codigo = 2398473"
```

Fonte: Elaborado pelo autor (2012)

Percebe-se que o sistema hipotético não está realizando nenhuma validação dos dados inseridos pelos usuários antes de concatená-los ao comando SQL a ser executado, tornando essa operação vulnerável a entradas de dados maliciosas. Nota-se que é possível inserir, inclusive, comandos SQL como código de entrada. Imagina-se que um usuário com o intuito de atacar o sistema entrou como código a sequência *0 OR 0=0*. O sistema então irá gerar a consulta ilustrada na Figura 4:

Figura 4 – Consulta SQL alterada por *SQL Injection*.

```
"SELECT informacoes
  FROM contrato
 WHERE codigo = 0 OR 0=0"
```

Fonte: Elaborado pelo autor (2012)

A Figura 4 ilustra uma consulta onde a cláusula *WHERE* foi dividida em uma operação lógica avaliando duas expressões com o operador OR. De acordo com Churcher (2008, tradução nossa), quando é utilizado o operador *OR* entre duas expressões, a operação lógica retornará verdadeiro se pelo menos uma das expressões for verdadeira. Assim, o usuário com intenções de roubar informações do sistema não precisa conhecer um código de contrato válido, contanto que a segunda expressão seja verdade. Utilizando a expressão $0=0$ o usuário garante que o sistema irá avaliar verdadeiro para todas as linhas consultadas, trazendo as informações de todos os contratos existentes no sistema.

O ataque de *SQL injection* exemplificado poderia ser evitado pela simples validação dos dados de entrada: o sistema poderia só aceitar valores inteiros como código para buscas. Daswani, Kern e Kesavan (2007, tradução nossa) ilustram outras técnicas para evitar *SQL injection*:

- a) *Blacklists*: consiste na criação de uma lista de caracteres que serão removidos automaticamente das entradas dos usuários. Assim, meta caracteres, como aspas, os quais o banco de dados interpreta como partes do comando SQL são removidos antes de dar continuidade com a operação;
- b) *Whitelists*: cria-se uma lista de caracteres permitidos como entradas de usuários. Se o usuário inserir caracteres que não estejam nessa lista, eles são removidos da entrada do usuário e em seguida a ação prossegue sem os caracteres não permitidos;
- c) *Escaping*: esta técnica implica na criação de um mecanismo que substitui seqüências de caracteres perigosas em seqüências benignas, assim nenhuma entrada de usuário é removida;
- d) *Prepared statements*: mecanismo utilizado por alguns sistemas de banco de dados que garante a distinção entre dados e controle através de *bind variables*. O *statement*, ou comando SQL, é inicialmente compilado e a execução ocorre como um segundo passo, onde as variáveis são preenchidas e o *statement* é executado. Tudo que é inserido no segundo passo é tratado como dado.

2.2.1.3 Ataques de autenticação

São ataques que exploram o evento de autenticação, explorando a usurpação da identidade de um usuário.

No âmbito das autenticações baseadas em algo que o usuário sabe, geralmente realizadas por senhas, Andress (2011, tradução nossa) cita a aplicação de força mínima nas senhas aceitas pelos sistemas, como também regularidade de renovação das mesmas, para se evitar esse tipo de ataque.

O uso combinado de mais de um método de autenticação diminui as chances de ataques de autenticação (DASWANI, KERN e KESAVAN 2007, tradução nossa). Pode-se usar, por exemplo, autenticação em duas fases, onde a primeira é baseada em uma informação que o usuário conhece, como sua senha pessoal, e a segunda é feita com um código gerado por um *OTP card*.

2.2.1.4 Ataques de autorização

Eventos de liberação de recursos para usuários que não possuem privilégios para acessar o mesmo se enquadram nessa categoria. Normalmente acontecem ao se explorar falta de design apropriado dos mecanismos de segurança de *software* e falhas humanas em autorizar usuários indevidamente (ANDRESS 2011, tradução nossa). O uso de técnicas apropriadas de autorização, como também uma forte política de controle dos eventos de liberação de autorizações em uma organização podem diminuir as ocorrências desse tipo de ataque.

2.3 TECNOLOGIAS DE SEGURANÇA EM SOFTWARE

No âmbito das vulnerabilidades existentes na construção e manutenção de sistemas corporativos, surgem mecanismos que auxiliam a manter um grau aceitável de confidencialidade, integridade e disponibilidade às informações manipuladas por estes sistemas. Tais mecanismos, segundo Daswani, Kern e Kesavan (2007, tradução nossa), devem ser projetados e construídos em paralelo com o projeto e construção do *software*, devido ao alto custo de se adicionar segurança a um sistema já construído.

Requisitos de segurança concretos e mensuráveis devem estar presentes no sistema a ser construído. McGraw (2006, tradução nossa) ilustra um conjunto de melhores práticas no campo da segurança e como aplicá-las nos artefatos gerados durante o ciclo de vida do *software*, categorizando objetivos ao longo dos níveis ou visões de um *software*, partindo da necessidade estratégica até a codificação, reforçando a importância da segurança em nível de requisitos.

Esta seção busca elencar algumas das principais tecnologias de segurança, que buscam atender requisitos de segurança existentes durante o ciclo de vida do *software*.

2.3.1 Criptografia

Criptografia é um ramo de estudo da criptologia que se refere à ação de transformar mensagens em uma derivação cifrada da mesma. SHIREY (2000, tradução nossa) define criptografia como:

Ciência matemática que lida com a transformação de dados para tornar seu significado ininteligível (ou seja, para esconder o seu conteúdo semântico), evitar a sua alteração não detectada ou impedir seu uso não autorizado. Se a transformação é reversível, criptografia também lida com a restauração de dados criptografados para sua forma inteligível. (SHIREY, 2000, p. 49, tradução nossa).

Talbot e Welsh (2006, tradução nossa) destacam a expansão do conceito original de criptografia, para aplicações atuais como protocolos de identificação e assinatura digital. No âmbito da segurança em *software*, criptografia busca proteger a confidencialidade dos dados, geralmente em operações que envolvem sua transferência. Assim como existem outras aplicações, como o uso em autenticação. (DASWANI, KERN e KESAVAN 2007, tradução nossa).

Uma função de criptografia é um algoritmo que recebe como entrada o texto original (*plaintext*) e uma chave (*key*) - usada no controle da operação de determinado algoritmo de criptografia - e produz como saída o texto criptografado, ou cifrado. A função que reverte o texto criptografado para o texto original recebe como entrada o texto criptografado e uma chave, produzindo como saída o *plaintext*. Um sistema criptográfico onde a mesma chave é utilizada para criptografar e

descriptografar é denominado de sistema de criptografia simétrica (TALBOT e WELSH 2006, tradução nossa).

Daswani, Kern e Kesavan (2007, tradução nossa) citam uma das principais desvantagens da criptografia simétrica: para se trocar informações criptografadas em um meio distribuído, as duas partes devem primeiramente acordar uma chave secreta - e compartilhada - utilizada para realizar a criptografia e, após a transferência dos dados, a descriptografia. Essa chave deverá ser acordada em um ambiente seguro, e geralmente é necessário contato físico para isso, aumentando os custos do processo. Nesse contexto surge a criptografia de chave pública ou criptografia assimétrica.

O modelo assimétrico utiliza duas chaves, uma chave privada e uma chave pública. A chave privada, secreta e não compartilhada, é utilizada como entrada na função que reverte o texto criptografado para o texto original. A chave pública, por sua vez, não é secreta e é publicada e utilizada como entrada para produzir texto cifrado. Todo dado criptografado com uma chave pública só poderá ser descriptografado pela chave privada equivalente. Dessa forma, não há necessidade de compartilhamento da chave privada (DASWANI, KERN e KESAVAN 2007, tradução nossa). Ainda sim é necessário que exista confiança na identidade da chave pública, ao realizar a operação de criptografia. A chave pública pode ser usurpada e, por consequência, a informação criptografada não poderá ser decifrada pela chave privada de seu destinatário.

2.3.1.1 Certificados

Para sanar o problema do modelo assimétrico exposto em 2.3.1, uma terceira parte, denominada *certificate authority* (CA), é inserida no processo. CA é uma entidade conhecida e confiável que atrela identidades a chaves públicas (DASWANI, KERN e KESAVAN 2007, tradução nossa). O resultado desse processo é um documento denominado certificado, que possui como conteúdo a identidade de uma pessoa ou instituição, sua chave pública e uma data de vencimento. Uma CA possui as responsabilidades de publicar e revogar os certificados, em caso de expiração, perda ou comprometimento de sua integridade. Logo, quando uma parte pretende criptografar uma mensagem para enviar para outra parte, a primeira irá solicitar a

uma CA o certificado da segunda, e irá utilizar este certificado como chave pública de criptografia. Quando a mensagem atingir seu destino, o destinatário irá utilizar sua chave privada para descriptografar a mensagem.

2.3.1.2 TLS

No âmbito da grande troca de informações em transações na web, surge o protocolo SSL (*Secure Sockets Layer*), que tem como motivação original prover segurança para as trocas de mensagens na internet. Inicialmente projetado pela Netscape em 1993, se tornando responsabilidade da organização internacional IETF em 1996 (THOMAS 2000, tradução nossa), SSL é usado como protocolo de autenticação, fornecendo confidencialidade e integridade durante transações entre clientes e servidores na *web* (DASWANI, KERN e KESAVAN, 2007, tradução nossa).

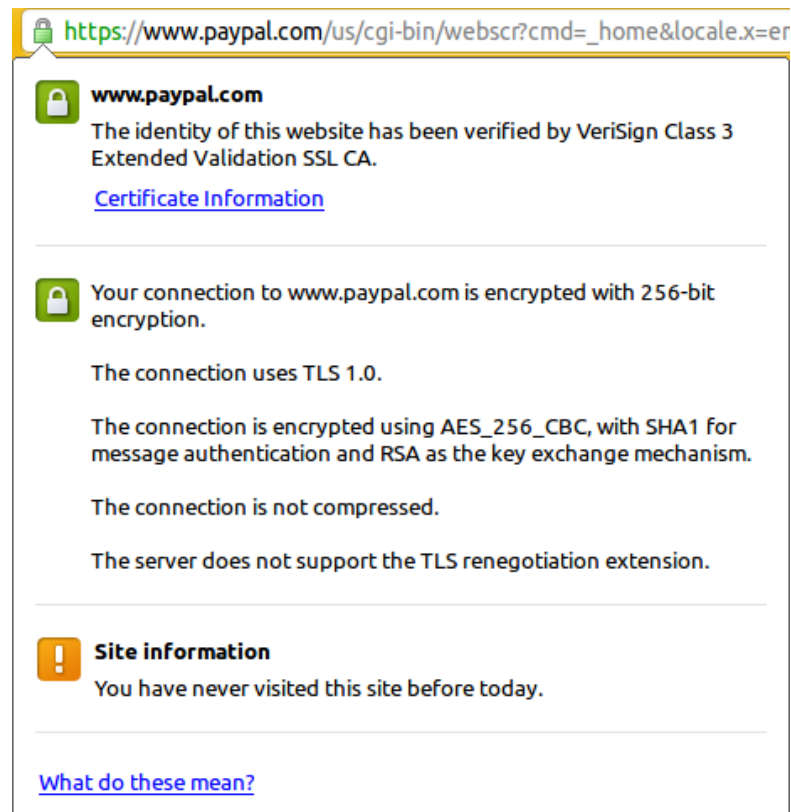
Em 1999, a primeira versão do protocolo TLS (*Transport Layer Security*) é publicada como uma evolução da versão corrente do SSL, sendo conhecido por TLS desde então (THOMAS, 2000, tradução nossa). A maioria dos clientes de internet mais populares atuais implementam a primeira versão do TLS, sendo que a especificação atualmente se encontra em sua versão 1.2 (DIERKS; RESCORLA, 2008, tradução nossa).

TLS utiliza criptografia simétrica e assimétrica em seus processos. Antes de iniciar a troca de dados, o cliente e o servidor devem realizar uma série de passos conhecidos como *handshake*. Um *handshake* completo com autenticação do servidor implica em duas trocas de mensagens. Inicialmente o cliente e o servidor trocam informações sobre as versões dos protocolos utilizados, assim como valores aleatórios que serão utilizados para computar as chaves. O servidor então envia para o cliente seu certificado, que realiza sua validação juntamente com a CA. Em seguida inicia-se a segunda troca de mensagens, onde o cliente gera um valor aleatório, criptografa esse valor utilizando a chave pública do servidor e o envia para o mesmo. O servidor descriptografa esse valor - com sua chave privada - e então, baseado nesse valor inicial e em outros valores aleatórios trocados no primeiro passo, o segredo é determinado e chaves que serão utilizadas para criptografia e autenticação são geradas a partir desse segredo, o servidor retorna uma

confirmação para o cliente, finalizando o processo de *handshake* (DASWANI, KERN e KESAVAN, 2007, tradução nossa).

Após a conclusão do *handshake*, cliente e servidor iniciam a conexão autenticada e confiável, onde os dados são criptografados e descriptografados com as chaves geradas. Quando um usuário de um cliente navegador quer utilizar SSL ou TLS ao acessar um *site*, utiliza-se o protocolo HTTP (*Hypertext Transfer Protocol*) como prefixo da *Uniform Resource Locator* (URL), por exemplo, <https://paypal.com>. O navegador irá então efetuar o *handshake* com o servidor e iniciar a conexão segura. Muitos deles indicam ao usuário que a identidade do *site* foi confirmada e que a conexão está estabelecida, como ilustrado na Figura 5. Percebe-se pela Figura 5 que o certificado do site foi validado pela CA VeriSign, e que a conexão foi estabelecida com TLS versão 1.0.

Figura 5 – Navegador exibindo informações sobre a conexão segura estabelecida com o site paypal.com.



Fonte: Elaborado pelo autor (2012)

2.3.2 OpenID

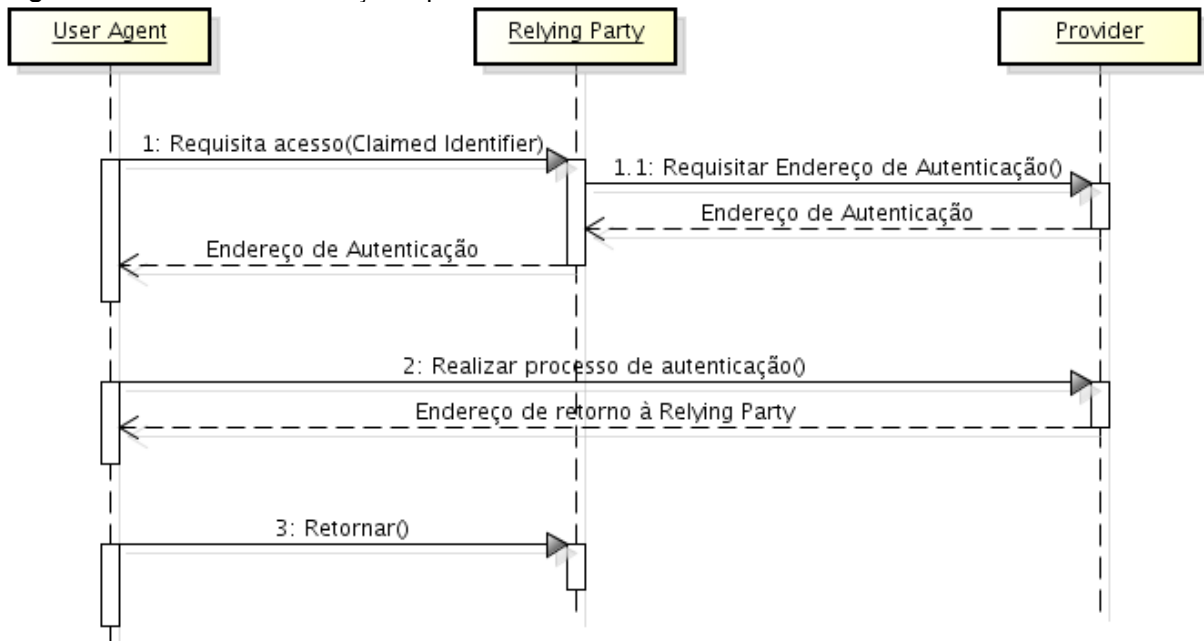
OpenID é um protocolo de autenticação aberto descentralizado, que surgiu em 2005 e vem adquirindo popularidade desde então como um dos principais padrões mundiais de autenticação descentralizada (OPENID FOUNDATION, 2009, tradução nossa). De acordo com OpenID Foundation (2007, tradução nossa):

A autenticação OpenID provê uma forma de provar que um usuário controla um *Identifier*. Isso é feito sem a necessidade da *Relying Party* acessar as credenciais do usuário como sua senha ou outras informações confidenciais como endereço de email. (OPENID FOUNDATION, 2007, p. 1, tradução nossa).

De acordo com OpenID Foundation (2007, tradução nossa) OpenID estende a terminologia padrão dos processos de autenticação com algumas terminologias, das quais destacam-se:

- a) *Identifier*: um *identifier* é um *Uniform Resource Identifier* (URI) escolhido pelo usuário como o seu identificador universal OpenID;
- b) *User-agent*: o *software* cliente pelo qual o usuário acessa o *provider*. Comumente trata-se de um navegador;
- c) *Relying party*: uma aplicação *web* que irá verificar se o usuário possui uma identidade;
- d) *OpenID provider*: uma entidade que provê o serviço OpenID, registrando URIs e fornecendo autenticações. A *relying party* irá se comunicar com o *provider* para validar identidades;
- e) *Claimed identifier*: o identificador que o usuário afirma possuir. Nota-se que o propósito final do protocolo é verificar tal afirmação.

Conforme observado na Figura 6, o fluxo de autenticação se inicia com o usuário apresentando seu *claimed identifier* à *relying party* via seu *user-agent*. A *relying party* então identifica o endereço do *provider* e requisita autenticação ao mesmo, direcionando o usuário para seu *provider*. O *provider* realiza a operação de autenticação com o usuário e redireciona seu *user-agent* de volta para a página da *relying party*, informando a mesma o resultado da tentativa de autenticação (OPENID FOUNDATION, 2007, tradução nossa).

Figura 6 – Fluxo de autenticação OpenID.

Fonte: Elaborado pelo autor (2012)

Percebe-se que nenhuma informação confidencial foi trocada entre o usuário e a *relying party*. A forma como o *provider* realiza a autenticação com o usuário está fora do escopo do protocolo (OPENID FOUNDATION, 2007, tradução nossa).

Várias entidades aderiram ao protocolo e se tornaram OpenID *providers*. Um mesmo usuário que possui identificador em vários *providers* poderá efetuar autenticação descentralizada utilizando qualquer uma de suas URIs através de OpenID. Em OpenID Foundation (2008, tradução nossa) encontra-se uma lista de 18 *providers* populares, já Wikipedia Contributors (2011, tradução nossa) identifica 27 *providers*.

2.3.3 OAuth

OAuth é um protocolo de autorização focado no acesso de clientes a recursos que pertencem a outrem (HAMMER-LAHAV, 2007, tradução nossa).

O protocolo permite que usuários autorizem terceiros a acessarem seus recursos sem a necessidade informar suas credenciais (HAMMER-LAHAV, 2010, tradução nossa).

A respeito do acesso a recursos dos usuários através de *software*, Hammer-Lahav (2010, tradução nossa) justifica:

No modelo tradicional de autenticação cliente-servidor, o cliente usa suas credenciais para acessar seus recursos hospedados pelo servidor. Com o aumento do uso de serviços distribuídos na web e cloud computing, aplicações terceiras requerem acesso a estes recursos hospedados no servidor. (HAMMER-LAHAV, 2010, p. 2, tradução nossa).

Isso aumenta os riscos à confidencialidade dos dados credenciais dos usuários, pois as aplicações terceiras tomam posse de tais credenciais para autenticar e autorizar os usuários no servidor. O modelo OAuth retira essa responsabilidade das aplicações terceiras e a retorna para os usuários.

De forma semelhante ao OpenID com autenticação, OAuth estende o modelo tradicional de autorização adicionando o papel do *resource owner*: Trata-se do verdadeiro detentor do recurso, no qual o cliente - o *software* utilizado pelo *resource owner* - irá requisitar ao servidor acesso a este recurso (HAMMER-LAHAV, 2010, tradução nossa).

O protocolo permite que *resource owners* autorizem clientes a acessarem seus recursos encontrados na web sem informar suas credenciais aos clientes. Isso é feito através de *tokens*. Um *token* é um identificador único gerado pelo servidor e utilizado pelo cliente ao requisitar recursos do servidor em nome do *resource owner*. Um *token* é gerado direcionado para um determinado cliente, contendo um conjunto temporário restrito de operações que esse cliente pode realizar (HAMMER-LAHAV, 2010, tradução nossa).

Como exemplo do processo de autorização OAuth, consideremos um usuário Aerys que possui seus contatos armazenados em um serviço que os disponibiliza pelo site contacts.net. O usuário utiliza o site agenda.net para visualizar seus contatos de contacts.net em forma otimizada e efetuar outras operações com esses dados. Os papéis desse exemplo, segundo o modelo OAuth, são verificados na Figura 7:

Figura 7 – Exemplo de papéis no modelo OAuth.

Elemento	Papel
Aerys	<i>resource owner</i>
contacts.net	servidor
agenda.net	cliente
contatos	recursos

Fonte: Elaborado pelo autor (2012)

Quando Aerys solicita a visualização de seus contatos em agenda.net, ele é redirecionado para contacts.net para que ele possa autorizar agenda.net a acessar seus recursos. Contacts.net solicita autenticação a Aerys por um método qualquer, em seguida solicitando a confirmação da autorização para as operações que agenda.net irá realizar. O servidor então gera o *token* e redireciona Aerys de volta para o cliente, informando o *token* nos cabeçalhos HTTP (HAMMER-LAHAV, 2010, tradução nossa). O cliente pode agora solicitar os dados ao servidor, informando sempre o *token* gerado, até que ocorra sua expiração ou revogação.

OAuth pode ser usado também em *softwares* que não estão em ambiente *web*. Uma possível versão *desktop* do agenda.net solicita que o *resource owner* acesse uma determinada URL (a forma de geração dessa URL foi previamente acordada entre o cliente e o servidor), na qual ele irá efetuar autenticação e aprovar a autorização do cliente *desktop*. Em seguida, essa página fornece um *token* para que o usuário insira-o no cliente.

2.3.4 Serviço de diretório

Serviço de diretório é uma tecnologia que basicamente manipula informações em um diretório, onde nomes são mapeados para um ou mais valores. Funções de compartilhamento de informações semi-estáticas numa organização são mais propriamente executadas em serviços de diretório que em banco de dados (KALALI, 2010, tradução nossa).

Serviços de diretório guardam as informações em forma hierárquica, em uma estrutura otimizada para leitura conhecida como *Directory Information Tree* (DIT) (KALALI, 2010, tradução nossa).

Um exemplo de uso desta tecnologia é no armazenamento de informações funcionais de uma organização. Uma pessoa pode possuir diversos atributos para formar sua identidade, como nome, atribuição, telefone, papéis, entre outros. Essa pessoa pode estar associada de forma hierárquica a outros objetos, como departamentos e outras pessoas.

Cada registro em um serviço de diretório possui um identificador único conhecido como *Distinguished Name* (DN). Esse identificador é construído pela associação de um ou mais atributos do objeto - tal associação é conhecida por

Relative Distinguished Name (RDN) - seguido do DN de seu objeto pai (KALALI, 2010, tradução nossa).

Em serviços de diretório, objetos pertencem a uma ou mais classes. Classes são utilizadas para classificação de objetos, especificando quais atributos um objeto pode ou deve conter. Essas classes são definidas em esquemas no *software* de serviço de diretório. Existe uma variedade de esquemas no mercado, onde cada fabricante define as regras de tais esquemas.

Devido ao crescimento, uso e importância dessa tecnologia, foi criado o protocolo *Lightweight Directory Access Protocol* (LDAP) definindo uma forma padrão e comum para acesso a serviços de diretório. Também foram criados esquemas padronizados e protocolos para armazenamento e transferência de dados em serviços de diretório (KALALI, 2010, tradução nossa).

2.3.5 Single Sign On

Single Sign On (SSO) é um recurso no qual usuários de um sistema efetuam autenticação através de uma entidade externa ao sistema no qual se deseja autenticar. Essa entidade externa é o sistema ou a central *Single Sign On*, que pode compartilhar seu serviço de autenticação de forma centralizada por vários sistemas. Como consequência, SSO se torna, de acordo com The Open Group (1999, tradução nossa), um recurso no qual o usuário se autentica uma vez e adquire autenticação em vários sistemas, normalmente de uma mesma organização. No modelo clássico, o usuário de sistemas de uma organização é requisitado para informar suas credenciais sempre que ele procura acesso a um novo sistema.

Daswani, Kern e Kesavan (2007, tradução nossa) atenta para a divergência existente entre conveniência e segurança. Sistemas se tornam menos convenientes para usuários ao passo que mais segurança é adicionada. Isso pode ser observado analisando a aceitação de usuários à obrigatoriedade de maior complexidade nas senhas para realizar autenticação em um determinado sistema. Quanto menor a força da senha, mais facilmente os usuários irão memorizá-las, e mais conveniente será o processo de autenticação. O conceito de SSO surge com o intuito de tornar o processo de autenticação múltipla mais conveniente, procurando manter o mesmo nível de segurança.

Os principais benefícios desse modelo são a eliminação da necessidade de memorização de várias senhas por parte dos usuários, a redução do tempo gasto ao realizar múltiplas autenticações e a diminuição dos custos de manutenção dos vários mecanismos de autenticação existentes utilizando o modelo tradicional, trazendo assim benefícios tanto para o aspecto de uso quanto para gerenciamento (THE OPEN GROUP, 1999, tradução nossa).

Mularen (2011, tradução nossa) introduz o *software Central Authentication Service* ou CAS (<http://www.ja-sig.org/cas>), mantido pela organização *Java in Administration Special Interest Group* (JA-SIG) que implementa um protocolo de mesmo nome, oferecendo uma solução SSO em um ambiente *web*. CAS provê um mecanismo de autenticação centralizada de usuários a múltiplos recursos, mantendo a configuração e armazenamento das credenciais dos usuários em um único ponto de acesso: um recurso acessível na rede conhecido como servidor CAS (MAZUREK, 2005, tradução nossa). Seu funcionamento se dá com os processos de autenticação dos sistemas *web* sendo direcionados ao servidor CAS. Este irá autenticar o usuário e retornar uma chave, denominada *ticket*, para o recurso no qual o usuário requisita acesso. O *ticket* será utilizado posteriormente para validar a autenticação junto ao portal CAS.

2.3.6 Segurança JEE

A plataforma JEE (*Java Enterprise Edition*) fornece recursos de segurança de forma intrusiva e não-intrusiva (KALALI, 2010, tradução nossa).

Kalali (2010, tradução nossa) explica os termos existentes no ambiente da segurança JEE:

- a) Usuário: um programa ou um operador de *software*. Pode ser membro de zero ou mais grupos;
- b) Grupo: conjunto de usuários que possuem um conjunto de características comuns que geralmente levam a um conjunto comum de permissões e níveis de acesso;
- c) *Realm*: um *realm* é o canal de acesso do servidor ao local onde as informações das identidades são guardadas, como um recurso na rede ou um arquivo em disco;

- d) Papel: um conceito em nível de aplicação que mapeia o nível de acesso - ou função - definido no *software* para usuários e grupos definidos no *realm*;
- e) *Principal*: uma identidade definida que irá efetuar a autenticação. Para isso, é necessária a apresentação das credenciais. Poderá ou não ser autenticada;
- f) Credencial: contém informação usada para autenticar um *principal*. Por exemplo, uma senha secreta, *tokens* biométricos, certificados digitais, entre outros.

A tecnologia JEE nos permite definir restrições de acesso em nível de URLs da aplicação (KALALI, 2010, tradução nossa). A Figura 8 ilustra um trecho de exemplo de configuração de restrição de acesso à interface administrativa de um sistema construído na plataforma JEE. Percebe-se que somente usuários que possuem a função *funcao_administrador* podem acessar as URLs do sistema que começam com */jsp/admin*.

Figura 8 – Configuração de uma restrição de segurança na plataforma JEE.

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Interface Administrativa</web-resource-name>
    <url-pattern>/jsp/admin/*</url-pattern>
    <http-method>PUT</http-method>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
    <http-method>DELETE</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>funcao_administrador</role-name>
  </auth-constraint>
</security-constraint>
```

Fonte: Adaptado de Kalali (2010).

O exemplo da Figura 8 mostra a configuração de autorização de forma declarativa e não-intrusiva, feita no *deployment descriptor*. JEE fornece outra forma de definir controles de acesso de forma declarativa: através de meta dados nas classes Java, definidos pelo uso de *annotations*. Existem *annotations* para definir os papéis e para restringir acesso a métodos dos componentes (KALALI, 2010, tradução nossa).

O suporte de autenticação a uma aplicação web na plataforma JEE é adicionado no *deployment descriptor*, onde se define o *realm* a ser utilizado e o método de autenticação. Diferentes métodos fornecem diferentes níveis de segurança e proteção. Entre eles temos o *HTTP Basic*, onde o servidor requisita as credenciais do usuário através de cabeçalhos HTTP, e o *Form Based*, onde a própria aplicação web requisita ao usuário suas credenciais - através de um formulário de autenticação, por exemplo - e as fornece ao container para que valide a autenticação (KALALI, 2010, tradução nossa).

A autenticação nos módulos clientes que são executados no *Application Client Container* (ACC) é obtida automaticamente pelo *container* e enviada ao servidor para a realização da autenticação e posterior autorização. Não há suporte padrão para autenticação em clientes limitados à plataforma *Java Standard Edition* (JSE) (KALALI, 2010, tradução nossa).

A plataforma JEE também suporta segurança programática ou intrusiva, nos casos onde a segurança declarativa não é suficiente para suprir os requisitos de autenticação, ou por requisitos do próprio *software*. É possível obter informações do usuário autenticado, como seus papéis e credenciais, como também realizar a autenticação de um *principal*, e assim tomar as decisões de segurança baseado nos dados obtidos (KALALI, 2010, tradução nossa).

2.4 SPRING SECURITY

Spring Security é o *framework* de controle de acesso padrão para sistemas corporativos construídos com o *meta-framework* Spring. O *framework* provê um conjunto de serviços de autenticação e autorização, como também facilita a integração de implementações de várias tecnologias de segurança em *software* à este conjunto, como criptografia, serviços de diretório e autenticação distribuída. Além disso, Spring Security pode ser usado em sistemas que não possuem a tecnologia Spring Framework como base (ALEX e TAYLOR 2011, tradução nossa).

Spring Security pode ser configurado de forma não-intrusiva e não-invasiva, apenas com o uso de arquivos de configuração, mantendo as classes Java livres de código de segurança. O *framework* foi construído de forma a prover uma arquitetura

conectável onde os principais componentes podem ser substituídos (ALEX e TAYLOR, 2011, tradução nossa).

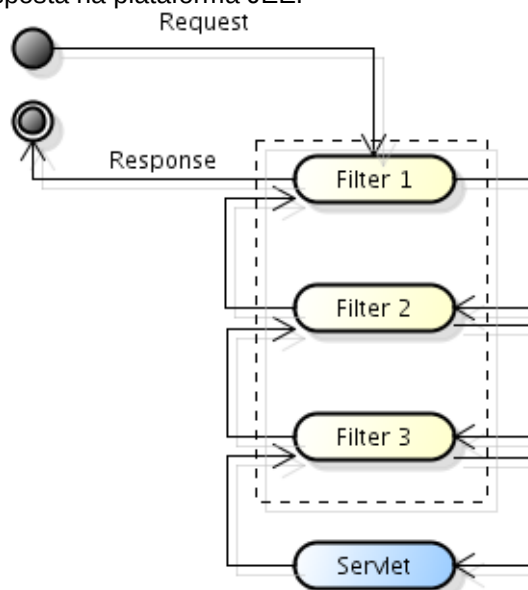
Em Alex e Taylor (2011, tradução nossa) encontramos alguns dos principais recursos do Spring Security, entre eles:

- a) Autenticação com certificados e suporte a criptografia: Spring Security provê recursos para leitura de certificados X.509, redirecionamento de requisições para canais criptografados e suporte a codificações SHA e MD5;
- b) Suporte a OpenID e OAuth: autenticação descentralizada e acesso a recursos em nome do usuário totalmente suportados;
- c) Autenticação em LDAP: autenticação em servidores LDAP;
- d) Integração com CAS: suporte a recursos avançados de *Single Sign On* com a tecnologia *Central Authentication Service*;
- e) Adaptação a modelos de segurança corporativo: os componentes do *framework* são altamente customizáveis podendo se adaptar a modelos de dados já existentes em uma organização;
- f) Outros mecanismos de autenticação: suporte à autenticação por meio de cabeçalhos HTTP, arquivos de configuração, banco de dados relacionais, entre outros. E ainda o suporte à inserção do próprio mecanismo de autenticação;
- g) Orientação à eventos e orientação à aspectos: habilidade de executar tarefas em determinados eventos relativos à autenticação e autorização, como também suporte a expressões AspectJ para aplicação de lógica de segurança transcendente à da aplicação;
- h) Autenticação em sistemas externos: em situações onde se deseja utilizar Spring Security apenas em decisões de autorizar usuários já autenticados, o *framework* permite o uso de autenticação por um sistema externo, como um container JEE.

Spring Security é utilizado no controle de acesso HTTP em serviços *web*, como também é capaz de fornecer segurança a componentes dentro do contexto do Spring Framework, provendo segurança em camadas internas da aplicação e facilitando testes unitários de controle de acesso (ALEX e TAYLOR, 2011, tradução nossa).

Spring Security re-utiliza elementos da arquitetura JEE, como *filters*, para formar sua estrutura de processamento de requisições. Dez *filters* são habilitados em corrente por padrão na configuração básica, em uma ordem pré-definida, cada um com um propósito relativo ao controle de acesso. Além dos *filters* padrões, Spring Security oferece outros *filters* para o processamento de requisições, como também é possível implementar *filters* adicionais e redefinir os *filters* padrões (MULARIEN, 2011, tradução nossa). Em cada requisição, a corrente de *filters* é processada antes e depois do processamento do *servlet*, como ilustra a Figura 9:

Figura 9 – Requisição e resposta na plataforma JEE.



Fonte: Adaptado de Mularien (2011)

Um dos principais componentes do processo de autenticação é o *AbstractAuthenticationProcessingFilter*, do pacote *org.springframework.security.web.authentication.filter* que recebe a requisição, cria um objeto *org.springframework.security.core.Authentication* e delega para a estratégia de autenticação utilizada. Objetos *Authentication*, na arquitetura do Spring Security, contêm as credenciais informadas para efetuar o processo, semelhante ao *Principal* da própria arquitetura JEE. Este objeto é então enviado ao *org.springframework.security.authentication.AuthenticationManager* - componente específico da estratégia utilizada - que irá validar as credenciais e preencher as demais informações no objeto *Authentication*, sendo este o responsável direto pela autenticação. O *AuthenticationManager* irá delegar a validação das credenciais a um

terceiro componente, o *AuthenticationProvider*, que pertence ao mesmo pacote e é responsável pela verificação das credenciais informadas junto ao repositório de usuários (MULARIEN, 2011, tradução nossa).

O último *filter* da corrente de *filters* do Spring Security é o *org.springframework.security.web.access.intercept.FilterSecurityInterceptor*, responsável por processar a decisão se uma requisição será aceita. Quando esse *filter* é executado, a autenticação já foi realizada. O *FilterSecurityInterceptor* irá então invocar o componente *org.springframework.security.access.AccessDecisionManager*, que irá utilizar as informações do elemento *Authentication* já existente para efetuar a autorização de requisições *web*. Componentes *AccessDecisionManager* implementam a interface do mesmo nome. Essa interface declara dois métodos: *supports* e *decide*. O primeiro reporta se um componente *AccessDecisionManager* suporta a requisição atual, enquanto que o segundo verifica se o acesso é permitido (MULARIEN, 2011, tradução nossa).

A configuração básica do Spring Security para controle de acesso de forma declarativa em sistemas *web* JEE pode ser feita com os seguintes passos:

- a) Adicionar as bibliotecas do Spring Security ao *classpath*;
- b) Declarar o *filter* básico do Spring Security, o *DelegateFilterProxy*, do pacote *org.springframework.web.filter*, ao *deployment descriptor* através da diretiva da Figura 10, delegando assim todas as requisições à estrutura do Spring Security. É necessário também declarar o *listener* *ContextLoaderListener*, do pacote *org.springframework.web.context*, que tem como função iniciar e finalizar o contexto *web* do Spring;
- c) Em seguida implementa-se o arquivo de configuração *Extensible Markup Language* (XML) do Spring Security, declarando os componentes necessários para suportar requisições *web*, ilustrado na Figura 11.

Figura 10 – *Filter DelegatingFilterProxy, e listener ContextLoaderListener.*

```

<filter>
  <filter-name>springSecurityFilterChain</filter-name>
  <filter-class>org.springframework.web.filter.DelegatingFilterProxy
</filter-class>
</filter>

<filter-mapping>
  <filter-name>springSecurityFilterChain</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>

<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>

```

Fonte: Elaborado pelo autor (2012)

A Figura 11 mostra as configurações necessárias para prover segurança padrão mínima a uma aplicação *web* JEE. O atributo *auto-config* com o valor *true* na diretiva *http* (elemento base de toda configuração de segurança *web*) irá induzir o Spring Security a registrar os componentes básicos JEE comentados nesta seção, assim como um formulário HTML de *login* com usuário e senha, suporte a autenticação HTTP *basic*, serviços de *logout*, entre outros (MULARIEN, 2011, tradução nossa). Alex e Taylor (2011, tradução nossa) explicam que o elemento *intercept-url* define um padrão (*pattern*) que é comparado com as URLs das requisições utilizando expressões *ant*, definindo os requisitos de acesso a essas URLs. Neste caso, é feita uma comparação simples se o usuário possui um dos papéis definidos no atributo *access*. Os elementos filhos de *authentication-manager* declaram credenciais e papéis (*authorities*) de forma estática no próprio arquivo, facilitando testes e a configuração básica (MULARIEN, 2011, tradução nossa).

Figura 11 – Configuração automática de segurança HTTP com Spring Security.

```

<http auto-config="true">
  <intercept-url pattern="/*" access="ROLE_USER" />
</http>
<authentication-manager alias="authenticationManager">
  <authentication-provider>
    <user-service>
      <user authorities="ROLE_USER" name="guest" password="guest" />
    </user-service>
  </authentication-provider>
</authentication-manager>

```

Fonte: Elaborado pelo autor (2012)

Após a configuração desses elementos, ao se acessar o endereço da aplicação através de um *browser*, a página de *login* é apresentada ao usuário. O mapeamento */** para o papel *ROLE_USER* (Figura 11) indica que o usuário que acessar qualquer URL da aplicação deverá ter esse papel. O usuário *guest* possui o papel *ROLE_USER*, já sendo possível efetuar autenticação e autorização com esses dados.

2.4.1 Spring Framework

Spring Framework é uma plataforma para aplicações corporativas em Java. Encontramos publicações que descrevem Spring Framework como *meta-framework*, ou somente *framework*. Spring Framework é projetado para ser não-intrusivo, de forma que a lógica envolvida nos objetivos da aplicação não possui dependências com o *framework* (JOHNSON et al, 2011, tradução nossa).

O núcleo do Spring Framework é baseado no princípio de Inversão de Controle, ou *Inversion of Control* (IoC), também conhecido como Injeção de Dependências, ou *Dependency Injection* (DI). Sistemas que seguem o princípio de IoC usam configuração ou meta dados para descrever as dependências entre seus componentes, e cabe ao *framework* de IoC - e não aos próprios componentes dos sistemas - satisfazer as dependências. O termo inversão vem do fato que a aplicação não controla sua estrutura, função atribuída ao *framework* IoC (MACHACEK et al, 2008, tradução nossa).

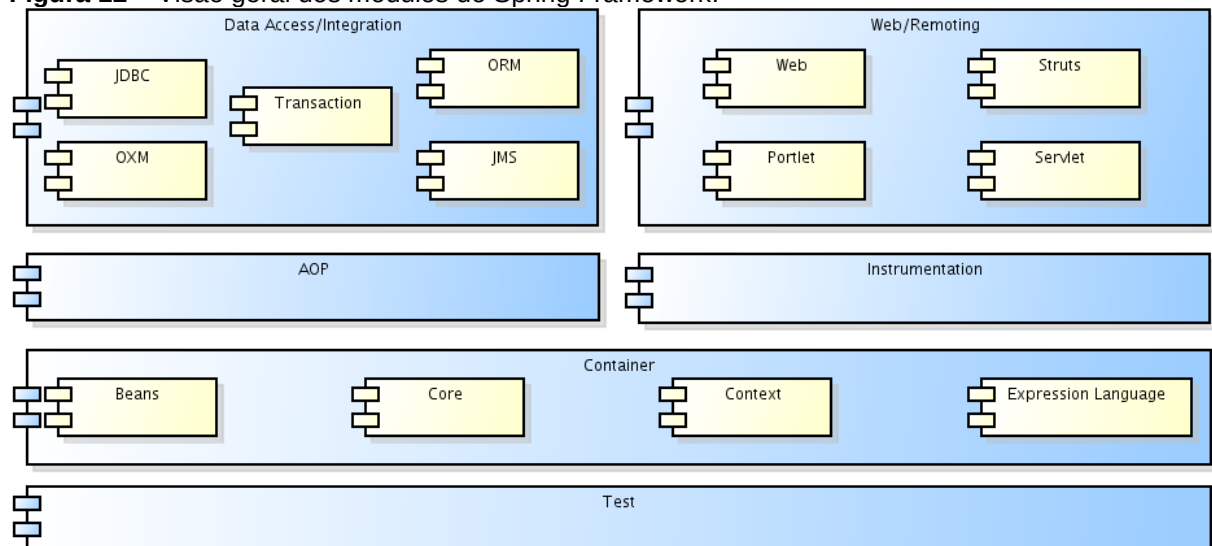
Componentes de sistemas baseados em Spring Framework vivem dentro do contêiner IoC, e são normalmente chamados de *beans*. O contêiner é responsável pela construção dos componentes da aplicação, a injeção das dependências necessárias ao funcionamento de tais componentes, e todas as outras fases que compõem o ciclo de vida de um *bean* (WALLS, 2011, tradução nossa). Na maioria dos casos, código Java não é necessário para instanciar um contêiner. Existem comportamentos padrões que um contêiner pode assumir, como também é possível configurá-lo através de XML (JOHNSON et al, 2011, tradução nossa).

O *framework* tem como objetivo simplificar o desenvolvimento e manutenção de sistemas construídos na linguagem Java, sendo amplamente adotado. Suas estratégias-chave são (WALLS, 2011, tradução nossa):

- a) Desenvolvimento minimamente invasivo com o uso de *plain old Java objects* (POJOs): Spring Framework evita, na medida do possível, inserir código relativo à sua *Application Programming Interface* (API) no código dos sistemas desenvolvidos com o *framework*;
- b) Baixo acoplamento através de desenvolvimento orientado à interfaces e Injeção de Dependências: com o contêiner realizando DI, o desenvolvimento orientado à interfaces torna-se uma prática aplicável, diminuindo ou anulando o acoplamento entre objetos concretos da aplicação. Essa configuração do contêiner para realizar DI é feita com o uso de meta dados (*Java annotations*) e/ou arquivos XML;
- c) Programação declarativa através de aspectos: o uso de aspectos como uma técnica que promove a separação de interesses em um sistema permite que os componentes da aplicação foquem em seu propósito, e ainda utilizem, de forma transparente, serviços como controle de ciclo de vida de transações, segurança e auditoria;
- d) Redução de código repetitivo com o uso de aspectos e *templates*: através da abstração de algumas tecnologias que forçam o desenvolvedor a escrever código repetido, como *Java Database Connectivity* (JDBC), *Java Message Service* (JMS), consumo de *web services*, entre outros.

O núcleo do Spring Framework é constituído por vinte *frameworks* distintos, nos quais Johnson et al (2011, tradução nossa) os qualifica como módulos, divididos em seis categorias bem definidas, como observado na (Figura 12).

Figura 12 – Visão geral dos módulos do Spring Framework.



Fonte: Adaptado de Walls (2011)

Essas categorias são explicadas por Walls (2011, tradução nossa):

- Container*: contém a parte central do Spring Framework - o contêiner que gerencia o ciclo de vida dos *beans* através da Injeção de Dependências. Os demais módulos são construídos com base nestes;
- Programação Orientada a Aspectos, ou *Aspect-Oriented Programming* (AOP): Spring Framework provê seu suporte à AOP neste módulo. Assim como Injeção de Dependências, AOP tem como objetivo final o baixo acoplamento entre componentes dos sistemas. AOP permite que interesses que transcendem aos objetivos principais dos sistemas, como segurança e transações, sejam desacoplados dos objetos nos quais eles são aplicados;
- Data Access/Integration*: trabalhar com a tecnologia JDBC normalmente resulta em código repetitivo para as tarefas elementares nos quais a tecnologia se propõe, como o ciclo de vida de conexões com banco de dados. Um dos objetivos principais destes módulos é abstrair essa repetição, aumentando a manutenibilidade do código de acesso ao banco de dados, provendo ferramentas para controle de ciclo de vida de conexões, gerenciamento de transações através de AOP, entre outros. Os módulos que compõem a categoria *Data Access/Integration* também provêm facilidades para a construção de componentes *Data Access Object* (DAO) e integração com tecnologias *Object Relational Mapping*

(ORM). E ainda incluem abstrações para a tecnologia JMS e recursos de serialização de objetos para XML;

- d) *Web/Remoting*: esta categoria traz um *framework Model-View-Controller* (MVC) completo que leva as técnicas de baixo acoplamento do Spring Framework à camada de visão de sistemas web, como também provê integração com vários *frameworks* MVC populares existentes. *Web/Remoting* ainda traz recursos para intercomunicação de sistemas, como *Remote Method Invocation* (RMI), *Java API for XML Web Services* (JAX-WS), entre outros;
- e) *Test*: módulos dedicados a testes de sistemas construídos com Spring Framework. Contém implementações de *mock objects* para testes unitários e testes de integração, e interoperabilidade com alguns *frameworks* de teste;
- f) *Instrumentation*: contém suporte à *class instrumentation* e provê implementações de *classloaders* para serem utilizados em servidores de aplicação.

Além do núcleo, Spring Framework possui um conjunto de *frameworks* e bibliotecas construídos em cima do núcleo, denominado *Spring Portfolio* (WALLS, 2011, tradução nossa). No *Spring Portfolio* - além do Spring Security - Walls (2011, tradução nossa) cita várias outras soluções, dentre as quais destacamos:

- a) Spring Social: uma extensão do Spring Framework para se trabalhar com redes sociais, como Twitter e Facebook;
- b) Spring Mobile: provê suporte ao desenvolvimento de aplicativos *web* para dispositivos móveis. Esse projeto inclui o Spring Android, que tem como objetivo levar as facilidades e simplicidades do Spring Framework para o desenvolvimento de aplicativos nativos na plataforma Android;
- c) Spring LDAP: busca a eliminação de repetição de código envolvido em operações de manipulação de serviços LDAP;
- d) Spring Rich Client: ferramentas baseadas no Spring Framework para desenvolvimento de aplicações com a tecnologia Swing;
- e) Spring Web Services: oferece um modelo de implementações de *web services* que determinam o próprio contrato, oferecendo integração com outros componentes do Spring Portfolio;

- f) Spring Web Flow: traz suporte a implementação de aplicações *web* baseadas em escopos conversacionais, provendo integração com tecnologias como *Java Server Faces*;

3 APLICAÇÕES DE SEGURANÇA COM SPRING SECURITY

O primeiro passo para a realização desse trabalho foi a análise dos recursos do Spring Security expostos em Alex e Taylor (2011) e a seleção de algumas funcionalidades normalmente encontradas no âmbito de segurança em sistemas corporativos.

A primeira funcionalidade observada foi a habilidade de adicionar novos usuários, preencher suas credenciais e atribuir papéis aos mesmos de forma dinâmica, assim como a posterior manipulação desses dados. Esse dinamismo é alcançado guardando esses dados em uma base de dados de credenciais. Na configuração básica exposta em 2.4, as credenciais dos usuários e seus papéis são definidos de forma volátil, no próprio arquivo de configuração do *framework*. No término da execução da aplicação, todas as alterações feitas a esses dados são perdidas, comportamento normalmente não desejável em sistemas em produção. Já em ambientes de testes de *software*, por exemplo, pode ser desejável que esses dados retornem ao estado original após a execução de um conjunto de testes.

Outro recurso observado foi o controle de acesso em *beans* do Spring Framework, protegendo a invocação de métodos. Melhores práticas indicam que sejam aplicadas regras de segurança a métodos que fazem parte da *service layer*, pois é nesse conjunto de componentes onde reside a maior parte da lógica de negócio de sistemas corporativos (ALEX; TAYLOR, 2011, tradução nossa). O uso de segurança na *service layer*, em comparação à segurança *web*, é mais robusta e mais difícil de ser quebrada, não possuindo riscos de mudanças em URLs (ALEX; TAYLOR, 2011, tradução nossa). Concentrar as regras de controle de acesso na *service layer* garante a consistência dessas regras em todos os ambientes onde o *bean* se encontra, como *web*, *desktop*, *jobs*, testes unitários, entre outros.

Como já foi mencionado, Spring Security foi construído visando garantir extensibilidade a seus componentes. É possível propagar os recursos de autenticação para o uso de outros mecanismos e paradigmas além da simples apresentação e validação de credenciais. Nesse âmbito foram selecionadas as seguintes tecnologias de autenticação:

- a) Serviços de diretório: por seu largo uso em ambientes corporativos para definição de credenciais funcionais em organizações, realização de autenticação e autorização baseados nos dados funcionais de usuários e serviços externos, e uso do padrão comum LDAP para busca e manipulação desses dados;
- b) OpenID: a importância e aceitação da autenticação descentralizada vem crescendo nos últimos anos, por ser uma forma mais segura de autenticação em serviços de terceiros, tornando o processo de autenticação mais rápido e removendo a necessidade do armazenamento de credenciais em clientes (OPENID FOUNDATION, 2009, tradução nossa). Spring Security dá suporte as principais características do protocolo, sendo assim, clientes que possuem Spring Security como tecnologia de controle de acesso podem utilizar *trusted providers* para efetuar autenticação e recuperação de informações de usuários.

Finalmente, verificou-se a integração nativa do *framework* com o *software Single Sign On Central Authentication Service* (introduzido em 2.3.5), oferecendo a possibilidade de se utilizar um provedor externo de autenticação de uma forma declarativa. Verificou-se também a gama de recursos oferecidos pelo CAS, a quantidade crescente de grandes organizações que utilizam este *software* e a variedade de plataformas que, de acordo com Jasig (2010, tradução nossa) integram/suportam essa tecnologia.

Em seguida foram feitos estudos de casos com implementações desses recursos com a tecnologia Spring Security. Essa seção tem por objetivo descrever como foram implementados os recursos de segurança em *software* elencados.

3.1 CONFIGURAÇÃO DA BASE DE DADOS DE CREDENCIAIS

A diretiva *authentication-provider* indica um componente *org.springframework.security.core.userdetails.UserDetailsService* que deverá ser utilizado como fonte de autenticação. O elemento interno *user-service* (Figura 13) induz o Spring Security a criar uma implementação em memória da interface *UserDetailsService*, que irá usar como fonte de dados as informações dos sub elementos *user* (ALEX; TAYLOR, 2011, tradução nossa).

Figura 13 – Criação de um *UserDetailsService* em memória.

```
<authentication-provider>
  <user-service>
    <user authorities="ROLE_USER" name="guest" password="guest" />
  </user-service>
</authentication-provider>
```

Fonte: Elaborado pelo autor (2012)

Outra forma de se definir qual *UserDetailsService* será usado é substituir o elemento filho *user-service* pelo atributo *user-service-ref*. Este atributo deverá referenciar um *bean* que implementa a interface *UserDetailsService* (Figura 14). Esse *bean* será então a nova fonte de autenticação (ALEX e TAYLOR 2011, tradução nossa).

Figura 14 – Referenciando um componente *UserDetailsService* que possui o identificador *uds*.

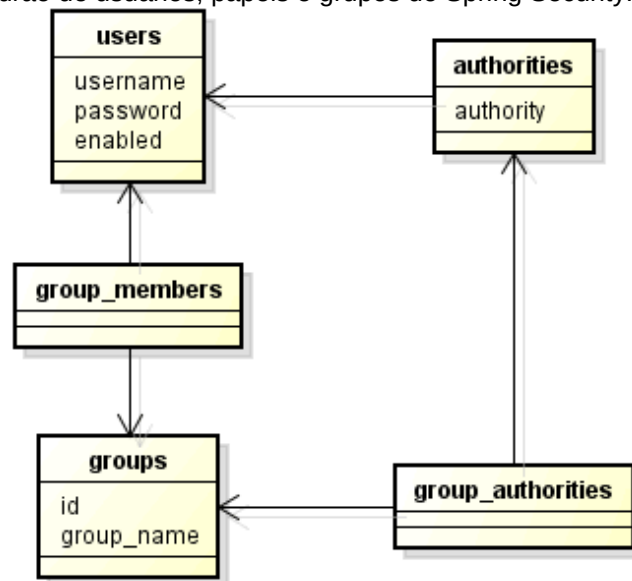
```
<authentication-provider user-service-ref="uds" />
```

Fonte: Elaborado pelo autor (2012)

A interface *UserDetailsService* define apenas um método: *loadUserByUsername*, que irá receber um nome de usuário e retornar uma implementação da interface *UserDetails*, do pacote *org.springframework.security.core.userdetails*, que representa um usuário com algumas informações básicas, suas credenciais e seus papéis (*authorities*).

Spring Security traz várias implementações de *UserDetailsService*. Entre elas, tem-se *JdbcDaoImpl*, do pacote *org.springframework.security.core.userdetails.jdbc*, que recupera todas as informações de usuários (credenciais, papéis, entre outros) de um banco de dados utilizando consultas JDBC. Essa classe recebe uma implementação de *javax.sql.DataSource* para executar as consultas. Seguindo o modelo de reusabilidade e desenvolvimento orientado a interfaces do Spring Framework, *JdbcDaoImpl* reutiliza componentes do módulo Spring Orm, como *org.springframework.jdbc.core.JdbcTemplate*, que manipula a criação e liberação de recursos, executa consultas e manipula seus resultados, entre outras funcionalidades (ALEX; TAYLOR, 2011, tradução nossa).

Figura 15 – Modelo padrão de usuários, papéis e grupos do Spring Security.



Fonte: Elaborado pelo autor (2012)

JdbcDaoImpl irá operar, por padrão, com o modelo de dados padrão para usuários e papéis do Spring Security, ilustrado na Figura 15. As consultas aos dados são totalmente manipuláveis, de forma que é possível alterá-las através da própria declaração do *bean*, no arquivo de configuração do Spring Security (Figura 16), podendo-se assim adaptar-se a um modelo de dados diferente, através das propriedades:

- a) *usersByUsernameQuery*: consulta usada para recuperar usuários;
- b) *authoritiesByUsernameQuery*: consulta usada para recuperar os papéis de um usuário, necessários para preencher as informações do objeto *UserDetails*;
- c) *groupAuthoritiesByUsernameQuery*: consulta que tem como objetivo recuperar os papéis de um grupo de um usuário. Por padrão o uso de grupos é desabilitado e essa consulta não é usada.

Figura 16 – Declaração de um *JdbcDaoImpl*, alterando duas de suas consultas padrão.

```
<bean id="uds"
  class="org.springframework.security.core.userdetails.jdbc.JdbcDaoImpl">
  <property name="dataSource" ref="beanDataSource" />
  <property name="usersByUsernameQuery"
    value="select nome from usuarios where nome = ?
          and habilitado = true" />
  <property name="authoritiesByUsernameQuery"
    value="select papel from papeis where nome = ?
          and expirado = false" />
</bean>
```

Fonte: Elaborado pelo autor (2012)

Spring Security fornece uma *tag* exclusiva para definição de um *UserDetailsService* buscando informações em banco de dados: *jdbc-user-service*. Quando essa diretiva é declarada como sub elemento de *authentication-provider*, o *framework* irá automaticamente configurar um *JdbcDaoImpl* como fonte de dados de credenciais e papéis. O usuário definirá no atributo *data-source-ref* qual *javax.sql.DataSource* será utilizado. Também é possível, nesta mesma diretiva, redefinir as consultas padrões (ALEX; TAYLOR, 2011, tradução nossa).

3.2 IMPLEMENTAÇÃO DE SEGURANÇA NA SERVICE LAYER

Como foi mencionado em 2.4, Spring Security provê controle de acesso a invocação de métodos de *beans* do Spring Framework. Isso é realizado pelo uso de orientação a aspectos. É possível configurar o controle de acesso em métodos, classes e interfaces de duas formas (ALEX; TAYLOR, 2011, tradução nossa):

- a) *Annotations* nos métodos, classes ou interfaces dos *beans*, informando os papéis (*roles*) que um usuário autenticado deverá possuir para acessar este método. Também é possível fazer uso de Spring EL, tecnologia que suporta manipulação de objetos em tempo de execução;
- b) AOP *interceptors* definidos em arquivos de configuração, permitindo a definição de regras de segurança que abrangem vários beans em uma única declaração.

Ambas as formas permitem realizar segurança declarativa em métodos de *beans* no contexto do Spring Framework.

3.2.1 Controle de acesso com *pointcuts*

Para adicionar suporte à segurança em métodos, classes e interfaces, declara-se o elemento *global-method-security* no arquivo de configuração (Figura 17). Os elementos filhos *protect-pointcut* definem *pointcuts* AOP e os controles de acesso que serão aplicados aos mesmos. Todos os *beans* no contexto do Spring Framework que possuem métodos que casam com a expressão serão contemplados com a regra de controle de acesso (Figura 17) (ALEX; TAYLOR, 2011, tradução nossa).

Figura 17 – Configuração de controle de acesso com *pointcuts* AOP.

```
<global-method-security>
  <protect-pointcut
    expression="execution(*info.exemplo.*Service.delete(..))"
    access="ROLE_ADMIN"/>
</global-method-security>
```

Fonte: Elaborado pelo autor (2012)

No exemplo da Figura 17, os métodos *delete* de todas as classes do pacote *info.exemplo* que terminam com “Service” só serão acessados por usuários que possuam o papel *ROLE_ADMIN*. De forma declarativa foi aplicado controle de acesso a um conjunto de métodos de *beans* do Spring Framework sem a necessidade de intrusão de código de segurança nesses *beans*.

3.2.2 Controle de acesso com *annotations*

A principal *annotation* para realizar limitações de acesso baseadas em papéis a um método, classe ou interface é a *@Secured*, do pacote *org.springframework.security.access.annotation*. Para ativá-la, deve-se adicionar à diretiva *global-method-security* o atributo *secured-annotations="enabled"* ao arquivo de configuração (Figura 18).

Figura 18 – Ativando o suporte à *annotation @Secured*.

```
<global-method-security secured-annotations="enabled" />
```

Fonte: Adaptado de Alex e Taylor (2011).

A Figura 19 ilustra exemplos de uso da *annotation* `@Secured` em métodos de uma classe: só será possível acessar o método `retrieve(String)` se existir um usuário autenticado no contexto do Spring Security, e se este usuário possua os papéis (*authorities*) `ROLE_USER` ou `ROLE_ADMIN`. Já `remove(Integer)` só poderá ser acessado por quem possua `ROLE_ADMIN`.

Figura 19 – Exemplos de uso da *annotation* `@Secured`.

```
@Secured({"ROLE_USER", "ROLE_ADMIN"})
public List<String> retrieve(String param){}
```

```
@Secured("ROLE_ADMIN")
public void remove(Integer id){}
```

Fonte: Elaborado pelo autor (2012)

As demais *annotations* para controle de acesso em *beans* fazem uso de Spring EL, estendendo os recursos de `@Secured` e provendo mais possibilidades de verificação. São elas `@PreAuthorize`, `@PreFilter`, `@PostAuthorize` e `@PostFilter`, todas do pacote `org.springframework.security.access.prepost`. Essas *annotations* suportam verificações de autorização antes e após a execução do método, como também filtragem de coleções de entrada e de retorno (ALEX e TAYLOR 2011, tradução nossa). Elementos declarados com essas *annotations* irão induzir o *framework* a avaliar a expressão informada e, caso o retorno não seja verdadeiro, uma exceção será lançada. Para ativar o uso dessas *annotations*, adiciona-se o atributo `pre-post-annotations="enabled"` a diretiva `global-method-security` (Figura 20).

Figura 20 – Ativando o suporte as *annotations* `@PreAuthorize`, `@PreFilter`, `@PostAuthorize` e `@PostFilter`.

```
<global-method-security pre-post-annotations="enabled" />
```

Fonte: Adaptado de Alex e Taylor (2011).

A Figura 21 exemplifica o uso de `@PreAuthorize` e `@PreFilter`. A classe `Contato` possui a propriedade `nome`, do tipo `String`, e a propriedade `acessoPublico`, do tipo `Boolean`. Spring Security só concederá acesso ao método `metodo1(Contato)` caso o objeto `Contato` passado por parâmetro possua o valor de seu atributo `nome` igual ao nome do usuário autenticado (obtido através de `authentication.name`) e o usuário possua o papel `ROLE_ADMIN`, avaliado por `hasRole` (Figura 22). O método `metodo3(List<Contato>)` poderá ser invocado por qualquer usuário que possua

qualquer papel. Porém, antes da execução do método, Spring Security irá avaliar a propriedade *acessoPublico* de cada objeto da lista de *Contatos* passada por parâmetro, e irá remover da lista os que possuem o valor *false* nessa propriedade.

Figura 21 – Exemplos de uso das *annotations* *@PreAuthorize* e *@PreFilter*.

```
@PreAuthorize("#c.nome == authentication.name and hasRole('ROLE_ ADMIN')")
public void metodo1(Contato c){}
```

```
@PreFilter("filterObject.acessoPublico")
public void metodo3(List<Contato> cs){}
```

Fonte: Elaborado pelo autor (2012)

A Figura 22 lista as expressões comuns que podem ser utilizadas tanto em segurança em método quanto em segurança *web*:

Figura 22 – Expressões comuns para controle de acesso.

Expressão	Descrição
<code>hasRole([role])</code>	avalia verdadeiro se o usuário corrente possui a <i>role</i> informada
<code>hasAnyRole([role1,role2])</code>	avalia verdadeiro se o usuário corrente possui qualquer uma das <i>roles</i> informadas
<code>principal</code>	retorna o objeto que representa o usuário corrente
<code>authentication</code>	retorna o objeto <i>Authentication</i> atual
<code>permitAll</code>	sempre avalia verdadeiro
<code>denyAll</code>	sempre avalia falso
<code>isAnonymous()</code>	avalia verdadeiro se o usuário corrente é <i>anonymous</i>
<code>isRememberMe()</code>	avalia verdadeiro se o usuário corrente é um usuário <i>remember-me</i>
<code>isAuthenticated()</code>	avalia verdadeiro se o usuário corrente não é <i>anonymous</i>
<code>isFullyAuthenticated()</code>	avalia verdadeiro se o usuário corrente não é <i>anonymous</i> e não foi autenticado por <i>remember-me</i>

Fonte: Adaptado de Alex e Taylor (2011, tradução nossa).

3.3 AUTENTICAÇÃO COM OPENID

O suporte a OpenID é habilitado com a diretiva *openid-login*, definida como filha da diretiva *http* (ALEX, TAYLOR, 2011, tradução nossa). Caso o sub-elemento *form-login* esteja configurado (isso pode ser feito inserindo *form-login* como filho de *http* ou com o atributo *auto-config="true"*, mencionado em 2.4) o *framework* irá automaticamente criar um formulário de *login* com usuário e senha. Após o incremento de *openid-login* (Figura 23), Spring Security irá automaticamente adicionar um formulário adicional para que o usuário efetue *login* com seu *identifier* (Figura 24).

Figura 23 – Habilitando autenticação via OpenID no Spring Security.

```
<http>
  <openid-login />
</http>
```

Fonte: Adaptado de Alex e Taylor (2011, tradução nossa)

Figura 24 – Formulário de *login* com usuário e senha (acima) e com OpenID (abaixo), após a configuração da diretiva *openid-login*.

Login with Username and Password

User:

Password:

Login with OpenID Identity

Identity:

Fonte: Elaborado pelo autor (2012)

Em caso de aplicações que não usam o formulário de *login* padrão do Spring Security, o formulário HTML para autenticação com OpenID deverá ser inserido manualmente na página de *login* da aplicação. Esse formulário deverá submeter os dados para a URL padrão *j_spring_openid_security_check*, onde o *identifier* do usuário deverá ser enviado no atributo de requisição *openid_identifier* (Figura 25).

Figura 25 – Exemplo de formulário de autenticação com OpenID.

```
<form action="j_spring_openid_security_check" method="post">
  <input id="openid_identifier" name="openid_identifier" type="text" />
  <input type="submit" value="Login com OpenID" />
</form>
```

Fonte: Elaborado pelo autor (2012)

Baseado no *identifier* informado pelo usuário, Spring Security tentará identificar o *provider*, e irá direcionar o *browser* do usuário para a página de autenticação do *provider*. Após a troca de credenciais e autenticação, o usuário é redirecionado novamente para o sistema. É nesse momento que o Spring Security efetua a autorização do recurso que será acessado. Para que a autorização ocorra com sucesso, é necessário que o usuário possua o papel (*role*) apropriado já definido para seu *identifier* (Figura 26), seja no arquivo de configuração ou em uma base de dados de credenciais. O atributo *password* não é requerido, pois, pelo modelo do OpenID, essa informação é trocada apenas entre o usuário e o *provider*.

Figura 26 – Definindo um usuário pelo seu OpenID *identifier*, com o papel *ROLE_USER*.

```
<authentication-manager alias="authenticationManager">
  <authentication-provider>
    <user-service>
      <user authorities="ROLE_USER"
        name="http://gewton.myopenid.com/" />
    </user-service>
  </authentication-provider>
</authentication-manager>
```

Fonte: Elaborado pelo autor (2012)

De forma semelhante ao *filter UsernamePasswordAuthenticationFilter*, do pacote *org.springframework.security.web.authentication*, responsável por processar autenticação de usuários por *login* e senha, o *filter OpenIDAuthenticationFilter* (localizado em *org.springframework.security.openid*) é o responsável por atender as requisições à URL padrão *j_spring_openid_security_check*, e invocar uma instância da interface *OpenIDConsumer*, localizada no mesmo pacote, que irá tratar do processo de autenticação. O *OpenIDConsumer* padrão do Spring Security é o *org.springframework.security.openid.OpenID4JavaConsumer*, que utiliza a tecnologia OpenID4 for Java (<http://code.google.com/p/openid4java/>). Além disso, *OpenIDAuthenticationFilter* também é o responsável por processar a resposta do

provider, após realizada a tentativa de autenticação (MULARIEN, 2011, tradução nossa).

No caso de sistemas que utilizam uma base de dados dinâmica de credenciais e oferecem autorização por OpenID, Spring Security oferece algumas formas de capturar um evento bem sucedido de autenticação e checar posteriormente se o usuário existe ou não na base. Em caso negativo, a aplicação pode registrar este usuário e atribuir papéis a ele, para que seja possível seu acesso a recursos do sistema. Uma maneira de reproduzir esse comportamento é criando uma implementação da interface *AuthenticationFailureHandler* (pacote *org.springframework.security.web.authentication*). De acordo com Alex e Taylor (2011, tradução nossa), esses componentes são responsáveis por lidar com tentativas falhas de autenticação. Essa implementação seria responsável por identificar a causa do erro e, em caso de usuário não cadastrado, efetuar a lógica apropriada. Deve-se registrar essa classe como um *bean* do Spring Framework e referenciá-lo no atributo *authentication-failure-handler-ref* da diretiva *openid-login* (Figura 27). Sendo assim o *framework* irá invocar a lógica dessa classe sempre que ocorrer uma falha no processo de liberação de um recurso após o usuário efetuar uma autenticação com seu OpenID *provider*.

Figura 27 – Configurando um *AuthenticationFailureHandler* como responsável por lidar com falhas de autenticação.

```
<bean id="failureHandler"
      class="com.foo.OpenIDAuthenticationFailureHandler" />

<openid-login authentication-failure-handler-ref="failureHandler">
```

Fonte: Adaptado de Mularien (2011).

3.4 CONTROLE DE ACESSO COM LDAP

Spring Security oferece suporte a LDAP através de componentes que utilizam estratégias distintas para autenticação e autorização, sendo totalmente configuráveis. O *framework* ainda provê implementações padrão também configuráveis para uma gama de situações, visto que existem diferentes cenários de configuração para um servidor LDAP como repositório de informações de usuários (ALEX; TAYLOR 2011, tradução nossa).

A diretiva *ldap-server* permite configurar um servidor LDAP em memória para uso em testes e ainda carregá-lo com dados iniciais (Figura 28), ou apontar para o endereço de um servidor existente, como exemplificado na Figura 29. As funcionalidades do servidor em memória definido na Figura 28 são providas pela tecnologia *Apache Directory Server* (Apache DS) (<http://directory.apache.org/>). O carregamento de dados do servidor em memória é definido através de arquivos *LDAP Data Interchange Format* (LDIF). Por padrão, as buscas ocorrerão na raiz *org.springframework*. Após definido, o servidor em memória poderá ser acessado pelo seu identificador ou através do endereço onde é servido, por padrão *ldap://localhost:33389/*.

Figura 28 – Configurando um servidor LDAP em memória e carregando os dados do arquivo *users.ldif*.

```
<ldap-server ldif="classpath:users.ldif" id="servidorLdapEmMemoria" />
```

Fonte: Elaborado pelo autor (2012)

Figura 29 – Configurando um servidor LDAP através de um serviço já existente acessado pela URL *springframework.org*, com o elemento raiz *com.security*.

```
<ldap-server url="ldap://springframework.org:389/dc=security,dc=com"
  id="servidorLdapExistente" />
```

Fonte: Elaborado pelo autor (2012)

Em seguida é necessário criar um componente *AuthenticationProvider* que utilize o servidor LDAP para realizar o controle de acesso. Esse componente é utilizado pelo *AuthenticationManager* do *framework* para efetuar a autenticação de usuários. Spring Security possui a diretiva *ldap-authentication-provider* (Figura 30) que automaticamente configura uma série de componentes para efetuar buscas e filtragens no servidor LDAP definido em *ldap-server* para autenticação e autorização (ALEX; TAYLOR 2011, tradução nossa).

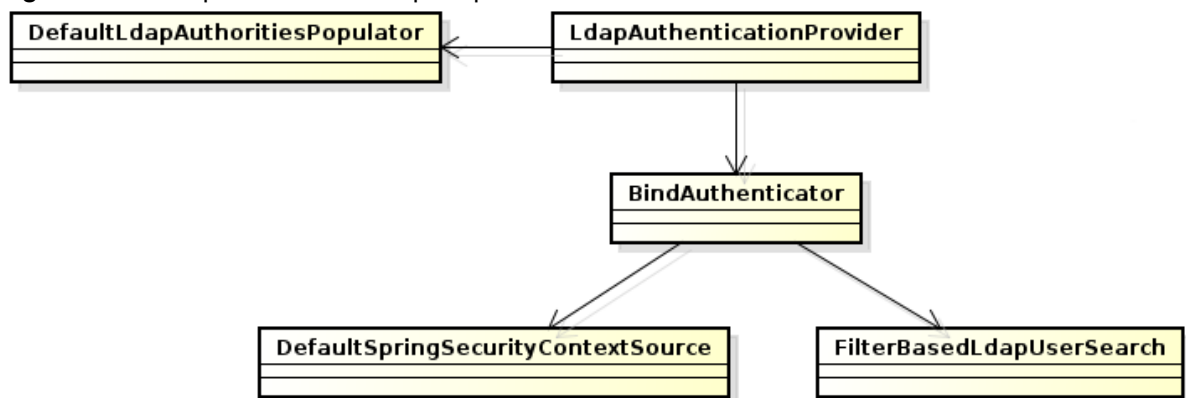
Figura 30 – Configurando um *AuthenticationProvider* que utiliza o servidor LDAP para efetuar controle de acesso.

```
<authentication-manager>
  <ldap-authentication-provider
    group-search-filter="member={0}"
    group-search-base="ou=groups"
    user-search-base="ou=people"
    user-search-filter="uid={0}" />
</authentication-manager>
```

Fonte: Adaptado de Alex e Taylor (2011).

Os componentes criados pela diretiva da Figura 30 irão realizar a busca a partir da DN `ou=people`, configurado no atributo `user-search-base` acrescida da raiz de busca definido na configuração do servidor LDAP, por exemplo: `ou=people,dc=springframework,dc=org`. A busca será realizada com o valor provido no atributo `user-search-filter`, onde `{0}` será substituído pela credencial provida pelo usuário. A mesma semântica vale para `group-search-base` e `group-search-filter`: por padrão, os resultados dessas buscas irão compor os papéis dos usuários, utilizados para autorização. A configuração da Figura 30 irá induzir o *framework* a criar os componentes ilustrados na Figura 31. Todos esses componentes implementam interfaces distintas e podem ser substituídos manualmente por outros componentes que obedecem aos mesmos contratos.

Figura 31 – Componentes criados para prover controle de acesso com LDAP.



Fonte: Elaborado pelo autor (2012)

Os componentes criados por padrão, todos localizados em sub-pacotes de `org.springframework.security.ldap`, são (Figura 31):

- LdapAuthenticationProvider*: componente *AuthenticationProvider* responsável por centralizar a delegação de operações de controle de acesso;
- DefaultLdapAuthoritiesPopulator*: implementação da interface *LdapAuthoritiesPopulator*, sua função é determinar os papéis de um usuário através de buscas em grupos;
- BindAuthenticator*: implementação de *LdapAuthenticator*, que tem como função a localização e autenticação de um usuário em um servidor LDAP;
- DefaultSpringSecurityContextSource*: utilizado por *BindAuthenticator*, essa implementação de *BaseLdapPathContextSource* é a responsável

por realizar a autenticação da aplicação no servidor LDAP (*binding*) e realizar os comandos de leitura e escrita;

- e) *FilterBasedLdapUserSearch*: responsável por efetuar a busca por filtragem em um *BaseLdapPathContextSource*. Essa classe implementa o contrato da interface *LdapUserSearch* e é também utilizada por *BindAuthenticator*.

Um cenário comum em ambientes corporativos é a utilização do servidor LDAP apenas como fonte de autenticação, deixando a busca por papéis em uma base de dados separada. Muitas vezes isso ocorre pela praticidade de manipulação e alteração da estrutura dessas informações em um Sistema Gerenciador de Banco de Dados (SGBD), ou pelo uso de estruturas legadas. Esse cenário é totalmente suportado pelo Spring Security através da substituição do componente *DefaultLdapAuthoritiesPopulator* por uma outra implementação da interface *LdapAuthoritiesPopulator* que efetua as buscas na base de dados. Spring Security provê a classe *UserDetailsServiceLdapAuthoritiesPopulator*, do pacote *org.springframework.security.ldap.authentication*. Trata-se de uma implementação de *LdapAuthoritiesPopulator* que delega a busca de papéis a uma instância de *UserDetailsService*. Essa instância pode ser, por exemplo, um *JdbcDaoImpl*, mencionado em 3.1, configurado para acessar a base de dados onde estão os papéis do usuário.

A Figura 32 ilustra a configuração de tal cenário, induzindo o *framework* a realizar um comportamento semelhante ao configurado na Figura 30, exceto que a busca por papéis do usuário não ocorrerão mais no servidor LDAP, e sim em um banco de dados acessado via JDBC pela instância de *JdbcDaoImpl*. O *bean* com o identificador *ldapProviderPapeisJdbc* é o *AuthenticatorProvider* que será responsável pelo controle de acesso. Seu identificador é informado como *authentication-provider*. Na construção desse *bean* são passadas as instâncias necessárias para seu funcionamento, como ilustrado na Figura 31. Entre elas, temos o *UserDetailsServiceLdapAuthoritiesPopulator* que, por sua vez, recebe um *JdbcDaoImpl* que acessa os dados através de um *DataSource* de identificador *beanDataSource*.

Figura 32 – Configurando autenticação via LDAP e autorização via SGBD.

```

<authentication-manager>
  <authentication-provider ref="ldapProviderPapeisJdbc" />
</authentication-manager>

<bean id="contextSource"
  class="org.springframework.security.ldap.
    DefaultSpringSecurityContextSource">
  <constructor-arg
    value="ldap://localhost:33389/dc=springframework,dc=org" />
</bean>

<bean id="ldapProviderPapeisJdbc"
  class="org.springframework.security.ldap.authentication.
    LdapAuthenticationProvider">
  <constructor-arg>
    <bean
      class="org.springframework.security.ldap.authentication.
        BindAuthenticator">
      <constructor-arg ref="contextSource" />
      <property name="userSearch">
        <bean id="userSearch"
          class="org.springframework.security.ldap.search.
            FilterBasedLdapUserSearch">
          <constructor-arg index="0" value="ou=people" />
          <constructor-arg index="1" value="(uid={0})" />
          <constructor-arg index="2" ref="contextSource" />
        </bean>
      </property>
    </bean>
  </constructor-arg>
  <constructor-arg>
    <bean
      class="org.springframework.security.ldap.authentication.
        UserDetailsServiceLdapAuthoritiesPopulator">
      <constructor-arg>
        <bean
          class="org.springframework.security.core.userdetails.jdbc.
            JdbcDaoImpl">
          <property name="dataSource" ref="beanDataSource" />
        </bean>
      </constructor-arg>
    </bean>
  </constructor-arg>
</bean>

```

Fonte: Elaborado pelo autor (2012)

3.5 AUTENTICAÇÃO EM UM SERVIÇO *SINGLE SIGN ON* COM *CENTRAL AUTHENTICATION SERVICE*

Na nomenclatura introduzida pelo protocolo CAS, o sistema que utiliza o servidor CAS é conhecido como serviço, e o usuário no *browser* utilizando o serviço é denominado cliente (MAZUREK, 2005, tradução nossa).

O fluxo básico de autenticação em *Central Authentication Service* (CAS) consiste das seguintes ações (MULARIEN 2011, tradução nossa):

- a) O usuário tenta acessar um recurso protegido do sistema;
- b) O mecanismo de segurança do sistema (ou serviço) direciona o cliente para o endereço do servidor CAS;
- c) Após apresentar suas credenciais e efetuar a autenticação, o servidor redireciona o *browser* do cliente de volta ao serviço, em uma URL específica de retorno, com um *ticket* único atribuído à requisição;
- d) O mecanismo de segurança do serviço envia uma requisição ao servidor CAS no intuito de validar o *ticket* apresentado. Caso a validade do *ticket* seja confirmada, o serviço procede normalmente com a autorização.

Nota-se, então, que o sistema não é mais responsável por efetuar regras de autenticação ou acessar o repositório de dados de autenticação. A configuração mínima em Spring Security para utilizar o servidor CAS é totalmente declarativa, e consiste em três etapas maiores:

- a) Declarar um componente `org.springframework.security.web.AuthenticationEntryPoint` que direcione o usuário não autenticado para o servidor CAS;
- b) Configurar o componente `AuthenticationManager` (introduzido em 2.4) para que utilize um `AuthenticationProvider` específico para CAS, que irá efetuar a validação do *ticket* junto ao servidor e popular as credenciais e papéis do usuário;
- c) Declarar um *filter* dentro da corrente de *filters* do Spring Security que será responsável por invocar o `AuthenticationProvider` declarado anteriormente.

Esta seção ilustra a configuração básica deste cenário em Spring Security. Configurações adicionais normalmente são feitas diretamente no servidor CAS, e não nas aplicações-serviço. Tais configurações estão fora do escopo deste trabalho.

3.5.1 Criação do *AuthenticationEntryPoint*

Componentes *AuthenticationEntryPoint* são responsáveis por realizar uma ação quando um usuário não autenticado tenta acessar um recurso protegido. Essa ação é manifestada em uma resposta do servidor, que pode ser um código de resposta HTTP, ou um redirecionamento. Um exemplo de *AuthenticationEntryPoint* é o *LoginUrlAuthenticationEntryPoint*, do pacote *org.springframework.security.web.authentication*, responsável por encaminhar o usuário para o formulário de *login*, em casos em que este tipo de autenticação é utilizado.

Spring Security provê a classe *org.springframework.security.cas.web.CasAuthenticationEntryPoint*, um *AuthenticationEntryPoint* que, se declarado, irá direcionar o usuário para o endereço de *login* do servidor CAS, definido na propriedade *loginUrl* (Figura 33). Além de *loginUrl*, essa classe deve ser alimentada com uma instância de *org.springframework.security.cas.ServiceProperties*, que possui atributos do servidor CAS e do serviço, utilizados no processo. O único atributo relevante para o cenário básico é o *service*, que deverá ser a URL de retorno que a central irá redirecionar o usuário após a autenticação. A Figura 33 ilustra um exemplo de declaração de um *CasAuthenticationEntryPoint* e de um *ServiceProperties*:

Figura 33 – Declaração de um *CasAuthenticationEntryPoint* e de um *ServiceProperties*.

```
<bean id="casAuthEntryPoint"
  class="org.springframework.security.cas.web.CasAuthenticationEntryPoint">
  <property name="loginUrl" value="http://centralcas.foo.com/" />
  <property name="serviceProperties" ref="casService" />
</bean>

<bean id="casService"
  class="org.springframework.security.cas.ServiceProperties">
  <property name="service"
    value="http://sistema1.foo.com/j_spring_cas_security_check" />
</bean>
```

Fonte: Elaborado pelo autor (2012)

Em seguida é necessário induzir o Spring Security a utilizar o *CasAuthenticationEntryPoint* declarado, através do atributo *entry-point-ref*, da diretiva *http*, introduzida em 2.4 (Figura 34):

Figura 34 – Informando que o novo *entry-point* será *casAuthEntryPoint* (instância de *CasAuthenticationEntryPoint*).

```
<http entry-point-ref="casAuthEntryPoint"></http>
```

Fonte: Elaborado pelo autor (2012)

Essa configuração é suficiente para que o usuário seja direcionado ao servidor CAS ao acessar um recurso protegido, que irá, por sua vez, retornar o usuário ao sistema com o *ticket* gerado, em caso de autenticação com sucesso. Todavia, não há ainda nenhum componente configurado para efetuar o processamento deste *ticket*. Logo, a autenticação realizada junto ao servidor não terá efeito, e o usuário será novamente direcionado ao endereço de autenticação.

3.5.2 Declaração do *filter* e do *AuthenticationProvider*

Spring Security provê um *filter* para interceptar o evento de retorno do servidor CAS e acionar o *AuthenticationProvider* para efetuar o processamento do *ticket*. Trata-se do *org.springframework.security.cas.web.CasAuthenticationFilter*, que deverá ser processado na posição *CAS_FILTER*. O posicionamento dos *filters* define em qual etapa da corrente um *filter* declarado manualmente será invocado (ALEX e TAYLOR, 2011, tradução nossa). A declaração de *filters* personalizados no Spring Security é feita com o elemento *custom-filter*, filho de *http* (Figura 35):

Figura 35 – Inserção de um *filter* na posição *CAS_FILTER*.

```
<http entry-point-ref="casAuthEntryPoint">
  <custom-filter ref="casAuthenticationFilter" position="CAS_FILTER"/>
</http>
```

Fonte: Elaborado pelo autor (2012)

A declaração da classe do *filter* se dá de forma simples, pois *CasAuthenticationFilter* necessita apenas de uma referência do *AuthenticationManager* da aplicação (Figura 36):

Figura 36 - Criação do *filter* *CasAuthenticationFilter*, populando sua propriedade *authenticationManager*.

```
<bean id="casAuthenticationFilter"
  class="org.springframework.security.cas.web.CasAuthenticationFilter">
  <property name="authenticationManager" ref="authenticationManager" />
</bean>
```

Fonte: Elaborado pelo autor (2012)

O componente *CasAuthenticationFilter* espera que um dos *AuthenticationProviders* seja capaz de processar um identificador especial gerado por ele, indicando ser uma autenticação em um servidor CAS. A classe *org.springframework.security.cas.authentication.CasAuthenticationProvider* é um *AuthenticationProvider* fornecido pelo Spring Security capaz de processar tal identificador. Esse componente deverá ser declarado como *bean* (Figura 38) e referenciado como um dos *authentication providers* do *authentication manager* (Figura 37):

Figura 37 – Configurando um *CasAuthenticationProvider* como um dos *authentication providers* da aplicação.

```
<authentication-manager alias="authenticationManager">
  <authentication-provider ref="casAuthenticationProvider"/>
</authentication-manager>
```

Fonte: Elaborado pelo autor (2012)

A declaração do *CasAuthenticationProvider* como *bean* envolve a configuração de várias propriedades, como exibido na Figura 38. As propriedades necessárias para a configuração mínima são:

- ticketValidator* - instância da classe *Cas20ServiceTicketValidator*, do pacote *org.jasig.cas.client.validation*, pertencente à API do CAS, responsável por efetuar a validação dos *tickets* junto ao protocolo de uma versão específica do CAS. Essa classe recebe como parâmetro do construtor o endereço da central CAS;
- serviceProperties* - instância de *ServiceProperties*, comentada anteriormente. Na Figura 38, trata-se da instância declarada com o identificador *casService*;
- key* - chave única utilizada internamente pelo *authentication provider*. Na Figura 38, seu valor é *chave-casAuthenticationProvider1*;
- authenticationUserDetailsService* - trata-se de uma classe que encapsula um componente *UserDetailsService* que será utilizado como fonte de dados de autorização. Aqui poderá ser utilizado qualquer implementação

dessa interface. Como informado em 3.1, componentes *UserDetailsService* são responsáveis por popular objetos *UserDetails* através de uma determinada fonte de dados, incluindo seus papéis. Na Figura 38, é encapsulado o *UserDetailsService* com identificador *userService*.

Figura 38 – Declaração de um componente *CasAuthenticationProvider*.

```
<bean id="casAuthenticationProvider"
    class="org.springframework.security.cas.authentication.
        CasAuthenticationProvider">
    <property name="ticketValidator">
        <bean
            class="org.jasig.cas.client.validation .Cas20ServiceTicketValidator">
            <constructor-arg value="http://centralcas.foo.com/" />
        </bean>
    </property>
    <property name="serviceProperties" ref="casService" />
    <property name="key" value="chave-casAuthenticationProvider1" />
    <property name="authenticationUserDetailsService">
        <bean
            class="org.springframework.security.core.userdetails.
                UserDetailsByNameServiceWrapper">
            <property name="userDetailsService" ref="userService" />
        </bean>
    </property>
</bean>
```

Fonte: Elaborado pelo autor (2012)

Com a configuração informada nessa seção, o fluxo de autenticação da aplicação é desviado para o servidor CAS, localizado em *http://centralcas.foo.com/*. A aplicação se torna ciente do evento de retorno do servidor e é capaz de efetuar a validação do *ticket*, recuperando as credenciais informadas pelo usuário junto ao servidor e efetuando autorização através da fonte de dados utilizada pelo *UserDetailsService*. Neste cenário, é necessário que exista um usuário de mesmo nome na fonte de dados utilizada pelo servidor CAS e na utilizada pelo *UserDetailsService*. Caso este usuário não exista na fonte de dados do *UserDetailsService*, o processo de autenticação irá falhar.

3.6 UTILIZANDO OS CÓDIGOS DE EXEMPLO

No Apêndice A são encontradas implementações dos mecanismos descritos nas seções 2.4, 3.1, 3.2, 3.3, 3.4 e 3.5. Trata-se de um projeto estruturado com Apache Maven (<http://maven.apache.org/>), que contém sub-projetos, onde cada um equivale a uma tecnologia validada com Spring Security. A saber:

- *ssconfig*: seção 2.4;
- *basedadoscredenciais*: seção 3.1;
- *servicelayersecurity*: seção 3.2;
- *openid*: seção 3.3;
- *ldap*: seção 3.4;
- *sso*: seção 3.5.

As implementações também estão disponíveis online através do endereço <https://github.com/bluefoot/springsecurity/> .

4 CONSIDERAÇÕES FINAIS

A aplicação e manutenção de segurança são fatores cruciais no *design* e desenvolvimento de sistemas corporativos. Uma área abrangente e mutante, onde periodicamente são introduzidos novas ameaças e mecanismos para mitigá-las. Nota-se que segurança deve ser introduzida em sistemas corporativos de forma holística, presente em todo o ciclo de construção e manutenção do sistema, fazendo parte do conjunto de requisitos da aplicação.

Durante o processo de realização deste trabalho, observou-se como Spring Security implementa de forma nativa as soluções para os problemas expostos, ao mesmo tempo que oferece enorme extensibilidade, sendo capaz de receber novos recursos ou adaptações às funcionalidades existentes. Os problemas de segurança em *software* selecionados para implementação cobrem uma gama de cenários e necessidades básicas encontradas no âmbito dos sistemas corporativos, e as soluções podem ser utilizadas na formação de estruturas de segurança em *software* de sistemas reais.

A tecnologia Spring Security possibilita estender esse trabalho com o estudo e implementação de recursos que não foram abordados neste momento, como:

- a) Autenticação em duas fases (*two-phase authentication*): consiste em efetuar autenticação utilizando pelo menos dois métodos de autenticação. Por exemplo: após informar algo que o ele sabe (apresentação de credenciais), o usuário deve utilizar algo que ele possui (uma chave física, ou uma segunda senha gerada por um aparelho que só ele possua) para prosseguir com a autenticação;
- b) Autenticação com certificados X.509: efetuar autenticação com o uso de certificados que comprovem a identidade dos usuários;
- c) Implementação de *Access Control Lists*: prover controle em nível de instâncias de objetos de domínio, baseado no estado dos objetos e nas credenciais dos usuários;
- d) Criptografia de senhas e uso de *salt*: além de alterar o algoritmo padrão de criptografia utilizado, realizar a adição de *salt*: um trecho de texto plano adicionado à senha do usuário – antes da escrita e leitura – provendo uma camada a mais de verificação;

- e) Autenticação com *logins* persistentes (*remember-me*): recurso que permite que usuários evitem efetuar *login* por um período de tempo, através do uso de *cookies*.

Por fim, este trabalho resultou na exploração de alguns dos principais conceitos de segurança em *software* e em um estudo de alguns recursos e características do *framework* Spring Security, através da implementação de soluções reais com esta tecnologia, servindo como um estudo básico de referência e avaliação do *framework* como solução de segurança em sistemas Java.

REFERÊNCIAS

ALEX, Ben; TAYLOR, Luke. **Spring Security Reference Documentation**. [S.l.: s.n.], 2011. Disponível em: <<http://static.springsource.org/spring-security/site/docs/3.0.x/reference/springsecurity-single.html>>. Acesso em: 14 out. 2011, 19:10:31.

ANDERSON, Ross. **Security engineering**: a guide to building dependable distributed systems. 2. ed. Indianapolis, IN: Wiley Publishing, 2008.

ANDRESS, Jason. **The basics of information security**. Waltham, MA: Syngress, 2011.

CBC NEWS. **PlayStation data breach deemed in 'top 5 ever'**. [S.l.: s.n.], 2011. Disponível em: <<http://www.cbc.ca/news/business/story/2011/04/27/technology-playstation-data-breach.html>>. Acesso em: 3 jul. 2011, 13:19:20.

CHURCHER, Clare. **Beginning SQL queries**. Berkeley, CA: Apress, 2008.

DASWANI, Neil; KERN, Christoph; KESAVAN, Anita. **Foundations of security**: what every programmer needs to know. Berkeley, CA: Apress, 2007.

DIERKS, Tim; RESCORLA, Eric. **The Transport Layer Security (TLS) Protocol**. [S.l.: s.n.], 2008. Disponível em <<http://tools.ietf.org/html/rfc5246>>. Acesso em: 26 set. 2011, 13:02:05.

ELMASRI, Ramez; NAVATHE, Shamkant. **Fundamentals of Database Systems**. 6. ed. London: Pearson, 2011.

EUROGAMER. **ICO confirms it will quiz Sony over PSN**. [S.l.: s.n.], 2011. Disponível em: <<http://www.eurogamer.net/articles/2011-04-27-ico-confirms-it-will-quizz-sony-over-psn>>. Acesso em: 5 out. 2011, 14:44:50.

FINANCIAL TIMES. **Hacking hits Sony's reputation**. [S.l.: s.n.], 2011. Disponível em: <<http://on.ft.com/sFG9o0>>. Acesso em: 5 out. 2011, 15:14:43.

HAMMER-LAHAV, Eran. **OAuth** - Introduction. [S.l.: s.n.], 2007. Disponível em: <<http://oauth.net/about/>>. Acesso em: 6 out 2011, 09:12:05.

HAMMER-LAHAV, Eran. **The OAuth 1.0 Protocol**. [S.l.: s.n.], 2010. Disponível em: <<http://tools.ietf.org/html/rfc5849>>. Acesso em: 6 out 2011, 12:31:10.

JASIG. **Central Authentication Service (CAS)**. [S.l.: s.n.], 2010. Disponível em: <http://www.jasig.org/sites/jasig.webchuckhosting.com/files/cas_brochure_v4.pdf>. Acesso em: 14 dez. 2011, 10:20:21.

JOHNSON, Rod et al. **Spring Framework Reference Documentation**. [S.l.: s.n.], 2011. Disponível em: <<http://static.springsource.org/spring/docs/3.1.0.RC1/spring-framework-reference/pdf/spring-framework-reference.pdf>>. Acesso em: 19 out. 2011, 21:20:01.

JUSTIA. **Johns v. Sony Computer Entertainment America LLC et al**. [S.l.: s.n.], 2011. Disponível em <<http://dockets.justia.com/docket/california/candce/3:2011cv02063/240051/>>. Acesso em: 5 out. 2011, 14:21:24.

KALALI, Masoud. **GlassFish security**. Birmingham, UK: Packt Publishing, 2010.

MACHACEK, Jan et al. **Pro Spring 2.5**. Berkeley, CA: Apress, 2008.

MAZUREK, Drew. **CAS Protocol**. [S.l.: s.n.], 2005. Disponível em <<http://www.jasig.org/cas/protocol>>. Acesso em 14 dez. 2011, 14:41:00.

MCGRAW, Gary. **Software Security: Building Security**. In: Boston, MA: Addison-Wesley Professional, 2006.

MCGRAW, Gary et al. **Processes to produce secure software v. 1, marc. 2004**. [S.l.: s.n.], 2004. Disponível em <http://www.cigital.com/papers/download/secure_software_process.pdf>. Acesso em: 26 jun. 2011, 14:15:40.

MEIER, J. D. et al. **Improving Web Application Security - Threats and Countermeasures**. Redmond, WA: Microsoft Press, 2003.

MULARIEN, Peter. **Spring Security 3**. Birmingham, UK: Packt Publishing, 2010.

OPENID FOUNDATION. **What is OpenID?**. [S.l.: s.n.], 2009. Disponível em: <<http://openid.net/get-an-openid/what-is-openid/>>. Acesso em: 27 jun. 2011, 13:45:20.

OPENID FOUNDATION. **OpenID Authentication 2.0**. [S.l.: s.n.], 2007. Disponível em: <http://openid.net/specs/openid-authentication-2_0.txt>. Acesso em: 27 jun. 2011, 14:47:10.

OPENID FOUNDATION. **Get an OpenID**. [S.l.: s.n.], 2008. Disponível em: <<http://openid.net/get-an-openid/>>. Acesso em: 28 jun. 2011, 16:30:50.

PC MAGAZINE. **PlayStation Hack to Cost Sony \$171M; Quake Costs Far Higher**. [S.l.: s.n.], 2011. Disponível em: <<http://www.pcmag.com/article2/0,2817,2385790,00.asp>>. Acesso em: 3 jul. 2011, 09:15:05.

REID, Paul. **Biometrics for Network Security**. [S.l.]: Prentice Hall PTR, 2003.

SHIREY, Robert W. **Internet Security Glossary**. [S.l.: s.n.], 2000. Disponível em <<http://www.ietf.org/rfc/rfc2828.txt>>. Acesso em: 09 set. 2011, 15:51:35.

SONY COMPUTER ENTERTAINMENT. **PlayStation Network and Qriocity Outage FAQ**. [S.l.: s.n.], 2011. Disponível em: <<http://blog.eu.playstation.com/2011/04/28/playstation-network-and-qriocity-outage-faq/>>. Acesso em: 3 jul. 2011, 09:10:50.

SPRING SOURCE. **Spring Security**. [S.l.: s.n.], 2008. Disponível em: <<http://static.springsource.org/spring-security/site/>>. Acesso em: 20 jun. 2011, 20:14:44.

TALBOT, John; WELSH, Dominic. **Complexity and Cryptography: an Introduction**. New York: Cambridge University Press, 2006.

THOMAS, Stephen A. **SSL & TLS Essentials: securing the web**. New York: Wiley Publishing, 2000.

THE OPEN GROUP. **Introduction to Single Sign-On**. [S.l.: s.n.], 1999. Disponível em: <http://www.opengroup.org/security/sso/sso_intro.htm>. Acesso em: 30 set. 2011, 17:36:41.

WALLS, Craig. **Spring in action third edition**. Shelter Island, NY: Manning, 2011.

WIKIPEDIA CONTRIBUTORS. **OpenID**. [S.l.: s.n.], 2011. Disponível em: <http://en.wikipedia.org/wiki/OpenID#OpenID_Providers>. Acesso em: 4 out. 2011,

20:39:54.

YE, Nong. **Secure Computer and Network Systems**. Indianapolis, IN: Wiley Publishing, 2008.

APÊNDICE A – Códigos de exemplo

Este CD contém os códigos de exemplo dos mecanismos desenvolvidos na seção 3, além do exemplo da configuração básica, descrito em 2.4.