Instituto de Computação
Universidade Estadual de Campinas

# Metodologia de Testes de Segurança para Análise de Robustez de Web Services pela Injeção de Ataques

**Marcelo Invert Palma Salas**

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Marcelo Invert Palma Salas e aprovada pela Banca Examinadora.

Campinas, 07 de Dezembro de 2012.

Prof[a]. Dr[a]. Eliane Martins (Orientadora)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

**FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP**
**Bibliotecária: Maria Fabiana Bezerra Muller – CRB8 / 6162**

Palma, Marcelo Invert Salas

M0000i Metodologia de Testes de Segurança para Análise de Robustez de Web Services por Injeção de Falhas/Marcelo Invert Palma Salas -- Campinas, [S.P.: s.n.], 2012.

Orientador: Eliane Martins

Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Tolerância a falhas (Computação). 2. Web Services. 3. WS-Security. 4. Engenharia de software. 5. Testes de Robustez. 6. Ataques a Web Services. 7. Testes de Segurança. 8. Injeção de Falhas. 9. Ataques de injeção. 10. Negação de serviços. I. Martins, Eliane. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

(mfbm/imecc)

Título em inglês: Security Testing Methodology for Robustness Analysis of Web Services by Fault Injection

1. Fault Tolerance (Computing). 2. Web Services. 3. WS-Security. 4. Software Engineering. 5. Robustness Testing. 6. Attacks on Web Services. 7 Security Testing. 8. Fault Injection. 9. Injection Attacks. 10. Denial of service.

Área de concentração: Tolerância a Falhas

Titulação: Mestre em Ciência da Computação

Banca examinadora: Profa. Dra. Eliane Martins

Prof. Dr. Paulo Lício (IC-UNICAMP)

Prof[a]. Dr[a]. Regina Lucia de Oliveira Moraes (Faculdade de Tecnologia - UNICAMP)

Data da defesa: 07/12/2012

Programa de Pós-Graduação: Mestrado em Ciência da Computação

Instituto de Computação
Universidade Estadual de Campinas

# Metodologia de Testes de Segurança para Análise de Robustez de Web Services por Injeção de Falhas

**Marcelo Invert Palma Salas**[1]

Dezembro de 2012

**Banca Examinadora:**

- Eliane Martins (Orientadora)
- Regina Lucia de Oliveira Moraes
  Faculdade de Tecnologia – UNICAMP
- Paulo Lício
- Ana Cristina Vieira de Melo
- Ricardo Dahab

[1] Suporte financeiro de: Bolsa do CNPq 2009-2010, Projeto Robust Web 2009-2012 e Universidade Estadual de Campinas 2009-2012.

# Resumo

Devido a sua natureza distribuída e aberta, os Web Services geram novos desafios de segurança da informação. Esta tecnologia Web, desenvolvida pela W3C e OASIS, é susceptível a ataques de injeção e negação de serviços. Desta forma, o atacante pode coletar e manipular informação para procurar vulnerabilidades nos serviços. Nesse estudo analisamos o uso do injetor de falhas (IF) WSInject, para emular ataques com testes de segurança nos Web Services. A motivação para o uso de um injetor de falhas, ao invés do uso de *vulnerabilities scanners* que são comumente usados na prática para testar a segurança, foi permitir melhor cobertura dos ataques. Em um estudo preliminar, usando um vulnerability scanner não comercial, foi possível determinar: (i) os serviços, bem como seus parâmetros e suas operações que seriam mais interessantes de utilizar durante a injeção de falhas, por terem sido os que apresentaram maior número de vulnerabilidades; (ii) um conjunto de regras para analisar os resultados dos testes de segurança. Esses resultados preliminares serviram de guia para os testes usando o injetor de falhas. As falhas foram injetadas em Web Services reais, sendo que alguns implementaram mecanismos de segurança de acordo com o padrão Web Services Security (WS-Security), como credencias de segurança (Security Tokens).

# Abstract

Due to its distributed and open nature, the Web Services give rise to new information security challenges. This technology, standardized by W3C and OASIS, is susceptible to both injection and denial of services (DoS) attacks. In this way, the attacker can collect and manipulate information in search of Web Services vulnerabilities. In this study we analyses the use of the WSInject fault injector, in order to emulate attacks with security tests on Web Services. The proposed approach makes use of WSInject Fault Injector to emulate attacks with Security Testing on Web Services. The motivation for using a fault injector, instead of vulnerabilities scanners, which are commonly used in practice for security testing, was to enable better coverage of attacks. In a preliminary study, using a non-commercial vulnerability scanner, it was possible to determine: (i) the Web Services to be tested as well as its parameters and operations more interesting to use during fault injection, by presenting the highest number of vulnerabilities; and (ii) a set of rules to analyze the results of security testing. These preliminary results served as a guide for the tests using the fault injector. The faults have been injected into real Web Services, and some of them have security mechanisms implemented, in compliance with the Web Services Security (WS-Security) with Security Tokens.

# Agradecimentos

A Deus, por me dar saúde e forças para finalizar o Mestrado. A meus pais, Felipe e Cecilia, e meus irmãos, Esven e Nadhia, por sempre me apoiarem, não só no desenvolvimento deste trabalho, mas em todas as decisões que me fizeram chegar até aqui. A minha família, avós, tios e primos pelo apoio incondicional.

A minha orientadora, professora Eliane Martins, pela paciência durante a orientação e dedicação deste trabalho, por me incentivar a ser um melhor pesquisador. Ao professor Marco Vieira pelas explicações, conselhos e apoios no desenvolvimento deste trabalho. A Terry Ingoldsby de Amenaza por sua confiança para me ajudar a desenvolver a árvore de ataques. Ao CNPq pela oportunidade de fazer o mestrado no Instituto de Computação da Universidade Estadual de Campinas. A Anderson Morais, André Valenti e William Maja por ser a base desta pesquisa.

Não poderia deixar de agradecer aos familiares e amigos que foram parte vital para finalizar o mestrado. A minha namorada Gabriela Batista por seus conselhos e apoio, *a Carlos Morato, Sergio Toro e Rommel Morón por su apoyo para postularme a becas para el Brasil*. Ao professor Paulo Lício por sua confiança para continuar meus estudos no IC. E a todos meus amigos que foram parte essencial da minha vida acadêmica durante os anos de mestrado e o inicio do doutorado, por sua amizade.

Por fim, agradeço aos professores e funcionários do Instituto de Computação que sempre me aconselharam e ensinaram. Ao Instituto de Computação e à UNICAMP por ser minha *alma mater* e me tornar um melhor profissional, para voltar a minha pátria amada Bolívia, orgulho de ser mestre da UNICAMP.

# Glossário

O documento tem base na terminologia da Norma Brasileira ABNT NBR ISO/IEC 27001 [1] e no Glossário da W3C[2] para Web Services [2]. A seguir descrevemos diversas terminologias técnicas utilizadas na presente dissertação.

**Administração de segurança:** configuração da segurança ou implantação de sistemas e aplicações que permitam administrar e habilitar o domínio de segurança.

**Ameaça:** causa potencial de um incidente indesejado, que pode resultar em danos a um sistema ou organização.

**Assinatura Digital:** um valor calculado com um algoritmo criptográfico e anexado a um conjunto de dados de tal forma que os destinatários dos dados podem usar a assinatura para verificar os dados, a origem e a integridade.

**Ataque**: é uma tentativa de destruir, expor, alterar, inutilizar, roubar, ganhar acesso ou fazer uso não autorizado de um ativo, através da qual um atacante, i.e. um usuário interno ou externo, intencionalmente viola uma ou mais propriedades de segurança do sistema (equivalente a uma tentativa de intrusão).

**Ativo:** qualquer coisa que tenha valor para a organização. Nota: Tem muitos tipos de ativos, incluindo: Informação, software, físico (hardware), serviços, pessoas e intangíveis (reputação e imagem).

**Autenticidade:** propriedade de ser "genuíno", i.e. para uma mensagem autenticidade é equivalente a integridade do conteúdo da mensagem, de sua origem e possivelmente de outras informações , como horário de emissão, nível de classificação, entre outros (integridade da meta-informação).

**Conexão:** a camada de transporte de circuito virtual estabelecida entre dois programas com a finalidade de comunicação.

**Confidencialidade:** propriedade que garante que usuários não-autorizados não obtenham conhecimento de informação sensível, i.e. ausência de revelação não-autorizada de informação.

**Controle de Acesso**: meios para garantir que o acesso a ativos estão autorizados e limitados a indivíduos com base nos requisitos de negocio e da segurança.

[2]W3C: A World Wide Web Consortium (W3C) é uma comunidade internacional qual as organizações estão associadas, uma equipe de trabalho de tempo integral para desenvolver padrões web.

**Credenciais:** os dados que são transferidos para estabelecer uma identidade principal solicitada.

**Criptografia:** transformação criptográfica de dados (chamado de “plaintext ou texto plano”) em um formulário (chamado “ciphertext ou texto cifrado”), que esconde o significado original dos dados para impedir que ele seja conhecido ou usado. Se a transformação é reversível, o processo de reversão correspondente é chamado de “descodificação” ou “decifrado”, que é uma transformação que restaura os dados criptografados para o seu estado original.

**Disponibilidade**: é prontidão para o uso imediato.

**Direitos de acesso:** a descrição do tipo de interações e autorizações que um sujeito pode ter com um recurso. i.e. incluem ler, escrever, executar, adicionar, modificar e apagar.

**Eficácia:** medida em que as atividades planejadas são realizadas e os resultados planejados alcançados.

**Eficiência:** relação entre os resultados alcançados e como os recursos foram utilizadas.

**Evento:** ocorrência de um determinado conjunto de circunstâncias.

**Intrusão:** é uma falha operacional intencionalmente maliciosa, no domínio do software, originada externamente ao sistema, resultante de um ataque bem-sucedido que explorou uma vulnerabilidade.

**Impacto:** mudanças adversas dos objetivos de negócio realizados.

**Integridade:** é a ausência de alteração imprópria de estado do sistema, i.e. a prevenção de modificação ou supressão não-autorizada de informação.

**Mecanismo de segurança:** um processo (ou um dispositivo que incorpora um processo) que pode ser usado em um sistema para implementar um serviço de segurança que é fornecido pelo mesmo sistema ou outro.

**Não-repúdio:** corresponde à disponibilidade e autenticidade de algumas meta-informações da mensagem, como a identidade do criador e, possivelmente, o horário de criação para evitar o repúdio de mensagem por parte do receptor.

**Política de segurança:** um conjunto de regras e práticas que especificam ou regulam a forma como um sistema ou organização fornece os serviços de segurança para proteger os recursos.

**Protocolo:** um conjunto de regras formais que descrevem como transmitir dados, especialmente em uma rede.

**Proxy:** um agente que transmite uma mensagem entre um agente solicitante e um agente prestador, é usado nos Web Services para ser o solicitador.

**Responsabilidade:** conhecer e gerenciar as pessoas que têm acesso a informações e recursos.

**Risco:** combinação da probabilidade de um evento e sua consequência.

**Análise de risco:** uso sistemático de informações para identificar fontes e estimar o risco. A análise de risco fornece uma base para a avaliação de risco, o tratamento de risco e de aceitação de risco.

**Avaliação de risco:** processo de análise e comparação de risco estimado com base em critérios de risco identificados para determinar a importância do risco.

**Critérios de risco:** termos de referência pelo qual o significado de risco é avaliado.

**Risco a Segurança da informação**: ameaça potencial que explora uma vulnerabilidade de um ativo, ou um grupo de ativos e, assim, prejudicar a organização.

**Sessão:** a interação duradoura entre entidades do sistema, muitas vezes envolvendo um usuário, caracterizada pela manutenção de alguns estados de interação para a duração da interação.

**Segurança da informação**: preservação da confidencialidade, integridade e disponibilidade da informação; adicionalmente, outras propriedades, tais como autenticidade, responsabilização, não-repúdio e confiabilidade.

**Vulnerabilidade:** é a falha de um ativo ou controle que pode ser explorado por uma ameaça.

# Lista de Acrônimos e Siglas

| | |
|---|---|
| **AIF** | **A**mbiente de Injeção de Falhas |
| **DoS** | *Denial of Services* |
| **ECA** | **e**vento-condição-acaõ |
| **F-Measure** | F1 ou Média Harmônica |
| **FN** | Falso Negativo |
| **FP** | Falso Positivo |
| **FT** | *Fuzz Testing* |
| **FS** | Falha de Software |
| **HTTP** | *Hypertext Transfer Protocol* |
| **HTTPS** | HyperText Transfer Protocol Secure |
| **IDE** | *Integrated Development Environment* |
| **IDS** | **I***ntrusion Detection Systems* |
| **IF** | Injetor de Falhas |
| **P** | Precisão |
| **QoS** | *Quality of Services* |
| **R** | *Recall* |
| **SAE** | Script de Ataque Executável |
| **SAG** | Script de Ataque Genérico |
| **SLA** | S*ervice-Level Agreement* |
| **SSL** | *Security Socket Layer* |
| **SOA** | *Service-Oriented Architecture* |
| **TP** | Testes de Penetração |
| **UBR** | *Universal Business Registry* |
| **UDDI** | *Universal Description, Discovery and Integration* |
| **UML** | *Unified Modeling Language* |
| **VNE** | Vulnerabilidade Não Encontrada |
| **VE** | Vulnerabilidade Encontrada |
| **XML** | *Extensible Markup Language* |

**XML-Enc** *XML Encrytion*

**XML-Sig** *XML Signature*

**WSDL** *Web Services Description Language*

**WSIT** *Web Service Interoperability Technologies*

**WSLA** *Web Service-Level Agreements*

**WS** Web Services

**WSS** WS-Security ou Web Services Security

**XSS** Cross-site Scripting

**XSRF** *Cross-site Request Forger*

# Sumário

# Lista de Tabelas

# Lista de Figuras

# Capítulo 1. Introdução

Os Web Services (Serviços Web) são aplicações de software modulares que podem ser descritos, publicados, localizados e invocados sobre uma rede, como a *World Wide Web*. Apesar de ter maior conectividade, flexibilidade e interoperabilidade, eles são mais suscetíveis a riscos de segurança, devido sua natureza distribuída e aberta.

As características dos Web Services que os tornam mais atrativos, tais como maior acesso a dados e conexões dinâmicas entre aplicações, geram novos desafios para manter a segurança da informação. Além das ameaças tradicionais, tem-se que contar com as novas ameaças, associadas às tecnologias e serviços. Um exemplo são os chamados ataques de injeção (*injection attacks*) e ataques de negação de serviços (*Denial of Services Attack*s), dois dos mais explorados em 2010, segundo a *Open Web Application Security Project*[3] (**OWASP Top 10**). Eles também estão dentro da lista dos 25 ataques mais perigosos, publicados em 2011 por CWE e SANS[4]. A proteção contra ataques desse tipo requer mecanismos de segurança aplicados à mensagem SOAP, para garantir seu transporte.

Um estudos em [3] afirmou que 70% das vulnerabilidades filtradas pelo uso de *firewall*, durante a última década, voltaram a ser reabertas pelo uso de aplicações de Web Services.

Motivados pelos problemas de segurança em Web Services, a IBM, Microsoft e VeriSign, trabalharam em um padrão que garantisse a confidencialidade, integridade e autenticação para esta tecnologia distribuída. Em abril de 2004 o padrão ***WS-Security*** foi publicado pela OASIS e em sequência as especificações para criptografia de documentos XML, assinaturas digitais e credenciais de segurança, respectivamente XML Encryption, XML Signature e Security Tokens, entre outras [4]. WS-Security introduz um grande número de novos padrões e tecnologias, mais resistentes a ataques de segurança do que outros sistemas de proteção de informação.

Com o objetivo de analisar a robustez dos Web Services frente a diversos tipos de ataques, nossa abordagem usa injetores de falhas (IF) para testar a segurança na troca de mensagens entre *Web Services* e seus clientes, ao invés de usar *vulnerability scanners*, com o

[3] https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project
[4] http://cwe.mitre.org/top25/index.html#Listing

objetivo de obter: (**i)** maior cobertura de ataques; e (**ii)** menor número de falsos positivos. Com relação a (**i)**, o uso de um injetor permite emular diversos tipos de ataques, variando os parâmetros e dados injetados. Em (**ii)** utilizamos um conjunto de regras, baseado em várias fontes. Também analisamos a robustez de Web Services que usam o padrão de segurança WS-Security.

## 1.1 Proposta e Contribuições

Os estudos mostram uma clara predominância dos problemas de Web Service, ainda com a utilização do padrão WS-Security, como perda e modificação da informação, o qual acarreta em maior probabilidade de falhas.

Nossa proposta analisa a robustez dos Web Services, usando a técnica de Injeção de Falhas que emula ataques contra estes, introduzindo falhas nas requisições de mensagens SOAP. Em alguns casos, os Web Services utilizam o padrão de segurança WS-Security. Para sua aplicação são considerados dois cenários: **1)** Injeção de Falhas a Web Services sem proteção de WS-Security; e **2)** Injeção de Falhas a Web Services com proteção de WS-Security, conforme a abordagem de Morais e Martins em [5] para remoção de vulnerabilidades em protocolos de segurança, baseada no modelo de Árvore de Ataques com o objetivo de detectar falhas e vulnerabilidades.

O banco de dados de ataques disponibiliza vários tipos de ataques conhecidos, cujos efeitos destes geram vulnerabilidades nos Web Services. Coletadas de diversas fontes como a OWASP Top 10, livros, Internet e outros, o banco de dados pode ser atualizado tão logo novos ataques sejam reportados. Sua informação pode ser utilizada para testar novos protocolos e padrões de segurança para tecnologias Web.

Os testes de segurança podem ser utilizados para validar um sistema tolerante a falhas, auxiliando na remoção e prevenção das mesmas, minimizando suas ocorrências e severidades, melhorando a dependabilidade de Web Services que utilizam WS-Security para sua proteção.

Para mostrar sua utilidade, realizamos um conjunto de experimentos organizado em duas fases. Na fase de pré-análise aplicamos o *Vulnerability Scanner* (**VS**) **soapUI** [6] a 69 Web Services reais, a fim de obter um conjunto de dados. Essa informação, baseada em resultados de outras pesquisas, ajudaram a desenvolver um conjunto de regras para determinar se um ataque foi bem sucedido e responder a outras questões de segurança. Já na segunda fase, aplicamos o

injetor de falhas (**IF**) WSInject[5], desenvolvido pelo grupo de pesquisa Robust Web [7] a 10 Web Services selecionados da anterior fase, sendo que 5 deles usam o padrão WS-Security e os outros não. Esta injetor de falhas emula ataques e injeta ataques contra Web Services.

Para sua aplicação selecionamos um conjunto de ataques (ver Seção 3.2) com a premissa de avaliar a segurança nos serviços. Selecionamos duas categorias de ataques: Ataques de Injeção e Negação de Serviço, utilizando tanto o VS soapUI quanto o IF WSInject. Os experimentos são executados com cenários de ataque, derivados dos descritos, com o intuito de violar as propriedades de segurança do padrão WS-Security.

O objetivo principal do projeto é desenvolver uma metodologia para aplicar testes de robustez por Injeção de Falhas em Web Services. Os testes confirmam que a proposta detectou eficientemente vulnerabilidades no padrão, além de descobrir novas vulnerabilidades e variações dos atuais cenários de ataques usados para testar a qualidade de segurança.

As principais contribuições deste trabalho são:

- Fazer um levantamento das vulnerabilidades e técnicas de teste de robustez para Web Services com o objetivo de determinar o conjunto de ataques a ser aplicados.
- Criar um conjunto de regras para analisar se um ataque foi bem sucedido e que responda a outras questões de segurança.
- Aplicar as regras ao VS soapUI para medir sua precisão e recuperação (*recall*) na detecção de ataques.
- Injetar falhas com WSInject a Web Services que utilizem o padrão WS-Security ou não, para tornar os testes de segurança o mais automatizado possível.
- Determinar quais sãos os principais ataques que revelam a maior quantidade de vulnerabilidades nos Web Services.
- Comparar o desempenho do IF WSInject para Ataques de Injeção e Negação de Serviços com o VS soapUI.

## 1.2 Organização

A dissertação está organizada da seguinte forma:

- O capítulo 2 apresenta uma visão geral da Arquitetura Orientada a Serviços e as plataformas para desenvolvimento de software baseadas nesta arquitetura.

[5] http://robustweb.ic.unicamp.br/

- O capítulo 3 trata dos principais conceitos de segurança na Arquitetura Orientada a Serviços que são necessários para entender a descrição da proposta, além de mencionar, em certo grau de profundidade, as principais vulnerabilidades em Web Services e metodologias de modelagem de ataques.
- O capitulo 4 relata as técnicas de detecção de vulnerabilidades e conceitos de injeção de falhas, testes de robustez e segurança. Também é apresentado o estado da arte da área, incluindo os desafios de pesquisas e desenvolvimento.
- O capítulo 5 contém a metodologia de ataques utilizada na proposta deste trabalho, cuja aplicação é feita nos capítulos a seguir.
- No capítulo 6 a metodologia é ilustrada em um estudo de caso.
- No capítulo 7 realizamos experimentos, em fase de pré-análise com o VS soapUI, que serviram para inferir as regras a fim de avaliar os resultados e selecionar os Web Services para as próximas fases.
- O capítulo 8 analisa os resultados dos testes experimentais usando o estudo de caso, com o injetor de falhas WSInject, demonstrando a viabilidade da proposta.
- O capítulo 9 fecha o trabalho relatando os resultados e contribuições que foram alcançadas e apresenta algumas direções para trabalhos futuros.

# Capítulo 2. Arquitetura Orientada a Serviços

O presente capítulo descreve o paradigma de computação e arquitetura orientadas a serviços, juntamente com as tecnologias atuais que podem ser consideradas nestas instâncias. Também apresentamos brevemente as ferramentas e plataformas para desenvolvimento de software orientado a serviços.

## 2.1 Visão Geral da Arquitetura Orientada a Serviços

A Computação Orientada a Serviços é o paradigma que utiliza serviços como elementos fundamentais para o desenvolvimento de aplicações [8]. A Arquitetura Orientada a Serviços (*Service-Oriented Architecture – SOA*) usa esse conceito para prover a interoperabilidade e fraco acoplamento entre serviços, através da definição de interfaces neutras e da utilização de protocolos de transporte padronizados.

Os serviços são unidades lógicas de software, definidos por componentes abertos e auto-descritivos, que podem encapsular um simples método ou um grande processo envolvendo múltiplos colaboradores, e possuem a capacidade de realizar tarefas que representam uma funcionalidade, do ponto de vista das entidades provedoras e requerentes [9]. Eles suportam a composição de aplicações distribuídas de forma rápida e com baixo custo, utilizando um protocolo padrão e aberto para comunicação (e.g. *Hypertext Transfer Protocol – HTTP*). Isto oferece uma infra-estrutura de computação distribuída tanto interna quanto externamente.

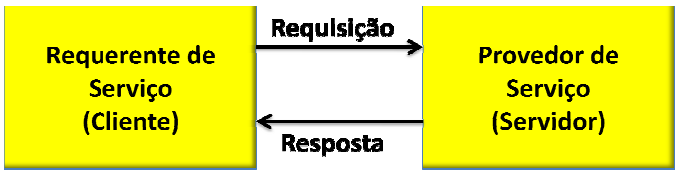



Figura 2.1 Visão simplificada da Arquitetura Orientada a Serviços.

A arquitetura SOA pode ser vista como uma aplicação da arquitetura cliente-servidor em um novo contexto, i.e. um requisitante de serviço (cliente) envia uma mensagem a um provedor (servidor), que por sua vez envia uma resposta, podendo ser a informação requisitada ou a confirmação de que alguma ação foi tomada, conforme ilustra a Figura 2.1. A transação pode ser síncrona ou assíncrona, que é independente do protocolo de transporte utilizado.

Para possibilitar dinamismo na localização dos serviços, SOA prevê a utilização de um mecanismo de registro (*registry*) e descoberta de serviços (*discovery*) que resulta em três papéis bem estabelecidos na arquitetura:

- **Provedor de serviços:** Entidade que fornece serviços acessíveis pela rede. Responsável pelo gerenciamento, manutenção, publicação do serviço.
- **Cliente de serviços:** Entidade que usa um serviço fornecido por um provedor. Para conhecimento das funcionalidades, localização e requisitos específicos do serviço (e.g. protocolos) realiza consultas a uma agência de registro de serviços.
- **Agência de registro de serviços:** Entidade que mantém registro dos serviços disponíveis, descrevendo funcionalidades, localização e protocolos utilizados pelo serviço, entre outras informações.

A interação entre estas três entidades se dá em quatro passos: **1)** o provedor publica o serviço na agência de registro, fornecendo informações relevantes quanto a sua utilização, como localização, interfaces de interação, protocolos de transporte e segurança, aplicações utilizadas, formato dos dados requeridos e parâmetros de qualidade de serviço; **2)** o cliente solicita à agência de registro uma busca de serviços que satisfaça certos parâmetros; **3)** a agência responde ao cliente com uma lista, possivelmente nula, com serviços que satisfazem os parâmetros recebidos; **4)** baseado em algum critério de avaliação, o cliente escolhe o serviço que mais lhe convém e envia uma requisição ao respectivo provedor, que por sua vez envia o resultado.

O uso da arquitetura SOA traz várias vantagens ao processo de desenvolvimento de software. Entre elas destacam-se: fraco acoplamento, interoperabilidade, composição, reusabilidade, alta granularidade e ubiquidade [10].

A tecnologia atual que possui mais conformidade com a arquitetura e o paradigma de computação orientados a serviços é o Web Service. Uma boa definição de **Web Services** [11]:

“Um **Web Service** permite a interoperabilidade entre aplicações de software desenvolvidas em linguagens de programação diferentes e executadas sobre qualquer plataforma na camada de Transporte do modelo OSI.” Ele é composto por um conjunto de serviços e padrões conhecido como Pilha de Padrões dos Web Services (*Web Services Protocol Stack*), ilustrada na Figura 2.2 [12].

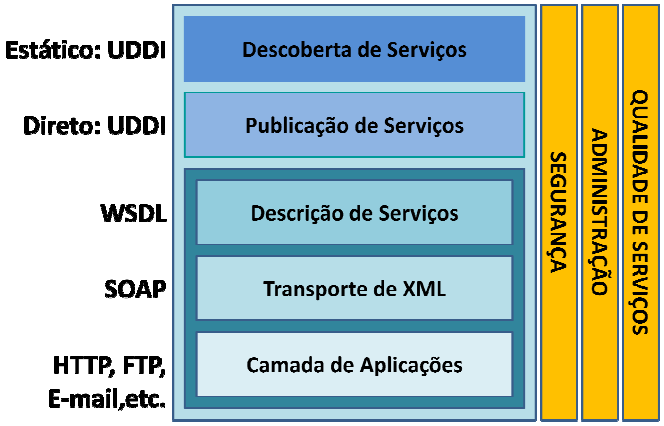



Figura 2.2 Pilha de padrões de Web Service.

Os Web Services se baseiam em quatro principais padrões [1]:

**XML** (*Extensible Markup Language*) é uma metalinguagem extensível de etiquetas desenvolvidas pela W3C. É um subtipo da Linguagem Padronizada de Marcação Genérica (*Standard Generalized Language – SGML*), capaz de descrever diversos tipos de dados. Seu propósito é facilitar o compartilhamento de informações através da Internet [9].

**SOAP,** originalmente chamado *Simple Object Access Protocol*, é um protocolo para troca de informações estruturadas em uma plataforma distribuída, utilizando tecnologias baseadas em XML.

**WSDL** (*Web Services Description Language*) é uma linguagem baseada em XML, utilizada para descrever Web Services. Trata-se de um documento escrito em XML que, além de descrever o serviço, especifica como acessá-lo e que operações ou métodos estão disponíveis.

**UDDI** (*Universal Description, Discovery and Integration*) especifica um método para publicar e descobrir diretórios de serviços em uma arquitetura orientada a serviços.

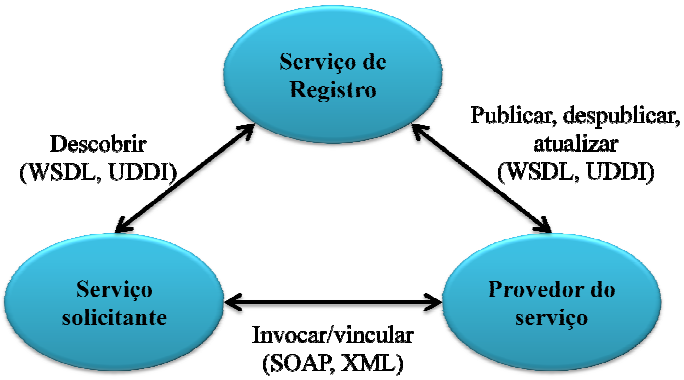



Figura 2.3 Atores e operações de Web Services.

A Figura 2.3 descreve um modelo básico da interação entre os atores envolvidos no consumo de Web Services, juntamente com o padrão de comunicação utilizado [4]. O “Provedor de serviço” é o titular, ele publica o Web Service no “Serviço de registro” usando o WSDL/UDDI. O “Serviço solicitante” descobre o serviço usando também o WSDL/UDDI no “Servidor do registro” e invoca o Web Service ao provedor, que por sua vez vincula o serviço fazendo um *Three-way Handshake*.

A Figura 2.4 ilustra três protocolos empilhados. **HTTP** serve de transporte de dados na camada de aplicação, enquanto **XML** é a camada utilizada como o padrão de formato de dados para a implementação tanto do serviço de interface como do intercâmbio de dados. Finalmente, UDDI, WSDL e SOAP são os protocolos responsáveis pela descoberta, descrição e invocação, respectivamente [11].

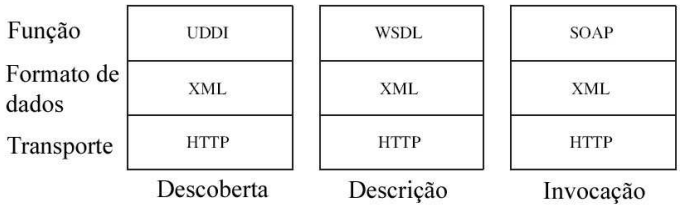



Figura 2.4 Pilha de protocolos de Web Service.

## 2.2 Plataformas Disponíveis para Web Services

O forte interesse de grandes empresas no desenvolvimento de aplicações baseadas na Arquitetura Orientada a Serviço, e mais especificamente na linguagem BPEL[6], impulsionou a criação de plataformas de desenvolvimento para Web Services. Encontram-se disponíveis soluções comerciais e de código aberto, destacando três versões na atualidade:

- **NetBeans com servidor Glassfish e biblioteca Metro:** NetBeans[7] é um ambiente de desenvolvimento integrado (IDE) gratuito e de código aberto para desenvolvedores de software. O IDE é executado é multiplataforma e oferece aos desenvolvedores ferramentas necessárias para criar aplicativos. **GlassFish[8]** é um servidor de aplicação de fonte aberta para a plataforma Java EE. **Metro**

[6] *Business Process Execution Language (BPEL)*.
[7] http://netbeans.org/
[8] http://glassfish.java.net/

**Glassfish**[9] é uma pilha de código aberto para Web Services, parte do projeto GlassFish, que pode ser usado em uma configuração isolada. O subsistema WSIT (*Web Service Interoperability Technologies*) da biblioteca Metro é um conjunto de implementações abertas dos padrões de Web Services para suportar recursos empresariais. Além de segurança, mensagens confiáveis e transações atômicas, Metro inclui WS-Security, WS-Trust e WS-SecurityPolicy entre outras.

- **Servidor Apache com biblioteca Axis[10]:** Axis é uma máquina para Web Services e funciona como aplicativo de servidor. "Apache Rampart[11]" é o módulo de segurança do Axis2. Ele protege mensagens SOAP de acordo com os padrões da pilha de WS-Security.
- **Web Services Enhancements – WSE V3.0[12]** é uma extensão para a plataforma de desenvolvimento Microsoft .NET Framework, que inclui um conjunto de bibliotecas que implementam os padrões de Web Services, principalmente em áreas como segurança, mensagens confiáveis e envio de arquivos anexos.

[9] https://metro.dev.java.net/discover/
[10] http://ws.apache.org/axis/
[11] http://axis.apache.org/axis2/java/rampart/
[12] http://www.microsoft.com/download/en/details.aspx?id=14089

# Capítulo 3. Segurança na Arquitetura Orientada a Serviços

Neste capítulo apresentamos os principais conceitos relacionados à dependabilidade e segurança na Arquitetura Orientada a Serviços, que são utilizados ao longo desse trabalho, principalmente na metodologia proposta no capítulo 5 e na aplicação da metodologia nos capítulos 6, 7 e 8.

Na Seção 3.1 são descritos os conceitos de dependabilidade, falha, erro, defeito, juntamente com a definição de segurança e suas propriedades. Ataques contra Web Services e as propriedades de segurança violadas por eles são descritos na Seção 3.2. Na Seção 3.3 detalhamos o protocolo WS-Security. Na Seção 3.4 é apresentada a metodologia de árvore de ataques usada neste proposta para representar os ataques contra o padrão WS-Security.

## 3.1 Dependabilidade e Terminologia Relacionada à Segurança

Segundo Laprie *et al*. [13], a **dependabilidade** (*dependability*) é "a capacidade de oferecer um serviço que se pode confiar plenamente e fornece a habilidade ao serviço para evitar falhas que são mais frequentes e graves do que é aceitável". Esta definição em [13] nos proporciona o critério para decidir se o serviço é confiável. A dependabilidade é um conceito integrado que compreende os seguintes atributos básicos descritos por Lopez [14]: **i) confiabilidade** (*reliability)* é a capacidade de o sistema estar operacional dentro das especificações e condições durante um período de tempo; **ii) Inocuidade** *(safety)* é probabilidade do sistema de não causar catástrofes ambientais, perdas de vidas humanas ou de valores financeiros em demasia; **iii) Manutenção** (*maintainability)* é a habilidade de estar sujeito a reparos e modificações. Os atributos de dependabilidade como confiabilidade, integridade e disponibilidade são descritos como parte dos atributos de segurança da informação.

No desenvolvimento de um sistema com os atributos de dependabilidade, o conjunto de métodos e técnicas que devem ser empregados, estão classificados em: **i) prevenção de falhas**, impede a ocorrência ou introdução de falhas; **ii) tolerância a falhas,** fornece o serviço esperado, mesmo na presença de falhas; **iii) validação**, verifica a presença de falhas e as remove; **iv) previsão de falhas**, estimativas sobre presença de falhas e suas consequência [15].

Um **defeito** (*failure*) de sistema é um evento que ocorre quando o serviço oferecido diverge do serviço correto, ou seja, o serviço não desempenha as funções como deveria. Um **defeito** é então uma transição do serviço correto para o serviço incorreto, i.e. a não implementação correta de uma função do sistema. Um sistema pode falhar quando ele não obedece à especificação ou porque a especificação não descreve, de maneira satisfatória, a função do sistema que implementa o serviço. Um **erro** (*error*) é a parte do estado do sistema que pode causar um subsequente defeito. Um **defeito** (*failure*) ocorre quando o erro alcança a interface do serviço e altera-o. Uma **falha** (*fault*) é a causa confirmada ou teórica de um erro. Uma falha é ativada quando ela produz um erro, se não ela continua inativa. As maneiras nas quais o sistema pode falhar são seus **modos de defeito** (*failure modes*), que são classificados de acordo com a severidade dos defeitos e descritos na Seção 4.1.

As **falhas** nos sistemas podem ser classificadas de acordo com seis critérios: **1)** fase de criação ou ocorrência são falhas de desenvolvimento ou operacionais; **2)** limites são falhas internas ou externas; **3)** domínio são falhas de hardware ou software; **4)** causas fenomenológicas são falhas naturais ou humanas; **5)** intenção são falhas acidentais, intencional ou não intencionalmente maliciosas; **6)** persistência são falhas permanentes ou transientes. Combinando os anteriores, obtemos três classes de falhas que servem para definir os mecanismos de defesa de um sistema: **a)** falhas de projeto que incluem vulnerabilidades de software, lógicas maliciosas e erros de hardware; **b)** falhas físicas que incluem defeitos de produção, deterioração, interferência física, ataques e erros de hardware; **c)** falhas de interação, que incluem interferência física, ataques (lógicas maliciosas e tentativas de intrusão) e erros de entrada.

Dada a importância de resguardar a informação e aos sistemas que administram a mesma, a **Segurança da Informação** (*Security Information*) protege a informação de diversos tipos de ataques, sejam intencionais ou acidentais. A Segurança da Informação está composta pelos atributos de dependabilidade, já discutidos. Apresentamos a seguir uma descrição mais detalhada das propriedades de Segurança da Informação [2]:

- ***Confidencialidade*** *(confidentiality)* – propriedade que garante que usuários não-autorizados não obtenham conhecimento de informação sensível, i.e. ausência de revelação não-autorizada de informação.
- ***Integridade*** *(integrity)* – é a ausência de alteração imprópria de estado do sistema, i.e. a prevenção de modificação ou supressão não-autorizada de informação.

- ***Disponibilidade*** *(availability)* – é prontidão para o uso imediato. Confidencialidade e integridade estão relacionadas em prevenir a ocorrência de eventos indesejáveis. Por outro lado, disponibilidade garante que eventos desejáveis certamente ocorram. Quando um usuário requisita um serviço para o qual ele está autorizado, o sistema deve garantir que o serviço é fornecido de uma maneira correta em um tempo adequado.

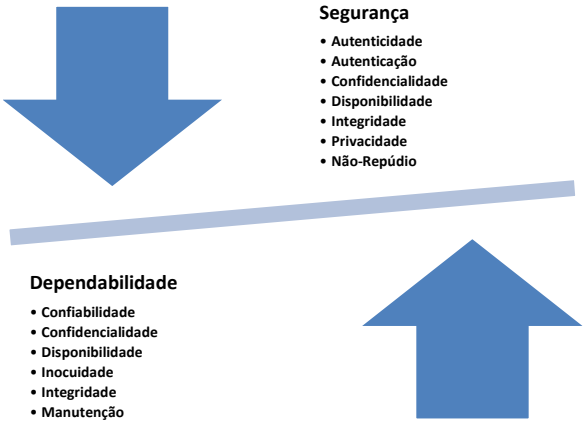



Figura 3.1 Atributos de Dependabilidade e Segurança.

Muitas propriedades de segurança podem ser definidas em termos de confidencialidade, integridade e disponibilidade, elas são: **Autenticidade** – propriedade de ser "genuíno", i.e. para uma mensagem autenticidade é equivalente a integridade do conteúdo da mensagem, de sua origem e possivelmente de outras informações , como horário de emissão, nível de classificação, entre outros (integridade da meta-informação); **Autenticação** – processo que garante confiança à autenticidade; **Não-repúdio** – corresponde à disponibilidade e autenticidade de algumas meta-informações da mensagem, como a identidade do criador e, possivelmente, o horário de criação para evitar o repúdio de mensagem por parte do receptor; **Privacidade** – é a habilidade de uma pessoa ou sistema de controlar a exposição e a disponibilidade de informações acerca de si. Relaciona-se com a capacidade de existir de forma anônima [14].

Fazendo uma comparação entre os atributos de Dependabilidade e as propriedades de Segurança da Informação, podemos observar na Figura 3.1 que ambas compartilham três propriedades ou atributos, nomeadamente confidencialidade, disponibilidade e integridade.

### 3.1.1 Tolerância de Falhas

Em 1990, a IEEE definiu “tolerância a falhas” como a capacidade de um sistema ou componente de continuar em normal funcionamento mesmo em presença de falhas, seja de hardware ou software [15].

Neste ponto, a prevenção e remoção de falhas não são suficientes quando o sistema exige alta confiabilidade ou alta disponibilidade. Nesses casos, o sistema deve ser construído usando técnicas de tolerância a falhas que garantem o funcionamento correto do sistema, mesmo na ocorrência de falhas, erros e defeitos, através de diversas técnicas. Estas técnicas dividem-se em 4 fases: detecção, confinamento, recuperação e tratamento.

Na **fase de detecção**, a falha se manifesta como um erro, para então ser detectada por algum mecanismo. Antes da sua manifestação como erro, a falha está latente e não pode ser detectada. Devido à latência da falha, até o erro ser detectado, pode ter ocorrido espalhamento de dados inválidos.

A **fase de confinamento** estabelece limites para a propagação do dano. Durante o projeto, devem ser previstas e implementadas restrições ao fluxo de informações para evitar fluxos acidentais e estabelecer interfaces de verificação para a detecção de erros.

A **fase de recuperação** ocorre após a detecção e envolve a troca do estado atual incorreto para um estado livre de falhas.

**A fase de tratamento** consiste em: **i)** localizar a origem do erro (falha); **ii)** localizar a falha de forma precisa; **iii)** reparar a falha; e **iv)** recuperar o restante do sistema. Nessa fase, geralmente é considerada a hipótese de falha única, ou seja, uma única falha ocorrendo por vez.

A simples aplicação de tolerância a falhas não garante que sistemas construídos com componentes frágeis sejam confiáveis. Uma solução é a combinação de técnicas de testes, com o objetivo de revelar falhas e demonstrar que o sistema em teste implementa os requisitos funcionais e não funcionais, verificando que o sistema foi construído corretamente.

### 3.1.2 A Problemática da Segurança da Informação

O modelo de **ataque–vulnerabilidade–intrusão** é uma especialização da sequência ***falha→erro→defeito*** aplicada a falhas maliciosas. Este modelo limita o espaço de falhas que interessa à composição **(ataque + vulnerabilidade) → intrusão** [16]. A Figura 3.2 descreve um

exemplo de ameaças a um sistema alvo, onde falhas de segurança podem ser causadas por dois tipos: falhas maliciosas (ataques) e falhas acidentais.

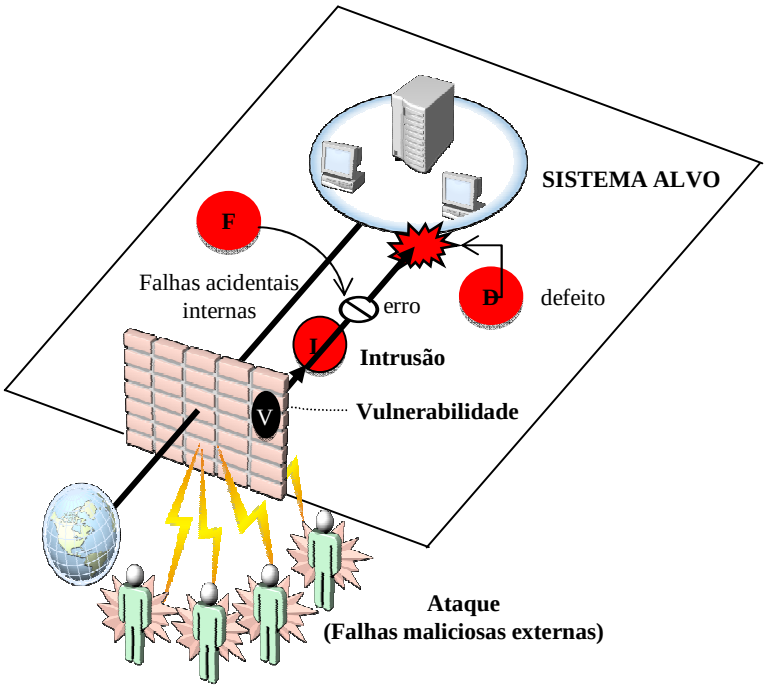



Figura 3.2 Ameaças a Segurança.

Alguns ataques podem não gerar intrusões e ser prevenidos pelos mecanismos de defesa existentes no sistema [15]. Um ataque é uma tentativa de intrusão, que por sua vez resulta de um ataque que foi bem-sucedido. No entanto, os defeitos do sistema podem ocorrer como consequência de falhas não-maliciosas, i.e. falhas que são consequências de defeitos em componentes externos ao sistema que interagem com ele, por exemplo, falhas de hardware ou sistema operacional [16]. A seguir definimos ataque, vulnerabilidade e intrusão.

**Ataque** [16] é uma tentativa de destruir, expor, alterar, inutilizar, roubar, ganhar acesso ou fazer uso não autorizado de um ativo, através da qual um atacante, i.e. um usuário interno ou externo, intencionalmente viola uma ou mais propriedades de segurança do sistema (equivalente a uma tentativa de intrusão). Os ataques são definidos tecnicamente como falhas de interação maliciosas, que tentam ativar uma ou mais vulnerabilidades, e.g. applets Java maliciosos, email com vírus. Um atacante é classificado como **passivo** ou **ativo**. O primeiro tenta aprender ou fazer uso de informações do sistema, mas não afeta os recursos do mesmo, enquanto o segundo tenta alterar os recursos do sistema ao afetar sua operação. Um ataque que ativa com sucesso uma vulnerabilidade causa uma intrusão**.**

**Vulnerabilidade** [16] é a falha de um ativo ou controle que pode ser explorado por uma ameaça. Pode ser uma falha maliciosa ou não-maliciosa, introduzida acidental ou intencionalmente, durante as fases de desenvolvimento (requisitos, análise, implementação do projeto ou configuração) do sistema, e.g. portas TCP/IP sem supervisão, que pode ser explorada para criar uma intrusão.

**Intrusão** [16] é uma falha operacional intencionalmente maliciosa, no domínio do software, originada externamente ao sistema, resultante de um ataque bem-sucedido que explorou uma vulnerabilidade. Uma falha de interação maliciosa, cometida por um usuário interno, pode ser classificada como uma intrusão, pois sua intenção foi executar uma operação em algum recurso que não lhe pertence. Uma intrusão pode levar o sistema a um estado errôneo. No caso em que um erro não é detectado nem isolado, um defeito pode ocorrer, i.e. as propriedades de segurança do sistema são violadas.

Há quatro métodos de prevenção de falhas maliciosas para garantir segurança, considerando o modelo **ataque**–**vulnerabilidade**–**intrusão**: prevenção, tolerância, remoção e previsão [16]:

- **Prevenção**, Através da inclusão de medidas de especificação formal, projeto rigoroso e gerenciamento de sistema, evita-se a introdução de vulnerabilidades, prevenindo a ocorrência de ataques, utilizando mecanismos como autenticação, autorização e firewalls. Prevenção de intrusão é a aplicação combinada de prevenção de ataque e vulnerabilidade, a fim de remover tais falhas.
- **Tolerância**, orientada a oferecer serviços corretos, mesmo na presença de intrusões, ataques e vulnerabilidades, i.e. assegura que o sistema forneça garantias de segurança, apesar dos ataques parcialmente bem-sucedidos. Prevenção e remoção de ataques são também formas de tolerância a vulnerabilidades.
- **Remoção**, orientada a reduzir o número ou a severidade de ataques e vulnerabilidades, inclui ações de manutenção que removem lógicas maliciosas, que são capazes de agir como agentes de ataque, restringindo o poder dos atacantes. Pode ser executada durante a fase de desenvolvimento do sistema, usando procedimentos de verificação, como métodos formais e modelos de testes, os quais identificam falhas que poderiam ser exploradas por um atacante.Também pode ser executada durante a fase de operação do sistema, que corresponde a

ações de manutenção, preventivas e corretivas, como aplicação de *patches* de segurança, isolamento de determinados serviços ou alteração periódica de senhas.

- **Previsão**, orientada a estimar a frequência de ataques e vulnerabilidades, juntamente com suas possíveis consequências. Inclui estatísticas, sobre as quais é possível inferir as dificuldades que um atacante teria que superar para tirar vantagens.

Nossa metodologia faz uso da Remoção de Falhas que provê a habilidade de detectar vulnerabilidades de um sistema na presença de falhas maliciosas intencionais (ataques). A metodologia faz uso da injeção de falhas que emulam diversos tipos de ataques. O objetivo é injetar ataques bem-sucedidos e conhecidos contra Web Services e WS-Security os quais são apresentados na seção a seguir.

## 3.2 Vulnerabilidades em Web Services

Garantir a segurança dos Web Service é um objetivo mais extenso e complexo do que simplesmente proteger os dados usando técnicas criptográficas. Na prática, isto implica que devemos levar em conta múltiplos fatores, tanto internos quanto externos, e analisar o sistema nas fases de processamento, armazenamento e transmissão de informação, cumprindo determinados requisitos: **i)** deve-se preservar as informações contra alterações tanto acidental quanto intencional, devido a falhas no software ou hardware, causadas por agentes externos (fogo, quedas de energia, entre outros) ou pelos próprios usuários; **ii)** é necessário impedir o acesso não autorizado ao sistema; **iii)** o sistema deve garantir que a informação esteja disponível quando necessário [14].

Nesta seção, discutimos os desafios da segurança em Web Services e descrevemos diferentes tipos de ataques e suas vulnerabilidades.

### 3.2.1 Desafios da Segurança em Web Service

Os Web Services são sistemas abertos que estão em constante comunicação com outros serviços. Seus clientes fazem requisições dos serviços através de um canal de comunicação, como a Internet, enviando e recebendo informações simultaneamente. Outro beneficio é a possibilidade de desenvolver os Web Services sobre diferentes linguagens e plataformas que

trabalham sobre XML e HTML. No entanto, isto gera vários desafios de segurança, como a proteção de informação na comunicação, a complexidade do Web Service, entre outros.

A segurança em Web Services se define como a proteção da mensagem SOAP, seus principais recursos de comunicação e armazenamento da informação. Holgersson define em [4] os principais desafios relacionados a padrões e interoperabilidade. Ambos desafios enfatizam a relativa imaturidade de Web Services sobre abordagens de segurança, qualidade de serviços (*Quality of Services – QoS*), escalabilidade, entre outros.

Ibrahim em [17], classifica e agrupa os desafios para a segurança em Web Services, que envolvem ameaças, ataques e problemas de segurança aos serviços. Eles são:

- *Ameaças ao nível de serviço*, descreve ataques: a WSDL e UDDI, por injeção de código malicioso, suplantação de identidade, Negação de Serviços XML, falsificação de esquemas XML, sequestro/roubo de sessão.
- *Ameaças ao nível de mensagem*, descreve ataques: de manipulação e injeção de mensagens, Web Services intermediário, reenvio de mensagens, abusos de validação de mensagens, interceptação da rede e mensagens confidenciais.

### 3.2.2 Ataques contra Web Services e WS-Security

Web Services evoluiu a uma tecnologia mais abrangente para integrar aplicativos e troca de dados na arquitetura orientada a serviços. No entanto, apresentam uma série de novos riscos [18]. Esta seção se concentra nas ameaças e vulnerabilidades identificadas e utilizadas nos testes de segurança para Web Services e WS-Security, obtidos de diversas referências [4], [19 - 26].

Os ataques estão organizados segundo o tipo de ataque (Negação de Serviços e Ataques de Injeção). Para cada ataque, descrevemos seu objetivo e impacto potencial. O restante dos ataques (Ataques de Força Bruta, Suplantação e Inundação) estão descritos na Tabela A.1 do Apêndice A.

#### 3.2.2.1 Ataques de Negação de Serviços (*Denial of Services Attacks*)

Mais conhecido como *DoS Attack*, é uma tentativa de tornar os recursos de um sistema indisponíveis para seus usuários. Não se trata de uma invasão ao sistema, mas sim da sua invalidação por sobrecarga. Os ataques de Negação de Serviço são feitos geralmente de duas formas: **i)** forçar o sistema vítima a reinicializar ou consumir todos os recursos, e.g. memória ou

processamento, de forma que ele não possa fornecer seu serviço; e **ii)** interromper a comunicação entre os usuários e o sistema alvo, de forma a prejudicar seu funcionamento. Os principais ataques de Negação de Serviços são:

- ***Replay Attack***, consiste em que o atacante faz várias requisições ao Web Services, em uma tentativa de sobrecarregar o servidor. Este tipo de atividade não é detectada como uma intrusão, já que o conteúdo das mensagens são válidas [24].
- ***Oversized Payload***, esse ataque usa um dos pontos fortes do XML, que é a capacidade de aninhar elementos dentro de um documento para atender as necessidades de relações complexas entre os elementos. Uma requisição contém um número finito de elementos, tais como uma ordem de compra que incorpora os endereços de cobrança, mas um atacante pode criar um documento XML que procura estressar ou até colapsar o analisador XML (*XML Parser*), criando, por exemplo, um documento contendo mais de 100.000 elementos de profundidade [4], [20 - 22], [24].
- ***Coercive Parsing* (*Recursive Payloads*)**, um dos primeiros passos no processamento de uma requisição é analisar o conteúdo da mensagem SOAP e transformá-lo em uma mensagem executável para ser processada. O XML pode se tornar complexo e extenso para analisar, especialmente quando se usa namespaces. Assim, o processo de análise de XML permite ao atacante esgotar os recursos computacionais, utilizando o analisador XML, e.g. abrir 100.000 etiquetas [4], [20 - 22], [24].
- ***Oversize Cryptography***, a utilização de elementos cifrados no cabeçalho da mensagem SOAP limita a verificação do esquema de validação de cada elemento. Este ataque consiste em enviar grandes blocos de mensagens cifrados para afetar o consumo da memória e a CPU. Outra versão consiste em modificar o cabeçalho, inserindo chaves cifradas e vinculando as mesmas, obrigando o usuário a utilizar várias chaves para decifrar cada elemento [4], [20 - 21], [24].
- ***Attack Obfuscation***, este ataque utiliza as limitações do analisador XML que não permite interpretar o conteúdo cifrado na mensagem SOAP, o que permite enviar ataques cifrados como Oversize Payload, Coercive Parsing, XML Injection, entre outros, obrigando o analisador XML a decifrar os segmentos cifrados e processar o ataque como parte da mensagem SOAP [4], [20 - 21], [24]

- ***XML Bomb***, seu objetivo é sobrecarregar o analisador XML, explorando o fato de que XML permite definir entidades e parâmetros dentro da requisição da mensagem SOAP, e.g. definir mais de 100.000 entidades, gerando uma requisição que fica armazenada na memória e gera alto consumo de CPU [26].
- ***Unvalidated Redirects and Forwards***, um atacante cria um enlace inválido para um serviço falso e o envia ao cliente, que por sua vez acessa, achando que será levado ao serviço legítimo e desejado. O cliente fornece suas informações, que são coletadas pelo atacante. Além disso, o acesso pode desencadear a instalação de códigos maliciosos, evitando controles de segurança [19]**.**
- ***Attacks through SOAP Attachment***, a mensagem SOAP permite anexar arquivos binários, os quais podem conter códigos maliciosos. Se um Web Services os armazena e distribui a outros, pode resultar em sérias consequências [6, 25].
- ***Schema Poisoning***, o esquema XML fornece instruções de formatação para o analisador XML, que descrevem instruções de pré-processamento, as quais são suscetíveis a modificações. Um atacante pode tentar comprometer o esquema armazenado e substituí-lo por outro similar, mas modificado. Este processo se chama envenenamento. Os ataques de DoS são fáceis de implementar se o esquema está comprometido [24].

#### 3.2.2.2 Ataques de Injeção (*Injection Attacks*)

Os ataques de injeção tais como SQL/XML Injection ou Cross-site Scripting (XSS) são consequências da interceptação e modificação de mensagens, que enganam ao Analisador Path para executar comandos mal intencionados ou acessar dados não autorizados. Eles são:

- ***XML Injection***, é uma técnica para modificar a estrutura XML de uma mensagem SOAP (ou qualquer outro documento XML), através da inserção de etiquetas (*tags*). XML Injection insere conteúdo malicioso no documento resultante. Do lado do servidor, este conteúdo é considerado como uma parte da estrutura da mensagem SOAP e pode provocar efeitos indesejáveis [27], [19 - 22].
- ***SQL Injection***, consiste em inserir ou injetar uma consulta SQL na mensagem SOAP requisitada. Existe um ataque bem sucedido quando o atacante pode ler os dados sigilosos do banco de dados, alterá-los (inserir, atualizar, eliminar) e executar

operações de administração como criação de tabelas ou até bancos de dados. Se o serviço não valida corretamente os dados, o atacante pode comprometer o Web Service, deixando o atacante executar consultas no servidor [19], [22], [24].

- ***XPath Injection***, similar a SQL Injection. Utiliza comandos XPath (XML Path Language) para construir consultas XPath, que permitem diversas consultas, i.e. conhecer a estrutura dos dados XML, elevar privilégios no servidor, modificar o banco de dados (documentos XML), acesso às tabelas de administrador, inacessíveis em consultas regulares, entre outros. Este ataque explora os comandos XPath para atacar o servidor, através do envio de informações intencionalmente malformadas para o Web Services e fazer consultas XPath [4], [23].
- ***Cross-site Scripting (XSS)***, utiliza vulnerabilidades existentes no Web Services para injetar códigos maliciosos no servidor, geralmente em JavaScript, através das operações ou atributos descritos no WSDL. Dada a relação de confiança estabelecida entre o Web Service e o servidor, o primeiro assume que o código recebido é legítimo e, portanto, o segundo permite o acesso a informações confidenciais, como o identificador de sessão. Com isso, o atacante envia requisições maliciosas ao Web Services, que não são validadas e são incluídas dentro das páginas geradas dinamicamente, para sequestrar a sessão e coletar informações das pessoas que visitam o site [6], [19], [23].
- ***Cross-site Request Forger* (*XSRF*)**, o ataque aproveita as sessões estabelecidas por Web Services vulneráveis para executar ações sem o consentimento da vítima, e.g. fechar a sessão até a transferência de fundos em um aplicativo bancário. Ao contrário do Cross-site Scripting (XSS), que explora a confiança de um usuário em um site, o XSRF explora a confiança que um site tem no navegador do usuário [19], [23].
- ***Fuzzing Scan***, o termo "Fuzzing" descreve a geração de entradas aleatórias em um sistema alvo, e.g. clicar o mouse ou teclado aleatoriamente na interface de uma aplicação ou a criação de dados aleatórios inseridos em alguma aplicação ou sistema. O ataque gera entradas aleatórias no Web Services, através das operações e parâmetros descritos no WSDL, com a esperança de provocar algum tipo de imprevisto ou erro. Ao longo de um período prolongado de tempo, *Fuzzing Scan* descobre vulnerabilidades no serviço, que não são revelados por um processo de teste

mais estruturado. Por padrão, os valores gerados estão entre 5 e 15 caracteres de comprimento, com até 100 mutações; um Teste de Fuzzing mais realista pode alterar essas configurações para estar entre 1 e 100 caracteres para mais de 100.000 requisições. Isso pode, naturalmente, levar algum tempo [6], [28].

- ***Invalid Types***, os Web Services têm possibilidades de criar vários tipos de variáveis ou dados. Isto permite manipular valores que o serviço não espera, especialmente se a entrada é restringida ao tipo de dado. Um atacante pode enviar valores que estão fora dos limites esperados das variáveis descritas no WSDL e obter informações úteis do sistema alvo, i.e. através de mensagens de erro [6], [19, 20].
- ***Parameter Tampering***, os parâmetros, descrito no WSDL, são usados para transmitir informação do cliente ao Web Services, a fim de executar operações remotas. O atacante analisa o WSDL com o intuito de descobrir operações privadas para executá-las e recuperar informações não autorizadas. Através dos parâmetros, são enviados conteúdo inesperado ou caracteres especiais para o serviço. Isto pode causar acesso ilegal a operações privadas até a negação do serviço [4], [6], [19], [22], [24], [26].
- ***Malformed XML***, aproveita as vulnerabilidades do analisador XPath, inserindo fragmentos mal formados de XML nas mensagens SOAP, por exemplo: deixando etiquetas abertas, adicionando atributos e elementos não definidos, entre outros. O ataque faz com que o Web Service exponha suas informações confidenciais e gere falhas no sistema alvo (*crash*), através de erros provocados no analisador XPath. Esta vulnerabilidade acontece quando uma aplicação Web não valida as informações recebidas de entidades externas (usuários ou outras aplicações) e processa a mensagem SOAP, gerando falhas, tanto no Web Services quanto no servidor [6].
- ***Frankenstein Message* (*Modify Timestamp*)**, este ataque utiliza o WSU, um namespace que controla o tempo de criação e expiração de mensagens SOAP, através da etiqueta <*Timestamp*>. Ao conhecer o tempo de criação e expiração, o receptor da mensagem SOAP decide se os dados são novos ou não. O atacante pode modificar o campo *Expires* para enviar mensagens SOAP que sejam rejeitadas pelo servidor. Em outra versão, o atacante modifica diversos elementos do WSU para realizar ataques de negação de serviço [25].

Apesar das pesquisas sobre vulnerabilidades em Web Services, os desenvolvedores carecem de conhecimento sobre os diversos tipos de ataques e como se proteger deles. Um desenvolvedor deve ponderar estas vulnerabilidades com os riscos de perda de informação. Para isto existe um conjunto de técnicas e protocolos criptográficos que asseguram a comunicação, autenticando e protegendo os usuários e meios de comunicação contra os ataques descritos. Estas técnicas são expostas a seguir.

## 3.3 Segurança em Web Services

A segurança em Web Services é um dos pontos fracos desta tecnologia. O problema não reside na falta de mecanismos de segurança, mas a falta de consenso sobre qual mecanismo deve ser adaptado à tecnologia. A presente seção descreve os principais mecanismos de segurança para Web Services (ver Figura 3.3).

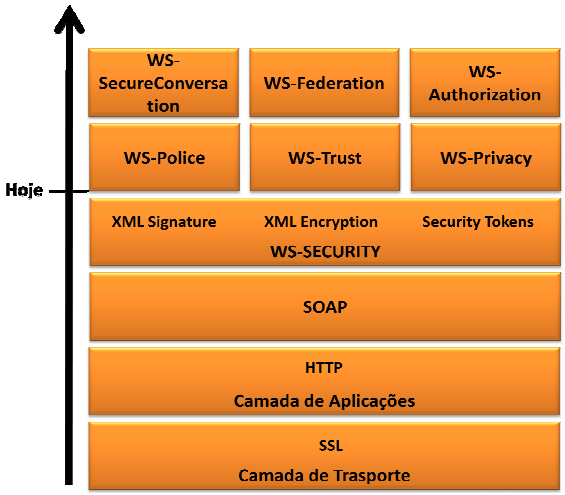



Figura 3.3 Mecanismos de segurança para Web Services.

Existem diversas tecnologias para proteger a comunicação entre clientes e servidores de Web Services, entre as quais temos a conexão ponto-a-ponto e fim a fim [27].

No contexto ponto-a-ponto procura-se garantir a segurança no transporte de dados. Para isso, existem diversos padrões, como HTTPS. A segurança ponto-a-ponto permite garantir a confidencialidade dos dados transportados, no entanto, quando uma mensagem passa por diversos terminais intermediários, antes de atingir o destinatário final, a segurança já não é garantida. Em troca, a segurança fim-a-fim visa proteger a troca de mensagens SOAP entre clientes e servidores, cifrando a informação da origem até o destinatário, não importando com os

intermediários. Entre estas tecnologias fim-a-fim, temos WS-Security, XML Signature, XML Encryption e Security Tokens. A seguir descrevemos essas tecnologias mencionadas anteriormente.

### 3.3.1 SSL (Secure Socket Layer)

Este protocolo desenvolvido pela Netscape em 1996, provê privacidade e integridade de dados entre duas aplicações, através da autenticação das partes envolvidas e da cifra dos dados transmitidos entre elas. Usa o mecanismo de segurança SSL sob HTTP, conhecido como HTTPS (*Hypertext Transfer Protocol Secure*), de fácil configuração. O protocolo não é adequado para taxas de transferências de dados elevadas. Por ser um mecanismo de proteção no nível de transporte, apresenta restrições para ser aplicado em Web Services. Não permite cifrar a informação ou usar sessões seguras entre mais de duas partes (fim-a-fim) [14].

### 3.3.2 WS-Security

Chamado também *Web Services Security* ou *WSS*[13], é um padrão para a proteção das mensagens SOAP. Desenvolvido pela IBM, Microsoft e Verisign em 2004, WS-Security contém especificações que garantem confidencialidade e integridade das mensagens, autenticação e autorização de usuários. Este padrão insere uma camada sobre a mensagem SOAP para construir serviços mais seguros e robustos, com ampla interoperabilidade, permitindo integrar todas as especificações descritas na Figura 3.3. Além de ser um modelo de segurança sólido e aberto, baseada em padrões, é de desenvolvimento rápido, permite cifrar os documentos XML e usar sessões seguras entre mais de duas partes [28] através do uso de outras especificações como XML-Signature, XML-Encryption e Security Tokens.

### 3.3.3 XML Encryption

*XML Encrytion* (XML-Enc) é uma especificação para prover **confidencialidade** e **autenticação** à mensagem SOAP, cifrando informações com o objetivo de proteger que usuários não autorizados a acessem sem a chave de decodificação. A tecnologia permite o uso de diversas chaves para criptografar as diferentes partes da mensagem SOAP. Portanto, a mesma mensagem

[13] WS-Security é parte da família WS-* de especificações, sua primeira versão foi publicada em 19 de abril de 2004. Em 17 de fevereiro de 2006 foi lançada a versão 1.1 [12, 13].

pode ter um número de receptores e cada receptor só tem acesso às suas próprias partes da mensagem [4], [29]. A etiqueta é <EncrytedData>. Sua sintaxe básica é descrita na Figura 3.4.

```
1:   <?xml version='1.0'?>   <-! Inicio da seção de encriptação na mensagem SOAP>
2:      <PaymentInfo xmlns='http://example.org/paymentv2'>
3:         <Name>John Smith</Name>
4:            <-! Inicio da seção cifrada da mensagem SOAP>
5:            <EncryptedData Type='http://www.w3.org/2001/04/xmlenc#Element'
6:                           xmlns='http://www.w3.org/2001/04/xmlenc#'>
7:               <CipherData> <-! Dados cifrados>
8:                  <CipherValue>elementos para cifrar</CipherValue>
9:                  </CipherData>
10:           </EncryptedData>
11:     </PaymentInfo>
```

Figura 3.4 Sintaxes de XML Encryption.

### 3.3.4 XML Signature

*XML Signature* (XML-Sig) é uma especificação que provê **integridade** e **autenticação**, tanto para verificar as credenciais de Security Token quanto para assegurar que a mensagem SOAP não foram modificadas durante a transmissão. Esta especificação, combina os certificados digitais com as credenciais para assegurar que o usuário é quem indica ser. Similar a XML Encryption, esta tecnologia permite assinar certas porções da mensagem SOAP [4], [30]. O elemento para assinar é <Signature>. A estrutura básica é descrita na Figura 3.5.

```
1:   <Signature>   <-! Inicio da seção da assinatura na mensagem SOAP>
2:      <SignedInfo>    <-! Contêm os dados assinados e descreve o algoritmo utilizado>
3:         <SignatureMethod/>   <-! Contêm o algoritmo hash e o hash da mensagem>
4:            <CanonicalizationMethod>   <-! Este método utilizado para assegurar a assinatura>
5:               <Reference>    <-! Especifica o recurso que é assinado por referência>
6:                  <Transforms> <DigestMethod> <DigestValue>
7:               </Reference>
8:            <CanonicalizationMethod/>
9:         </SignedInfo>   <-! A etiqueta contêm >
10:      <SignatureValue/>   <-! Contêm a assinatura em Base 64>
11:     <KeyInfo/>
12:     <Object/>   <-! contêm dados assinados>
13:  </Signature>
```

Figura 3.5 Sintaxes de XML Signature.

A combinação dos três elementos básicos – *Security Tokens, XML Encryption e XML Signature* – provê novas possibilidades para o intercâmbio seguro de mensagens SOAP. O primeiro facilita a autenticação e autorização dos usuários a serviços e documentos XML. O segundo cifra a informação de usuários não autorizados. O último fortalece a autenticação e identifica as transações, detectando mensagens adulteradas e evitando o repúdio da mensagem.

## 3.4 Security Tokens

*Security Tokens* é uma especificação de segurança para prover **autenticação** e **autorização** nos Web Services, com o objetivo de determinar a identidade do usuário,

juntamente com seus direitos de acesso ao servidor de aplicações do Web Service. Representado, na mensagem SOAP, pela etiqueta <wsse:SecurityToken>, provê diversos tipos de credenciais de segurança, como a identificação por usuário/senha (*username*/*password*), até os mais complexos, baseados em certificados como X.509 e Kerberos [4], [31]. Sua sintaxe básica é detalhada na Figura 3.6.

```
1:  <wsse:SecurityToken wsu:Id="..."> <-! Security Token de autenticação por username>
2:          <wsse:Username>...</wsse:Username> OR <wsse:SecToken>...</wsse:SecToken>
3:  </wsse:SecurityToken>
```

Figura 3.6 Sintaxes de Security Tokens.

## 3.4.1 A Etiqueta <Security>

A segurança em Web Services inicia no cabeçalho (*Header*) com a etiqueta <Security. >. A etiqueta contém a segurança relacionada a dados e informações necessárias para implementar diversos mecanismos como credencias de segurança (Security Tokens), assinaturas digitais e encriptação. Esta etiqueta está presente repetidas vezes para permitir fornecer diferentes funções de segurança para vários receptores, permitindo que cada receptor possa decifrar parcial ou totalmente a mensagem SOAP, independentemente de ser o receptor intermediário ou o receptor final [32].

Dentro de <Security> está a etiqueta <role> que especifica funções de cada receptor e provê privilégios de segurança para diferentes destinatários. A etiqueta <role> não pode ser repetida nem ser omitida, já que permitiria que qualquer atacante pudesse acessar e modificar a mensagem SOAP sem privilégios. Esta estrutura é descrita na Figura 3.7.

```
1:   <SOAP:Envelope xmlns:SOAP="...">
2:          <SOAP:Header>
3:                 <wsse:Security SOAP:role="..." SOAP:mustUnderstand="...">
4:                        <wsse:UsernameToken>
5:                               …
6:                        </wsse:UsernameToken>
7:                               …
8:                 </wsse:Security>
9:          </SOAP:Header>
10:         <SOAP:Body Id="MsgBody">
11:         <!-- SOAP Body data -->
12:         </SOAP:Body>
13:  </SOAP:Envelope>
```

Figura 3.7 Estrutura de uma mensagem SOAP usando <Security> no cabeçalho (Header).

## 3.4.2 Security Tokens

Os objetivos do Security Token são: **1)** comprovar a identidade do emissor; **2)** permitir acesso aos serviços do provedor; e **3)** identificar o provedor de serviços, i.e. autenticar ao cliente. Para isso, o WS-Security pode implementar três tipos de Security Tokens: **Username Tokens** para autenticação personalizada, onde o usuário e a senha podem ser modificados, Tokens de autenticação binária do tipo **Kerberos** e autenticação por **Certificado Digital X.509** para autenticar os usuários e servidores de Web Services [32].

Para nossa pesquisa utilizaremos a primeira opção, **Username Tokens**. Ele usa a autenticação personalizada por validação de usuário e senha, dentro da etiqueta <UsernameToken>, fornecendo apoio por meio de diversos sub-elementos para autenticar aos clientes de um Web Services. Estes sub-elementos são descritos na Figura 3.8.

**/Username:** Usuário associado com o Tokens.
**/Password:** Senha do usuário associado com o Tokens.
**/Password/@Type:** Tipo de senha fornecida, 2 tipos pré-definidos:
- **PasswordText:** Senha em texto simples ou claro.
- **PasswordDigest:** Senha implícita, de função resumem (valor hash) com o criptosistema SHA-1[14] em base64-encoded codificada em UTF8-encoded.

**/Nonce:** Cadeia aleatória para cada mensagem SOAP.
**/Created:** Data e hora da criação do Tokens.

Figura 3.8 Sub-elementos da etiqueta <UsernameToken>.

## 3.4.3 Segurança em Web Services com Username Token

Existem diversas formas de utilizar o elemento <UsernameToken> dependendo da forma como o elemento <Password> seja usado. A maneira mais fácil de identificação seria transmitir o nome de usuário e omitir a senha. Um trecho da mensagem SOAP pode ser visto na Figura 3.9.

```
1:      <UsernameToken>
2:            <Username>MeuUsuario</Username>
3:      </UsernameToken>
```

Figura 3.9 Código de transmissão do nome de usuário por <UsernameToken>.

Como podemos observar na Figura 3.9, utilizar somente o usuário como mecanismo de autenticação é inseguro. Sem nenhum método de comprovação de identidade, é apenas útil em situações em que algum outro mecanismo de autenticação como o SSL é usado, sendo que o nome de usuário é usado como uma identificação do usuário. No entanto, WS-Security nos

[14] http://www.ietf.org/rfc/rfc3174.txt

permite implementar uma melhor autenticação. Transmitindo o elemento <Password> como parte da etiqueta <UsernameToken>, a mensagem ficaria segura pela identificação do usuário. Um trecho da mensagem é descrita na Figura 3.10.

```
1:      <UsernameToken>
2:          <Username>MyName</Username>
3:          <Password Type="PasswordText">MinhaSenha</Password>
4:      </UsernameToken>
```

Figura 3.10 Código de transmissão do nome de usuário por <UsernameToken>.

Se alguém interceptar a mensagem SOAP, da Figura 3.10, pode facilmente descobrir a senha e autenticar-se. Isto acontece porque o atributo "*Type*" da etiqueta <Password> é dada como texto claro (*PasswordText*). Para evitar isto, a senha será enviada como uma função hash[15]. Tornando impossível para um intermediário ver a senha real. Esta senha geralmente usa o algoritmo SHA-1 em base 64, descrita na Figura 3.11.

```
1:  <UsernameToken>
2:      <Username>MyName</Username>
3:      <Password Type="PasswordDigest">fm6SuM0RpIIhBQFgmESjdim/yj0=</Password>
4:  </UsernameToken>
5:
```

Figura 3.11 Código de transmissão do nome de usuário por <UsernameToken>.

Existe a possibilidade que o atacante utilize a senha cifrada para autenticar sua mensagem com o Web Services. Nesta caso o atacante não precisa conhecer a senha do usuário. Para resolver esse problema, WS-Security fornece alguns meios adicionais para fazer a autenticação de forma mais segura através do *Password_Digest*, que é uma função hash, obtida da combinação da: senha de texto claro, Nonce[16] e o tempo da criação do Security Token. Esta função criptográfica usa SHA-1 em base 64, e é resumida em:

Password_Digest = Base64(SHA-1(Nonce + Created + Password))

```
1:   <UsernameToken>
2:       <Username>MyName</Username>
3:       <Password Type="PasswordDigest">fm6SuM0RpIIhBQFgmESjdim/yj0=</Password>
4:       <Nonce>Pj+EzE2y5ckMDx5ovEvzWw==</Nonce>
5:       <Created>2004-05-11T12:05:16Z</Created>
6:   </UsernameToken>
```

Figura 3.12 Autenticação por senha, Nonce e tempo de criação.

[15] Função hash é uma sequência de bits geradas por um algoritmo de dispersão, representada geralmente em base hexadecimal.

[16] Nonce é um número aleatório ou pseudoaleatório gerado por um protocolo de autenticação para garantir que as mensagens SOAP utilizadas não possam ser usadas para ataques de repetição.

Neste cenário, o cliente cria uma senha com os três atributos descritos na Figura 3.12 e transmite o valor para o Web Services. O servidor usa o mesmo mecanismo de autenticação para recuperar a senha relacionada com o nome do usuário, o Nonce e o tempo de criação da mensagem, gerando uma função hash que é comparada com o PasswordDigest da mensagem SOAP recebida. Caso as funções hash sejam idênticas, o Servidor dá acesso ao usuário.

No entanto, não é suficiente lidar com a senha para autenticar uma mensagem SOAP. Um atacante poderia interceptar o UsernameToken de um usuário e fazer requisições ao nome dele. Para evitar isto, podemos utilizar um atributo chamado *Timestamp* ou tempo de expiração de mensagens SOAP. O remetente define o tempo de expiração, geralmente em segundos. Caso o servidor identifique a mensagem em um tempo maior do que o especificado, a mensagem é rejeitada. A implantação é simples, mas precisa que os relógios do servidor e do cliente fiquem sincronizados para evitar que mensagens válidas sejam rejeitadas.

Quanto às questões de sincronização, WS-Security fornece o cabeçalho <timestamp>. Ele pode ser usado para expressar o tempo de criação e expiração de uma mensagem SOAP.

```
1:  <Timestamp>
2:        <Created>...</Created>
3:        <Expires>...</Expires>
4:  </Timestamp>
```

Figura 3.13 Autenticação por tempo de criação e expiração.

Tal como acontece com o cabeçalho <Security>, os múltiples elementos <timestamp> podem ser especificados e direcionados para diferentes *roles* e usuários. Na Figura 3.13 observamos uma mensagem SOAP com o elemento <timestamp>.

## 3.5 Modelando Ataques

A **modelagem de ataques** é usada para descrever as etapas de um ataque bem-sucedido, podendo variar de modelos simples de árvores [33] a métodos baseados em redes de Petri [34]. Nos últimos anos surgiram também os modelos baseados em UML [35], como por exemplo, diagramas de sequência, casos de uso e diagramas de estado.

O que distingue os modelos de ataques dos métodos de modelagem são aspectos funcionais. O ponto de vista de um atacante é enfatizado, aproximando mais aos problemas de segurança, com o intuito de: **i)** descobrir vulnerabilidades em sistemas novos; **ii)** evitar vulnerabilidades durante o desenvolvimento do software; e **iii)** avaliar implementações existentes

para vulnerabilidades conhecidas [5]. A seguir descrevemos o modelo de Árvore de Ataques para representar os ataques contra Web Services.

### 3.5.1 Árvore de Ataques

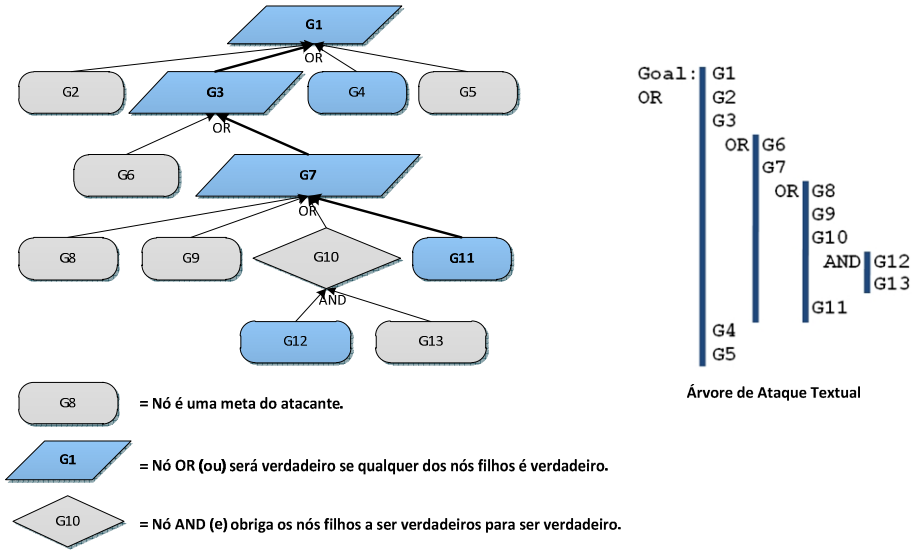
As **árvores de ataques** são estruturas de dados que podem descrever possíveis ataques a um sistema, de forma organizada para facilitar a análise de segurança. Ela representa os passos de um ataque e suas interdependências. Pode ser usada para representar e calcular probabilidades, riscos, custos, entre outras ponderações. Além disso, ela possui fácil representação de uma base de conhecimento de ataques. Essas vantagens foram observadas a partir de trabalhos de análise de segurança [5], [33], que usaram diferentes tipos de modelos, por exemplo redes de Petri e modelos UML para representação dos ataques que:

- Concentram-se em objetivos que podem ser transformados em ataques contra os Web Services que utilizam o protocolo WS-Security;
- Permitem descrever, de uma forma mais estruturada do que linguagem natural, as ações executadas por uma vulnerabilidade encontrada;
- São fáceis de compreender por pessoas com pouca prática em modelos formais; são concisos e possibilitam que muitas pessoas possam contribuir para sua manutenção;
- Permite gerar uma estrutura hierárquica, na qual objetivos em níveis mais altos são divididos em sub-objetivos, até que o nível de refinamento (detalhes) desejado seja alcançado, simplificando a navegação e possibilitando que várias pessoas trabalhem em diferentes ramos simultaneamente;
- É uma estrutura modular que permite o reuso da árvore de ataques para representar outros ataques, i.e. uma nova árvore de ataques pode acoplar módulos menores.
- É um projeto modular que permite aos analistas trabalharem simultaneamente em componentes separados e integrá-los para completar a árvore de ataques. Além de tornar o processo de criação automático, permite a atualização dos componentes (ataques) em face de novas tecnologias, possibilitando o uso de bibliotecas de árvores de ataques.
- Permite definir padrões de ataque com base em ataques mais comuns [36].

O nó raiz da árvore de ataques representa a realização do objetivo final do ataque. Cada nó filho representa um sub-objetivo que precisa ser realizado para que o objetivo do nó pai tenha sucesso. Nós pais podem estar relacionados com seus filhos por uma relação de tipo OR ou AND. Numa relação do tipo OR, se qualquer um dos sub-objetivos dos nós filhos é realizado então o nó pai é bem sucedido, enquanto que na relação de tipo AND todos os sub-objetivos precisam ser alcançados. As folhas da árvore, e.g. nós que não são mais decompostos, denominados pontos de influência, representam ações do atacante.

**Cenários de ataque individuais** podem ser gerados percorrendo a árvore em profundidade, e.g. um cenário de ataque é um caminho da raiz até a folha da árvore, assim gerando uma combinação mínima de eventos das folhas, no sentido que, se qualquer nó for omitido do cenário de ataque então o objetivo da raiz não é alcançado. O conjunto completo de cenários de ataque de uma árvore mostra todos os ataques que estão disponíveis para um atacante que possui recursos "infinitos".

Fazendo analogia com geração de casos de teste, o objetivo é cobrir todas as ações representadas nas folhas. Árvores de ataques podem ser representadas gráfica ou textualmente. A Figura 3.14 (a) mostra um exemplo de uma notação gráfica e a Figura 3.14 (b) mostra a notação textual correspondente. A descrição de cada nó é feita em linguagem natural. Os cenários de ataque que podem ser gerados a partir da árvore de ataques também são mostrados. A notação <a, b, c> representa um cenário, com as folhas sendo consideradas nesta ordem: a → b → c.




(a) Árvore de ataques gráfica (b) Árvore de ataques textual

Figura 3.14 Árvore de ataques e cenários de ataque.

Cenários de ataque: <G2>, <G6>, <G2>, <G8>, <G9>, <G12, G13>, <G11>, <G4>, <G5>. Cada nó folha da árvore é associado com um nível aproximado de recursos requeridos para realizar um ataque específico, tais como custo financeiro, resolução de problemas, entre outros. Os recursos necessários em qualquer nó da árvore de ataques são um reflexo direto da complexidade de alcançar a vulnerabilidade. Essas métricas de recursos incorporadas no modelo da árvore de ataques determinam a probabilidade dos ataques, pois influenciam no comportamento dos atacantes, já que um atacante, com pouca ou muita experiência, pode executar um ataque se tem os recursos requeridos para executá-lo e pode interagir com ele [37].

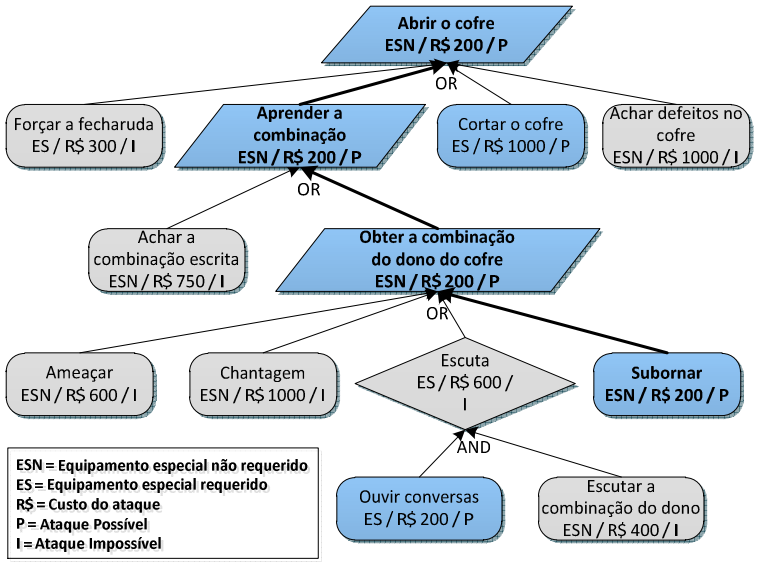



Figura 3.15 Árvore de ataques para abrir o cofre.

Usando os atributos associados aos nós como custo, probabilidades ou valores lógicos, é possível selecionar cenários de ataque baseados em critérios, i.e. os cenários que são mais prováveis ou os menos custosos. A Figura 3.15 mostra um exemplo da aplicação da árvore de ataques e descreve os atributos sendo usados nas folhas da árvore representando o custo e um valor lógico (P: possível ou I: impossível) para cada folha [38]. Cada nó é uma meta para conseguir abrir o cofre, neste caso temos quatro nós de segundo nível, cada um deles usa valores booleanos para analisar a factibilidade de cumprir seu objetivo, estas opções variam desde: **i)** facilidade (fácil e difícil); **ii)** custo (caro e barato); **iii)** legais e ilegais; e **iv)** equipamentos necessários e não necessários.

A necessidade para modelar as ameaças, compreender quais são os objetivos dos ataques, descobrir o que procuram os atacantes, conhecer como aconteceu um ataque e saber onde gastar o orçamento de segurança são os objetivos da árvore de ataques [34].

Neste exemplo (Figura 3.15), para alcançar o objetivo final de “Abrir o cofre”, o atacante deveria “Forçar a fechadura” **OR** “Aprender a combinação do cofre”. Para “Aprender a combinação do cofre” ele deveria “Achar a combinação escrita” **OR** “Obter a combinação do dono do cofre”. Uma forma de “Obter a combinação do dono do cofre” é por “Escuta”, a qual requer um equipamento especial. A “Escuta” deve ser feita de modo que o atacante possa “Ouvir uma conversa” **AND** “Escutar ao dono do cofre dizer a combinação”. Atacantes não podem alcançar o objetivo a menos que os sub-objetivos sejam satisfeitos. O cenário de ataque <“Subornar”> é o ataque mais barato (R$ 200) e não requer nenhum equipamento especial.

### 3.5.2 Padrão de Ataque

**Padrão de ataque** (*attack pattern*) é uma representação genérica de um ataque malicioso e intencional, que geralmente ocorre em contextos específicos. Cada padrão de ataque contém: **i)** o objetivo do ataque especificado pelo padrão; **ii)** uma lista de precondições para que aconteça; **iii)** os passos para executar o ataque; e **iv)** uma lista de pós-condições se o ataque foi bem-sucedido.

As precondições incluem suposições feitas sobre o atacante ou o estado do sistema que são necessárias para que o ataque tenha sucesso, como por exemplo habilidades, recursos, acesso ou conhecimento que um atacante deve possuir e o nível de risco que ele deve estar disposto a tolerar. As pós-condições incluem o conhecimento adquirido pelo atacante e alterações do estado do sistema, que resultaram dos passos das vulnerabilidades encontradas [34].

O tipo mais comum de vulnerabilidade de segurança é o tratamento incorreto de *buffer overflow* [34], que é uma condição anômala onde a aplicação tenta armazenar dados além dos limites de um buffer. O padrão de ataque para um estouro de *buffer* (*buffer overflow*) é representado na Figura 3.16 [36].

Uma árvore de ataques pode ser refinada desde o nó raiz, usando uma combinação de extensões manuais e aplicação de padrões de ataque. Extensões manuais dependem do conhecimento de segurança em construir árvores de ataques, enquanto a aplicação de padrões de ataque depende mais do conhecimento que está implementado na biblioteca. Tais informações

são providas por um especialista em segurança. A utilização dessas árvores servirá para modelar ataques contra WS-Security, usando as vulnerabilidades descritas na Seção 3.2.2.

***Padrão de ataque de Buffer Overflow:***

**Objetivo:** Explorar a vulnerabilidade de *Buffer Overflow* para desempenhar funções maliciosas no sistema alvo.

**Pré-condição:** O programa tem uma ou mais operações que o atacante pode acessar remotamente.

***Ataque:***

AND 1. Identificar a existência de vulnerabilidades de Buffer Overflow, por exemplo, ingressando uma cadeia longa de caracteres como entrada para ser executada por um programa. Isto permite enviar código malicioso (*shellcode*) a um endereço da memória.

2. Identificar o endereço da memória que armazena o *shellcode*.

3. Reescrever o valor de IP (*Instruction Pointer*)e o FP (*Frame Pointer*) para executar o *shellcode*.

4. Solicitar através de um programa o nome de um arquivo do sistema, que faz referencia ao endereço de memória modificado em IP, para que o programa execute o *shellcode*.

**Pós-condição:** O sistema alvo desempenha a função maliciosa (*shellcode*).

Figura 3.16 Padrão de ataque de Buffer Overflow.

Nesta seção descrevemos os mecanismos de proteção de Web Services e as técnicas para ataca-lhos. Desta forma, chega a ser tão importante a pesquisa, tanto para o desenvolvimento de técnicas de proteção quanto o desenvolvimento de testes de segurança, pois enquanto a ausência de falhas por definição é indemonstrável e não robusta, a presença de falhas de um sistema é demonstrável.

# Capítulo 4. Avaliação da Segurança

Este capítulo aborda os principais métodos de validação usando injeção de falhas. Na Seção 4.1 são apresentadas diferentes técnicas para detectar vulnerabilidades. Na Seção 4.2 descrevemos e exemplificamos o conceito de injeção de falhas. O teste de robustez é exposto na Seção 4.3. Finalmente, na Seção 4.4 revisamos o estado da arte de testes de robustez e segurança para Web Services, detalhando as propostas e ferramentas para descobrir vulnerabilidades.

## 4.1 Técnicas de Detecção de Vulnerabilidades

Devido à existência de vulnerabilidades em Web Services, se desenvolveram uma série de ferramentas, linguagens e técnicas que seguem as boas práticas dos testes de software e os padrões mais adequados para sua aplicação, com o objetivo de detectá-las [16]. Esta validação de segurança em Web Services pode ser realizada em duas fases, a fase estática e a fase dinâmica:

**i)** A fase estática tenta localizar falhas inseridas durante o desenvolvimento do projeto como um estado não alcançável ou possíveis erros humanos introduzidos no código. Nesse caso são utilizados métodos de análise estática (e.g. inspeção de código, analisadores de vulnerabilidade estáticos), ou prova de teorema, os quais não necessitam executar o sistema.

**ii)** A fase dinâmica se foca na verificação da implementação durante sua execução, i.e, verificar o sistema exercitando seu código, onde entradas reais são fornecidas para verificar os mecanismos de segurança; os testes de segurança usando injeção de falhas se enquadram nessa categoria.

Cada fase tem suas técnicas, estáticas e dinâmicas. As **técnicas estáticas** analisam e inspecionam o código; são técnicas de detecção antecipada que carregam muitos benefícios como redução de custo e teste; não precisam da execução dos serviços para sua aplicação. Já as **técnicas dinâmicas** requerem da execução da implementação; nesta categoria temos os Testes de Penetração (TP), Fuzz Testing (FT) e Injeção de Falhas descritos a seguir.

Os **Testes de Penetração (TP)** emulam ataques, com o objetivo de revelar vulnerabilidades. Os testes são automatizados pelo uso de ferramentas denominadas Vulnerability Scanners (VS). Existem diversas VS, tanto comerciais (e.g. HP Web Inspect, IBM

Rational AppScan) quanto de código aberto (e.g. WSDigger, WebScarab). As vulnerabilidades detectadas variam de uma ferramenta para outra. Uma avaliação de várias versões comerciais de VS mostrou que essas ferramentas têm como principais limitações a baixa cobertura das vulnerabilidades existentes e a alta porcentagem de falsos positivos [18]. A vantagem de usar injeção de falhas sobre teste de penetração é que a primeira permitem maior cobertura de ataques.

**Fuzz Testing (FT)** [39] é uma técnica que fornece entradas inválidas, inesperadas ou aleatórias a um sistema e observa possíveis defeitos como lançamento de exceções imprevistas, colapso (*crash*) do cliente ou servidor, entre outros. É uma técnica muito usada para testar a segurança de sistemas computacionais. Como exemplos de trabalhos em Web Services, podemos citar as ferramentas H-Fuzzing [28] e SQL Fuzzing [40]. Uma vantagem dos fuzz tester sobre outras técnicas é revelar a presença de falhas mais difíceis de serem descobertas e que podem ser exploradas por um atacante. No entanto, a cobertura de ataques conhecidos pode ser baixa. Os testes por injeção de falhas permitem maior capacidade de controle dos ataques gerados.

## 4.2 Injeção de Falhas

**Injeção de Falhas** é uma técnica que pode ser utilizada para avaliar aspectos de dependabilidade dos sistemas de computação, podendo ser implementado em hardware ou software, para emular as anomalias, defeitos ou erros no sistema alvo e observar seu comportamento sob um ambiente estressante. Esta técnica se remonta à 1970, quando foi usado para induzir falhas no hardware. Este tipo de injeção de falhas, chamado *Hardware Implemented Fault Injection* (*HWIFI*), emula falhas no hardware. Esta técnica pode ser usada para validar um sistema tolerante a falhas, auxiliando na remoção e prevenção de falhas, minimizando suas ocorrências e severidades [41, 42].

Os protótipos baseados nesta técnica são classificados em [43]: **1) injeção de falhas por hardware**, procura falhas de lógica ou eletrônica para verificar a eficácia de mecanismos tolerantes a falhas implementadas em hardware; **2) injeção de falhas por software**, é usado para injetar falhas que procuram de corromper dados ou código nos sistemas operacionais e aplicações; e **3) injeção de falhas por simulação**, usada na fase de projeto, sendo útil para validar a eficácia de mecanismos de tolerância a falhas e sua dependabilidade, onde as falhas são

introduzidas num modelo do sistema alvo. Nesta pesquisa utilizaremos de **falhas por software** para analisar a robustez dos Web Services.

Neste contexto, as falhas são introduzidas por um injetor – software responsável por injetar falhas no sistema – antes ou durante a execução. Os testes são constituídos por dois conjuntos de entrada: a carga de trabalho (***workload***) e a carga de falhas (***faultload***) [43]. A primeira representa as entradas usuais do sistema, que servem para ativar suas funcionalidades, enquanto a segunda representa as falhas a serem introduzidas no sistema.

A injeção de falhas baseada em mensagem manipula o conteúdo e a troca de mensagens entre nós do sistema alvo, o qual é usado em Testes de Robustez, discutido na Seção 4.4 [11]. Esta técnica utiliza o modo de defeito, que descreve o impacto de defeitos de subsistemas no sistema distribuído. Os modos de defeito comumente assumidos incluem [41]: falhas bizantinas, falhas temporais, falhas de omissão e falhas de crash; **falhas bizantinas** (ou Arbitrárias) podem ser emuladas corrompendo o conteúdo das mensagens, ou enviando mensagens contraditórias a diferentes nós (e.g. duplicar mensagens); **falhas temporais** ou falhas de desempenho podem ser emuladas atrasando a entrega de mensagens por um período maior do que o especificado ou adiantando a entrega de mensagens; **falhas de omissão** podem ser emuladas interceptando algumas mensagens enviadas por um nó (falhas de omissão de envio), ou algumas mensagens recebidas por um nó (falhas de omissão de recebimento); **falhas de crash** podem ser emuladas interceptando todas as mensagens enviadas ou recebidas por um nó específico.

O **ambiente de injeção de falhas (AIF)** descrito na Figura 4.1, consiste em um Sistema de Injeção de Falhas composto por: analisador de dados (*data analyzer*), biblioteca de carga de trabalho (*workload library*), biblioteca de falhas (*fault library*), coletor de dados (*data collector*), controlador (*controller*), injetor de falhas (*fault injector*), gerador de carga de trabalho (*workload generator*), monitor (*monitor*), sistema alvo (*target system*) [43].

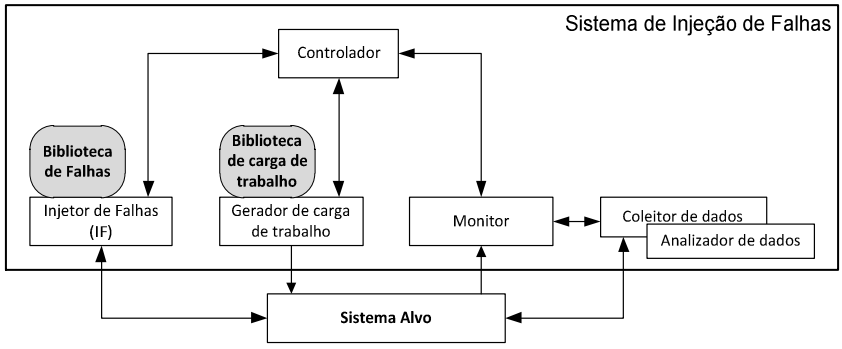



Figura 4.1 Ambiente de Injeção de Falhas.

Como parte da pesquisa, utilizamos o método de **injeção de falhas por software**, centrada no desenvolvimento de ferramentas de software, denominada **injetor de falhas (IF)**. Esta técnica é atraente porque não exige um hardware caro e pode ser utilizada sobre aplicações e sistemas operacionais, o que é difícil com a injeção de falhas por hardware. No entanto, a abordagem de software tem suas deficiências [43]:

- Não pode injetar falhas em locais que são inacessíveis ao software.
- A utilização do software pode perturbar o trabalho em execução no sistema alvo, e até mesmo alterar a estrutura do software original. Um planejamento cuidadoso do ambiente de injeção pode minimizar a perturbação.

Podemos classificar os métodos de injeção de falhas pelo momento em que são injetadas:

- **Injeção na compilação**, durante a compilação, a instrução do programa deve ser modificada antes que a imagem binária seja carregada e executada. Este método injeta falhas a nível de código fonte do programa alvo, emulando efeitos transitórios no hardware e software. O código modificado altera as instruções do programa alvo e a injeção gera uma imagem errônea de software. Quando o sistema alvo é executado, a falha fica ativa [16].
- **Injeção na execução**, durante a execução, são necessário mecanismos para acionar a injeção de falhas. Os mecanismos de ativação utilizados são: **tempo limite**, um temporizador expira em um tempo predeterminado, provocando a injeção; e**xceção/armadilha**, uma exceção de hardware ou software transfere controle para o injetor quando se apresentam certos eventos ou condições; e **inserção de código**, as instruções são adicionadas ao sistema alvo que permitam a injeção de falhas [41].

O uso de **injeção de falhas por software** para obtenção de medidas tais como cobertura de falhas e avaliação do mecanismo de tolerância a falhas em sistemas, tem sido proposta e pesquisada por décadas. A maior preocupação é garantir que as falhas injetadas representem falhas reais, pois é a condição necessária para obter resultados significativos. Como explicado até agora, típicas propostas incluem a inserção de erros em nível de código fonte e a emulação de falhas externas [44]. Contudo, essa técnica também pode ser usada para avaliar a robustez do software, inserindo artificialmente em condições e entradas excepcionais, e observando a resposta e o comportamento do sistema que está sob essas condições.

Uma dificuldade na injeção de falhas é determinar se as saídas produzidas pelo sistema estão corretas ou não; é o denominado **problema do oráculo.** Este problema é comum a qualquer técnica de verificação dinâmica (teste), e não existe uma solução genérica.

Propostas tradicionais de injeção de falhas por software e sistemas de comunicação são provadamente efetivas em capturar defeitos de software usando mensagens trivialmente alteradas, perdidas, duplicadas ou atrasadas, que simulam problemas de transmissão.

## 4.3 Testes de Robustez

A IEEE [27] define **robustez** como "o grau em que um sistema funciona corretamente na presença de fatores excepcionais ou sobre condições estressantes". Robustez é um atributo de dependabilidade, que mede o comportamento de um sistema sob condições não-padronizadas.

Dado os conceitos, podemos definir que **teste de robustez** é uma metodologia de garantia de qualidade voltado a testar a robustez do software, que lida com ativar as falhas ou vulnerabilidades do sistema que resultam em mau funcionamento, denominado como *defeito de robustez*. Segundo Koopman [45] os **defeitos de robustez** podem ser classificados de acordo com os critérios de CRASH, como se observa na Figura 4.2 [27].

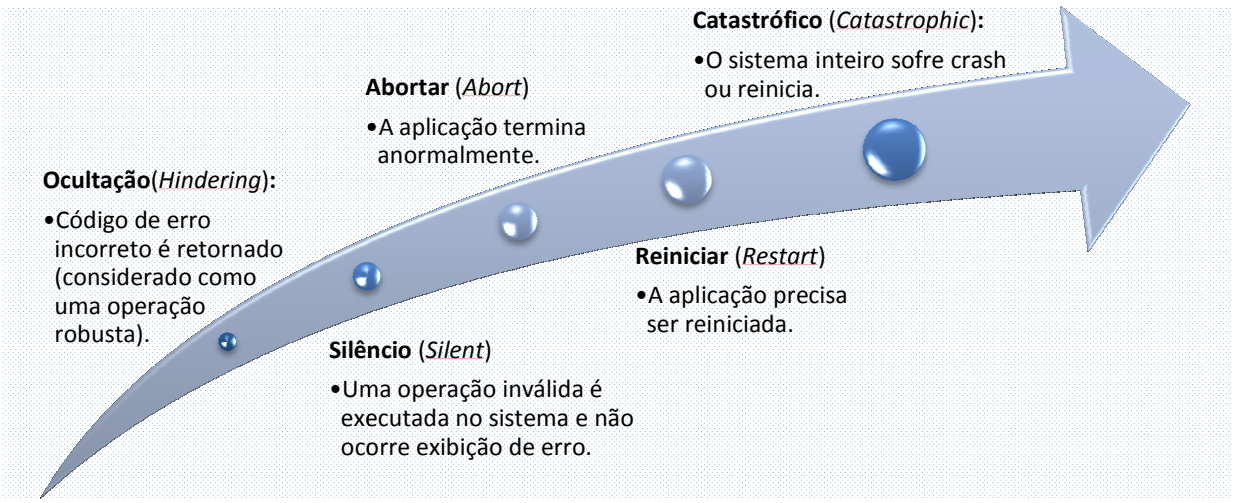



Figura 4.2 Testes de Robustez.

Os testes de robustez podem ser reproduzidos por i**njeção de falhas**. Neste caso, é necessário definir dois conjuntos de entradas: as **atividades** (*workload*), entradas para ativar a funcionamento normal do sistema e as **falhas** (*faultload*), entradas excepcionais e as condições estressantes aplicadas ao sistema. Dependendo de como estas duas cargas são balanceadas, os testes de robustez podem ser usados para propósitos de verificação ou avaliação da robustez.

Nesta abordagem, o foco é obter uma **carga de falhas** emulada representativa, i.e. induzir erros e defeitos que são similares àqueles provocados por falhas em ambientes reais. As abordagens, descritas a seguir, estão categorizadas conforme a maneira como as entradas são escolhidas:

- **Uso de entradas aleatórias:** É uma técnica simples, denominada Teste Fuzz [39], descrita na Seção 4.1. Consiste na geração de entradas aleatórias para o sistema.
- **Uso de entradas inválidas:** Essa técnica consiste em selecionar valores, os quais incluem: valores limites e fora do domínio de entradas permitidas [5].
- **Testes de tipo específico:** Nesta técnica, entradas válidas e inválidas são definidas para os tipos de dados usados nas funções do sistema. Os testes de robustez são gerados combinando valores definidos para todos os parâmetros [11].

## 4.4 Testes Segurança para Web Services

Conhecido como *Security Testing*, permite avaliar as vulnerabilidades em aplicações e serviços frente a diferentes tipos de ataques de segurança – como XPath Injection – e descobrir novas vulnerabilidades antes que sejam exploradas por atacantes. Estes testes são geralmente mal interpretados e, por conseguinte mal desenvolvidos porque os testadores não fazem uso de uma metodologia sistemática para gerar cenários de ataques de acordo com os objetivos dos testes de segurança.

Devido ao fato de que arquitetura orientada a serviços (SOA) e Web Services são usados em contextos heterogêneos, os serviços são obrigados a satisfazer padrões de alta qualidade e testados por ferramentas de teste automatizado. As clássicas técnicas de testes e ferramentas tradicionais já não são adequadas para testar sistemas baseados em SOA. Este novo desafio tem impulsionado a comunidade acadêmica a pesquisar técnicas e desenvolver novas metodologias e ferramentas.

Segundo Melo [46] as estratégias de testes aplicado a Web Services depende do nível de teste abordado e as perspectivas do testador. Indiferentemente da abordagem, os testes podem ser aplicados usando testes funcionais ou não funcionais adaptados a partir de técnicas de componentes. Ambas técnicas são descritas a seguir:

- **Testes funcionais**, segundo Myers [47], os testes funcionais são orientados a avaliar o comportamento externo do componente de software, sem se considerar o

comportamento interno do mesmo. São também chamados testes de caixa preta porque são executados através de dados fornecidos para a entrada, os resultados obtidos são comparados com os resultados esperados, previamente conhecidos, sem conhecer a estrutura interna do sistema. A diferença de outros testes derivados dos detalhes de implementação, os testes funcionais são derivados da especificação do sistema.

- **Testes não funcionais**, Em contraste com os testes funcionais mencionado anteriormente, as técnicas não funcionais verificam atributos de um componente ou sistema que não se relacionam com a funcionalidade, e.g. **testas de segurança**, desempenho ou performance, recuperação, estresse, entre outros [46].

Os Web Services não possuem interface direta com o usuário final, o que os torna mais difíceis de tratar manualmente, porem são bons candidatos para uso de testes não funcionais automatizados [48]. Segundo Canfora e Di Penta [49], os **testes não funcionais** são cruciais em SOA por uma série de razões:

**i)** Provedores de serviços e consumidores usam o SLA[17], em qual o provedor garante aos consumidores certas funcionalidade com um determinado nível de QoS. No entanto, sob certas condições de execução, causados por inesperadas entradas ou cargas de serviço, não garantem o cumprimento da QoS.

**ii)** A ausência de robustez de serviço, i.e. a falta de ações de recuperação adequada para comportamentos inesperados, podem causar efeitos colaterais indesejáveis.

**iii)** Os serviços são frequentemente expostos através da Internet, assim, eles podem estar sujeitos a ataques de segurança, por exemplo XML Injection.

Um passo importante para realizar um teste em Web Services, é determinar os pontos de entrada e o esquema de comunicação descrita no WSDL, reconhecendo as operações, parâmetros e a estrutura básica da mensagem SOAP, que seja susceptível a ataques [50].

## 4.5 Trabalhos Relacionados

Para a presente dissertação, não foram encontrados trabalhos diretamente relacionados, mas sim, trabalhos que abordam a: **1)** utilização de testes de segurança e injeção de falhas; **2)**

[17] SLA é um acordo firmado entre o fornecedor e o cliente, que descreve o serviço fornecido, as metas de nível de serviço, além dos papéis e responsabilidades das partes envolvidas no acordo.

análise de segurança para Web Services; **3)** utilização de árvore de ataques. Apesar disso, um grande número de trabalhos, na área de testes de segurança, são descritos a seguir, incluindo sua abordagem e ferramentas usadas. Alem disso, a Tabela 4.1 apresenta um resumo das principais abordagens relacionadas a nossa pesquisa.

Tabela 4.1Características das abordagens e ferramentas.

| Abordagem / Ferramenta | 1) Testes não funcionais | 2) Acesso ao código fonte | 3) Duração dos experimentos (Curta ou Longa) | 4) Portabilidade | 5) Robustez para Web Services | 6) Robustez para WS-Security |
|---|---|---|---|---|---|---|
| **WebScarab [53]** | ✓ | ✓ | Curta | ✓ | ✓ | |
| **Wsrbench [54]** | ✓ | | Curta | | ✓ | |
| **HPLoadRunner [51]** | ✓ | | Curta | | ✓ | |
| **CDLChecker [52]** | ✓ | ✓ | Longa | | ✓ | |
| **WS-Diamond [55]** | ✓ | ✓ | Longa | | ✓ | |
| **IDEA – Volcano [56]** | ✓ | ✓ | Longa | | | |
| **H-Fuzzing [57]** | ✓ | ✓ | Longa | | | |
| **SQL Fuzzing [40]** | ✓ | ✓ | Longa | ✓ | ✓ | |
| **RV4WS [58]** | ✓ | ✓ | Longa | | ✓ | |
| **Seo - IDS [59]** | ✓ | | Curta | | ✓ | |
| **WS-TAXI [60]** | ✓ | ✓ | Longa | ✓ | ✓ | |
| **SoapUI [6, 60]** | ✓ | ✓ | Longa | ✓ | ✓ | |
| **TCP App [63, 64]** | ✓ | ✓ | Longa | ✓ | ✓ | |
| **VS.WS [65]** | ✓ | | Longa | ✓ | ✓ | |
| **HP WebInspect [18]** | ✓ | | Longa | ✓ | ✓ | |
| **IBM Rational [18]** | ✓ | | Longa | ✓ | ✓ | |
| **Acunetix WVS [18]** | ✓ | | Longa | ✓ | ✓ | |
| **WSInject [7]** | ✓ | ✓ | Curta | ✓ | | ✓ |

Os seguintes aspectos foram considerados em relação a cada uma das abordagens aqui descritas: **1)** A abordagem utiliza testes de segurança ou injeção de falhas; **2)** os autores provêem acesso ao código fonte; **3)** A duração dos experimentos, citados, são de longa ou curta duração; **4)** A ferramenta é fácil de ser portada; **5)** Analisa a robustez para Web Services; **6)** Analisa a robustez pra WS-Security. A seguir descrevemos as pesquisas relacionadas a nosso trabalho:

Em [51], os autores desenvolveram um agente móvel mediante a integração das ferramentas **HP LoadRunner** e **IBM Aglet**, com o intuito de construir um ambiente prático para testar Web Services, além de desenvolver um algoritmo para geração de casos de testes automatizados, que analisa a interface dos serviços e cria casos de testes por injeção de falhas baseados nos limites dos valores nas variáveis (parâmetros). Não entanto, sua implementação requer a instalação e configuração das ferramentas.

**CDLChecker** [52], é uma ferramenta automatizada para testar WS-CDL (*Web Service Choreography Description Language*) através da geração de assertivas, que servem como oráculos. Depois de cada execução, o conjunto de condições coletadas viram novas entradas para a seguinte simulação, prévia análise de um solucionador de SMT (*Satisfiability Modulo Theories*), que decide se as atuais assertivas satisfazem o conjunto de condições. Este método possibilita modificar as assertivas para melhorar a eficiência do SMT, além de ser um modelo reutilizável para analisar diversos tipos de ataques. No entanto, a duração dos experimentos chega a ser extensa.

Existe uma grande quantidade de ferramentas denominadas injetores de falhas (IF) para analisar a robustez de Web Services, entre elas destacamos:

***WebScarab*** [53] é um analisador e injetor de falhas para aplicações web, a ferramenta, descrita em Java, insere uma camada entre o cliente e o servidor, interceptando e modificando as mensagens, quando a ferramenta é configurada.

***wsrbench*** [54], desenvolvida pela Universidade de Coimbra, é uma ferramenta online, disponível na sua web site. Está voltada a testar a robustez em Web Service por meio da injeção de falhas. À finalização dos testes, um e-mail é enviado aos testadores com os resultados. A wsrbench opera em duas fases: 1) o serviço é chamado apenas com entradas válidas para se obter uma medida do funcionamento do serviço (*gold run*); 2) são introduzidas falhas para se verificar a ocorrência de mudanças no comportamento do serviço.

**WS-Diamond** [55], é uma ferramenta que analisa a qualidade de serviços compostos, usando técnicas de injeção de falhas. Os autores avaliaram a capacidade de tolerância a falhas dos Web Services, inspecionando a reação dos processos compostos durante a injeção de falhas. Foram usados as técnicas de caixa preta e caixa branca como parte da metodologia de análise de resultados. Sua falta de portabilidade restringe sua aplicação.

**IDEA** (*Automatic Security Testing for Web Applications*) [56] é uma metodologia, cujo objetivo foi detectar automaticamente as vulnerabilidades em aplicações Web, usando uma técnica dinâmica de descoberta chamada “Análise de Fluxo de Dados”, que melhora a cobertura dos scripts, para a geração automática de dados nos testes. A proposta também desenvolveu **Volcano**, uma ferramenta para testes de segurança, que emula ataques de SQL Injection. Uma vantagem é que a metodologia pode ser re-usável para emular outros tipos de ataques.

**Zhao et al** [57], apresentam um novo método heurístico para geração de dados aleatórios denominado ***H-Fuzzing***, o qual tem uma alta cobertura de testes. Seu objetivo é coletar as entradas aleatórias, gerando uma relação entre elas, para reduzir a grande quantidade de dados aleatórios. É usado um programa Fuzzer que tenta descobrir vulnerabilidades de segurança através do envio de entradas aleatórias para os Web Services.

**SQL Fuzzing Tool** [40], é uma ferramenta que detecta e corrige diversos erros (*bugs*) de segurança durante o ciclo de desenvolvimento do Web Service. A metodologia utilizada, permite acesso ao código fonte, podendo ser aplicada sobre qualquer tipo de linguagem de programação. Do mesmo jeito, fornece uma técnica de penetração e possibilita que a ferramenta seja portável.

**Cao et al** [58], apresentam uma metodologia para fazer testes passivos[18] de conformidade comportamental, baseados em um conjunto de regras de segurança, para Web Services. A metodologia proposta pode ser usada para testar serviços em tempo de execução (*online*) ou não (*offline*). A definição das regras de segurança foram feitas em linguagem Nomad[19], e foi proposto um algoritmo que pode verificar múltiplas instancias simultaneamente. Além disso, o algoritmo foi implementado na ferramenta **RV4WS** (*Runtime Verification engine for Web Service*) que ajuda na automatização dos testes com sua abordagem passiva.

**Seo et al** [59], desenvolve diversos módulos de **IDS** (*intrusion detection systems*) para monitorar e analisar os padrões de ataques contra Web Services. Como parte da proposta, apresenta uma nova classificação de ataques, que os categoriza a partir da causa do ataque.

**Bartolini et al** [60] apresenta um framework chamado **WS-TAXI** (*Web Services Testing by Automatically generated XML instances*) em que se combinam as operações de Web Services com geração de teste, controlada por dados (*data-driven*). Este framework é uma integração de dois software existentes: **soapUI** [6], uma ferramenta conhecida para testar Web Services e TAXI[20], uma ferramenta de geração automática de casos de testes a partir do esquema de XML. **WS-TAXI** oferece um conjunto completo de teste para sua execução.

**Paiva, A. e Eliane Martins** [61, 62], apresentam uma abordagem de injeção de falhas para verificar protocolo de segurança, com o objetivo de detectar vulnerabilidades. Além disso, os autores usam o modelo de árvore de ataque para descrever diversos tipos de ataques conhecidos, derivando cenários de teste de injeção para verificar as propriedades de segurança do

[18] Um teste passivo monitora os resultados de um sistema em execução, sem intromissão nenhuma.
[19] Nomad é uma linguagem de programação de quarta geração (4GL).
[20] http://www.cs.unicam.it/polini/Articoli/AST2007.pdf

protocolo em avaliação. Os cenários de teste são convertidos para um script de injeção de falhas, depois de fazer algumas transformações. O injetor de falhas emula os ataques. O atacante é emulada usando o injetor de falhas. Esta abordagem baseada em modelos facilita a usabilidade e durabilidade dos ataques de injeção gerados, bem como a geração de scripts de injeção de falhas.

**Laranjeiro et al** [63], propõem uma metodologia para testar Web Services, com o objetivo de detectar problemas de robustez e atenuar os mesmos a través da aplicação de **TPC-App** Web Services[21], que analisa o desempenho dos serviços. Os resultados mostram que esta ferramenta pode ser facilmente utilizado por desenvolvedores para melhorar a robustez das implementações de Web Services.

**Marco Vieira et al** [18], [64-69], propõem diferentes abordagens de detecção de vulnerabilidades para injetar falhas, usando três ferramentas: **1)** ***HP WebInspect***, realiza testes de segurança em aplicações Web, avaliando sua robustez; **2)** ***IBM Rational AppScan***, é uma suíte de ferramentas de segurança automatizada para aplicações Web. **3)** ***Acunetix Web Vulnerability Scanner***, é uma ferramenta de teste de segurança automatizado para aplicações Web e Web Services, que audita as mesmas através da verificação de vulnerabilidades exploráveis por invasão, usado para executar testes de penetração. Descrevemos os artigos a seguir:

Em [18], os autores apresentam uma avaliação experimental de vulnerabilidades de segurança. Pelo qual, quatro VS (***HP WebInspect V1*** e ***V2***, ***IBM Rational AppScan***, ***Acunetix Web Vulnerability Scanner***) foram usados para identificar falhas de segurança em 300 Web Services públicos. Os resultados indicam que muitos serviços são vulneráveis a diferentes tipos de ataques.

Em [66], os autores comparam duas técnicas, tanto testes de penetração quanto análise estático de código, que são utilizados para detectar vulnerabilidades por SQL Injection nos Web Services. Para compreender os pontos fortes e limitações das técnicas, diversas ferramentas de código aberto e comerciais foram usadas para detectar vulnerabilidades nos serviços. Os resultados sugerem que analisadores de código estáticos são capazes de detectar mais vulnerabilidades de SQL Injection que as ferramentas de teste de penetração.

Em [65], os autores desenvolveram um scanner de vulnerabilidade chamado ***VS.WS***, que foi comparado, juntamente com outros scanners comerciais, focando a cobertura de detecção de vulnerabilidades por ataques de SQL Injection. Alem disso, foi proposto uma metodologia para

[21] http://www.tpc.org/tpc_app/default.asp

comparar o nível de cobertura de ataques para VS. Os resultados indicam, que cada ferramenta tem uma cobertura baixa de detecção de vulnerabilidades e uma alta taxa de falsos positivos.

Em [64], os autores abordam uma metodologia de Benchmarking TPC-App para avaliar e comparar a eficácia das ferramentas de detecção de vulnerabilidades por ataques de SQL Injection em Web Services. Os resultados demonstram que o benchmark retrata a eficácia das ferramentas de detecção de vulnerabilidade e sugerem que o método proposto pode ser aplicado em cenários reais.

Em [67-69] propõem uma abordagem para proteger aos Web Services contra ataques de SQL Injection e XPath Injection. Alem disso, avaliam e comparam o desempenho e o tempo de recuperação das infra-estruturas de Web Services frente a estes ataques, usando diversas soluções de benchmarking.

De fato a maioria dos trabalhos apresentados até agora são baseados na técnica de injeção de falhas e testes de penetração. Apenas os trabalhos de Paiva [61, 62] usam uma modelagem de ataques, que pode ser utilizada para representar diversos tipos de ataques para a geração de casos de testes, os quais poderiam ser portados para outras ferramentas. Nossa proposta tem como objetivo gerar casos de testes baseados em diversas classes de ataques reais (descrito na Seção 3.2), os quais são previamente modelados variando o mínimo possível de parâmetros, e assim diminuindo drasticamente a carga de falhas e aumentando a eficiência da metodologia.

Neste trabalho, focamos em testes de segurança para Web Services com o padrão WS-Security usando a fase dinâmica (verificação durante a execução). Nossa abordagem se baseia em ataques reais e modelos desses ataques reais, o que permite o reuso da carga de falhas, permitindo também usar outros injetores.

# Capítulo 5. Geração de Ataques

Uma das dificuldades para encontrar vulnerabilidades em Web Services, durante a fase de execução, é determinar os **cenários de ataque** apropriados para os testes. Estes cenários podem ser obtidos de diversas fontes, i.e. Internet, livros, entre outras. Contudo, é demorado encontrar e armar um banco de ataques relevantes e automatizá-los de acordo com o ambiente de testes. Nosso propósito neste capítulo é utilizar, parcialmente, a metodologia de ataques desenvolvida por Morais e Martins [5], usando a abordagem descrita na Figura 5.1, para desenvolver o projeto de injeção de falhas para Web Services e o padrão WS-Security.

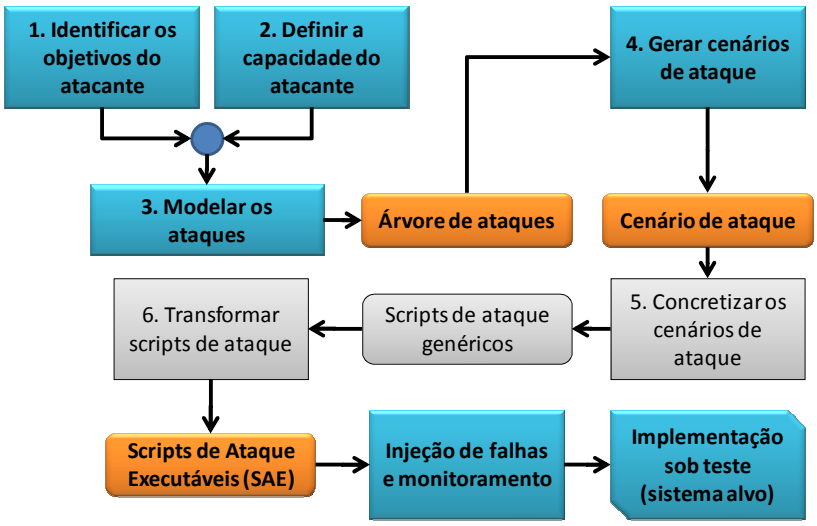



Figura 5.1 Metodologia de testes de Segurança.

A Figura 5.1 ilustra as etapas da abordagem, descritas no restante do capítulo. A aplicação desta metodologia será parcial, devido à utilização de ferramentas como WSInject e soapUI, que emulam diversos tipos de ataques. No entanto, é necessário refinar os cenários de ataque e monitorar a injeção de falhas para obter melhores resultados.

## 5.1 Identificação dos Objetivos do Atacante

Para identificar os objetivos do atacante foi necessário fazer uma pesquisa sobre vulnerabilidades em Web Services com o objetivo de reunir informações sobre propriedades de segurança vulneradas a diversos ataques. Para isso, se decidiu pesquisar artigos e livros que apresentem vulnerabilidades no contexto de Web Services.

Tabela 5.1 Ataques organizados por Propriedades de Segurança para Web Services.

| ***Proprie-dades*** | **Ataques** | **Referências** | **Classificação de Ataques** | **Nível de camada de ataque** | **Tam.** | **Medir o impacto** | **WSS protege aos serviços deste ataque?** |
|---|---|---|---|---|---|---|---|
| ***P1. Confiden-cialidade*** | *A.01. WSDL Scanning* | [20-22], [24] | *Spoofing Attacks* | Mensagem | 1 | Medium | Protege |
| | *A.02. Insufficient Transport Layer Protection* | [19] | *Spoofing Attacks* | Processo | 1 | Medium | Protege |
| ***P2. Integri-dade*** | *A.03. Metadata Spoofing* | [20, 21], [24] | *Spoofing Attacks* | Mensagem | 1+ | Medium | Não Protege |
| | A.04. XML Injection | [19-22], [64] | *Injection Attacks* | Mensagem | 1 | High | Protege |
| | A.05. SQL Injection | [19], [22], [24] | *Injection Attacks* | Mensagem | 1 | High | Protege |
| | *A.06. XPath Injection* | [6], [23] | *Injection Attacks* | Mensagem | 1 | High | Protege |
| | *A.07. Cross-site Scripting (XSS)* | [19] | *Injection Attacks* | Mensagem | 1+ | High | Protege |
| | *A.08. Fuzzing Scan* | [6], [57] | *Injection Attacks* | Mensagem | 1 | High | Protege |
| | *A.09. Invalid Types* | [6], [20, 21] | *Injection Attacks* | Mensagem | 1 | Medium | Protege |
| | *A.10. A. Parameter Tampering* | [4], [6], [19], [22], [24], [54] | *Injection Attacks* | Mensagem | 1 | Medium | Protege |
| | *A.11. Malformed XML* | [6] | *Injection Attacks* | Mensagem | 1 | High | Protege |
| | *A.12. Frankenstein Message: Modify Timestamp* | [25] | *Injection Attacks* | Mensagem | 1+ | Medium | Protege |
| ***P3. Disponibi-lidade*** | *A.13. Replay Attack* | [24] | *Denial-of-Service A* | Mensagem | 1 | Low | Não Protege |
| | *A.14. Oversize Payload* | [19-22], [24] | *Denial-of-Service A* | Mensagem | 1 | Medium | Não Protege |
| | *A.15. Coercive Parsing (Recursive Payloads)* | [19-22], [24] | *Denial-of-Service A* | Mensagem | 1 | Medium | Não Protege |
| | *A.16. Oversize Cryptography* | [19-21], [24] | *Denial-of-Service A* | Mensagem | 1 | Medium | Não Protege |
| | *A.17. Attack Obfuscation* | [19-21], [24] | *Denial-of-Service A* | Mensagem | 1 | Medium | Não Protege |
| | *A.18. XML Bomb* | [6] | *Denial-of-Service A* | Mensagem | 1 | Medium | Não Protege |
| | *A.19. Unvalidated Redirects and Forwards* | [19] | *Denial-of-Service A* | Mensagem | 1 | Medium | Não Protege |
| | A.20. Attacks through SOAP Attachment | [6], [25] | *Denial-of-Service A* | Mensagem | 1 | Medium | Não Protege |
| | A.21. Schema Poisoning | [24] | *Denial-of-Service A* | Mensagem | 1 | Medium | Não Protege |
| | *A.22. Insecure Cryptographic Storage* | [19] | *Brute Force Attacks* | Banco de Dados | 1 | High | Não Protege |
| | *A.23. WS-addressing Spoofing* | [5], [20, 21] | *Spoofing Attacks* | Mensagem | 2 | Medium | Protege |
| | *A.24. Middleware Hijacking* | [4], [20-22] | *Spoofing Attacks* | Processo | 2+ | Medium | Não Protege |
| | A.25. Maliciuos Content | [26] | *Spoofing Attacks* | Mensagem | 1 | Medium | Não Protege |
| | *A.26. Instantiation Flooding* | [20, 21], [24] | *Flooding Attacks* | Processo | 1 | High | Não Protege |
| | *A.27. Indirect Flooding* | [20, 21] | *Flooding Attacks* | Processo | 2+ | High | Não Protege |
| | *A.28. BPEL State Deviation* | [20, 21] | *Flooding Attacks* | Processo | 1 | High | Não Protege |
| ***P4. Controle de Acesso*** | *A.29. Broken Authentication and Session Management* | [19] | *Brute Force Attacks* | Mensagem | 1 | High | Protege |
| | *A.30. SOAPAction* | [20, 21], [25] | *Spoofing Attacks* | Mensagem | 1 | Medium | Protege |
| | *A.31. Security Misconfiguration* | [19] | *Spoofing Attacks* | Todos os níveis | 1+ | Medium | Protege |
| | *A.32. Unauthorized Access* | [4] | *Spoofing Attacks* | Mensagem | 1 | Medium | Protege |
| | *A.33. Routing Detours* | [24] | *Spoofing Attacks* | Mensagem | 1+ | Medium | Protege |
| | A.34. Attack on WS-Trust, WS-SecureConversation | [25] | *Spoofing Attacks* | Mensagem | 2+ | Medium | Protege |
| | *A.35. Cross-site Request Forgery* | [19] | *Injection Attacks* | Mensagem | 2+ | Medium | Protege |
| | *A.36. Attack on WS-Security* | [25] | *Spoofing Attacks* | Mensagem | 1+ | Medium | Protege |

Enquanto algumas das vulnerabilidades são causadas por deficiências de implementação, a maioria delas explora falhas básicas do protocolo, i.e. abusando da flexibilidade de SOAP. Estes ataques estão disponíveis em diferentes fontes de informação e servirão de base para modelar os ataques na etapa 3 (Seção 5.3). Para maior detalhe sobre os ataques, consulte a Seção 3.2 e Apêndice A.

A Tabela 5.1 contém os 36 tipos de ataques para Web Services, reunindo várias pesquisas sobre vulnerabilidades. Eles estão organizados a partir dos seguintes campos:

i. **Propriedades de segurança violadas**, composto por confidencialidade (**C**), integridade (**I**), disponibilidade (**D**), controles de acesso (autorização e autenticação - **CA**);
ii. **Nome do ataque;**
iii. **Referências**, onde o leitor pode encontrar mais informação do ataque;
iv. **Classificação do ataque**, usando o modelo de classificação de [23], que categoriza em 5 tipos de ataques: Denial-of-Service Attacks, Brute Force Attacks, Spoofing Attacks, Flooding Attacks, Injection Attacks;
v. **Nível de camada de ataque** que descrevem o nível onde será inserido o ataque e descreve características inerentes a uma vulnerabilidade (**nível de mensagem**, **nível de processo** e **nível do banco de dados**);
vi. **Tamanho**, indica o "número habitual ou mínimo de ataques" para atingir o objetivo;
vii. **Medir o impacto**, para avaliar esta campo, usamos a medição de impacto da OWASP Top 10 [19], que classifica seus ataques em três níveis (**baixa**, **média** e **alta**). Esta informação, juntamente com o estudo de diversas vulnerabilidades no Capítulo 7 e 8, nos permitiu classificar cada um dos ataques.
viii. **WS-Security (WSS) protege o serviço deste ataque?**, WS-Security protege a confidencialidade, integridade, autenticação e autorização, pelo qual podemos responder se: **Protege** e **Não Protege**.

## 5.2 Definição da Capacidade do Atacante

O modelo de intrusão de **Dolev-Yao** [70] – implementado na maioria das abordagens existentes de verificação estática da segurança, supõe que o intruso possui todos os meios para

interferir na rede e pode capturar o tráfego de rede desejado para análise. Supõe-se também que o intruso possui tempo ilimitado para atacar a rede, e que suas capacidades, em termos de memória disponível e tempo de processamento são ilimitadas para quebrar o sistema. Além disso, o intruso pode interceptar ou emitir mensagens, dividir ou constituir mensagens, e pode regenerar mensagens, mas ele não pode evitar que participantes legítimos recebam mensagens. Também se assume que as funcionalidades criptográficas usadas no protocolo são perfeitas, i.e. ataques de criptografia não são possíveis de serem executados.

Portanto, baseado no modelo de Dolev-Yao, consideramos que o atacante possui as seguintes **capacidades**:

- Controle parcial da rede, podendo capturar mensagem SOAP para simples análise.
- Capacidade de **interceptar**, **corromper** (modificar, remover, introduzir) cadeias ou expressões, **atrasar** ou **duplicar** (replicar) mensagens do tráfego.
- Conhecimento do estado de todos os participantes, i.e. o atacante intercepta as mensagens, podendo fazer o papel do cliente ou apenas atua como mediadora de comunicação entre este e o servidor.
- O atacante sabe reconhecer os pontos de acesso, operações e parâmetros de um arquivo WSDL do Web Services testado.

A capacidade do atacante atual pode ser atualizada ou redefinida de acordo com as necessidades do ambiente de teste usado ou devido a outras necessidades específicas.

## 5.3 Modelagem dos Ataques

Uma vez que as ameaças ao sistema de comunicação foram identificadas, utilizamos o método de **árvore de ataques** (descrito na Seção 3.4) a fim de que os ataques possam ser obtidos de preferência automaticamente. A partir das informações de ataques disponíveis em linguagem natural – obtidas na etapa 1 (Seção 5.1) – são definidos os requisitos de ataque para construir a Árvore. A abordagem que usamos para analisar e sistematizar as informações de ataques é a mesma usada no trabalho de Edge et al [71].

Dado que não existem orientações específicas para construir **árvores de ataques** propomos que a mesma seja construída de uma maneira topo base (*top-down*), i.e. começa-se pela raiz até chegar às folhas para facilitar a categorização dos ataques de acordo com as

propriedades de segurança identificadas na etapa 1 (Seção 5.1) e dividir os ataques de acordo com os mecanismos e elementos do sistema alvo explorado. Os passos para a construção são:

1. O nó raiz representa o objetivo genérico final de todos os ataques, que é obter sucesso contra a implementação do protocolo ou padrão alvo. Portanto o nó raiz é do tipo **OR**.
2. O segundo nível representa as **"propriedades de segurança violadas pelos ataques"**, de acordo com a categorização feita na etapa **1** (Seção 5.1). Cada nó desse nível representa uma propriedade.
3. O terceiro nível representa os **mecanismos** explorados pelo atacante, e.g. Coercive Parsing, Cross-Site Scripting , entre outros, para violar as propriedades de segurança do nível 2. Essa informação é obtida da **descrição do ataque** na Seção 5.1. Se a "**descrição** do ataque" não disponibiliza essa informação esse nível é omitido.
4. Níveis subsequentes representam as "descrições dos ataques" ou passos da descrição de um ataque, que foram identificados na etapa **1** (Seção 3.2 e Apêndice A), para realizar os sub-objetivos do nível 3.
5. Por último, são associados atributos a cada nó folha, i.e. valores lógicos de acordo com a capacidade do atacante definida na Seção 5.2. Outros atributos relacionados às informações coletadas das vulnerabilidades na etapa **1** (Seção 5.1) também podem ser considerados, como o nível da camada do ataque ou medição do impacto.

Esse modelo pode ser atualizado na medida em que novos ataques são descobertos.

## 5.4 Geração de Cenários de Ataque

Nesta etapa os cenários de ataque são produzidos de forma automática segundo o critério descrito na Seção 3.4, que indica quais atributos associados aos nós devem ser considerados para realizar a busca na árvore, a fim de cobrir todos os ataques que satisfaçam estes atributos. O critério usado é "cobrir todos os cenários possíveis de acordo com a capacidade do atacante", onde foram considerados os atributos da capacidade do atacante descritos na etapa 3 (Seção 5.3) para a seleção dos cenários. Esta etapa é completamente automatizada, pois a ferramenta que auxilia na construção da **árvore de ataques** também seleciona os **cenários de ataque**. A saída dessa etapa são os **cenários de ataque** descritos no mesmo formato das folhas da árvore, que possuem a **descrição do ataque**. Os cenários obtidos podem ser usados para criar uma biblioteca de ataques, o qual é útil para testar outros protocolos, facilitando sua reusabilidade [5].

## 5.5 Concretização dos Cenários de Ataque

Os **cenários de ataque** gerados na etapa 4 (Seção 5.4) estão descritos em linguagem textual, i.e. no mesmo nível de abstração da árvore de ataques. Esse tipo de descrição chega a ser útil para os analistas de teste e especialistas em segurança por sua fácil configuração, mas não para ser processado por uma ferramenta. Nesta etapa, os analistas devem realizar um conjunto de passos de refinamento, com o intuito de transformar os cenários de ataque em linguagem textual para um script executável pela ferramenta, sobre o contexto de Web Services. Esta etapa foi definida para permitir que os scripts executáveis sejam obtidos de forma automática, análoga a [5], onde os autores prepararam scripts para um injetor que atuava no nível do kernel Linux.

### 5.5.1 Primeiro Passo – Padrão de Ataque

Utilizamos o conceito de **padrão de ataque**, descrito na Seção 3.4.2, para refinar os cenários de ataque de forma estruturada. Isto nos permite: **1)** caracterizar o ataque em elementos bem definidos, com passos específicos para realizar o ataque; e **2)** descrever os requisitos necessários para gerar o mesmo. A seguir descrevemos os elementos que lho conformam:

i. **Objetivo do ataque**, sua finalidade é detalhar o cenário de ataque.
ii. **Lista de precondições**, é um conjunto de requisitos para que o ataque aconteça. A lista é extraída da **descrição do ataque**, como eventos na rede, com o intuito de ativar o ataque, e.g. um pacote é enviado do servidor para um cliente.
iii. **Passos para executar o ataque**, são tarefas que o atacante deve executar para que o ataque tenha sucesso. Também são extraídos da **descrição do ataque**.

Para esta etapa de concretização do cenário de ataque, não é necessário utilizar a **lista de pós-condições** do **padrão de ataque**, caso o ataque seja bem sucedido. Para exemplificar utilizaremos o ataque de **XML Injection** descrito na Figura 5.2, que injeta uma requisição maliciosa, como parte da mensagem SOAP, enviada pelo cliente. Seu objetivo é modificar a estrutura XML de uma mensagem (ou qualquer outro documento XML) através da inserção de etiquetas (*tags*), para executar uma operação restrita.

A **descrição do ataque** seria: se o atacante intercepta a mensagem SOAP e for uma requisição e se existe uma cadeia <String>, modificar os parâmetros de operação por um parâmetro não padrão, fazendo uso da informação fornecida pelo WSDL, será enviada uma mensagem SOAP alterada ao serviço. Se o Web Service usa algum esquema seguro de

comunicação como HTTPS ou algum padrão de segurança como WS-Security (XML Encryption), esta requisição possivelmente será rejeitada. Se o Web Service não fornece pelo menos uma operação, não será possível fazer a injeção do XML mal formado. As "propriedades de segurança violadas pelo ataque" são integridade e controle de acesso (autorização e autenticação). O cenário para esse ataque seria <Ataque de XML Injection para modificar a estrutura XML>, que conteria a **descrição do ataque** e a "propriedade de segurança violada pelo ataque". A descrição fará uso dos seguintes elementos: **i)** o objetivo do ataque; **ii)** a lista de precondições; **iii)** os passos para executar o ataque. A saída desse passo é o cenário de ataque mostrado na Figura 5.2.

| | | |
|---|---|---|
| 1: | **Objetivo**: | Ataque de XML Injection para modificar a estrutura XML. |
| 2: | **Precondições**: | O cliente envia uma requisição ao Web Services através de uma mensagem SOAP. |
| 3: | | O cliente não usa um esquema seguro de comunicação. |
| 4: | | O WSDL descreve ao menos um parâmetro. |
| 5: | **Ataque:** | |
| 6: | AND | 1. Caso seja requisição. |
| 7: | | 2. Caso contenha a cadeia <String> procurada. |
| 8: | | 3. Modificar os parâmetros da requisição por uma mensagem não padrão. |
| 9: | | 4. Gerar a nova mensagem SOAP. |

Figura 5.2 Padrão de ataque para o Cenário de Ataque.

## 5.5.2 Segundo Passo – Regra ECA

O *padrão de ataques* permite descrever o cenário de ataque de uma maneira mais estruturada e sobre uma linguagem natural. Nessa seção utilizamos a notação ***e**vento-condição-acaõ* (**ECA**) [72] para a representação do cenário descrito na forma de **padrão de ataque**. A razão de usar a regra ECA são: **1)** sua sintaxe simples; e **2)** sua facilidade para representar comportamentos reativos usando apenas uma regra, por exemplo, ações executadas como efeito ou consequência de eventos e condições. A regra **ECA** possui uma sintaxe genérica que funciona da seguinte forma: **ON** *evento* **IF** *condição* **DO** *ação*, em que:

- **Evento** é uma atividade externa, como a recepção ou transmissão de uma mensagem específica que indica quando será ativado a regra, e.g. corresponde ao elemento **Precondição** do padrão de ataque da Figura 5.2.
- **Condição** determina os casos em que a regra é ativada, podendo ser estabelecida em termos de: **1)** conteúdo de campos da mensagem; ou **2)** valor de alguma variável de estado. A descrição do **padrão de ataque** corresponde às restrições para a execução das ações. Na Figura 5.2, identificamos por "Caso" e procedemos a sua descrição [73].

- **Ação** são às atividades realizadas pelo atacante e levadas em consideração pela **capacidade do atacante** descritas na Seção 5.2, caso a regra fosse ativada.

Utilizamos um vocabulário de **palavras-chave** para descrever as partes de ***e**vento-**c**ondição-acão* da regra **ECA**. Esse método é usado na abordagem de "teste dirigido por palavras-chave[22]" ou teste dirigido por tabela, onde as **palavras-chaves** são determinadas por um testador, de acordo com o domínio do software utilizado pelos testes e durante a fase de planejamento. Desta forma, os casos de testes podem ser descritos de maneira independentes da ferramenta usada para executá-los [5].

Tabela 5.2 Ações de Falhas de Interface e Falhas de Comunicação.

| *AÇÕES POR FALHAS DE INTERFACE* | | |
|---|---|---|
| **NOME** | **PALAVRAS-CHAVES** | **DESCRIÇÃO** |
| **StringCorruption Fault** | **stringCorrupt**(String fromString, String toString) | Substitui as ocorrências de fromString por toString. Trata os dados apenas como Strings, ignorando completamente a sintaxe XML. Pode ser usada para substituir caracteres XML, como '<' e '>'. |
| **XPathCorruption Fault** | **xPathCorrupt**(String xPathExpression, String newValue) | Substitui todas as casos de uma expressão XPath[23] pelo valor especificado. Pode ser usada para modificar atributos ou elementos. |
| **Multiplication Fault** | **multiply**(String xPathExpression, int multiplicity) | Multiplica parte de uma mensagem por um número específico de repetições. Exemplos: multiply("/", 2) duplica todo o conteúdo da mensagem. multiply("/Envelope/MyElement", 3) triplica somente o elemento "MyElement". |
| **Emptying Fault** | **empty**() | Esvazia a mensagem SOAP, entregando uma mensagem HTTP sem qualquer conteúdo. |
| **CoerciveParsing Fault** | **coerciveParse**(int depth) | Cria uma sequência de tags de abertura de elementos XML sem fechá-los, permitindo criar uma árvore XML muito profunda. |
| *AÇÕES POR FALHAS DE COMUNICAÇÃO* | | |
| **DelayFault** | **delay**(int Milliseconds) | Atrasa a entrega de uma mensagem em milissegundos. |
| **ConnectionClosing Fault** | **closeConnection**() | Fecha abruptamente a conexão entre cliente e Proxy. |

A *ação* pode conter vários passos. Sua sintaxe é estabelecida em relação ao conjunto de **palavras-chave** da Tabela 5.2, onde o domínio é o padrão de segurança WS-Security. As **palavras-chave** são classificadas pelas falhas de interface e comunicação, não relacionadas à capacidade do atacante, que são definidas na Tabela 5.3. Nesta tabela se exibe os dados da regra **ECA**, que contém as **palavras-chaves** com sua correspondente legenda, descrevendo outras

[22] http://safsdev.sourceforge.net/FRAMESDataDrivenTestAutomationFrameworks.htm
[23] http://www.w3.org/TR/xpath/

palavras usadas na comunicação. O domínio de software considerado é um sistema de comunicação, onde participantes trocam mensagens SOAP [48].

Tabela 5.3 Dados da regra ECA mais a legenda da Tabela.

| PARTE | PALAVRAS-CHAVE |
|---|---|
| **EVENTO** | env(A,B,String,<EP>,Po_A>,<Po_B>)<br>rec(B,A,String,<EP>,<Po_B>,<Po_A>) |
| **CONDIÇÃO** | contains(String stringPart)<br>uri(String uriPart)<br>isRequest()<br>isResponse()<br>soapAction(String soapAction) |
| **AÇÃO** | GenerateNewMessage(message)<br>stringCorrupt(String fromString, String toString)<br>xPathCorrupt(String xPathExpression, String newValue)<br>multiply(String xPathExpression, int multiplicity)<br>empty()<br>coerciveParse(int depth)<br>delay(int delayInMilliseconds<br>closeConnection() |
| **Legenda** | **env:** função de envio de mensagem. **rec:** função de recepção de mensagem.<br>**A:** remetente. **B:** destinatário.<br>**String:** mensagem enviada ou recebida. **<EP>** especificação de protocolo (e.g. SOAP).<br>**<Po_A>** porta de envio da máquina do remetente.<br>**<Po_B>** porta de destino da máquina do destinatário.<br>**contains(String stringPart):** seleciona mensagem SOAP contento a string especifica.<br>**uri(String uriPart):** seleciona tanto as requisições enviadas à URI contendo a string especifica como as respostas a essas requisições.<br>**isRequest():** seleciona somente requisições, tanto de um cliente para um serviço como de um processo BPEL para um serviço parceiro.<br>**isResponse():** seleciona somente respostas, tanto de um serviço para um cliente como de um serviço parceiro para um processo BPEL.<br>**soapAction(String soapAction):** seleciona as requisições contendo SoapAction nos cabeçalhos HTTP. |

Consideramos dois tipos de ações, por *falhas de interface* e por *falhas de comunicação* descritas na Tabela 5.2. As primeiras são aquelas que afetam o conteúdo das mensagens, como a corrupção de dados. As segundas afetam a entrega das mensagens, como a queda da conexão.

Naturalmente a semântica será estruturada em: **ON** *evento* **IF** *condição* **DO** *ação*, i.e. a partir do momento que a atividade *evento* e *condição* (regra) são satisfeitas, a regra ativa a parte *ação*, que por sua vez executa as atividades do atacante. A partir dessa semântica definiremos

cada regra **ECA** usando as partes *condição* e *ação*, sendo que a atividade *evento* é usada para todas as regras. Também várias atividades de *condição* podem ser agrupadas em uma atividade *condição* semanticamente vinculada pelo operador lógico **AND**, onde todas as *condições* são satisfeitas para a execução da regra.

Como exemplo, podemos usar o padrão de ataque da Figura 5.2, o qual é convertido para o formato **ECA** na Figura 5.3. O elemento "**Precondição**" é mapeado para a parte *evento*; os elementos 3 e 4 ("*passo para executar o ataque*") são mapeados para a parte de *ação*, pois "*stringCorrupt(String fromString, String toString)*" – corrução_cadeia(Cadeia_receptor, Cadeia_corrupta) – é uma **palavra-chave** estabelecida e GenerateNewMessage(message) também; os elementos, 1 e 2 são mapeados para a parte *condição*. A saída desse passo é o **padrão de ataque** do **Cenário de Ataque** descrito no formato **ECA** usando **palavras-chave**.

| **Regra 1:** | |
|---|---|
| **ON** *evento*: | env(A,B,String,<EP=SOAP,<Po_A>,<Po_B>) |
| **IF** *condição*: | (1. isRequest() == True) AND (2. contains(String) == True) |
| **DO** *ação*: | 3. stringCorrupt(String, String_Corrupt)<br>4. GenerateNewMessage(message) |

Figura 5.3 Formato ECA para padrão de ataque XML Injection.

O injetor de falhas (IF) WSInject utiliza os scripts executáveis dos **cenários de ataque** a partir do Formato **ECA**, prontos para sua utilização nos testes de segurança, através das palavras-chave descritas na Tabela 5.2. Esta ferramenta junto com o *vulnerability scanner* (VS) soapUI são descritos no Capítulo 6 e 7.

Apesar das etapas a seguir não serem parte da metodologia proposta, que propõe a geração de **script de ataque executável** (**SAE**) a partir de um modelo de ataques e obtido dos **scripts de ataque genéricos** (**SAG**), vale a pena fazer algumas considerações, dado que é importante na validação experimental. Enquanto os ataques são injetados, é útil monitorar e coletar as mensagens sendo trocadas, para determinar se os ataque planejados foram efetivamente realizados, ou ainda, para monitorar o uso de recursos [5].

# Capítulo 6. Estudo de Caso

Neste capítulo apresentamos a aplicação da abordagem descrita no capítulo 5 para testes de segurança em Web Services (WS). Como estudo de caso foi usado Web Services e WS-Security, membro da família WS-* de padrões de Web Services e publicado pela OASIS, o qual foi exposto na Seção 3.3. As razões para a escolha deste padrão foram: **i)** as vulnerabilidades de Web Services são bem conhecidas e documentadas; **iii)** as implementações existentes são recentes para experimentar com testes de segurança; **ii)** WS-Security é uma extensão flexível e rica em recursos para SOAP para aplicar segurança aos Web Services e proteger aos Web Services contra diversos tipos de ataques.

O presente capítulo ilustra as etapas da metodologia proposta no Capítulo 5. Na Seção 6.1, analisamos a estrutura da especificação **Username Token** como parte de WS-Security. Na Seção 6.2 identificamos os **objetivos do atacante** extraídas de diversas fontes, descritos na etapa 1. Na Seção 6.3 explicamos o funcionamento do injetor de falhas (IF) WSInject, que implementará a **capacidade do atacante**, detalhado na etapa 2. Na seção 6.4 modelamos a XML Injection para atacar o padrão WS-Security, através da **arvore de ataque** detalhado na etapa 3. Na Seção 6.5 geramos os **cenários de ataque** a partir da **árvore de ataques** construída na etapa 4, como também utilizamos o Formato **ECA** para obter os **scripts de ataques executáveis** (**SAE**), a ser utilizados pelo IF WSInject, com o objetivo de testar a segurança nos Web Services, descritos na etapa 5.

## 6.1 Web Services com WS-Security e Security Tokens

Nesta seção, fazemos uso de diversos conceitos da Seção 3.3 e 3.4, que proporcionam um estado da arte da segurança para Web Services e credencias de segurança (Security Tokens).

A especificação Security Token [31] faz uso do tipo de conexão fim-a-fim, que comprova a identidade do cliente por meio das credenciais de segurança. O protocolo SOAP, descrito na Seção 2.1.1, fornece um framework dentro da mensagem, sob o qual os Web Services podem ser construídos e transportados por outros protocolos como HTTP. Este protocolo (SOAP) contém um envelope que se divide em duas partes: um cabeçalho (*Header*) que armazena informações específicas que possam ser de utilidade para os intermediários da mensagem SOAP (como

autenticação) e, o corpo (*Body*) que contém a mensagem para ser transmitida ao receptor [32]. Na Figura 6.1 mostramos a estrutura de uma mensagem SOAP [32].

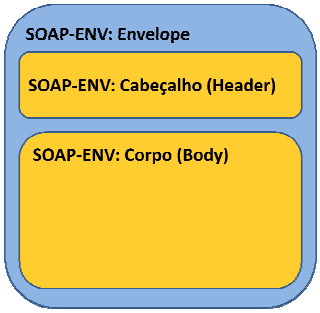



Figura 6.1 Estrutura do Protocolo SOAP.

Nesta etapa, analisaremos a estrutura de uma mensagem SOAP usando a etiqueta <Security> e <UsernameToken > no cabeçalho (Header). Este padrão utiliza *Password_Digest*, compostos por: **i)** a senha de texto claro; **ii)** um Nonce; e **iii)** o tempo da criação do Security Token, para proteger a mensagem SOAP. Esta função criptográfica usa SHA-1 em base 64. Resumindo: Password_Digest = Base64(SHA-1(Nonce + Created + Password)), como podemos observar na Figura 6.2 [32].

```
1:   <UsernameToken>
2:          <Username>MyName</Username>
3:          <Password Type="PasswordDigest">fm6SuM0RpIIhBQFgmESjdim/yj0=</Password>
4:          <Nonce>Pj+EzE2y5ckMDx5ovEvzWw==</Nonce>
5:          <Created>2004-05-11T12:05:16Z</Created>
6:   </UsernameToken>
```

Figura 6.2 Autenticação por senha, Nonce e tempo de criação.

## 6.2 Objetivos do Atacante

As vulnerabilidades de segurança e ataques contra Web Services são definidas na Seção 5.1. Estes ataques foram coletados de diferentes trabalhos e refletidos na Tabela 5.1, que revelam diversas classes de vulnerabilidades reais, tanto na implementação quanto na utilização de Web Services e WS-Security.

Os ataques foram classificados de acordo com as propriedades de segurança violadas. Os detalhes e a descrição dos ataques usados nos experimentos estão descritos nas seções a seguir através da **árvore de ataques**. Dessa forma, foi executada a etapa 1 da metodologia de geração de ataques descrita no Capítulo 5.

## 6.3 Capacidade do Atacante

O objetivo da etapa 2 (Seção 5.2) é descrever a **capacidade do atacante**, a qual é implementada pelo injetor de falhas (IF) WSInject. Esta ferramenta, que foi utilizada em diverso trabalhos [48, 74], realiza testes de robustez para injeção de falhas em Web Services. O objetivo da ferramenta é injetar falhas em tempo de execução e no nível de mensagem SOAP, aplicável tanto a Web Services simples quanto compostos. Seu funcionamento é baseado na interceptação de mensagens trocadas entre cliente e serviço. No caso de serviços compostos, ela também é capaz de interceptar mensagens trocadas entre o serviço composto e seus serviços parceiros. O IF WSInject pode ser executado tanto em linha de comando quanto em modo de interface gráfica. A Figura 6.3 ilustra seu funcionamento no modo de interface gráfica. O procedimento básico para o funcionamento do IF WSInject é:

1. O usuário configura o Proxy (menu Proxy), selecionando a porta de entrada de dados para interceptar as mensagens SOA;
2. O usuário observará as mensagens interceptadas na Interfase Gráfica, composta pela requisição (*Request*) e a resposta (*Response*);
3. Para injetar falhas, o usuário deve ir ao menu File e carregar um arquivo em formato de texto com o ataque, prévia configuração do Proxy (passo 1).

Neste momento, o injetor de falhas começará a interceptar as mensagens SOAP. Quando a especificação cumpra com as condições desejadas, injetará a falhas dentro da mensagem SOAP.

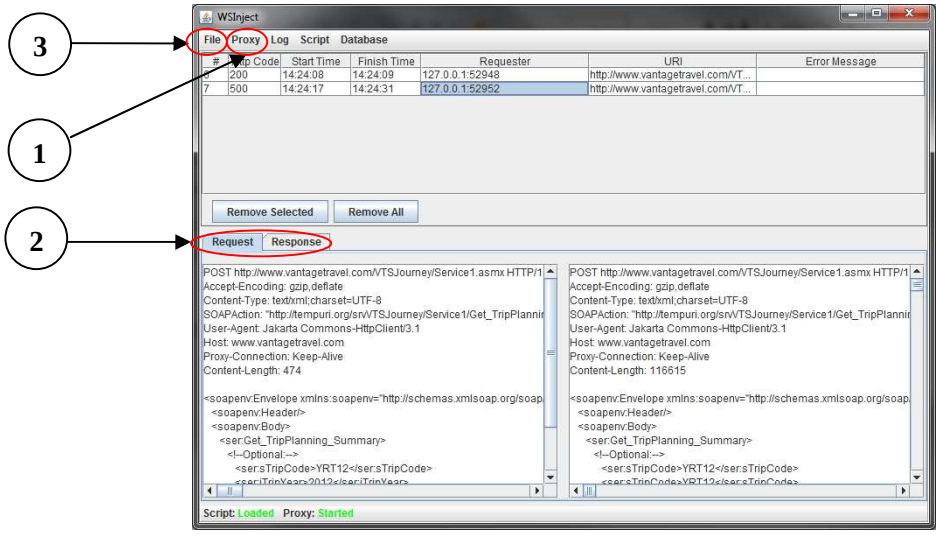



Figura 6.3 WSInject em funcionamento.

O WSInject utiliza *scripts* para descrição das falhas a serem injetadas, que são arquivos de texto contendo um ou mais *FaultInjectionStatements* (comandos de injeção de falhas). Cada *FaultInjectionStatement* é composto de um *ConditionSet* (conjunto de condições) e uma *FaultList* (lista de falhas), descritas na Tabela 5.2. Os *FaultInjectionStatements* funcionam como comandos do tipo **condição-ação**. Ao interceptar uma mensagem e satisfazer um conjunto de condições, a lista de falhas é injetada. A Figura 6.4 mostra um exemplo de *script* executável [7]. Em negrito temos as **palavras-chave** que especificam condições e ações. A primeira linha contém uma condição e duas ações; nesse caso, cada vez que a URI de uma chamada ao serviço ou sua resposta contiver a cadeia “Hotel”, todas as ocorrências de “Name” são substituídas por “Age” e uma duplicação do conteúdo é gerada. Na segunda linha, toda vez que uma mensagem contiver a cadeia "caught exception" e for uma resposta, o conteúdo da mensagem é esvaziado.

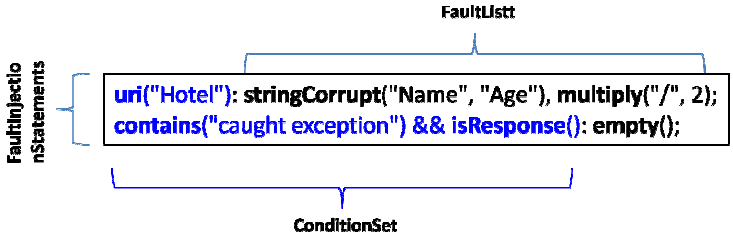



Figura 6.4 Exemplo de script para o WSInject.

## 6.4 Modelagem dos ataques

Para modelar os ataques utilizamos a ferramenta SecureITree V3.4 [37]. Esta ferramenta, utilizada em diversas pesquisas [5, 61, 62], nos ajudou a construir a **árvore de ataques** para Web Services, já que demonstrou sua eficiência e funcionalidade para essa tarefa.

A **árvore de ataques** foi construída e estruturada de acordo com os passos propostos na etapa 3 (Seção 5.3), usando os ataques da Seção 6.2 (etapa 1 da metodologia). A Figura 6.5 mostra a representação textual da **árvore de ataques** dos Web Services. Podemos observar que várias classes de ataques são mostradas na **árvore de ataques**, as quais violam diferentes tipos de propriedades de segurança, i.e. ataque de Oversize Payload violam *disponibilidade*, Fuzzing Scan viola a propriedade de *integridade*. As folhas da árvore de ataques possuem três atributos associados, que são valores lógicos que representam atributos de um ataque, e estão compostos por: <**capacidade do atacante, arquitetura de teste, dispor de um Web Services**>. Estes valores lógicos podem ser <**P**ossível, **I**mpossível>.

**OR 1 – Objetivo: Atacar Web Services**

- **OR 1.1** Violar ***Confidencialidade***
  - **OR 1.1.1** WSDL Scanning **<P, I, P>***
  - **1.1.2** Insufficient Transport Layer Protection **<P, P, P>**
- **OR 1.2** Violar **Integridade**
  - **OR 1.2.1** Metadata Spoofing **<P, I, P>****
  - **1.2.2** XML Injection **<P, P, P>**
  - **1.2.3** SQL Injection **<I, P, P,>**
  - **1.2.4** XPath Injection **<P, P, P >**
  - **1.2.5** Cross-site Scripting (XSS) **<P, P, P>**
  - **1.2.6** Fuzzing Scan **<P, P, P>**
  - **1.2.7** Invalid Types **<P, P, P>**
  - **1.2.8** Parameter Tampering **<P, I, P>**
  - **1.2.9** Malformed XML **<P, P, P>**
  - **1.2.10** Frankenstein Message: Modify Timestamp **<P, P, P>**
- **OR 1.3** Violar **Disponibilidade**
  - **OR 1.3.1** Replay Attack **<P, P, P>**
  - **1.3.2** Oversize Payload **<P, P, P>**
  - **1.3.3** Coercive Parsing **<P, P, P>**
  - **1.3.4** Oversize Cryptography **<P, P, P>**
  - **1.3.5** Attack Obfuscation **<P, P, P>**
  - **1.3.6** XML Bomb **<P, P, P>**
  - **1.3.7** Unvalidated Redirects and Forwards **<P, I, P>**
  - **1.3.8** Attacks through SOAP Attachment **<P, I, P>**
  - **1.3.9** Schema Poisoning **<I, I, I>**
  - **1.3.10** Insecure Cryptographic Storage **<I, I, I>**
  - **1.3.11** WS-addressing Spoofing **<I, I, I>**
  - **1.3.12** Middleware Hijacking **<I, I, I>**
  - **1.3.13** Maliciuos Content **<I, I, I>**
  - **1.3.14** Instantiation Flooding **<I, P, P>***
  - **1.3.15** Indirect Flooding **<I, P, P>***
  - **1.3.16** BPEL State Deviation **<I, P, P>**
- **OR 1.4** Violar **Controle de Acesso (Autenticação e Autorização)**
  - **OR 1.4.1** Broken Authentication and Session Management **<P, I, P>**
  - **1.4.2** SOAPAction **<P, I, P>**
  - **1.4.3** Security Misconfiguration **<P, I, P>**
  - **1.4.4** Unauthorized Access **<P, P, P>**
  - **1.4.5** Routing Detours **<P, P, P>**
  - **1.4.6** Attack on WS-Trust, WS-SecureConversation **<P, I, I>**
  - **1.4.7** Cross-site Request Forgery **<P, I, P>**
  - **1.4.8** Attack on WS-Security **<P, I, I>**

* **Acesso de Controle fica comprometido pelo ataque.**

** **Todas as propriedades de segurança ficam comprometidas pelo ataque.**

Figura 6.5 Árvore de ataques textual para Web Services.

A **capacidade do atacante** é definida na etapa 2 (Seção 5.2), a **arquitetura de teste**, embora alguns ataques estejam dentro do escopo da capacidade do atacante proposta, eles não podem ser injetados devido às limitações técnicas e funcionais do injetor (WSInject – escolhido na Seção 6.2), esta informação é detalhada na Seção 6.3 e **dispor de um o Web Services**, para aplicar um determinado ataque, precisamos que o Web Services utilize certos padrões, i.e. no ataque de *Oversize Cryptography* é preciso que o Web Services use o padrão WS-Security para injetar a falha.

O uso dos três atributos são úteis no sentido em que separamos as limitações da metodologia proposta das limitações da arquitetura de teste usada. Por exemplo, os valores <**P**ossível, **I**mpossível> para os atributos que estão colocados do lado direito da folha <1.1.1> WSDL Scanning <**P**, **I**, **P**> da Árvore de Ataques da Figura 6.5 indicam que: **i)** o atacante tem a capacidade para executar a ação; **ii)** o injetor de falhas não é capaz de emular o ataque; **iii)** temos disponível, pelo menos, um Web Services com as propriedades necessárias para reproduzir o ataque. Em resumo, o ataque não pode ser executado usando a arquitetura de testes atual. Nesse caso, o testador tem a opção de escolher outro injetor de falhas apropriados para sua execução ou não selecionar esse tipo de ataque ao momento de realizar os testes.

## 6.5 Geração e Concretização dos Cenários de Ataques

Os Cenários de Ataques foram selecionadas de acordo com os valores dos atributos das folhas. Para produzir os Cenários de Ataque, o ***critério*** usado foi "cobrir todos os ataques que satisfazem os valores **<P, P, P>** para os três atributos das folhas", pois indicam que estão dentro da capacidade do atacante e podem ser emulados pelo injetor WSInject (descrito na Seção 6.4), que cumpre com as propriedades dos Web Services a testar. A ferramenta *SecureITree* [37] foi usada para fazer a busca e selecionar os **cenários de ataques** de uma maneira automatizada. Os cenários que satisfazem ao ***critério*** foram: <1.1.2>, <1.2.2>, <1.2.4>, <1.2.5>, <1.2.6>, <1.2.7>, <1.2.9>, <1.2.10>, <1.3.1>, <1.3.2>, <1.3.3>, <1.3.4>, <1.3.5>, <1.3.6>, <1.4.4>, <1.4.5>. Estes são os **cenários de ataques** selecionados de acordo com a etapa 4 (Seção 5.4).

Um novo atributo foi introduzido chamado **"WS-Security protege ao Web Service"** com os seguintes resultados: Possível (Protege) e Impossível **(**Não Protege), descrito textualmente na Figura 6.6 da **árvore de ataque textual** para Web Services com WS-Security.

Este novo atributo permite separar os ataques que são rejeitados com WS-Security dos que não, e esta informação está detalhada na Tabela 5.1 de propriedades de segurança para Web Services.

**OR 1 – Objetivo: Atacar Web Services**
- **OR 1.1** Violar ***Confidencialidade***
  - **OR 1.1.2** Insufficient Transport Layer Protection **<P, P, P, P>**
- **OR 1.2** Violar **Integridade**
  - **OR 1.2.2** XML Injection **<P, P, P, P>**
  - **1.2.4** XPath Injection **<P, P, P , P>**
  - **1.2.5** Cross-site Scripting (XSS) **<P, P, P, P>**
  - **1.2.6** Fuzzing Scan **<P, P, P, P>**
  - **1.2.7** Invalid Types **<P, P, P, P>**
  - **1.2.9** Malformed XML **<P, P, P, P>**
  - **1.2.10** Frankenstein Message: Modify Timestamp **<P, P, P, P>**
- **OR 1.3** Violar **Disponibilidade**
  - **OR 1.3.1** Replay Attack **<P, P, P, I>**
  - **1.3.2** Oversize Payload **<P, P, P, I>**
  - **1.3.3** Coercive Parsing **<P, P, P, I>**
  - **1.3.4** Oversize Cryptography **<P, P, P, I>**
  - **1.3.5** Attack Obfuscation **<P, P, P, I>**
  - **1.3.6** XML Bomb **<P, P, P, I>**
- **OR 1.4** Violar **Controle de Acesso (Autenticação e Autorização)**
  - **OR 1.4.4** Unauthorized Access **<P, P, P, P>**
  - **1.4.5** Routing Detours **<P, P, P, P>**

Figura 6.6 Árvore de Ataque Textual para Web Services.

Anexando este novo atributo da Figura 6.6, podemos dividir em dois **cenários de ataque**, cenário A de ataques que são **rejeitados por WS-Security** e cenário B de ataques que **não são rejeitados por WS-Security**, a seguir:

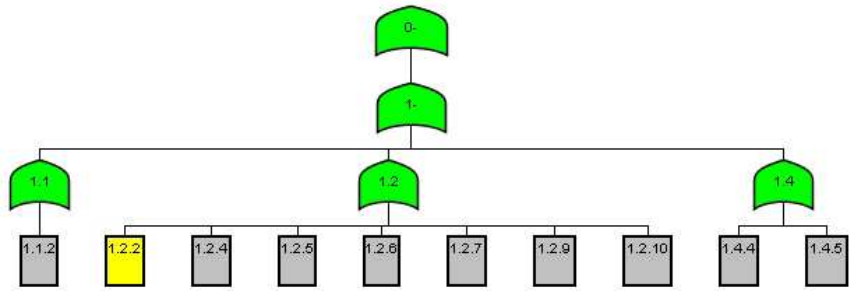



Figura 6.7 Sub-árvore gráfica de Ataques que são rejeitados por WS-Security.

***Cenário A) Ataques que são rejeitados por WS-Security:*** A sub-árvore classifica os ataques que WS-Security rejeita, lembremos na Seção 6.1 que o padrão WS-Security protege as propriedades de confidencialidade, integridade, autorização e autenticação nas mensagens SOAP, isto nos permite determinar se o ataque é efetivo em função da propriedade que é afetada

pelo Ataque. A lista de ataques se resume em: <1.1.2>, <1.2.2>, <1.2.4>, <1.2.5>, <1.2.6>, <1.2.7>, <1.2.9>, <1.2.10>, <1.4.4>, <1.4.5> e é descrita na Figura 6.7.

Dos dez ataques obtidos, selecionamos cinco ataques por seu impacto considerado crítico (*High*), segundo a OWASP Top 10 [19] e obtido da Tabela 5.1. Os ataque selecionados são: XML Injection, XPath Injection, Cross-site Scripting (XSS), Fuzzing Scan, Malformed XML.

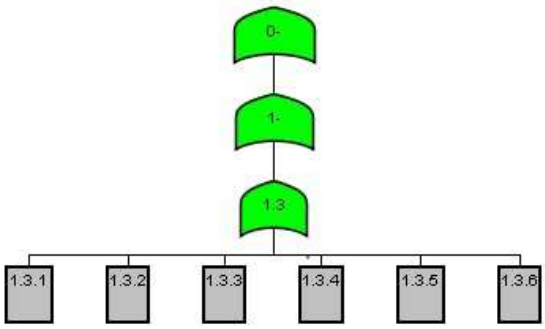



Figura 6.8 Sub-árvore de Ataques gráfica que Não são rejeitados por WS-Security.

***Cenário B) Ataques que não são rejeitados por WS-Security:*** A sub-árvore classifica os ataques que WS-Security não consegue rejeitar; geralmente os ataques estão associados à propriedade de disponibilidade que o padrão WS-Security não protege. A lista de ataques se resume em: <1.3.1>, <1.3.2>, <1.3.3>, <1.3.4>, <1.3.5>, <1.3.6> descrita na Figura 6.8.

Dos seis ataques obtidos, selecionamos cinco ataques onde seu impacto é considerado perigoso, segundo a OWASP Top 10 [19] e obtido da Tabela 5.1. Estes ataques são: Oversize Payload, Coercive Parsing, Oversize Cryptography, Attack Obfuscation, XML Bomb.

Para ilustrar os passos de refinamento da etapa 5 (Seção 5.5) e os métodos propostos para obter o **SAE** de forma automatizada, continuaremos usando o Cenário de Ataque <1.2.2 - Ataque de *XML Injection*> que contém a "descrição do ataque" e a "propriedade de segurança violada pelo atacante". O cenário de ataques é baseado em *XML Injection* para executar operações restringidas, descrito na Seção 3.2. Nesta versão, o atacante intercepto a mensagem SOAP de uma requisição e procura uma cadeia <String> na mensagem SOAP, modifica os parâmetros de operação por um parâmetro não padrão fazendo uso da informação fornecida pelo WSDL e enviando a requisição da mensagem SOAP alterada ao Web Service.

A Figura 6.9 mostra o primeiro passo da concretização do cenário <1.2.2> para o contexto do Web Services. Nesse passo foi utilizado o padrão de ataque, detalhado na Seção 5.4 e citado na etapa 5 (Seção 5.5.1), onde o **cenário de ataque** – descrição do ataque – é detalhado

usando os seguintes elementos: **i)** o "*objetivo do ataque*"; **ii)** a "lista de precondições"; **iii)** os "*passos para executar o ataque*".

**Objetivo**: Ataque de *XML Injection* para executar uma operação restrita.

**Precondições**: O cliente faz uma requisição ao Web Service.

O atacante consegue ao menos uma operação usando o WSDL ou a requisição feita pelo cliente.

O cliente não usa um esquema seguro de comunicação como HTTPS ou WS-Security.

**Ataque:** AND 1.2.2.1. Caso seja uma requisição.

1.2.2.2. Caso contenha a cadeia <Operação1>.

1.2.2.3. Modificar os parâmetros da Operação1 por uma mensagem não padrão.

1.2.2.4. Gerar uma nova mensagem SOAP.

Figura 6.9 Padrão de ataque para o Cenário de Ataque <1.2.2>.

O padrão de ataque da Figura 6.9 foi baseado no cenário <*Ataque de XML Injection para modificar a estrutura XML*> da Seção 5.5.1, mostrado na Figura 5.2. Foram feitas as seguintes alterações nos elementos desse padrão de ataque: **i)** o **"Objetivo"** e **"Precondição"** foram atualizadas; **ii)** os elementos ("*passo para executar o ataque*") 1.2.2.2 e 1.2.2.3 procuram a transação da Operação1 para modificar-se por uma cadeia não padrão; **iii)** os elemento ("*passo para executar o ataque*") 1.2.2.4 foi atualizado, detalhando a importância de gerar a nova mensagem depois de modificar a mesma; o elemento 1.2.2.1 não foi modifica.

Em seguida o padrão de ataque da Figura 6.9 é convertido para o **script de ataque executável** (**SAE**) para o WSInject, onde o elemento **"**Precondição**"** é mapeado para a atividade *evento*, os elementos 1.2.2.3 e 1.2.2.4 (*passo para executar o ataque*) são mapeados para a parte de *ação*, pois "**stringCorrupt(String fromString, String toString)**" – corrução_cadeia(Cadeia_receptor, Cadeia_corrupta) – é uma palavra-chave estabelecida e GenerateNewMessage(message) é outra palavra-chave estabelecida, e os elementos, 1.2.2.1 e 1.2.2.2 verificam consecutivamente se a mensagem é uma requisição **isRequest()** e contem uma cadeia para ser substituída **stringCorrupt(String fromString, String toString)**, estas palavras-chaves foram estabelecidos na Tabela 5.2, e são mapeados para a atividade *condição*. O **SAE** criado para esse ataque é mostrada na Figura 6.10.

**Condição:** isRequest():

**Do:** stringCorrupt("</ser:Username>","</ser:Username><ser:Username>hacker</ser:Username>");

Figura 6.10 SAE para o ataque XML Injection <1.2.2> com o IF WSInject.

Agora, explicaremos as operações realizadas na Figura 6.16 com as palavras-chave, de acordo com a Tabela 5.2.

- Na ativação **ON** do **SAE** está relacionada a ativação do Proxy do injetor de falhas WSInject, usado para interceptar as mensagens SOAP pelo WSInject.
- Na atividade **IF** (*condição*), a palavra-chave **isRequest()** da primeira condição é usada para a operação **isRequest**: Boolean, pois ela seleciona somente as mensagens SOAP de tipo requisições, enviadas do cliente para o servidor.
- Na atividade **DO** (*ação*) a palavra-chave **stringCorrupt(String, String_Corrupt)** é usada pela operação **stringCorrupt**: Action, que é utilizada para substituir as ocorrências de *fromString* por *toString*, a fim de substituir os caracteres XML pelo ataque de *XML Injection*. Consecutivamente, é gerado a nova mensagem SOAP, para ser enviada ao Web Services. Para a aplicação do ataque <1.2.2> *XML Injection*, substituímos as ocorrências da operação </ser:Username> por um script de ataque de *XML Injection*, e.g. <ser:Username> por <ser:Username>hacker</ser:Username>, com o objetivo de modificar a estrutura XML de uma mensagem SOAP, através da inserção de etiquetas (*tags*), provocando efeitos indesejáveis no Web Services.

A saída dessa etapa 5 poderia ser reusada para outros ataques similares, necessitando de pequenas alterações. Por exemplo, o *script* de ataque *XPath Injection* deveria ter a operação *XPathCorruption* em vez de *stringCorruption* para substituir todas as ocorrências de uma expressão *XPath* pelo valor especificado ou modificar os atributos e elementos.

A abordagem usada nessa seção favorece a reusabilidade do modelo de ataque, tanto através dos **cenários de ataque** obtidos da **árvore de ataques** quanto do **script de ataques executáveis** para o WSInject, obtido a partir da concretização do **cenário de ataque**, ao contrário da maioria das abordagens discutidas na Seção 4.5.

O **script de ataque executável** (**SAE**) será usado no Capítulo 8 na geração dos scripts, para a emulação de ataques contra Web Services e WS-Security.

# Capítulo 7. Experimentos de Pré-Análise

Este capítulo descreve a fase de pré-análise, que nos ajudou a identificar o comportamento de um Web Services em presença de falhas. Os serviços são testados com o *vulnerability scanner* (VS) soapUI, uma ferramenta que injeta vulnerabilidades aos Web Services através do add-on Security Testing[24]. Na Seção 7.1 detalhamos a arquitetura de testes, que serviu para executar os experimentos e reproduzir ataques reais contra os Web Services.

Na Seção 7.2 mostra a obtenção uma amostra de 69 Web Services de 22.272, selecionados aleatoriamente da UBR (*Universal Business Registry*) Seekda. A seguir, configuramos o add-on Security Testing para realizar testes de penetração automatizados sobre os serviços selecionados. Na Seção 7.3 injetamos um conjunto de ataques (Cross-site Scripting, Fuzzing Scan, Invalid Types, Malformed XML, SQL Injection, XML Bomb e XPath Injection) contra a segurança dos Web Services utilizando o add-on Security Testing. O resultado é outro conjunto de dados classificados em ataques bem sucedidos e ataques detectados, que serão analisados em próximas seções.

Dado que a abordagem de teste é caixa preta e o acesso aos serviços são feitos remotamente, na Seção 7.4 desenvolvemos um conjunto de regras para identificar quando encontramos uma vulnerabilidade e assim podermos classificar as respostas obtidas. Para finalizar, na Seção 7.5 aplicamos o conjunto de regras geradas na Seção 7.4 para analisar o conjunto de dados da Seção 7.3 e apresentamos os comentários finais do capítulo.

## 7.1 Arquitetura de Testes

Nossa abordagem inicial faz uso da técnica de testes de penetração. Sua arquitetura utiliza o Vulnerability Scanner (VS) soapUI, uma ferramenta premiada pela ATI[25]. Juntamente com o add-on Security Testing, o VS injeta diversos tipos de ataques aos Web Services. A arquitetura é descrita na Figura 7.1.

A partir da versão 4.0.1, o VS soapUI adiciona o *add-on Security Testing*. Este add-on envia um conjunto de requisições maliciosas aos Web Services, com o objetivo de: **i)** reproduzir

[24] http://soapui.com/Security/getting-started.html
[25] http://www.automatedtestinginstitute.com

diferentes tipos de ataque; **ii)** tentar provocar um comportamento não robusto; **iii)** identificar possíveis vulnerabilidades de segurança; **iv)** tratar as possíveis vulnerabilidades presentes nos Web Services [6]. Instalamos o VS soapUI com Security Testing versão 4.5 em um laptop com sistema operacional (SO) Windows 7, CPU Intel Core2 Duo 2GHz e memória RAM 3GB.

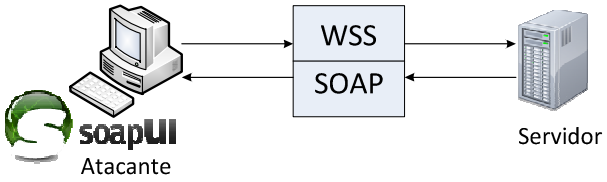



Figura 7.1 Arquitetura de Teste.

## 7.2. Configuração de Ataques

A presente seção descreve o procedimento utilizado para selecionar uma amostra representativa de Web Services e a configuração do VS soapUI com o add-on Security Testing para a fase de pré-analise.

### 7.2.1 Seleção da Amostra de Web Services

Segundo Marconi e Lakatos [75], a amostra é uma parcela convenientemente selecionada do universo (população); que serve para recolher informação generalizada e possivelmente representativa do universo. Para nosso experimento, no universo é um conjunto de 22.272 Web Services armazenados no Banco de Dados de "Seekda[26]", uma UBR (*Universal Business Registry*) que armazena a descrição de 22.272 Web Services, fornecendo acesso a uma plataforma de busca de serviços públicos e privados. Como primeiro passo, calculamos o tamanho da amostra adequada para nossa pesquisa utilizando a Fórmula 7.1:

$$n = \frac{Z^2 * p * q * N}{N * e^2 + Z^2 * p * q} \quad (7.1)$$

Onde: **n** é o tamanho da amostra procurada.

**N** são 22.272 Web Services (Universo).

**e** (Erro de estimação) é 10% ou 0.1.

**Z** (Nível de confiança) é 1.65 (da tabela de distribuição normal) para o 90% de confiabilidade e 10% de erro.

[26] http://webservices.seekda.com

**p** é 0.5 ou 50% para a probabilidade de encontrar vulnerabilidades em Web Services.

**q** é 0.5 ou 50% para a probabilidade de não encontrar vulnerabilidades em Web Services. O complemento de p (q = p - 1).

Ao substituir todos os dados, temos:

$$n = \frac{1.96^2 * 0.5 * 0.5 * 22.272}{22.272 * 0.1^2 + 1.96^2 * 0.5 * 0.5}$$

$$n = 68{,}743125$$

$$n \cong 69 \text{ Web Services}$$

A amostra representativa contém 69 Web Services, com 90% de precisão e um erro de $\pm$10%; o que permite comprovar quantos Web Services têm vulnerabilidades e quais são os principais ataques contra Web Services. Existe a possibilidade de cometer um erro, devido ao fato de obter os resultados a partir da análise de uma amostra não significativa; este erro é chamado **erro de amostragem**. Para evitá-lo, fazemos uso de números pseudo-aleatórios para selecionar os 69 Web Services.

Por questões de segurança, não descrevemos o WSDL de cada Web Services, no entanto, detalhamos algumas das características dos serviços.

Classificamos os Web Services em quatro classes, segundo o uso fornecido: **i)** 32 Web Services (46,38%) de uso comercial proporcionam serviços empresarias aos clientes e empresas; **ii)** governamentais (1,45%) servem para interagir entre o governo e os cidadãos; **iii)** 26 Web Services (37,68%) servem para fazer consultas e validação da informação sobre diversos áreas como música, meteorologia, entre outros; e **iv)** acadêmicos (14,49%) apóiam as atividades acadêmicas e de pesquisas. Dos Servidores de Web Services testados 86% usa SO Microsoft Windows Server 2003/2008 com IIS (*Internet Information Server*) 6.0 ou superior, 10% usa SO Linux com o Servidor Apache, 3% também usa SO Linux com o Servidor Jetty V6.1.17 e 1% usa Windows 7 Professional com Glassfish Server V2.1.

## 7.2.2 Configuração do Vulnerability Scanner soapUI

O seguinte passo é a configuração do VS soapUI com o add-on Security Testing para executar os testes de segurança nos Web Services, selecionadas na Seção 7.2.1. Com o objetivo de obter uma maior cobertura da injeção de falhas, selecionamos um conjunto de ataques (SQL

Injection, XPath Injection, Cross-site Scripting (XSS), Fuzzing Scan, Invalid Types, Malformed XML, XML Bomb). Estes ataques são classificados na Figura 7.2 de acordo com a propriedade de segurança violada. Na Figura 7.3 descrevemos a arquitetura do atacante, usando o VS soapUI com o add-on Security Testing para injetar os ataques mencionados.

**OR 1 – Objetivo: Atacar Web Service**
- **OR 1.2** Violar **Integridade**
  - **OR 1.2.3** SQL Injection
  - **1.2.4** XPath Injection
  - **1.2.5** Cross-site Scripting (XSS)
  - **1.2.6** Fuzzing Scan
  - **1.2.7** Invalid Types
  - **1.2.9** Malformed XML
- **OR 1.3** Violar **Disponibilidade**
  - **OR 1.3.6** XML Bomb

Figura 7.2 Árvore de ataque textual para o VS soapUI.

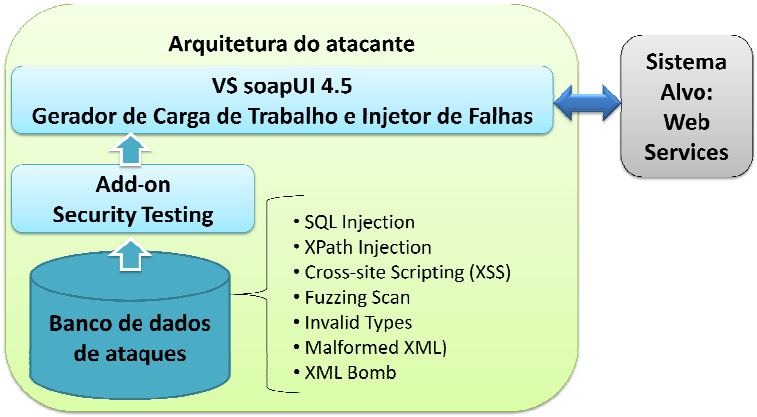



Figura 7.3 Arquitetura do VS soapUI com add-on Security Testing.

Os ataques de Malicious Attachment e Boundary Scan foram descartados da seleção de ataques injetados, já que o Vulnerability Scan (VS) SoapUI não conseguiu gerar ataques bem sucedidos nos serviços. A maioria dos Web Services não anexam arquivos dentro da mensagem SOAP, sendo um requisito de Malicious Attachment.

Nesta etapa enviamos um conjunto de requisições aos Web Services. Sua saída é um log descrito na Figura 7.4, que classifica as resposta em dois tipos: **i)** *Alerts*, que detalha a quantidade de alertas ou possíveis vulnerabilidades achadas no Web Service; e **ii)** *No Alerts*, para o caso em que o VS soapUI não conseguiu achar nenhuma vulnerabilidade no Web Service [68].

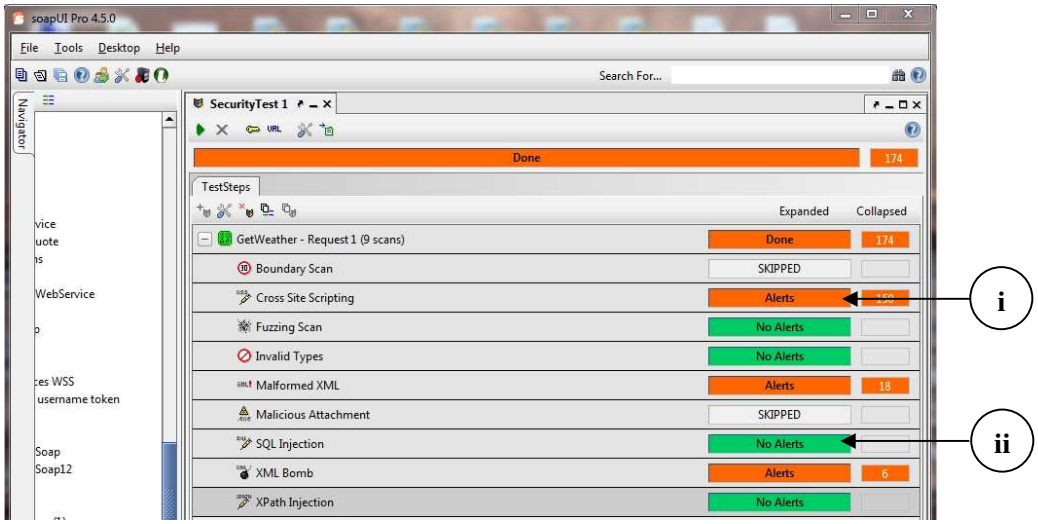



Figura 7.4 Execução do Security Testing no soapUI em um Web Service.

## 7.3. Execução dos Ataques pelo VS soapUI

Nesta seção, injetamos uma diversidade de ataques contra os 69 Web Services selecionados na Seção 7.1. Utilizamos o add-on Security Testing para injetar os ataques descritos na Seção 7.2. A seguir detalhamos os resultados da injeção dos ataques.

No total foram registrados 35.087 requisições pela injeção de 7 ataques contra os 69 Web Services. O VS classificou como “alertas” (possíveis vulnerabilidades encontradas) 54,57% das respostas, e 45,43% foram classificados como “sem alertas” (vulnerabilidades não encontradas).

Os ataques que superam os 50% de respostas classificadas como alertas são: **1º** XML Bomb (DoS) com 66,88%; **2º** Cross-site Scripting com 65,93%; e **3º** Malformed XML com 65,51%. O restante dos ataques são descritos na Tabela 7.1.

Tabela 7.1 Porcentagem de ataques por ataques.

| Ataque | Total | Alertas | % | Sem Alertas | % |
|---|---|---|---|---|---|
| **Cross-site Scripting (XSS)** | 19.108 | 12.598 | 65,93% | 6.510 | 34,07% |
| **Fuzzing Scan** | 5.900 | 2.818 | 47,76% | 3.082 | 52,24% |
| **Invalid Types** | 3.415 | 1.363 | 39,91% | 2.052 | 60,09% |
| **Malformed XML** | 1.670 | 1.094 | 65,51% | 576 | 34,49% |
| **SQL Injection** | 2.517 | 1.158 | 46,01% | 1.359 | 53,99% |
| **XML Bomb (DoS)** | 637 | 426 | 66,88% | 211 | 33,12% |
| **XPath Injection** | 1.840 | 920 | 50,00% | 920 | 50,00% |
| **Total** | **35.087** | **20.377** | **54,57%** | **14.710** | **45,43%** |

Na Figura 7.5, observamos a porcentagem de ataques classificados como alertas e sem alertas para os 7 ataques injetados. XPath injection conseguiu ter a mesma quantidade de respostas classificas tanto como alertas (50%) e não alertas (50%). Isto não representa que os 69 Web Services apresentam o mesmo comportamento. Porem analisamos a presença de vulnerabilidades nos Web Services na Tabela 7.2.

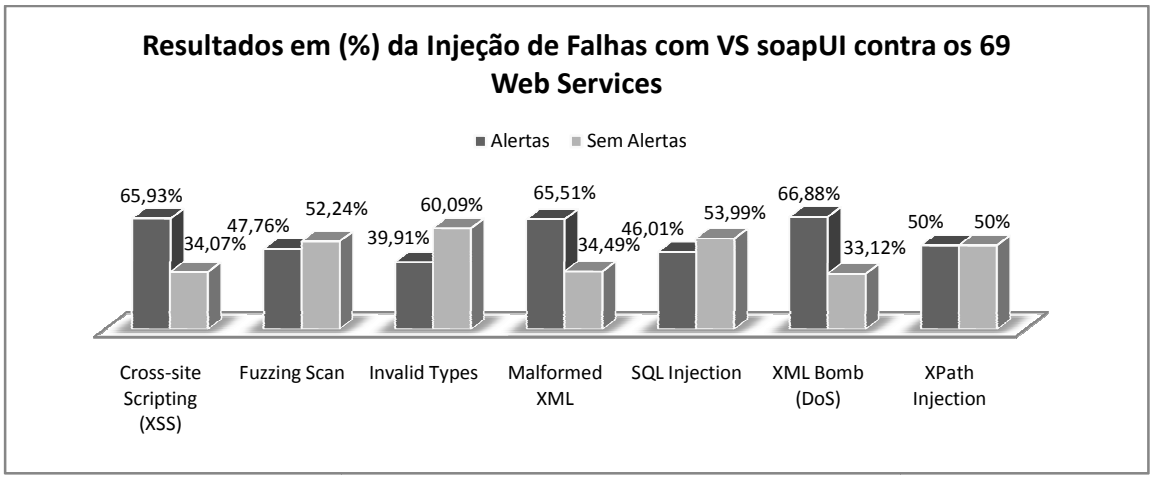



Figura 7.5 Ataques contra os 69 Web Services.

A Tabela 7.2 mostra quantos Web Services são susceptíveis a um tipo especifico de ataque, o qual é injetado pelo VS soapUI. Podemos destacar três tipos de ataques: **1º** com 64/69 Web Services vulneráveis a Cross-site Scripting; **2º** com 57/69 Web Services vulneráveis a Malformed XML; e **3º** com 48/69 Web Services vulneráveis a XML Bomb (DoS). O restante dos ataques está descrito na Tabela 7.2.

Tabela 7.2 Porcentagem de alertas nos 69 Web Services

| Ataque | Alerts #/69 Web Services | % |
|---|---|---|
| **Cross-site Scripting (XSS)** | 64 | 92,75% |
| **Malformed XML** | 57 | 82,61% |
| **XML Bomb (DoS)** | 48 | 69,57% |
| **Invalid Types** | 31 | 44,93% |
| **XPath Injection** | 30 | 43,48% |
| **Fuzzing Scan** | 29 | 42,03% |
| **SQL Injection** | 28 | 40,58% |

É importante analisar esta informação para compreender quais são as vulnerabilidades mais frequentes segundo o VS soapUI. Por exemplo, apenas 5 Web Services não apresentam vulnerabilidades a ataques de Cross-site Scripting, segundo o VS soapUI.

## 7.4. Geração das Regras

O objetivo desta seção é identificar quando um ataque não encontrou uma vulnerabilidade (vulnerabilidade não encontrada ou **VNE**) e quando um ataque provavelmente ativou uma vulnerabilidade no Web Service (vulnerabilidade encontrada **VE)**, reduzindo potenciais falsos positivos (**FP**) e falsos negativos (**FN**) para ataques de injeção e negação de serviços.

Utilizamos a abordagem caixa preta para injetar falhas em tempo de execução e em nível de mensagem SOAP, o VS soapUI envia requisições incluindo ataques, descritos na Seção 7.2.2.

Para determinar se um Web Services apresenta vulnerabilidades aos diversos ataques selecionados na Seção 7.2.2, inicialmente analisamos as assertivas do add-on Security Testing do VS soapUI, descrita na Tabela 7.3 [6]. O add-on classifica as respostas como válidas (`VALID – OK`) quando Web Services respondeu com uma mensagem robusta e mal sucedido (`FAILED`) quando um ataque encontrou uma vulnerabilidade.

Tabela 7.3 Assertivas oferecidas pelo soapUI.

| Assertiva | Descrição |
|---|---|
| **Invalid HTTP Status Codes** | Verifica a existência de códigos de status HTTP inválidos nas respostas |
| **Schema Compliance** | Valida as respostas das requisições contra o XML-schema definido no WSDL |
| **Simple Contains** | Verifica a existência de uma string nas respostas das requisições |
| **Sensitive Information Exposure** | Verifica se a última mensagem recebida não contém informação sensível do sistema operacional |
| **SOAP Fault** | Verifica se a resposta da requisição não contém um soap-fault |
| **XPath Match** | Compara o conteúdo das respostas das requisições contra uma expressão XPATH |

O add-on Security Testing analisa o conteúdo das respostas, tentando achar informações sensíveis que forneçam indícios de vulnerabilidades encontradas. Por exemplo, a assertiva de “Sensitive Information Exposure” verifica se a resposta contém informação sensível do sistema operacional tal como uma rota do diretório. As restantes assertivas são descritas na Tabela 7.4. Dado que o soapUI também realiza testes de stress, são oferecidas regras para determinar se um

serviço atende aos requisitos de desempenho; elas são conhecidas como (*Web Service-Level Agreements*) ou Acordo de Nível de Web Services [6].

Também utilizamos a ferramenta *Message Viewer* do VS soapUI para analisar o comportamento de um Web Services frente a diversos ataques. Esta ferramenta nos permite ver a **requisição** e **resposta** em tempo de execução. Por exemplo, na Figura 7.6 observamos de lado esquerdo, a **requisição** com ataque de Cross-site Scripting (**em negrito**); de lado direito a **resposta** com o código de status HTTP 500 *Internal Server Error*, descrevendo falha(s) no servidor, seguido de um conjunto de etiquetas que descrevem a falha (**em negrito**).

| Requisição | Resposta |
|---|---|
| 1: POST ... HTTP/1.1 | 1: **HTTP/1.1 500 Internal Server Error** |
| 2: Content-Type: text/xml;charset=UTF-8 | 2: <?xml version="1.0" encoding="utf- |
| 3: SOAPAction: "..." | 3: 8"?><soap:Envelope xmlns:soap="..."> |
| 4: <soapenv:Envelope xmlns: "... "> | 4: <soap:Body> |
| 5: <soapenv:Header/> | 5: **<soap:Fault>** |
| 6: <soapenv:Body> | 6: **<faultcode>soap:Client** |
| 7: <web:ConversionRate> | 7: **</faultcode>** |
| 8: **<web:FromCurrency><SCRIPTa=">'>"SRC=** | 8: **<faultstring>System.Web.Services.Protocols.SoapExc** |
| 9: **"http://soapui.org/xss.js"></SCRIPT>** | 9: **eption: Server was unable to read request.** |
| 10: **</web:FromCurrency>** | 10: **...** |
| 11: <web:ToCurrency>BOB</web:ToCurrency> | 11: **</faultstring>** |
| 12: </web:ConversionRate> | 12: **<detail/>** |
| 13: </soapenv:Body> | 13: **</soap:Fault>** |
| 14: </soapenv:Envelope> | 14: </soap:Body></soap:Envelope> |

Figura 7.6 Exemplo de *Message Viewer* gerado pela soapUI.

Outro aspecto importante é analisar os logs armazenados pelo VS soapUI. A linha 4 da Figura 7.7 descreve a existência de uma vulnerabilidade encontrada através da injeção do script (linha 4 e 5) de Cross-site Scripting. Nas linhas 5, 6 e 7 exibem uma mensagem alertando sobre a existência de informação do sistema na resposta. Além disso, o log descreve o resultado do ataque `FAILED` (linha 4), o tempo da resposta (linha 1) e uma descrição da vulnerabilidade encontrada (linha 5 à 12).

```
1:  SecurityTest started at 2012-04-17 11:04:59.503
2:  Step 1 [ConversionRate - Request 1]
3:          SecurityScan 2 [Cross Site Scripting]
4:          [Cross Site Scripting] Request 3 - FAILED - [FromCurrency=<SCRIPT a=">'>"
5:          SRC="http://soapui.org/xss.js"></SCRIPT>]: took 216 ms
6:                  -> [Stacktrace] Can give hackers information about which software or language
7:                  you are using - Token [(?s).*(s|S)tack ?(t|T)race.*] found [stack trace]
8:                   -> [Stacktrace] Can give hackers information about which software or
9:                  language you are using - Token [(?s).*at
10:                 [\w\$]+(\.[\w\$<>\[\],]+|\.\.ctor)+(\((([\w\$<>\`\[\]]+ [\w\$<>]+,
11:                 )*(([\w\$<>\`\[\]]+ [\w\$<>]+))\)|\(\)).*] found [at
12:                 System.Web.Services.Protocols.WebServiceHandler.CoreProcessRequest()
```

Figura 7.7 Log gerado pelo VS soapUI pela injeção de XSS contra um Web Services.

Os códigos de status HTTP proporcionam informação acerca da resposta do Web Service. Estes códigos nos permitem, facilmente, analisar se uma requisição foi executada ou existe problema(s) para sua execução [27].

Segundo Kohlert e Arun em [76]: **1)** as interações HTTP são de natureza requisição e resposta; **2)** a maioria dos Web Services usam HTTP como protocolo de transferência de mensagens SOAP, somente em uma direção (*one-way*). **3)** o requisitor tem que esperar pela resposta HTTP, mesmo que não haja nenhuma mensagem SOAP no corpo da entidade de resposta HTTP; **4)** temos em conta que a finalização de uma requisição com resposta HTTP 200 significa que a transmissão da mesma foi completada, não que a requisição foi aceita ou processada pelo Web Service.

Tabela 7.4 Lista de códigos de status HTTP.

| **Códigos HTTP** | **Descrição** |
|---|---|
| 200 – OK | Padrão de resposta para requisições HTTP bem sucedida [27]. Ajudará a identificar a existência de uma **possível vulnerabilidade encontrada** quando o sistema executou a requisição sem detectar o ataque e, **ataque detectado** quando o sistema responde com uma mensagem robusta, descrevendo a existência de erro na requisição. |
| 400 – Bad Request | A requisição não pode ser cumprida devido a sintaxe ruim [27]. Consideramos uma **resposta robusta**, já que o servidor detectou o ataque. |
| 500 – Internal Server Error | O servidor não cumpriu com uma requisição aparentemente válida ou encontrou uma condição inesperada, impedindo de executar a requisição com êxito [27]. Analisamos a resposta do servidor usando o *tag* <soap:Fault> dentro do corpo (*body*) da mensagem SOAP, que fornece os erros e a informação de status da mensagem SOAP, contendo os sub-elementos:<br>• <faultcode> Código de identificação da falha.<br>• <faultstring> Explicação descritiva da falha.<br>• <faultactor> Informação de que/quem fez acontecer a falha.<br>• <details> Informação que descreve o erro no servidor.<br>Além disso, os valores de *faultcode* podem ser classificados em 4 tipos:<br>• VersionMismatch: O servidor encontrou um *namespace* inválido no envelope da mensagem SOAP.<br>• MustUnderstand[27]: ausência de um elemento obrigatório no cabeçalho da mensagem SOAP.<br>• Client: A mensagem enviada foi estruturada de forma incorreta ou contém informações incorretas para sua autenticação.<br>• Server: Aconteceu um problema com o servidor para que a mensagem não possa ser processada. |

Já que a maioria dos Web Services usam o protocolo de comunicação HTTP, as respostas podem ser associadas a uma lista de códigos de status HTTP (ver Tabela 7.4), para decidir se

[27] O atributo MustUnderstand serve para indicar se uma entrada no cabeçalho é obrigatório ou opcional

nosso ataque conseguiu achar uma vulnerabilidade ou não [27]. Além disso, podemos classificar as respostas fornecidas pelos Web Services em três tipos: **1)** vulnerabilidades encontradas (**VE**) para respostas com HTTP 200/500 quando o servidor processou a mensagem SOAP e forneceu informações sensíveis, como rotas de diretórios do servidor, linguagem de programação, entre outros; **2)** vulnerabilidades não encontradas (**VNE**) para respostas com HTTP 400 quando o Web Service encontrou uma requisição com sintaxe ruim; e falha de software (**FS**) quando o erro não foi provocado pelo ataque, senão por um erro de software do servidor ou Web Service.

Tendo em conta as assertivas do add-on Security Testing, a análise do comportamento dos Web Services em presença de ataques através do *Message Viewer* e logs, os códigos de status HTTP e a pesquisa em [64]. Descrevemos a seguir na Tabela 7.5, as regras para classificar as respostas dos ataques contra Web Services.

Tabela 7.5 Regras de analise de vulnerabilidades nos Web Services.

| Regras | Descrição |
|---|---|
| **Regra 1** | **SE** a resposta contém mensagem com código "**HTTP 200 *OK***" |
| | E respondeu com uma mensagem SOAP descrevendo a existência de erro de sintaxe na requisição, ENTÃO existe uma **vulnerabilidade não encontrada (VNE)**.<br>CASO CONTRÁRIO o usuário tem acesso a páginas não autorizadas,<br>OU o usuário é redirecionado para outro Web Service,<br>OU o usuário executou trechos de código no servidor (e.g. Java script),<br>ENTÃO existe uma **vulnerabilidade encontrada (VE)**. |
| **Regra 2** | **SE** a resposta contém mensagem de erro do tipo "*bad request message*" ou o código "**HTTP 400**" (e.g. "*Request format is invalid: Missing required soap: Body element*") |
| | ENTÃO existe uma **vulnerabilidade não encontrada (VNE)**. |
| **Regra 3** | **SE** a resposta contém mensagem com código "**HTTP 500 *Internal Server Error***" |
| | E houve divulgação de informação (e.g. software, linguagem de programação, bibliotecas de funções, banco de dados, rotas de diretórios (*Path*) do servidor, conexão do usuário),<br>ENTÃO existe uma **vulnerabilidade encontrada (VE)**.<br>CASO CONTRÁRIO não houve divulgação de informação<br>ENTÃO existe uma **vulnerabilidade não encontrada (VNE)**. |
| **Regra 4** | **SE** na ausência de ataques, a resposta contém mensagem com código "**HTTP 500 *Internal Server Error***" |
| **Regra 4.1** | Considerando a ocorrência da Regra 4, se na presença de ataque teve como resposta o código "HTTP 200 OK",<br>ENTÃO existe uma **vulnerabilidade encontrada (VE)**. |
| **Regra 4.2** | Considerando a ocorrência da Regra 4, se na presença de ataque teve como resposta o código "HTTP 400",<br>ENTÃO existe uma **vulnerabilidade não encontrada (VNE)**. |
| **Regra 4.3** | Considerando a ocorrência da Regra 4, se na presença de ataque teve como resposta o código "HTTP 500 Internal Server Error",<br>ENTÃO existe **falha de software (FS)**, porque as respostas não foram provocadas pelo ataque, senão por um erro de software. |
| **Regra 5** | **SE** o servidor não responde, se considera como colapso (*crash*),<br>ENTÃO é **inconclusivo**. |
| **Regra 6** | **SE** nenhuma das regras acima pode ser aplicada,<br>ENTÃO o resultado é tido como **inconclusivo**, pois não há forma de confirmar se realmente existe uma vulnerabilidade. |

A regra 1, 2 e 3 utilizam os códigos de status 200, 400 e 500 respectivamente para classificar a resposta como vulnerabilidade encontrada ou não. A regra 4 analisa o caso em que o Web Service gera uma resposta com HTTP 500 em ausência de ataques e em presença deles gere uma resposta com um dos códigos, HTTP 200, 300 e 500. A regra 5 descreve o caso em que o Web Service não responde, considerando como inconclusiva. A regra 6 é uma exceção ao restante das regras, para o caso em que nenhuma das outras regras possa classificar o resposta. Estas regras são descritas na Figura 7.8.

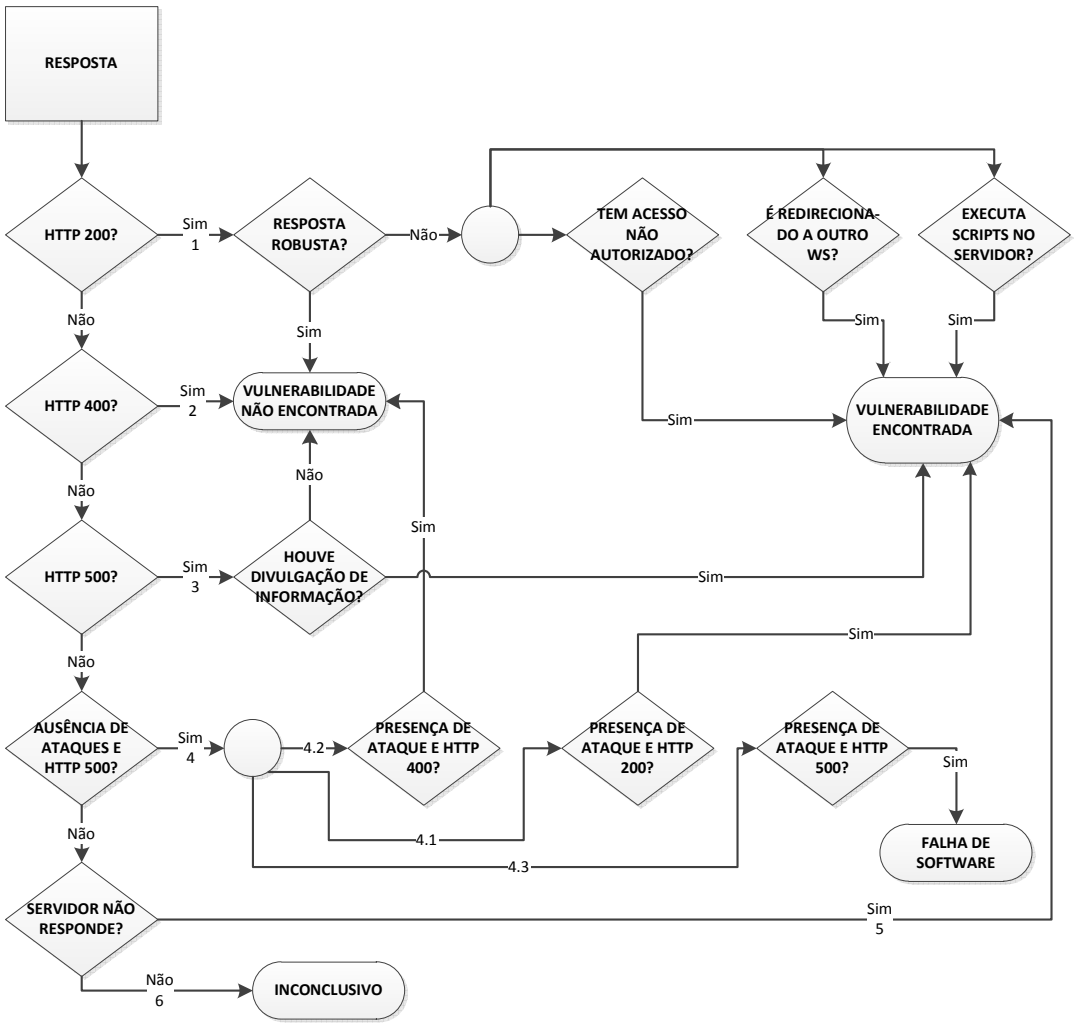



Figura 7.8 Analise de repostas para Web Services.

A partir da utilização destas regras, podemos classificar em 6 categorias as respostas das requisições feitas pelo add-on, aos 69 Web Services, as quais são:

- Vulnerabilidade encontrada (VE): Detecção de uma vulnerabilidade (`FAILED`) no Web Service pelo add-on Security Testing, que realmente sucedeu.
- Vulnerabilidade não encontrada (VNE): detecção de uma resposta robusta (`VALID – OK`) pelo add-on, que realmente não aconteceu.
- Falsos positivos (FP): detecção de vulnerabilidade (`FAILED`) pelo add-on, que não aconteceu.
- Falsos negativos (FN): detecção de uma resposta robusta (`VALID – OK`) pelo add-on, que realmente aconteceu.
- Falha de software (FS): Quando o erro não foi provocado pelo ataque, senão por um erro de software do servidor ou do Web Service.
- Inconclusivo: resposta não possível de ser classificada, segundo as regras de análise de respostas para Web Services da Tabela 7.5. Se o testador tivesse uma abordagem caixa branca, as presentes regras poderiam ser refinadas para determinar melhor o acontecido no servidor ao ser recebido uma mensagem SOAP corrompida.

## 7.5. Aplicação das Regras aos Resultados da Fase de Pré-analise

Para determinar se encontramos uma vulnerabilidade no Web Service, inicialmente utilizamos os logs com os resultados da injeção de ataques do VS soapUI. Estes logs contêm as requisições feitas pelo cliente e as respostas do Web Service. Na linha 1 da Figura 7.9 descreve a hora e data do inicio dos testes. Na linha 2 descreve a existência de vulnerabilidades no Web Service pela injeção de diversos ataques e a duração dos testes em milissegundos (379103 ms). A linha 3 mostra o início dos testes de segurança de Cross-site Scripting, descrevendo a existência de Alertas e a duração desta fase (292770 ms). Na linha 4 começa a descrição da requisição 1 (Request 1) com o resultado de **FAILED**, citada em §7.4, para alertar sobre a presença de vulnerabilidades no Web Service. Nas linhas 5 e 6 mostram o script do ataque de Cross-site Scripting injetado na requisição e o tempo que o servidor demorou em responder (27887 ms). Nas linhas 7 e 8, o servidor responde com as mensagens error: Unexpected element: CDATA e Unexpected element: CDATA para descrever a existência de caracteres XML inesperados na requisição. Na linha 9 começa a descrição da requisição 2 (Request 2) com o resultado de **OK**, citada em §7.4, para ataques mal sucedidos porque o servidor respondeu com uma mensagem

robusta. Na linha 10 descreve o script injetado na requisição e o tempo que o servidor demorou em responder (15483 ms).

```
1  SecurityTest started at 2011-10-26 10:11:11.568
2  Step 1 [Request 1] Alerts: took 379103 ms
3  SecurityScan 1 [Cross Site Scripting] Alerts, took = 292770
4  [Cross Site Scripting] Request 1 - FAILED –
5  [CountryName=<SCRIPT>document.write("<SCRI");</SCRIPT>PT
6  SRC="http://soapui.org/xss.js"></SCRIPT>]: took 27887 ms
7  -> error: Unexpected element: CDATA
8  -> Unexpected element: CDATA
9 [Cross Site Scripting] Request 2 - OK - [CountryName=<SCRIPT a=">'>"
10 SRC="http://soapui.org/xss.js"></SCRIPT>]: took 15483 ms
```

Figura 7.9 Log do Security Testing produzido pela soapUI.

Como segundo passo, utilizamos a ferramenta *Message Viewer* do add-on Security Testing para analisar cada requisição e resposta injetada pelo VS soapUI. Inicialmente, revisamos o código de status HTTP que proporcionam informação acerca do estado da resposta do Web Service e identifica possíveis vulnerabilidades na mensagem SOAP. No exemplo da Figura 7.10, observamos que a resposta da requisição 1 (Request 1) contém o código de status HTTP 400 *Bad Request*, i.e. a requisição não pode ser cumprida devido a sintaxe ruim, citada em §7.4, considerado como uma resposta robusta do servidor que detectou o ataque.

Já que a requisição não foi executada devido à sintaxe ruim (HTTP 400) e o add-on Security Testing classificou a resposta como uma vulnerabilidade encontrada ou possível alerta (`FAILED`) no Web Service. Utilizando a regra 2 da Tabela 7.5 “para toda resposta que contem o código de status HTTP 400 bad request message, será classificada como vulnerabilidade não encontrada (VNE)”, nós classificamos como falso positivos (FP) porque não achamos nenhuma vulnerabilidade encontrada, a comparação do VS soapUI.

```
1:  HTTP/1.1 400 Bad Request
2:  Cache-Control: private
3:  Transfer-Encoding: chunked
4:  Content-Type: text/html
5:  Server: Microsoft-IIS/7.0
6:  X-AspNet-Version: 4.0.30319
7:  X-Powered-By: ASP.NET
8:  Date: Tue, 13 Mar 2012 20:55:40 GMT
9:  Bad Request
```

Figura 7.10 Resposta robusta a um Ataque de Cross-site Scripting (XSS).

No exemplo da Figura 7.11, observamos que a resposta da requisição 2 (Request 2) contêm o código de status HTTP 200 *OK* para requisições HTTP bem sucedidas, i.e. a transmissão da mesma foi completada, mas não que a solicitude foi aceita ou processada pelo Web Service, citada em §4. A seguir, procuramos dentro da resposta algum indicio que sugira que o analisador XPath processou a requisição. Na linha 15, o servidor cria uma coleção de dados tabulados através do script <NewDataSet /> como resposta à injeção do ataque de Cross-site Scripting. As restantes das linhas descrevem a estrutura de uma resposta XML.

A requisição com o script de ataque "[CountryName=<SCRIPT a=">'>" SRC="http://soapui.org/xss.js"></SCRIPT>]" foi executada, gerando um novo objeto NewDataSet para a criação de uma coleção de dados. Além disso, a mensagem SOAP respondeu com o código de status HTTP 200 e o add-on Security Testing classificou a resposta como uma vulnerabilidade não encontrada (OK). Utilizando a regra 1 da Tabela 7.5 "Se a resposta contém mensagem com código HTTP 200 Ok e o usuário executou trechos de código no servidor (e.g. Java script), então existe uma vulnerabilidade encontrada (VE)", achamos vulnerabilidades (**em negrito**) no Web Services, pelo que classificamos como falso negativo à resposta do VS soapUI.

```
1:   HTTP/1.1 200 OK
2:   Cache-Control: private, max-age=0
3:   Content-Length: 455
4:   Content-Type: text/xml; charset=utf-8
5:   Server: Microsoft-IIS/7.0
6:   X-AspNet-Version: 4.0.30319
7:   X-Powered-By: ASP.NET
8:   Date: Mon, 26 Oct 2011 10:12:06 GMT
9:
10:  <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
11:  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
12:  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
13:     <soap:Body>
14:        <GetISOCountryCodeByCountyNameResponse xmlns="http://www...X.NET">
15:           <GetISOCountryCodeByCountyNameResult>&lt;NewDataSet
16:  /></GetISOCountryCodeByCountyNameResult>
17:        </GetISOCountryCodeByCountyNameResponse>
18:     </soap:Body>
19:  </soap:Envelope>
```

Figura 7.11 Vulnerabilidade encontrada pela injeção de Cross-site Scripting (XSS).

Este procedimento descrito anteriormente é utilizado para analisar as respostas dos ataques injetados pelo add-on Security Testing aos 69 Web Services (Seção 7.3). Os resultados são sumarizados na Tabela 7.6. No total, registramos 35.087 respostas pela injeção dos 7 ataques aos 69 *Web Services*, classificados em: vulnerabilidades encontradas (VE), vulnerabilidades não encontradas (VNE), falsos positivos (FP) e falsos negativos (FN).

Tabela 7.6 Resumo de ataques contra os 69 Web Services.

| TOTAL | Vulnerabilidades Encontradas (VE) | Vulnerabilidades Não Encontradas (VNE) | Falsos Positivos (FP) | Falsos Negativos (FN) |
|---|---|---|---|---|
| **35087** | 9.857 | 6.015 | 10.520 | 8.695 |
| **100.00%** | 28,09% | 17,14% | 29,98% | 24,78% |

Os resultados sugerem que o VS SoapUI apresenta uma grande porcentagem de FP (29,98%) e FN (24.78%). Somando FP e FN, representam quase 2/3 do total dos testes. Os falsos positivos superam as vulnerabilidades encontradas (28.09%). Por último temos os ataques detectados que representam o 17,14%. Na Figura 7.12, observamos a classificação dos ataques em porcentagens.

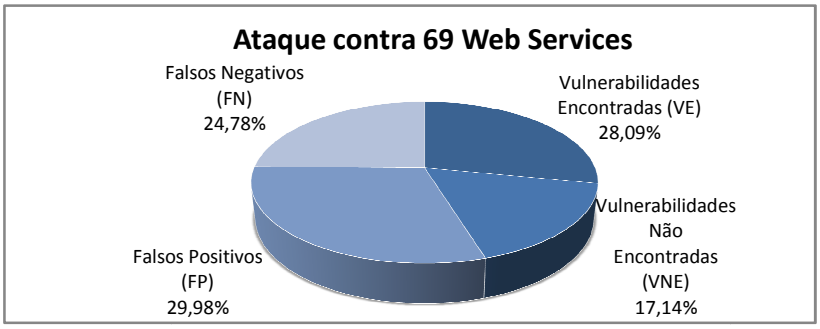



Figura 7.12 Ataques contra os 69 Web Services.

Tabela 7.7 Classificação das respostas por ataques contra Web Services.

| Ataque | Total | Vulnerabilidades Encontradas (VE) | Vulnerabilidades Não Encontradas (VNE) | Falsos Positivos (FP) | Falsos Negativos (FN) |
|---|---|---|---|---|---|
| **Cross-Site Scripting (XSS)** | **19.108** | 3.727 | 1.534 | 8.871 | 4.976 |
| **Fuzzing Scan** | **5.900** | 2.511 | 2.005 | 307 | 1.077 |
| **Invalid Types** | **3.415** | 1.222 | 1.035 | 141 | 1.017 |
| **SQL Injection** | **2.517** | 1.061 | 700 | 97 | 659 |
| **XPath Injection** | **1.840** | 841 | 520 | 79 | 400 |
| **Malformed XML** | **1.670** | 328 | 173 | 766 | 403 |
| **XML Bomb (DoS)** | **637** | 167 | 48 | 259 | 163 |
| **Total** | **35.087** | **9.857** | **6.015** | **10.520** | **8.695** |

A Tabela 7.7 fornece uma classificação por quantidade de VE pela injeção dos 7 ataques contra os 69 Web Services. O ataque de Cross-site Scripting (XSS) representa a maioria das respostas injetadas, chegando a um total de 3.727 VE, seguido de Fuzzing Scan com 2.511 VE,

Invalid Types com 1.222 VE, SQL Injection com 1.061 VE, XPath Injection com 841 VE, Malformed XML com 328 VE e XML Bomb com 167 VE. No total 9.857 VE foram achadas nos 69 Web Services. Também podemos, na Tabela 7.7, observar a classificação das VNE, FP e FN.

Na Figura 7.13 exibimos os dados da Tabela 7.7 em porcentagens. Os ataques com maior número de VE são: XPath Injection com 45,71%; Fuzzing Scan com 42,56% e SQL Injection com 42,15%. Os ataques com maior número de VNE são: Fuzzing Scan com 33,98%; Invalid Types com 30,31%; XPath com 28,29% e SQL Injection com 27,81%. Na Figura 7.13 podemos observar estes resultados. Com relação aos FP, observamos que três ataques superam o 40% das requisições feitas, eles são Cross-site Scripting (XSS) com 46,43%, Malformed XML com 45,87% e XML Bomb com 40,66%. Os ataques com maior número de FN são: Invalid Types com 29,78% e Cross-site Scripting com 26,04%.

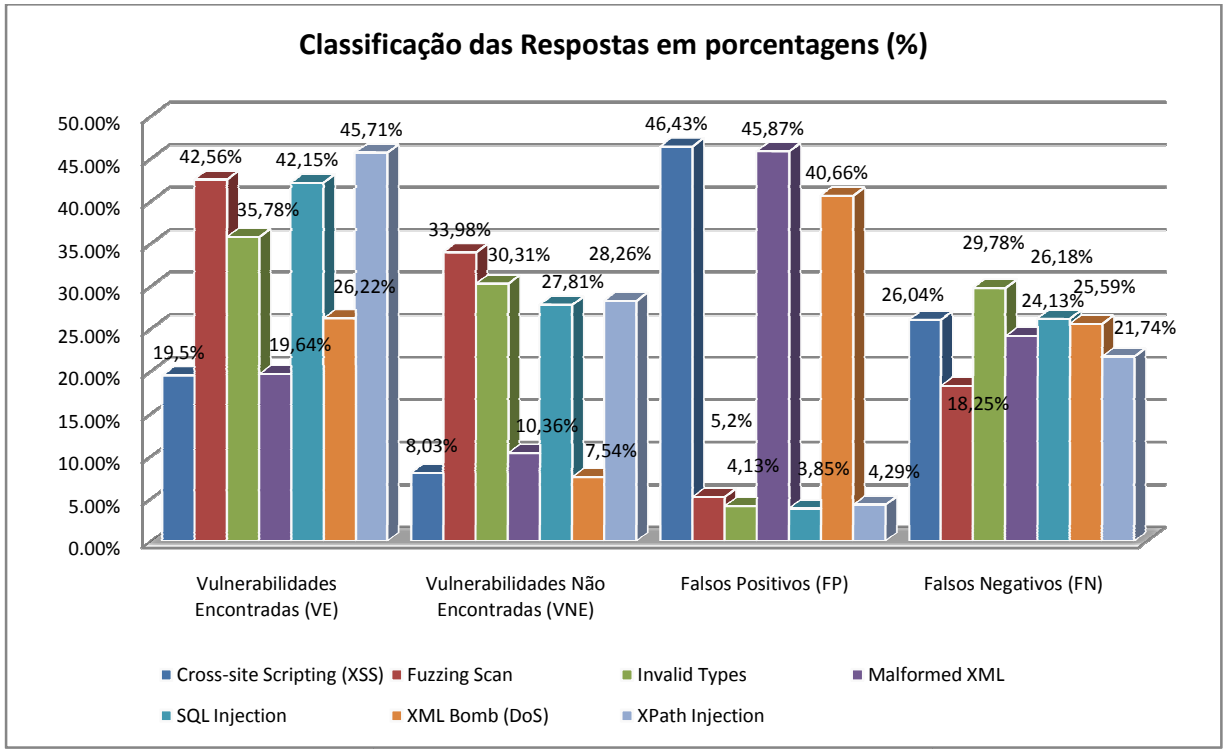



Figura 7.13 Ataques injetados pelo VS soapUI contra Web Services em (%).

A Tabela 7.8 descreve a quantidade de Web Services que apresentam VE, VNE, FP ou FN entre suas respostas. O resultado máximo é 69, por exemplo, quando os 69 Web Services apresentaram VE em relação a um ataque ou 0 quando nenhum Web Services dos 69 apresentou

vulnerabilidades a um ataque. Podemos observar que 40 Web Services apresentaram vulnerabilidades pela injeção do ataque de Cross-site Scripting.

Tabela 7.8 Presença de vulnerabilidades detectadas nos 69 Web Services.

| Ataque | Vulnerabilidades Encontradas (VE) | Vulnerabilidades Não Encontradas (VNE) | Falsos Positivos (FP) | Falsos Negativos (FN) |
|---|---|---|---|---|
| **Cross-site Scripting (XSS)** | 40 | 23 | 46 | 37 |
| **Fuzzing Scan** | 26 | 21 | 4 | 12 |
| **Invalid Types** | 29 | 20 | 3 | 25 |
| **Malformed XML** | 31 | 24 | 42 | 38 |
| **SQL Injection** | 25 | 20 | 3 | 22 |
| **XML Bomb (DoS)** | 12 | 4 | 36 | 18 |
| **XPath Injection** | 26 | 22 | 4 | 20 |

Na Figura 7.14 exibimos os dados da Tabela 8 em porcentagens. 57,97% dos Web Services testados apresentam vulnerabilidades encontradas contra o ataque de Cross-site Scripting, no entanto o 66,67% das respostas estão classificadas como FP. O restante dos ataques podem ser revisados na Figura 7.14.

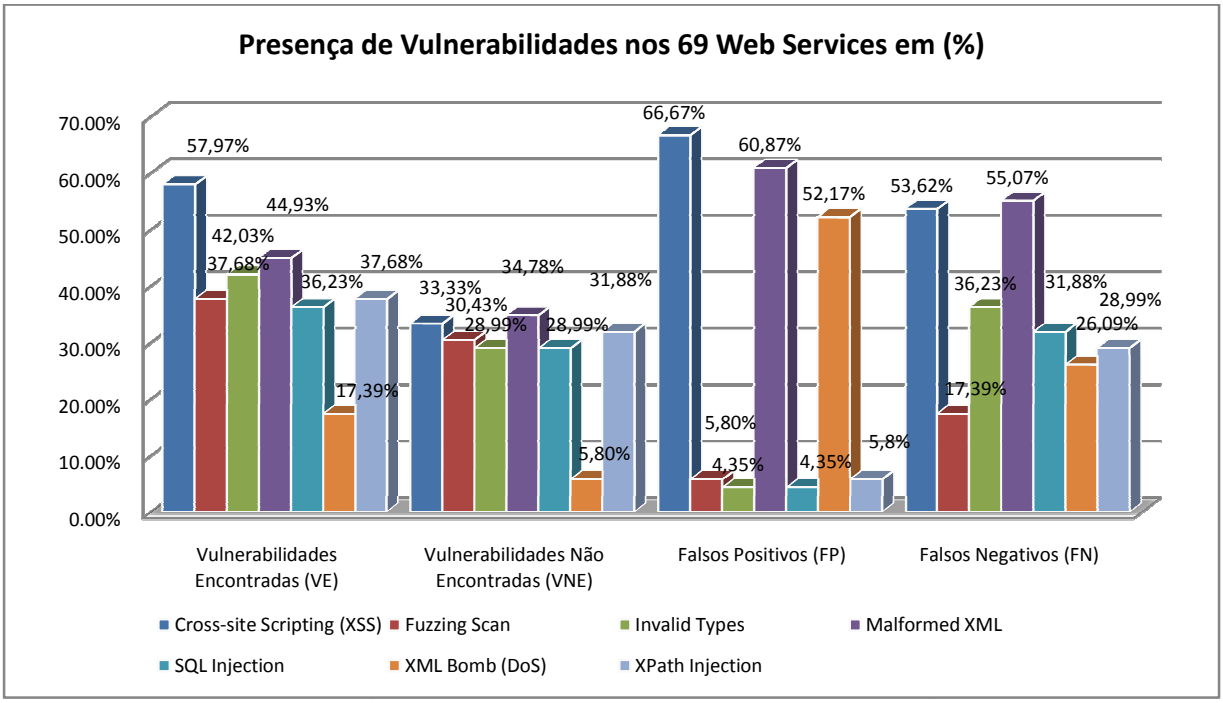



Figura 7.14 Presença de vulnerabilidades detectadas nos 69 Web Services em porcentagens.

Para conhecer o nível de vulnerabilidade de um Web Service, coletamos as respostas classificadas como vulnerabilidades encontradas e falsos negativos (VE+FN). Caso um Web Services apresente ao menos uma destas classes de resposta, será classificado como Web Services vulnerável na Tabela 7.9. Lembramos que ±10% representa o nível de confiabilidade, obtido na Seção 7.2. Por exemplo, podemos dizer que 84,06% ±10% dos Web Services testados são suscetíveis a ataques de Cross-site Scripting segundo nossa pesquisa.

Tabela 7.9 Porcentagem de Web Services Robustos e Vulneráveis.

| Ataques | Web Services Vulneráveis (VE+FN) | % |
|---|---|---|
| **Cross-site Scripting (XSS)** | 58 | 84,06% |
| **Malformed XML** | 54 | 78,26% |
| **Invalid Types** | 48 | 69,57% |
| **SQL Injection** | 40 | 57,97% |
| **XPath Injection** | 40 | 57,97% |
| **Fuzzing Scan** | 38 | 55,07% |
| **XML Bomb (DoS)** | 29 | 42,03% |

## 7.5.1 Configuração do Vulnerability Scanner soapUI

Como parte dos resultados, avaliamos a precisão (**P**), recall (recuperação - **R**) e a Média Harmônica (**F-Measure**) para os resultados obtidos do add-on Security Testing [77]. A seguir definimos as três medidas:

A precisão (**P**) é a fração de casos recuperados que são relevantes. Para nosso contexto, precisão é a recuperação das vulnerabilidades nos Web Services (*True Positive*) evitando falsos positivos (FP). Precisão (**P**) é definido na Fórmula 7.2.

$$P = \frac{\mathrm{TP}}{\mathrm{TP}*\mathrm{FP}} \qquad (7.2)$$

O *recall* (**R**), chamado também Recuperação, é a fração dos casos pertinentes que são recuperados. Para nosso contexto, recall permite analisar a precisão da ferramenta add-on Security Testing em relação à recuperação das vulnerabilidades (*True Positive*) e as vulnerabilidades classificadas como faltos negativos (FN). Recall (**R**) é definido na Fórmula 7.3.

$$R = \frac{\mathrm{TP}}{\mathrm{TP}*\mathrm{FN}} \qquad (7.3)$$

Com respeito à Média Harmônica (**F-Measure** ou **F1**), é uma medida de precisão de testes para recuperação de informação. Considera tanto a precisão (**P**) quanto o recall (**R**) para

calcular a recuperação da informação. **F-Measure** pode ser interpretado como a média ponderada da precisão (**P**) e recall (**R**), onde a pontuação de F-Measure atinge a "1" para o melhor valor e "0" para o pior. Para nosso contexto, "1" representa a recuperação de todas as vulnerabilidades no Web Service pelo add-on Security Testing e "0" para o caso em que VS não recuperou nenhuma vulnerabilidade. **F1** é definido na Fórmula 7.4.

$$F1 = 2 * \frac{P*R}{P+R} \quad (7.5)$$

Na Tabela 7.10 apresentamos os resultados de P, R e F-Measure para todos os ataques.

Tabela 7.10 Avaliação do VS soapUI por Precisão, Recall (recuperação) e Média Harmônica.

| Ataque | Vulnerabilidades Encontradas (VE) | Falsos Positivos (FP) | Falsos Negativos (FN) | Precisão (P) | Recall (R) | F-Measure (F1) |
|---|---|---|---|---|---|---|
| **Fuzzing Scan** | 26 | 4 | 12 | 86,67% | 68,42% | **76,47%** |
| **XPath Injection** | 26 | 4 | 20 | 86,67% | 56,52% | **68,42%** |
| **Invalid Types** | 29 | 3 | 25 | 90,63% | 53,70% | **67,44%** |
| **SQL Injection** | 25 | 3 | 22 | 89,29% | 53,19% | **66,67%** |
| **Cross-site Scripting** | 40 | 46 | 37 | 46,51% | 51,95% | **49,08%** |
| **Malformed XML** | 31 | 42 | 38 | 42,47% | 44,93% | **43,66%** |
| **XML Bomb (DoS)** | 12 | 36 | 18 | 25,00% | 40,00% | **30,77%** |

O add-on Security Testing do VS soapUI tem uma precisão (**P**) de capturar vulnerabilidades, acima do 85% para os seguintes ataques: Invalid Types (90,63%), SQL Injection (89,29%), Fuzzing Scan (86,67%) e XPath Injection (86,67%). O add-on tem uma recuperação (**recall - R**) acima do 50% para os seguintes ataques: Fuzzing Scan (68,42%), XPath Injection (56,52%), Invalid Types (53,70%), SQL Injection (53,19%) e Cross-site Scripting (51,95%). Por último, usando **F-Measure**, podemos medir o nível de eficiência da ferramenta para os 7 tipos de ataques: 76,47% para Fuzzing Scan, 68,42% para XPath Injection, 66,67% para SQL Injection e 67,44% para Invalid Types, 49,08% para Cross-site Scripting, 43,66% para Malformed XML e 30,77 para XML Bomb.

Podemos concluir que a qualidade do conjunto de scripts de ataques, que utiliza o add-on Security Testing para ataques como Fuzzing Scan, fazem a diferença entre encontrar vulnerabilidades e diminuir o número de falsos positivos e negativos. Por exemplo, o add-on Security Testing enviou 19.108 requisições com o ataque do Cross-site Scripting, das quais apenas 3.727 conseguiram encontrar vulnerabilidades, representando só o 19,51% das

requisições enviadas aos 69 Web Services e distribuídas em 58 Web Services, obtendo 49,08% de eficiência. Fazendo uma comparação com Fuzzing Scan, o add-on Security Testing enviou 5.900 requisições, das quais 2.511 conseguiram encontrar vulnerabilidades, representando 42,56% das requisições enviadas e distribuídas em 38 Web Services, obtendo 76,47% de eficiência.

Esta fase de pré-análise teve por objetivo identificar as operações e parâmetros que seriam mais interessantes de usar como alvo, ou seja, ataques com mais probabilidade de ocorrência de defeitos (*failures*) para ser replicados na fase de injeção (Capítulo 8). No próximo capítulo utilizaremos a Tabela 7.5 para analisar as respostas dos Web Services e emularemos diversos tipos de ataques com o injetor de falhas (IF) WSInject.

Os experimentos da fase de pré-análise nos permitiram determinar os Web Services, suas operações e parâmetros que seriam mais interessantes de usar como alvo, i.e. injetar naqueles que levaram mais probabilidade de ocorrência de defeitos (*failures*).

# Capítulo 8. Ataques com o IF WSInject

Este capítulo apresenta a segunda fase da dissertação, composta pela emulação de diversos tipos de ataque pelo injetor de falhas (IF) WSInject, utilizado para analisar a robustez dos Web Services e o padrão WS-Security.

Para iniciar, detalhamos a arquitetura de testes na Seção 8.1, que servirá para executar os experimentos e reproduzir ataques reais.

Na Seção 8.2 detalhamos o procedimento da abordagem de testes de segurança para WS-Security, composta por: **1) preparação**, que inclui o uso dos 10 ataques selecionados na Seção 6.5 utilizando o modelo de **árvore de ataques** contra Web Services e a geração dos **scripts de ataque executável** (SAE) para o IF WSInject, por último, selecionamos 10 Web Services dos 69 da fase de pré-análise (Seção 7.2) , dos quais, 5 Web Services usam o padrão WS-Security e os outros não; **2) execução** da campanha de injeção de ataques contra Web Services e WS-Security; e **3) análise de resultados** para a classificação das respostas dos ataques.

Na Seção 8.3 aplicamos a abordagem caixa preta aos 5 Web Services, utilizando um conjunto de 8 ataques destinados a injetar vulnerabilidades nos serviços. Em contrapartida, na Seção 8.4, aplicamos a mesma abordagem contra os outros 5 Web Services com o padrão WS-Security e Security Tokens, utilizando 10 ataques, cujo objetivo é analisar a robustez do padrão.

Na Seção 8.5, analisamos os resultados da Seção 8.3 e 8.4 para determinar a robustez dos Web Services e o padrão WS-Security contra diversos ataques. Finalmente, na Seção 8.6, concluímos o trabalho, mostrando suas principais contribuições e apresentando os comentários finais em relação aos resultados experimentais do capítulo.

## 8.1 Arquitetura de Testes

A fase de injeção de falhas utiliza a técnica de testes de segurança através do IF WSInject. A ferramenta descrita na Seção 6.3, atua como um Proxy entre o cliente (sistema operacional Windows 7, Core 2 Duo 2GHz e 3Gb RAM) e os Web Services, interceptando as mensagens trocadas.

Um aspecto importante nos testes de Web Services é a geração do tráfego de rede – a carga de trabalho. Ela representa as requisições que ativam o Web Service alvo. Para ser o mais

realista possível durante os testes, dever-se-ia gerar tráfego bem próximo do fluxo real de mensagens SOAP. Para gerar a carga de trabalho (*workload*) foi utilizada o *add-on* Load Testing[28] do VS soapUI, representando o cliente fazendo requisições. Esta arquitetura é mostrada na Figura 8.1.

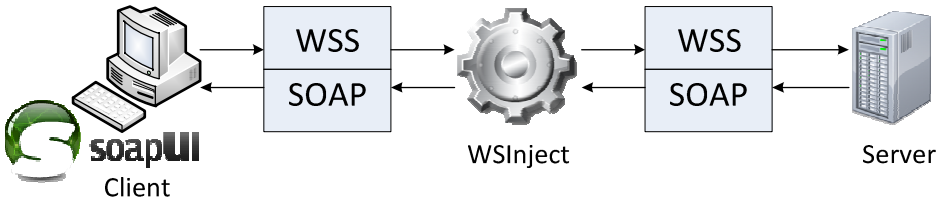



Figura 8.1 Arquitetura de testes para a fase de injeção de falhas.

A arquitetura tem como objetivo: **1)** emular as requisições feitas aos Web Services através do add-on *Load Testing* para emular um cliente real fazendo requisições; **2)** interceptar as mensagens SOAP enviadas e injetar um ataque com o IF WSInject; e **3)** coletar os logs gerados pelo WSInject e o Load Testing do VS soapUI para sua análise. Os logs estão compostos pelas requisições e respostas dos Web Services testados.

Dado que WSInject se comporta como um servidor *proxy* HTTP, sua utilização requer pouca instrumentação; basta configurar o cliente para se conectar ao Web Service alvo através do *proxy*. A interceptação e modificação das mensagens trocadas entre o cliente e o servidor são transparentes entre eles. Dessa forma, a ferramenta não necessita do código fonte do serviço nem de interferir na plataforma de execução, o que a torna possível de ser utilizada tanto por provedores quanto por clientes do Web Services.

## 8.2 Abordagem proposta

Uma vez definida a arquitetura de testes, foram necessárias as seguintes etapas:

1. **Preparação:** nesta etapa foram determinados os ataques a realizar; a carga de falhas a ser utilizada; e selecionar os Web Services que serão testados.
2. **Execução:** esta etapa visa configurar a carga de trabalho para gerar tráfego de mensagens SOAP próximo à realidade. Além disso, são gerados os dados que serão analisados.
3. **Análise dos resultados:** analisa os dados coletados na etapa anterior e determina a existência de vulnerabilidades.

[28] http://soapui.com/Getting-Started/load-testing.html

## 8.2.1 Preparação

Embora seja possível emular diversos tipos de ataques com o WSInject, nos centramos em ataques de injeção e negação de serviços, composto por um conjunto de 10 tipos de ataques selecionados na Seção 6.5. Utilizamos a ferramenta SecurITree V3.4 [37] para a geração da árvore de ataques contra Web Services e WS-Security, permitindo classificar os ataques segundo a propriedade que atinge: integridade ou disponibilidade. Como podemos observar na Figura 8.2, cada ataque cumpre com os critérios <P, P, P>, que definem respectivamente: **i)** a capacidade do atacante; **ii)** a possibilidade de emular o ataque pelo IF WSInject; e **iii)** as características necessárias dos Web Services para injetar o ataque.

**OR 1 – Objetivo: Atacar o Web Services e WS-Security**
- **OR 1.2** Violar **Integridade**
  - **OR 1.2.2** XML Injection **<P, P, P, P>**
  - **1.2.4** XPath Injection **<P, P, P , P>**
  - **1.2.5** Cross-site Scripting (XSS) **<P, P, P, P>**
  - **1.2.6** Fuzzing Scan **<P, P, P, P>**
  - **1.2.9** Malformed XML **<P, P, P, P>**
- **OR 1.3** Violar **Disponibilidade**
  - **OR 1.3.2** Oversize Payload **<P, P, P, I>**
  - **1.3.3** Coercive Parsing **<P, P, P, I>**
  - **1.3.4** Oversize Cryptography **<P, P, P, I>**
  - **1.3.5** Attack Obfuscation **<P, P, P, I>**
  - **1.3.6** XML Bomb **<P, P, P,I>**

Figura 8.2 Árvore de ataque textual para Web Services com WS-Security.

O **4º** atributo, orientado à utilização do padrão WS-Security com Security Tokens, descrito na Seção 6.1, classifica em possível (**P**) quando protege o serviço e impossível (**I**) quando não. Utilizando o mesmo critério da Seção 6.5, podemos dividir em dois cenários:

***Cenário A) Ataques contra Web Services:*** Esta sub-árvore classifica os ataques contra Web Services e esta composta por XML Injection, XPath Injection, Cross-site Scripting (XSS), Fuzzing Scan, Malformed XML, Oversize Payload, Coercive Parsing, e XML Bomb.

***Cenário B) Ataques contra WS-Security:*** Esta sub-árvore classifica um conjunto de ataques contra WS-Security e esta composta por XML Injection, XPath Injection, Cross-site Scripting (XSS), Fuzzing Scan, Malformed XML, Oversize Payload, Coercive Parsing, XML Bomb, Oversize Payload, Coercive Parsing, Oversize Cryptography, Attack Obfuscation e XML

Bomb. É importante destacar a utilização de dois ataques, Oversize Cryptography e Attack Obfuscation. Eles foram incluídos porque atacam exclusivamente a WS-Security.

Para exemplificar a utilização da metodologia desenvolvida no Capítulo 6, selecionamos dois dos ataques de cada cenário: **Cenário A)** *XML Injection*; **Cenário B)** *Attack Obfuscation*, descritos a seguir.

**Cenário A)** *XML Injection* explora a estrutura XML de uma mensagem SOAP (ou qualquer outro documento XML) através da adição de etiquetas XML. O ataque insere conteúdo malicioso na mensagem SOAP a ser enviada, tanto para o servidor quanto ao cliente. Este conteúdo é considerado como uma parte da estrutura da mensagem SOAP e pode provocar efeitos indesejáveis [19 – 22], [64]. Por exemplo, suponha que o servidor mantém o arquivo XML mostrado na Figura 8.3 com informações do usuário Alice.

```
1: <users>
2:         <user>
3:                 <name>Alice</name>
4:                 <password>ghj348</password>
5:                 <account>4859-3</ account >
6:         </user>
7:         <user>
8:                 ...
9: </users>
```

Figura 8.3 Exemplo de trecho de arquivo XML com informações de usuários.

Uma aplicação Web que necessite autenticar um usuário e transferir dinheiro de uma conta para outra do mesmo banco, pede ao cliente que forneça seu nome e senha de usuário (ver Figura 8.4). O Servidor recebe um comando XML da forma:

```
1: <user:userReq>
2:       <user:name>Alice</user:name>
3:       <user:password>********</user:password>
4:       <user:account>4859-3</user:account>
5:       <user:amount_Sus>1000</user:amount_Sus>
6:       <bank:name_sender>Bob<bank:name_sender>
7:       <bank:account_sender>2598-0<bank:account_sender>
8: </user:userReq>
```

Figura 8.4 Autenticação de usuário para transferência de dinheiro.

Um atacante pode agregar tags ao "*name_sender*" e "*account_sender*" requisitado (ver Figura 8.5), sem necessidade de ter o conhecimento da senha:

```
1:      <bank:name_sender>Bob<a>John</a><bank:name_sender>
2:      <bank:account_sender>2598-0<b>4982-3</b><bank:account_sender>
```

Figura 8.5 Injeção de XML na requisição para transferência de dinheiro.

Com essa modificação, o atacante "*John*" adiciona duas operações para receber US$. 1000 em sua conta 4982-3 como parte da transação.

Para emular esse tipo de ataque, o injetor deve interceptar mensagens SOAP contendo comandos XML e corromper os valores dos parâmetros. Para definir os valores a serem usados, nos baseamos na fase de pré-análise (Seção 7.5) e na literatura [20, 21]. Os **scripts de ataque executável** (SAE) gerados para o IF WSInject da Seção 6.5 são descritos na Tabela 8.1.

Tabela 8.1 Scripts usados para emular XML Injection por WSInject.

| Script WSInject | Descrição |
|---|---|
| `isRequest(): stringCorrupt("</ser:Username>", "</ser:Username><ser:Username2>hacker</ser:Username2>");` | O script usa a condição **isRequest()** que seleciona somente as requisições de um cliente para um servidor. Para cada requisição o injetor usa **stringCorrupt** adiciona informações as ocorrências da operação </ser:Username> por uma consulta XML Injection (ex.: "`</ser:Username><ser:Username>hacker</ser:Username>`" or ("`<ser:Username>EdgarFilings`", "`<ser:Username><a>EdgarFilings</a></ser:Username>`"), agregando "Tags" à requisição da mensagem SOAP para realizar operações não permitidas ou provocar efeitos indesejáveis no WS. |
| `isRequest(): stringCorrupt("</ser:Password>", "</ser:Password><ser:Password2>administrator</ser:Password2>");` | |
| `isRequest(): stringCorrupt("</ser:Identifier>", "</ser:Identifier><ser:Identifier>xignite</ser:Identifier>");` | |
| `isRequest(): stringCorrupt("</ser:Tracer>", "</ser:Tracer><ser:Tracer>666</ser:Tracer>");` | |
| `isRequest(): stringCorrupt("<ser:Username>EdgarFilings", "<ser:Username><a>EdgarFilings</a></ser:Username>");` | |

**Cenário B)** O segundo ataque, *Attack Obfuscation*, aproveita as limitações do analisador XML (*XPath Parser*), que não interpreta o conteúdo cifrado na mensagem SOAP. Este ataque consiste em usar elementos cifrados no cabeçalho para inserir ataques cifrados como *Oversize Payload*, *Coercive Parsing*, *XML Injection*, entre outros, obrigando ao analisador XML a decifrar os segmentos e processar o ataque como parte da mensagem. Este ataque surge da utilização de WS-Security, que usa elementos para cifrar partes da mensagem SOAP e está presente nos Web Services que usam este padrão [4], [20, 21], [24].

Na Figura 8.6 (a), utilizamos a etiqueta *Nonce*[29], que usa o conteúdo cifrado para autenticar usuários mediante o uso de credencias de segurança (Security Tokens). Neste caso, o analisador XML não detecta o ataque inserido na linha 9 da requisição. Na Figura 8.6 (b) observamos o ataque decifrado para modificar a hora do servidor, desenvolvido em JavaScript. Na Tabela 8.2, podemos encontrar outros **scripts de ataque executável** para *Attack Obfuscation* a ser injetados pelo IF WSInject.

[29] A etiqueta *Nonce* serve como mecanismo de autenticação, gerando números aleatórios, que são adicionados à mensagem através da função resumo (*hash*) e tem Base64 ( SHA-1 (nonce + created + password ) ).

**Attack Obfuscation - Requisição**

```
1:  <SOAP:Envelope xmlns:SOAP="...">
2:    <SOAP:Header>
3:      <wsse:Security xmlns:wsse="...">
4:        <wsse:UsernameToken wsu:Id="...”>
5:          <wsse:Username>admin
6:          </wsse:Username>
7:          <wsse:Password ...>adminadmin
8:          </wsse:Password>
9:  <wsse:NonceEncodingType=...Base64Binary">
10: <#>N:AES:64<#>1xxYGhxVbNyR+G4HRsXninGS/CT
11: AAaTq2BuYBq39FqQzWEJ21DAhkiPAwlwTgCDLtJSe
12: gjCOvFQQDie11NiRWb6E54Yp9FH5p7neBgwVIe7uJ
13: 6OBmjhkl87BqIvIXmQx3VV3TJQtDyxe2VH2TE2...
14:         ... <!– Attack Obfuscation-->
15:         vYDNFj9Elw0EM0YI6Yt4jomx4=
16:          </wsse:Nonce>
17:       </wsse:UsernameToken>
18:       <wsu:Timestamp wsu:Id="TS-18">
19:       <wsu:Created>2012-...</wsu:Created>
20:       <wsu:Expires>2012-...</wsu:Expires>
21:       </wsu:Timestamp>
22:     </wsse:Security>
23:     </soapenv:Header>
24:     <soapenv:Body>
25:        ...
26:     </soapenv:Body>
27: </soapenv:Envelope>
```

(a) Attack Obfuscation cifrado

**Ataque decifrado para modificar a hora do servidor**

```
1:   import java.io.File;
2:   import java.io.IOException;
3:   import javax.xml.parsers.ParserConfiguration...
4:   import org.xml.sax.SAXException;
5:   public class ChangeTime {
6:    public static void main(String[  ] args)
7:    throws IOException, SAXException, ... {
8:    // Instantiate an Time object
9:    Time inst = new Time(new File(args[0]));
10:   // Get current
11:   System.out.println("Current Time");
12:   System.out.println("Hour: " + inst.getHour());
13:   System.out.println("Minute: " ...;
14:   System.out.println("Second: " + ...");
15:     inst.setTimezone("MDT");
16:     inst.setHour(12);
17:     inst.setMinute(23);
18:     inst.setSecond(05);
19:     inst.setMeridiem("p.m.");
20:   // Print the new XML
21:   System.out.println("New Time");
22:   System.out.println(inst.makeTextDocument());
23:  }
```

(b) Attack Obfuscation decifrado

Figura 8.6 Ataque de Attack Obfuscation.

Tabela 8.2 Scripts usados para emular Attack Obfuscation por WSInject.

| Script WSInject | Descrição |
|---|---|
| `isRequest(): stringCorrupt("Base64Binary\">", "Base64Binary\"><#>N:AES:64<#>1xxYGhxVbNyR+G4HRsXninGS/CTA ... <!-troca a hora do servidor--> ENKpq+hJvy7z1JYg7OyLI3eqcZbNXvYDNFj9Elw0EM0YI6Yt4jomx4=");` | O script usa a condição **isRequest()** que seleciona somente as requisições de um cliente para um servidor. Para cada requisição o injetor usa **stringCorrupt** para adicionar cadeias, cifradas ou não, em etiquetas que usem cifrado binária em base 64, e.g. a etiqueta *Nonce*. Para injetar Attack Obfuscation, usamos o script a seguir: `isRequest(): stringCorrupt ("Base64Binary\">", "</ser:Username>"Base64Binary\"> fU7qvq... ==")` Injeta um script para trocar a hora do relógio do servidor. |
| `isRequest(): stringCorrupt("Base64Binary\">", "Base64Binary\"><#>N:AES:64<#>1xxYGhxVbNyR+G4HRsXninGS/CTA ... <!-descarrega um arquivo pdf no servidor--> fyPt+82Hz6mWjAMZYOdKTlpaj91/G4F6gp5o6LS+fn/m+ztgAlnnQ==");` | |
| `isRequest(): stringCorrupt("Base64Binary\">", "Base64Binary\"><#>N:AES:64<#>1xxYGhxVbNyR+G4HRsXninGS/CTA ... <!-insere um script com um loop infinito--> E0B1vM0hMlRuqdPrkbwSQkjQbCdMGqTP8ftZB1VveElU95xdjHdlg==");` | |

Descritos os dois cenários de ataque, podemos prosseguir com a seleção dos Web Services, que são o alvos de nossos ataques. A fase de pré-análise nos permitiu identificar os Web Services, suas operações e parâmetros que seriam mais interessantes usar como alvo, i.e. injetamos naqueles que levaram mais probabilidade de ocorrência de defeitos (*failures*).

Para a seleção dos Web Services tomamos como base 69 Web Services da Seção 7.2. Selecionamos 10 Web Services dos 69, dos quais 5 usam o padrão WS-Security e Security Tokens, de acordo com os seguintes critérios:

- Os Web Services devem conter ao menos uma operação em que se possa injetar ataques.
- Os Web Services deverão ter um comportamento definido e fácil de identificar, ante uma injeção de falha.
- Os Web Services com WS-Security, devem conter os atributos necessários para a execução do ataque, e.g. devem usar Security Tokens com a etiqueta *Nonce* para injetar os ataques de Oversize Cryptography e Attack Obfuscation.

Por questões de segurança não podemos divulgar a URL nem o nome dos serviços.

## 8.2.2 Execução

Para executar os testes de segurança em Web Services utilizamos o gerador de tráfego *add-on Load Testing*[30]do *VS soapUI*, descrito na Seção 8.1. Este gerador de fluxo de trabalho (*workload*) é utilizado nos dois cenários da etapa de preparação, para ataques contra Web Services e WS-Security. Descrevemos as duas campanhas de injeção de falhas a seguir:

**Cenário A)** Na Figura 8.7 da campanha de teste de Web Services, ilustra-se a injeção de ataques para os 5 Web Services que não usam o padrão WS-Security. Para todos os Web Services foram selecionado 8 ataques descritos na etapa de preparação, cada cenário de ataque fica constituído de 5 scripts e cada script especifica a corrupção de um determinado valor (parâmetro, operação ou estrutura do envelope da mensagem SOAP), conforme ilustrado na Tabela 8.1. Para cada script, a carga de trabalho consistiu no envio de 100 requisições. O processo foi repetido para todos os Web Services, fazendo um total de 20.000 ataques realizados.

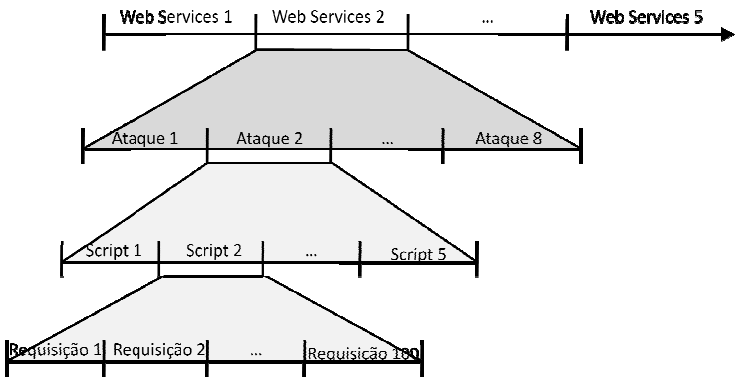



Figura 8.7 Campanha de Injeção de Ataques contra Web Services.

[30] http://www.soapui.org/Getting-Started/load-testing.html

**Cenário B)** Na campanha de injeção de ataques contra WS-Security, descrita na Figura 8.8, selecionamos 5 Web Services que usam o padrão WS-Security. Para todos os Web Services foram selecionados 10 ataques da etapa de preparação. Cada cenário de ataque fica constituído de 5 scripts, e cada script especifica a corrupção de um determinado valor (parâmetro, operação ou estrutura do envelope da mensagem SOAP), conforme ilustrado na Tabela 8.2. Para cada script, a carga de trabalho consistiu no envio de 100 requisições. O processo foi repetido para todos os Web Services, fazendo um total de 25.000 ataques realizados.

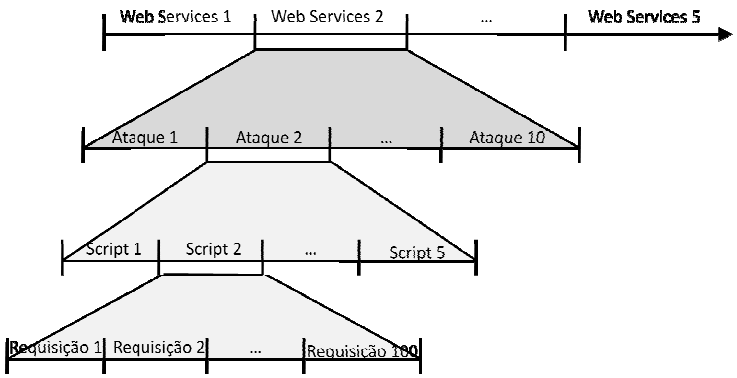



Figura 8.8 Campanha de Injeção de Ataques contra WS-Security.

Cumpre notar que o número de ataques possíveis de serem realizados, em que cada parâmetro de cada operação de cada serviço com diferentes valores, seria inviável de ser realizado dentro do tempo previsto para a realização dos testes. Por essa razão, optamos por realizar apenas um subconjunto dos testes.

## 8.2.3 Análise dos Resultados

Um aspecto importante nesta etapa é conseguir identificar quando uma vulnerabilidade foi efetivamente detectada, excluindo potenciais falsos positivos e negativos. Com base nos experimentos da fase de pré-análise da Seção 7.5 e na interpretação dos códigos de status HTTP na Tabela 7.4, criamos um conjunto de 6 regras para analisar os resultados da fase de injeção, descrito na Tabela 7.5.

Estas regras nos permitem identificar o comportamento dos Web Services na presença de falhas que caracterizam uma vulnerabilidade encontrada. Dessa forma, classificamos em 4 categorias as respostas das requisições feitas aos 10 Web Services pelo IF WSInject:

- Vulnerabilidade encontrada (VE): detecção de uma vulnerabilidade no Web Service.
- Vulnerabilidade Não Encontrada (VNE): detecção de uma resposta robusta no Web Services.
- Falha de Software (FS): Quando o erro não foi provocado pelo ataque, senão por um erro de software do servidor ou Web Service.
- Inconclusivo: resposta não possível de ser classificado, segundo as regras de ataques bem sucedido da Tabela 7.5.

Dado que fazemos uso de abordagem caixa preta, utilizamos como fontes de informação os *logs* armazenados pelas ferramentas (soapUI e WSInject). Os *logs* contêm as requisições feitas pelo cliente, bem como as respostas enviadas pelo servidor. Descrevemos abaixo os dois cenários utilizados durante a etapa de preparação e um exemplo de cada ataque injetado pelo IF WSInject.

***Cenário A) Ataques contra Web Services:*** realizado o ataque de XML Injection e alterado o parâmetro <ser:username> na Figura 8.9, concluímos que encontramos uma vulnerabilidade. Na linha 7 e 8 da requisição atacamos o Web Service com a injeção de um *script*, descrito na Tabela 8.1, por meio do IF WSInject. Na linha 1 da resposta, o serviço não detectou o ataque e respondeu ao cliente com o código de status HTTP 200 OK.

**Script XML Injection:** isRequest():stringCorrupt("</ser:Username>", "</ser:Username><ser:Username2>hacker</ser:Username2>");

| Requisição | Resposta |
|---|---|
| 1: <soapenv:Envelope xmlns:...">  2: <soapenv:Header>  3: <ser:Header>  4: <!--Optional:-->  5: <ser:Username>admin</ser:Username>  6: !--Optional:-->  7: **<ser:Username>admin</ser:Username>**  8: **<ser:Username2>hacker</ser:Username2>**  9: ...  10: </soapenv:Body>  11: </soapenv:Envelope> | 1: **HTTP/1.1 200 OK // O WS não reconheceu o ataque**  2: Content-Type: text/html; charset=utf-8  3: Content-Length: 31046  4:  5: <soap:Envelope ...>  6: <soap:Body>  7: **<!-- BEGIN: navigation -->**  8: ** **  9: **<div class="xMenu"**  10: **id="ctl00_ctl00_ucNavigation_mnMainNavigation">**  11: …  12: </soap:Body>  13: </soap:Envelope> |

Figura 8.9 Exemplo de ataque XML Injection contra Web Service.

Usando a regra 1 da Tabela 7.5, concluímos que a resposta obtida na Figura 8.9 é classificada como vulnerabilidade encontrada (VE) porque o ataque não foi detectado.

***Cenário B) Ataques contra WS-Security:*** Nesta etapa, também utilizamos o IF WSInject para injetar *Attack Obfuscation* contra o padrão WS-Security, descrito na Figura 8.10. Nas linhas de 9 à 23 da requisição injetamos um *script* cifrado na mensagem SOAP, que cifra o ataque. A linha 1 da resposta descreve que o ataque foi bem sucedido, já que respondeu com o código de status HTTP 500 – *Internal Server Error*. O servidor encontrou uma condição inesperada que o impediu de cumprir a requisição e divulgou informação de rotas de diretórios (*Path*), devido a um erro interno.

**Script 1:** isRequest(): stringCorrupt("Base64Binary\">", "Base64Binary\"><#>N:AES:64<#>
1xxYGhxVbNyR+G4HRsXninGS/CTAAaTq2BuYBq39FqQzWEJ21DAhkiPAwlwTgCDLt...
...AghExOK0MSENKpq+hJvy7z1JYg7OyLI3eqcZbNXvYDNFj9Elw0EM0YI6Yt4jomx4=");

**Requisição**

```
1:  <soapenv:Envelope xmlns:per="...">
2:   <soapenv:Header>
3:    <wsse:Security xmlns:wsse="...">
4:     <wsse:UsernameToken wsu:Id="...">
5:      <wsse:Username>aaa</wsse:Username>
6:      <wsse:Password ...#PasswordText">
7:       aaa
8:      </wsse:Password>
9:      <wsse:Nonce ... Base64Binary">
10:      <#>N:AES:64<#>1xxYGhxVbNyR+G4HRsX
11:         ninGS/CTAAaTq2BuYBq39FqQzWEJ21
12:         DAhkiPAwlwTgCDLtJSegjCOvFQQDie
13:         11NiRWb6E54Yp9FH5p7neBgwVIe7uJ
14:         6OBmjhkl87BqIvIXmQx3VV3TJQtDyx
15:         ...
16:         w0EM0YI6Yt4jomx4=0cIEFegWJ2pE9
17:         Zs7yjNapg==</wsse:Nonce>
18:     <wsu:Created>2012-08-
19: 16T18:16:19.504Z</wsu:Created>
20:     </wsse:UsernameToken>
21:    </wsse:Security>
22:   </soapenv:Header>
23:  <soapenv:Body>
24:   <per:PersonReq>
25:
26: <per:PersonID>marcinv</per:PersonID>
27:   </per:PersonReq>
28:  </soapenv:Body>
29: </soapenv:Envelope>
```

**Resposta**

```
1:  HTTP/1.1 500 Internal Server Error
2:  Cache-Control: private
3:  Content-Length: 2145
4:
5:  <?xml version="1.0" encoding="utf-8"?>
6:   <soap:Envelope xmlns:soap="..." xmlns:xsi="..."
7:   xmlns:xsd="...">
8:    <soap:Body>
9:     <soap:Fault>
10:     <faultcode>soap:Server</faultcode>
11:     <faultstring>
12:      System.Web.Services.Protocols.SoapException:
13:      Server was unable to process request.
14:      System.Data.SqlClient.SqlException: Cannot
15:      insert the value NULL into column 'ApiKey',
16:      ...
17:      at Frontur.Db.DbConnection.Update() in
18:      C:\Projects\Frontur.Db\DbConnection.cs:line
19:      ...
20:      C:\Projects\AppServer\trunk\src\WebserviceAp
21:      i.cs:line 87
22:      String apiKey) in c:\Website\App\AppServer
23:      2.9.6\dev\UserToken.asmx:line 21   --- End
24:      of inner exception stack trace ---
25:     </faultstring>
26:     <detail/>
27:    </soap:Fault>
28:   </soap:Body>
29:  </soap:Envelope>
```

Figura 8.10 Exemplo de log gerado pelo IF WSInject.

Usando a regra 3 da Tabela 7.5, concluímos que a resposta obtida na Figura 8.10 é classificada como vulnerabilidade encontrada porque houve divulgação de rotas de diretórios

(*Path*). Os resultados de injeção de ataques com o IF WSInject são apresentados nas seções 8.3 e 8.4, juntamente com o análise de resultados de ambos cenários.

## 8.3 Ataques contra Web Services

Nesta fase, utilizamos o WSInject para emular e injetar 8 ataques contra 5 Web Services. Os resultados alcançados na Tabela 8.3 foram analisados pelas regras descritas na Tabela 7.5 que verificam a presença de vulnerabilidades nos Web Services.

Tabela 8.3 Resumo de Ataques contra Web Services.

| Ataque | Classificação da Resposta | Resp. | % | WS | WS 1 | WS 2 | WS 3 | WS 4 | WS 5 |
|---|---|---|---|---|---|---|---|---|---|
| **Oversize Payload** | **Total de respostas** | **2.500** | 100,00% | 5 | **500** | **500** | **500** | **500** | **500** |
| | Vul. Não Encontradas | **478** | 19,12% | 3 | 200 | 78 | - | - | 200 |
| | Vul. Encontradas | **2.022** | 80,88% | 5 | 300 | 422 | 500 | 500 | 200 |
| **Coercive Parsing** | **Total de respostas** | **2.500** | 100,00% | 5 | **500** | **500** | **500** | **500** | **500** |
| | Vul. Não Encontradas | **1.224** | 48,96% | 3 | 400 | 324 | - | - | 500 |
| | Vul. Encontradas | **1.276** | 51,04% | 4 | 100 | 176 | 500 | 500 | - |
| **XML Bomb** | **Total de respostas** | **2.500** | 100,00% | 5 | **500** | **500** | **500** | **500** | **500** |
| | Vul. Não Encontradas | **1.231** | 49,24% | 3 | 400 | 331 | - | - | 500 |
| | Vul. Encontradas | **1.269** | 50,76% | 4 | 100 | 169 | 500 | 500 | - |
| **XML Injection** | **Total de respostas** | **2.500** | 100,00% | 5 | **500** | **500** | **500** | **500** | **500** |
| | Vul. Não Encontradas | **235** | 9,40% | 1 | - | 235 | - | - | - |
| | Vul. Encontradas | **2.265** | 90,60% | 5 | 500 | 265 | 500 | 500 | 500 |
| **XPath Injection** | **Total de respostas** | **2.500** | 100,00% | 5 | **500** | **500** | **500** | **500** | **500** |
| | Vul. Não Encontradas | **0** | 0,00% | 0 | - | - | - | - | - |
| | Vul. Encontradas | **2.500** | 100,00% | 2 | 500 | 500 | 500 | 500 | 500 |
| **XSS** | **Total de respostas** | **2.500** | 100,00% | 5 | **500** | **500** | **500** | **500** | **500** |
| | Vul. Não Encontradas | **936** | 37,44% | 3 | 300 | 236 | - | - | 400 |
| | Vul. Encontradas | **1.564** | 62,56% | 5 | 200 | 264 | 500 | 500 | 100 |
| **Fuzzing Scan** | **Total de respostas** | **2.500** | 100,00% | 5 | **500** | **500** | **500** | **500** | **500** |
| | Vul. Não Encontradas | **0** | 0,00% | 0 | - | - | - | - | - |
| | Vul. Encontradas | **2.500** | 100,00% | 5 | 500 | 500 | 500 | 500 | 500 |
| **Malformed XML** | **Total de respostas** | **2.500** | 100,00% | 5 | **500** | **500** | **500** | **500** | **500** |
| | Vul. Não Encontradas | **812** | 32,48% | 3 | 200 | 312 | - | - | 300 |
| | Vul. Encontradas | **1.688** | 67,52% | 5 | 300 | 188 | 500 | 500 | 200 |

Pode-se notar que os 8 ataques puderam ser injetados pelo IF WSInject. A Tabela 8.3 classifica os resultados dos ataques em 2 tipos de resposta: vulnerabilidades encontradas, vulnerabilidades não encontradas. Não se apresentou respostas classificadas como falhas de

software ou respostas inconclusivas. Os 5 Web Services (**WS**) também são classificados individualmente pelo tipo de resposta.

Na Figura 8.11 descrevemos os resultados da Tabela 8.3 em porcentagens. Os Web Services testados têm muitas vulnerabilidades encontradas, especialmente Oversize Payload (80,88%) e XML Injection (90,6%). Em troca, XML Bomb (49,24%) e Coercive Parsing (48.96%) apresentam respostas sem vulnerabilidades encontradas.

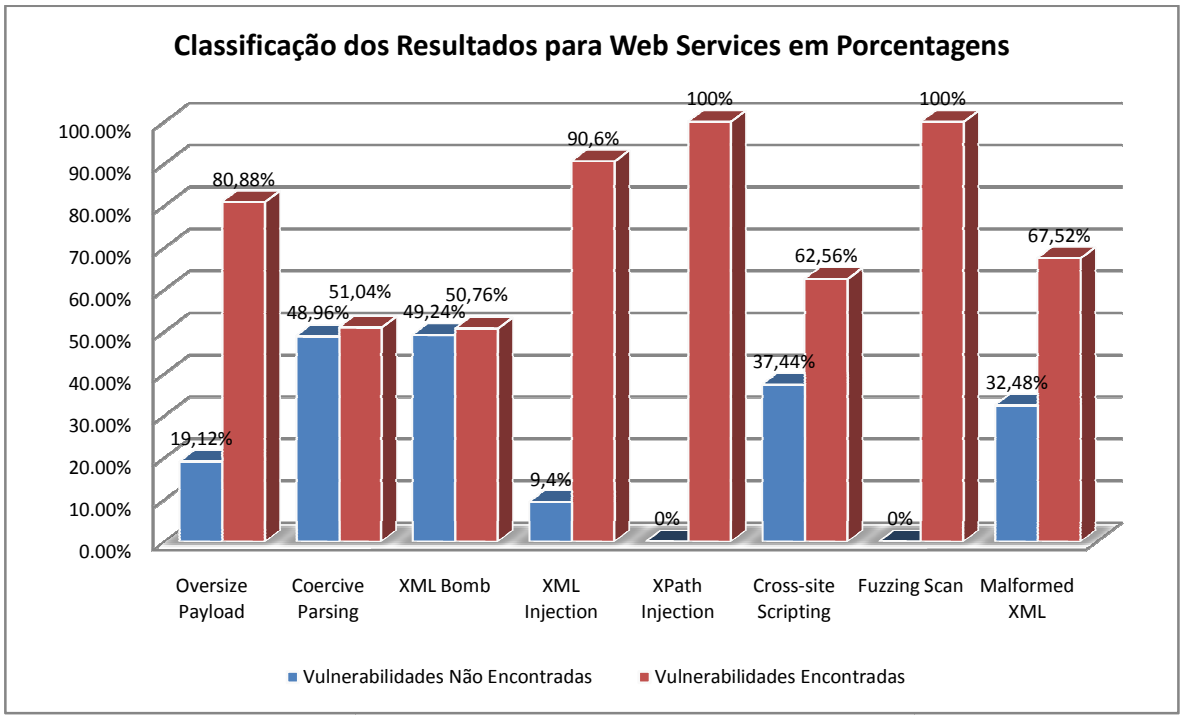



Figura 8.11 Análise dos resultados para Web Services.

Cabe destacar que o IF WSInject obteve maior cobertura de ataques em comparação ao VS soapUI, introduzindo novas vulnerabilidades como Oversize Payload, Coercive Parsing e XML Injection.

## 8.4 Ataques contra WS-Security

Nesta seção apresentamos os resultados da injeção de 10 ataque a 5 Web Services com o padrão WS-Security usando credencias de segurança (Security Tokens) como artefato de **autenticação** e **autorização**. Estes resultados são descritos na Tabela 8.4.

Tabela 8.4 Resumo de Ataques contra WS-Security.

| Ataque | Tipo de Resposta (TR) | Resp. | % | WS | WS 1 | WS 2 | WS 3 | WS 4 | WS 5 |
|---|---|---|---|---|---|---|---|---|---|
| **Oversize Payload** | **Total de Respostas** | **2.500** | 100% | 5 | **500** | **500** | **500** | **500** | **500** |
| | Vul. Não Encontradas | **1.100** | 44% | 4 | 500 | 100 | 300 | 200 | - |
| | Vul. Encontradas | **1.400** | 56% | 4 | - | 400 | 200 | 300 | 500 |
| **Coercive Parsing** | **Total de Respostas** | **2.500** | 100% | 5 | **500** | **500** | **500** | **500** | **500** |
| | Vul. Não Encontradas | **1.400** | 56% | 4 | 100 | 400 | 500 | 400 | - |
| | Vul. Encontradas | **1.100** | 44% | 4 | 400 | 100 | - | 100 | 500 |
| **Oversize Cryptography** | **Total de Respostas** | **2.500** | 100% | 5 | **500** | **500** | **500** | **500** | **500** |
| | Vul. Não Encontradas | **500** | 20% | 1 | - | - | - | 500 | - |
| | Vul. Encontradas | **2.000** | 80% | 4 | 500 | 500 | 500 | - | 500 |
| **Attack Obfuscation** | **Total de Respostas** | **2.500** | 100% | 5 | **500** | **500** | **500** | **500** | **500** |
| | Vul. Não Encontradas | **2.000** | 80% | 4 | 500 | 500 | 500 | 500 | - |
| | Vul. Encontradas | **500** | 20% | 1 | - | - | - | - | 500 |
| **XML Bomb** | **Total de Respostas** | **2.500** | 100% | 5 | **500** | **500** | **500** | **500** | **500** |
| | Vul. Não Encontradas | **1.900** | 76% | 4 | 500 | 400 | 500 | 500 | - |
| | Vul. Encontradas | **600** | 24% | 2 | - | 100 | - | - | 500 |
| **XML Injection** | **Total de Respostas** | **2.500** | 100% | 5 | **500** | **500** | **500** | **500** | **500** |
| | Vul. Não Encontradas | **700** | 28% | 4 | 200 | 200 | 100 | 200 | - |
| | Vul. Encontradas | **1.800** | 72% | 5 | 300 | 300 | 400 | 300 | 500 |
| **XPath Injection** | **Total de Respostas** | **2.500** | 100% | 5 | **500** | **500** | **500** | **500** | **500** |
| | Vul. Não Encontradas | **100** | 4% | 1 | - | 100 | - | - | - |
| | Vul. Encontradas | **1.400** | 96% | 5 | 500 | 400 | 500 | 500 | 500 |
| **XSS** | **Total de Respostas** | **2.500** | 100% | 5 | **500** | **500** | **500** | **500** | **500** |
| | Vul. Não Encontradas | **1.100** | 44% | 4 | 200 | 300 | 300 | 300 | - |
| | Vul. Encontradas | **1.400** | 56% | 5 | 300 | 200 | 200 | 200 | 500 |
| **Fuzzing Scan** | **Total de Respostas** | **2.500** | 100% | 5 | **500** | **500** | **500** | **500** | **500** |
| | Vul. Não Encontradas | **1.000** | 40% | 2 | - | 500 | 500 | - | - |
| | Vul. Encontradas | **1.500** | 60% | 3 | 500 | - | - | 500 | 500 |
| **Malformed XML** | **Total de Respostas** | **2.500** | 100% | 5 | **500** | **500** | **500** | **500** | **500** |
| | Vul. Não Encontradas | **1.400** | 56% | 4 | 400 | 300 | 400 | 300 | - |
| | Vul. Encontradas | **1.100** | 44% | 5 | 100 | 200 | 100 | 200 | 500 |

Na Tabela 8.4 observamos que todos os ataques detectaram vulnerabilidades, ainda com a utilização do padrão WS-Security. Com respeito à Tabela 8,3 de Injeção de Ataques contra Web Services, existe uma diminuição na porcentagem de vulnerabilidades encontradas. Essa diferença ocorre porque os Web Services utilizam credenciais de segurança (Security Tokens), que verificam a autenticidade do cliente além de outros parâmetros. No entanto, não é suficiente para evitar certos tipos de ataques como Oversize Cryptography, onde WS-Security não protege contra este ataque. O restante dos resultados é descrito na Figura 8.12.

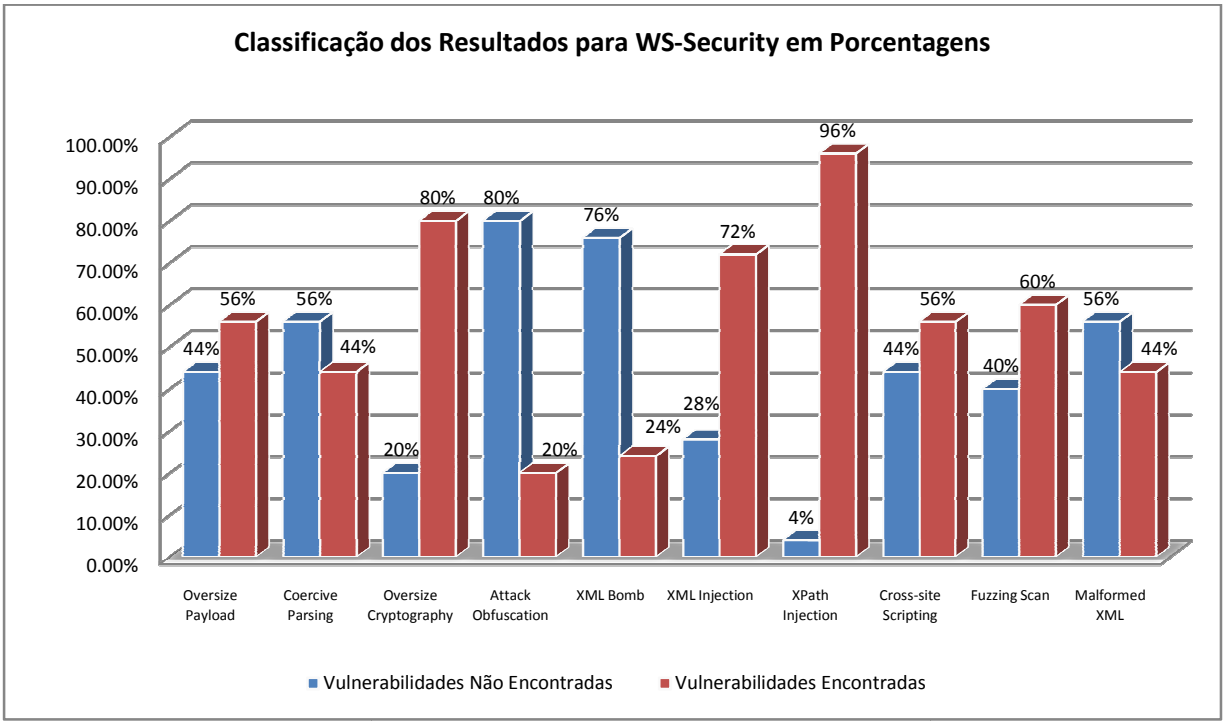



Figura 8.12 Análise dos resultados para WS-Security.

Dentro dos ataques detectados temos Attack Obfuscation (80%), XML Bomb (76%), Malformed XML (56%) e Coervice Parsing (56%). Os ataques não detectados requerem outros mecanismos do segurança, os quais não são parte dos Web Services testados, e.g. XML Encryption, XML Signature.

## 8.5 Análise de Ataques contra Web Services e WS-Security

Neste capítulo foram apresentados vários experimentos demonstrando o uso prático da abordagem de testes de segurança e do WSInject para avaliar a robustez dos Web Services. Além de mostrar os resultado dos experimentos, se detalhou a metodologia adotada assim como as ferramentas e critérios utilizados.

Entre outros objetivos, procurou-se fazer um paralelo entre os resultados alcançados usando Web Services (Seção 8.3) e o padrão WS-Security (Seção 8.4) para analisar o beneficio da aplicação do padrão aos serviços testados. Por exemplo, na Figura 8.13 observamos que o uso deste padrão ajuda a diminuir o número de vulnerabilidades encontradas para todos os ataques injetados nos 10 Web Services não homogêneos e selecionados pseudo-aleatóriamente de 22.272 serviços (Seção 7.2).

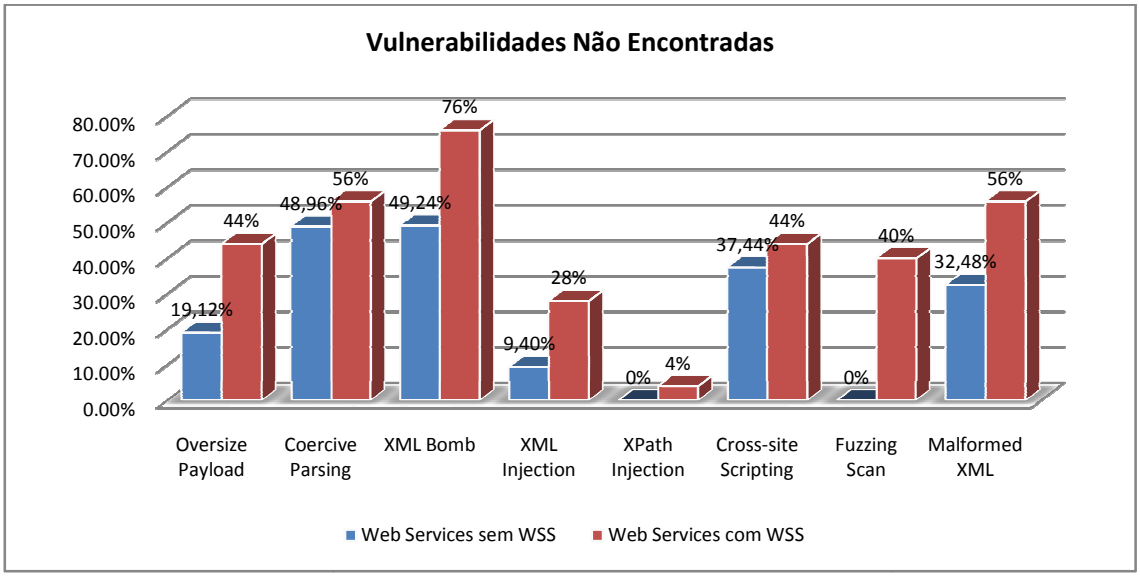



Figura 8.13 Comparação das duas abordagens para Vulnerabilidades Não Encontradas.

Outro objetivo é encontrar a maior quantidade de vulnerabilidades nos Web Services. Na Figura 8.14, comparamos o desempenho dos serviços com WS-Security contra os Web Services sem segurança, reduzindo o número de vulnerabilidades encontradas (VE) para todos os ataques testados. Destacamos uma redução significativa para os ataques de XML Injection (de 90,60% para 72%), Malformed XML (de 63,52% para 44%), Oversize Payload (de 80,88% para 56%) e XML Bomb (de 50,76% para 24%).

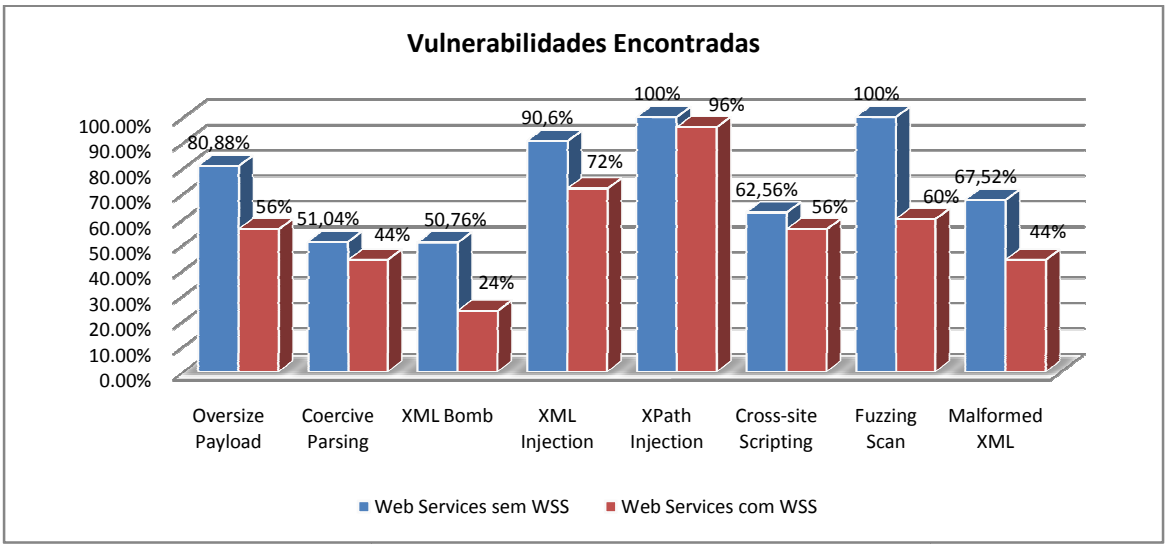



Figura 8.14 Comparação das duas abordagens para ataques bem sucedidos AS.

Esta redução de vulnerabilidades encontradas é devido ao uso de credencias de segurança, mais conhecidos como Security Tokens, que comprovam a identidade do emissor e permite acesso aos serviços do provedor, sempre e quando o provedor tenha certeza da identidade do cliente. Caso não exista uma autenticação correta, o servidor retorna um *HTTP 400 – Bad Request* ou *HTTP 500 – Internal Server Error* descrevendo um erro de Sintaxes na resposta.

## 8.6 Considerações Finais

O foco desse trabalho foi obter: **1)** armar um banco de dados de ataques contra Web Services; **2)** desenvolver uma árvores de ataques para selecionar os tipos de ataques a ser injetados; **3)** utilizar uma técnica de modelagem de ataques para sistematiza a construção e manutenção dos cenários de ataques para que sejam usados por testadores; 4**)** usar um injetor de falhas para emular diversos tipos de ataques e ter uma maior cobertura dos mesmos; **5)** usar as regras da Tabela 7.5 para classificar os resultados em vulnerabilidades encontradas, vulnerabilidades não encontradas, falhas de software e respostas inconclusivas.

Também podemos concluir que a utilização de outras técnicas e padrões de segurança para Web Services como XML Encryption ou XML Signature, descritos na Seção 3.3, podem reduzir as vulnerabilidades encontradas e melhorar a robustez contra estes tipos de ataques.

# Capítulo 9. Conclusões e Recomendações

A proposta da utilização de uma metodologia para analisar a robustez de Web Services por testes de segurança através do injector de falhas WSInject nos permitiu emular diversos tipos de ataques, melhorando a cobertura dos mesmos e conseguindo um maior numero de vulnerabilidades encontradas nos Web Services. Já a redução dos falsos positivos e negativos se deve à criação das regras (ver Tabela 7.5) e sua aplicação aos resultados para melhorar a detecção de ataques, mas ainda não é suficiente para garantir proteção contra estes.

A seguir descrevemos os resultados e contribuições, como os trabalhos futuros do presente trabalho.

## 9.1 Resultados e Contribuições

O presente trabalho testa a robustez dos Web Services através de testes de segurança, emulando diversos ataques pelo injetor de falhas (IF) WSInject para dois cenários: a) ataques contra Web Services e b) ataques contra WS-Security, descritos no Capítulo 8.

Coletamos um conjunto de ataques de diversas fontes, com o objetivo de concretizar um banco de dados de ataques. Os ataques foram selecionados, de forma automatizada, pela ferramenta SecureITree V3.4 através de uma árvore de ataques para Web Services. Esta árvore foi construída e estruturada de acordo com os passos propostos na Seção 5.3. As folhas da árvore de ataques da Seção 6.4 possuem três atributos associados, que são valores lógicos que representam atributos de um ataque e estão compostos por: **<capacidade do atacante, arquitetura de teste, dispor de um Web Services**>.

Um novo atributo foi introduzido na Seção 6.5, chamado **"WS-Security protege ao Web Service"** com os seguintes resultados: Possível (Protege) e Impossível **(**Não Protege), para classificar os ataques em dois cenários: ***Ataques que são rejeitados por WS-Security*** e ***Ataques que não são rejeitados por WS-Security***. Utilizando a abordagem em [24], geramos um conjunto de scripts de ataque denominados SAEs (Scripts de Ataque Executáveis) para emular diversos ataque com o IF WSInject.

Para nosso experimento no Capítulo 7, selecionamos pseudo-aleatóriamente uma amostra de 69 Web Services de 22.272 serviços do banco de dados Seekda. Esta amostra nos permite obter resultados com 90% de precisão e um erro de $\pm 10\%$; o que permite comprovar quantos

Web Services têm vulnerabilidades e quais são os principais ataques contra Web Services. Na Seção 7.2 configuramos o add-on Security Testing do VS soapUI para enviar requisições com ataques aos 69 Web Services. Geramos um conjunto de regras (Seção 7.4) que nos permite determinar quando encontramos uma vulnerabilidade. Na Seção 7.5, aplicamos estas regras aos resultados obtidos na Seção 7.3, que permite constatar a grande quantidade de falsos positivos e falsos negativos, produzidos pelo vulnerability scanner soapUI.

Na Seção 8.2, emulamos ataques de injeção (*injection attacks*) e negação de serviço (*denial-of-services attacks - DOS*) com o IF WSInject. Estes ataques atingem os atributos de integridade e disponibilidade respectivamente. Através da análise dos resultados, da Seção 8.5, descrevemos a eficiência da técnica de injeção de falhas comparadas com vulnerabilities scanners, obtendo maior cobertura de ataques e a possibilidade de combinar ou criar novos ataques. Entre as contribuições da dissertação, podemos citar:

- Utilização de árvore de ataques para geração de cenários de ataque de forma sistemática para propósitos de injeção de ataques.
- Geração de um banco de dados para ataques a Web Services, contendo propriedades ou atributos violados por ataques, nível do ataque injetado (mensagem, processo ou banco de dados), impacto do ataque (baixo, médio, alto) e proteção pelo padrão WS-Security.
- A abordagem injeta ataques conhecidos e permite a criação de variações destes ataques (novos ataques), de forma localizada, variando os parâmetros do ataque original, que podem ser executados usando o atual injetor de falhas WSInject.
- A abordagem é capaz de executar diferentes classes de ataques de acordo com a capacidade do atacante definida e a ferramenta de injeção de falhas usada, como executar as ações **interceptar** ou **atrasar**, o que não é possível usando abordagens baseadas em Fuzz e Testes de Penetração, que se limitam à ação **corromper**.
- Emulamos 10 ataques usando o IF WSInject contra 5 Web Services com WS-Security (Security Tokens) e 5 Web Services sem WS-Security.

## 9.2 Trabalhos Futuros

A seguir listamos algumas sugestões para o desenvolvimento de trabalhos futuros.

- Analisar outros elementos de segurança do padrão WS-Security, como XML-Encryption ou XML-Signature, como estudo de caso para realizar Testes de Segurança.
- Usar a abordagem com um injetor que trabalhe no nível de camada de aplicação e transporte, para analisar a robustez de diversos protocolos de segurança.
- Usar as informações coletadas durante a interceptação das mensagens na injeção dos ataques para uma análise mais refinada do comportamento do padrão alvo.
- Comparar Web Services homogêneos que usem diversos padrões de segurança e injetar diversos ataques com o WSInject.
- Combinar diferentes metodologias de testes para garantir melhor análise de vulnerabilidades para protocolos de segurança.
- Sistematizar a criação de novos ataques a partir de ataques conhecidos já reportados, i.e, definir uma abordagem para obter de forma sistemática variações de ataques bem sucedidos que caracterizem novos ataques.

# Apêndice A: Ataques contra Web Services e WS-Security

Os seguintes ataques estão organizados usando a classificação de [23].

| Classificação de Ataques | Ataques | Propie-dades | Nível de ataque | Tam. | Impacto | WSS protege o Web Services? |
|---|---|---|---|---|---|---|
| ***V.1. Denial of Service Attacks*** | *A.01. Replay Attack* | *D* | *Mensagem* | *1* | *Low* | *Não Protege* |
| | *A.02. Oversize Payload* | *D* | *Mensagem* | *1* | *Medium* | *Não Protege* |
| | *A.03. Coercive Parsing (Recursive Payloads)* | *D* | *Mensagem* | *1* | *Medium* | *Não Protege* |
| | *A.04. Oversize Cryptography* | *D* | *Mensagem* | *1* | *Medium* | *Não Protege* |
| | *A.05. Attack Obfuscation* | *D* | *Mensagem* | *1* | *Medium* | *Não Protege* |
| | *A.06. XML Bomb* | *D* | *Mensagem* | *1* | *Medium* | *Não Protege* |
| | *A.07. Unvalidated Redirects and Forwards* | *D* | *Mensagem* | *1* | *Medium* | *Não Protege* |
| | *A.08. Attacks through SOAP Attachment* | *D* | *Mensagem* | *1* | *Medium* | *Não Protege* |
| | *A.09. Schema Poisoning* | *D* | *Mensagem* | *1* | *Medium* | *Não Protege* |
| ***V.2. Brute Force Attacks*** | *A.10. Insecure Cryptographic Storage* | *D* | *Banco de Dados* | *1* | *High* | *Não Protege* |
| | *A.11. Broken Authentication and Session Management* | *CA* | *Mensagem* | *1* | *High* | *Protege* |
| ***V.3. Spoofing Attacks*** | *A.12. SOAPAction* | *CA* | *Mensagem* | *1* | *Medium* | *Protege* |
| | *A.13. WSDL Scanning* | *C* | *Mensagem* | *1* | *Medium* | *Protege* |
| | *A.14. Insufficient Transport Layer Protection* | *C* | *Processo* | *1* | *Medium* | *Protege* |
| | *A.15. WS-addressing Spoofing* | *D* | *Mensagem* | *2* | *Medium* | *Protege* |
| | *A.16. Middleware Hijacking* | *D* | *Processo* | *2+* | *Medium* | *Não Protege* |
| | *A.17. Metadata Spoofing* | *I* | *Mensagem* | *1+* | *Medium* | *Não Protege* |
| | *A.18. Security misconfiguration* | *CA* | *Todos os níveis* | *1+* | *Medium* | *Protege* |
| | *A.19. Unauthorized access* | *CA* | *Mensagem* | *1* | *Medium* | *Protege* |
| | *A.20. Routing Detours* | *CA* | *Mensagem* | *1+* | *Medium* | *Protege* |
| | *A.21. Attack on WS-Security* | *I* | *Mensagem* | *1+* | *Medium* | *Protege* |
| | *A.22. Attack on WS-Trust, WS-SecureConversation* | *CA* | *Mensagem* | *2+* | *Medium* | *Protege* |
| | *A.23. Maliciuos Content* | *D* | *Mensagem* | *1* | *Medium* | *Não Protege* |
| ***V.4. Flooding Attacks*** | *A.24. Instantiation Flooding* | *D* | *Processo* | *1* | *High* | *Não Protege* |
| | *A.25. Indirect Flooding* | *D* | *Processo* | *2+* | *High* | *Não Protege* |
| | *A.26. BPEL State Deviation* | *D* | *Processo* | *1* | *High* | *Não Protege* |
| ***V.5. Injection Attacks*** | *A.27. XML Injection* | *I* | *Mensagem* | *1* | *High* | *Protege* |
| | *A.28. SQL Injection* | *I* | *Mensagem* | *1* | *High* | *Protege* |
| | *A.29. XPath Injection* | *I* | *Mensagem* | *1* | *High* | *Protege* |
| | *A.30. Cross site Scripting (XSS)* | *I* | *Mensagem* | *1+* | *High* | *Protege* |
| | *A.31. Cross Site Request Forgery* | *CA* | *Mensagem* | *2+* | *Medium* | *Protege* |

| | | | | | | |
|---|---|---|---|---|---|---|
| | *A.32. Fuzzing Scan* | *I* | *Mensagem* | *1* | *High* | *Protege* |
| | *A.33. Invalid Types* | *I* | *Mensagem* | *1* | *Medium* | *Protege* |
| | *A.34. A. Parameter Tampering* | *I* | *Mensagem* | *1* | *Medium* | *Protege* |
| | *A.35. Malformed XML* | *I* | *Mensagem* | *1* | *High* | *Protege* |
| | *A.36. Frankenstein Message: Modify Timestamp* | *I* | *Mensagem* | *1+* | *Medium* | *Protege* |

Tabela A.1 Ataques a Web Services.

Organizamos a Tabela A.1 de vulnerabilidades para Web Services pelos seguintes critérios: **i)** informações gerais de vulnerabilidades fazendo referências; **ii)** métricas de "exploraridade" e de impacto que descrevem características inerentes de uma vulnerabilidade: nível de camada do ataque (no nível de **message**, nível de **processo** da mensagem, ou no nível do **banco de dados**), o valor de **tamanho** que indica o "número habitual ou mínimo" para reproduzir o ataque, **propriedades de segurança violadas** (confidencialidade (**C**), integridade (**I**), disponibilidade (**D**), controles de acesso (**CA**) composto por autorização e autenticação); **iii)** métricas que descrevem o efeito da vulnerabilidade dentro do ambiente de uma organização: potencial das perdas, porcentuais de sistemas vulneráveis e nível de requisitos impactados que classificam aos ataques em três níveis (**baixa**, **média** e **alta**); iv) métrica que descreve se o uso de WS-Security protege ao Web Service do ataque (**Protege**, **Não Protege**).

# Referências Bibliográficas


[1] ABNT NBR ISO/IEC 27001: Tecnologia da informação - Técnicas de segurança - Sistemas de Gestão de Segurança da Informação – Requisitos [relatório técnico]. Associação Brasileira de Normas Técnicas. Rio de Janeiro, Brasil – RJ: ABNT; 31 Mar 2006.

[2] Haas H, Brown A. Web Services Glossary [relatório técnico]. W3C Web Services Architecture Working Group. W3C Working Group: 11 Fev 2002 [acesso em 11 Aug 2011]. Disponível em: http://www.w3.org/TR/ws-gloss/

[3] Siddavatam I, Gadge J. Comprehensive Test Mechanism to Detect Attack on Web Services. In: Proceedings of the 16th IEEE International Conference on Networks, 2008. ICON'08. IEEE Computer Society Press; Nova Delhi, India, 12-14 Dec. 2008.

[4] Holgersson J, Soderstrom E. Web Service Security-Vulnerabilities and Threats Within the Context of WS-Security. In: Proceedings of the 4th Conference on Standardization and Innovation in Information Technology. SIIT 2005. Geneva: ITU; Sep 2005.

[5] Morais A, Martins E. Injeção de Ataques Baseados em Modelo para Teste de Protocolos de Segurança. Dissertação (Mestrado em Ciências da Computação). Instituto de Computação, Universidade Estadual de Campinas; 15 Mai 2009.

[6] soapUI [software]. Version 4.5. Eviware. soapUI; the Web Services Testing tool – Security Testing Tool [acesso 20 May 2012]. Disponível em: http: //www.soapui.org

[7] Valenti AW, Maja WY, Martins E, Bessayah F, Cavalli A. WSInject: A Fault Injection Tool for Web Services [relatório técnico]. Instituto de Computação, Universidade Estadual de Campinas. Campinas, Brazil, July 2010.

[8] Laranjeiro N, Viera M. Testing Web Services for Robustness: A Tool Demo. In: Proceedings of the 12th European Workshop on Dependable Computing. EWDC 2009. Toulouse, França: Mai 2009. Disponível em: http://hal.archives-ouvertes.fr/hal-00381072

[9] Quin L. Extensible Markup Language (XML). XML Activity Lead. W3C Working Group [acesso em 11 Aug 2011]. Disponível em: http://www.w3.org/XML

[10] Marques, Victor; Cardozo, Eliri, "Uma Arquitetura Orientada a Serviços para Desenvolvimento, Gerenciamento e Instalação de Serviços de Rede", Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação. 2006.

[11] Moorsel AV, Bondavalli A, Pinter G, Madeira H, Majzik I, Durães J, et al. Assessing, Measuring, and Benchmarking Resilience [relatório técnico ]. AMBER – State of the Art. D2.2 ,University of Newcastle Upon Tyne. 30 Jun 2009. Disponível em: http://www.amber-project.eu

[12] Fonseca J, Vieira M, Madeira H. Testing and Comparing Web Vulnerability Scanning Tools for SQL Injection and XSS Attacks. In: Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing, PRDC 2007. IEEE Computer Society; Melbourne, Australia, 17-19 Dec 2007.

[13] Laprie J, Randell B. Basic Concepts and Taxonomy of Dependable and Secure Computing. In: IEEE Transaction on Dependable and Secure Computing. Volume 1 Edição 1. Jan 2004.

[14] Lopez MJL. Criptografía y Seguridad en Computadoras. Universidad de Jaén. 4ta ed. [acesso em 11 Nov 2011]. Universidad de Jaén, Mar. 2010. Disponível em: http://wwwdi.ujaen.es/~mlucena/wiki/pmwiki.php

[15] Weber TS, Tolerância a falhas: conceitos e exemplos. Intech Brasil, São Paulo, Volume 52, 2003.

[16] Cachin C, Camenisch J, Dacier M, Deswarte Y, Dobson J, Horne D, et al. MALICIOUS - and Accidental-Fault Tolerance in Internet Applications: Reference Model and Use Cases [relatório técnico]. MAFTIA. University of Newcastle Upon Tyne LAAS report no. 00280, Project IST-1999-11583; Aug 2000.

[17] Ladan MI. Web services: Security Challenges. In: Proceedings of the World Congress on Internet Security, 2011. WorldCIS'11. IEEE Press; Londres, Reino Unido, 21-23 Feb. 2011.

[18] Vieira M, Antunes N, Madeira H. Using Web Security Scanners to Detect Vulnerabilities in Web Services. In: Proceedings of the IEEE/IFIP International Conference on Dependable Systems & Networks. DSN '09. IEEE Computer Society; Lisbon, Porgugal, 2009.

[19] Williams J, Wichers D. OWASP Top 10 – 2010. OWASP Foundation [acesso em 11 Aug 2011]. Disponível em: https://www.owasp.org/index.php/Top_10_2010

[20] Meiko J, Nils G, Ralph H. A Survey of Attacks on Web Services. Computer Science - Research and Development, 01 Nov 2009. Springer Berlin, Heidelberg; ISSN: 1865-2034, volume 24, Edição 4. 2009.

[21] Meiko J, Nils G, Ralph H, Norbert L. SOA and Web Services: New Technologies, New Standards - New Attacks. In: Proceedings of the Fifth European Conference on Web Services, 2007. ECOWS '07. IEEE Computer Society Press; Halle, Germany: 26-28 Nov. 2007.

[22] Ghourabi A, Abbes T, Bouhoula A. Experimental Analysis of Attacks Against Web Services and Countermeasures. In: Proceedings of the 12th International Conference on Information Integration and Web-based Applications & Services, 2010. iiWAS '10. ACM, Paris, França, 8-10 Nov 2010.

[23] Rodrigues D, Estrella JC, Branco KRLJC, Vieira M. Engineering Secure Web Services. In: Performance and Dependability in Service Computing: Concepts, Techniques and Research Directions. IGI Global. Jul 2011.

[24] Lindstrom P. Attacking and Defending Web Service [relatório técnico]. 2004. Spire Security, LLC. Malvern, USA [acesso em Aug 2011]. Disponível em: http://www.forumsys.com/resources/resources/whitepapers/Attacking_and_Defending_WS.pdf

[25] Patel V, Mohandas R, Pais AR. Attacks on Web Services and Mitigation Schemes. In: Proceedings of the International Conference on Security and Cryptography, 2010. SECRYPT'10. Springer; Atenas, Grécia, 26-28 July 2010.

[26] Loh YS, Yau WC, Wong CT, Ho WC. Design and Implementation of an XML Firewall. In: Proceedings of the International Conference on Computational Intelligence and Security, 2006. IEEE Press: Guangzhou, China, 3-6 Nov. 2006.

[27] Della-Libera G, Dixon B, Farrel J, Garg P, Hondo M, Kaler C, et al. Security in a Web Services World A Proposed Architecture and Roadmap [relatório técnico]. IBM Corp. - Microsoft Corp; 7 Avr 2002 [acesso em 11 Aug 2011]. Disponível em: http://msdn.microsoft.com/en-us/library/ms977312.aspx

[28] Lawrence K, Kaler C, Nadalin A, Monzillo R, Hallam-Baker P. Web Services Security: SOAP Message Security 1.1 - WS-Security 2004. OASIS Standard Specification. 11 Feb

2006. [acesso em 11 Aug 2011]. Disponível em: Aug/2011. Disponível em: www.oasis-open.org/committees/wss/

[29] Eastlake D, Reagle J, Imamura T, Dillaway B, Simon E. XML Encryption Syntax and Processing. W3C Recommendation: 10 Dec 2002 [acesso em 11 Aug 2011]. Disponível em: http://www.w3.org/TR/xmlenc-core

[30] Eastlake D, Reagle J, Solo D, Hirsch F, Roessler T, Bartel M, et al. XML Signature Syntax and Processing. 2nd ed. W3C Recommendation: 10 Jun 2008. [acesso 11 Aug 2011]. Disponível em: http://www.w3.org/TR/xmldsig-core

[31] Lawrence K, Kaler C, Nadalin A, Monzillo R, Hallam-Baker P. Web Services Security: UsernameToken Profile 1.1. OASIS Standard Specification: 01 Feb 2006. [acesso em 11 Aug 2011] Disponível em: www.oasis-open.org/committees/wss

[32] Hartman B, Flinn D, Beznosov K, Kawamoto S. Mastering Web Services Security. Wiley Publishing. ISBN-13:978-0471267164. Indianapolis-IN. USA. 2003.

[33] Schneier B, Attack Trees. Modeling Security Threats. Dr. Dobb's Journal: Dec. 1999 [acesso 11 Aug 2011]. Disponível em: http://www.schneier.com/paper-attacktrees-ddj-ft.html

[34] McDermott JP, Attack Net Penetration Testing. In: Proceedings of the 2000 Workshop on New Security Paradigms. NSPW '00. ACM SIGSAC, ACM Press, Cork, Ireland, Feb. 2001.

[35] Hussein M, Zulkernine M. UMLintr: a UML Profile for Specifying Intrusions. In: Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems. ECBS 2006. IEEE Computer Society Press; Potsdam, Germany, 27-30 Mar. 2006.

[36] Moore AP, Ellison RJ, Linger RC. Attack Modeling for Information Security and Survivability [relatório técnico]. CMU/SEI-2001-TN-001, 2001.

[37] SecureITree [software]. Version 3.4. Calgary-AL, Canada. Amenaza Technologies Limited [acesso em 12 Avr 2011]. Disponível em: http://www.amenaza.com

[38] Steffan J, Schumacher M. Collaborative Attack Modeling. In: Proceedings of the 2002 ACM symposium on Applied computing. SAC '02. ACM Press; Madrid, Espanha, Mar. 2002.

[39] Miller BP, Koski D, Pheow C, Maganty LV, Murthy R, Natarajan A, et al. Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services [relatório técnico]. In: Computer Sciences Technical Report #1268, University of Wisconsin-Madison, Avr 1995.

[40] Raul G. Case study: Experiences on SQL language fuzz testing. In: Proceedings of the Second International Workshop on Testing Database Systems. DBTest 09. ACM Press; Providence-RI, USA, 29 Jun – 02 Jul 2009.

[41] Cristian F, Aghili H, Strong R, Volev D. Atomic Broadcast: From Simple Message Diffusion to Byzabtube Agreement. In: Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing. IEEE Computer Society Press, Pasadena-CA, USA, 27-30 Jun 1995.

[42] Carreira JV, Costa D, Silva JG. Fault Injection Spot-Checks Computer System Dependability. Spectrum. IEEE. Volume 36, Edição 8, Aug 1999.

[43] Hsueh MC, Tsai TK, Iyer RK. Fault Injection Techniques and Tools. IEEE Computer Society Press. Computer; Volumen 30, Edição 4: Apr. 1997.

[44] Arlat J, Aguera M, Amat L, Crouzet Y, Fabre JC, Laprie JC, et al. Fault Injection for Dependability Validation: a Methodology and some Applications. In: the IEEE Transactions on Software Engineering; volume 16, Edição 2; Feb 1990.

[45] Koopman P, DeVale J. The Exception Handling Effectiveness of POSIX Operating Systems. In: IEEE Transactions on Software Engineering. volume 26, edição 9. Sep 2000.

[46] de Melo ACV, Silveira P. Improving Data Perturbation Testing Techniques for Web Services. In: the International Journal on Information Sciences. February 2011.

[47] Myers GJ. Sandler C, Badgett T. 2011. The Art of Software Testing. 3rd ed. Wiley Publishing. New Jersey, USA.

[48] Valenti AW, Martins E. Testes de Robustez em Web Services por Meio de Injeção de Falhas. Dissertação (Mestrado em Ciências da Computação) – Instituto de Computação, Universidade Estadual de Campinas. 29 07 2011.

[49] Canfora G, Penta M. Service-Oriented Architectures Testing: A Survey. In Software Engineering, Springer-Verlag, Berlin, Heidelberg, 2009.

[50] Meucci M (editor). The OWASP Testing Guide v3. OWASP Foundation. 16 December 2008 [acesso em 11 Aug 2011]. Disponível em: https://www.owasp.org/images/5/56/OWASP_Testing_Guide_v3.pdf

[51] Zhang J, Xu D. A Mobile Agent-Supported Web Services Testing Platform. In: Proceedings of the IEEE/IFIP International Conference on Embedded and Ubiquitous Computing 2008. EUC '08. Volume 2. IEEE Computer Society Press, Shanghai, China, 17-20 Dec 2008.

[52] Zhou L, Ping J, Xiao H, Wang Z, GeguangPu, Ding Z. Automatically Testing Web Services Choreography with Assertions. In: Proceedings of the 12th international Conference on Formal Engineering Methods and Software Engineering. ICFEM'10. Springer-Verlag; Berlin, Heidelberg: 2010.

[53] Rogan D. OWASP WebScarabLite [software]. Version 20070504-1631. Open Web Application Security Project 2011. [acesso 11 Aug 2011]. Disponível em: http://www.owasp.org/software/webscarab.html

[54] Laranjeiro N, Canelas S, Vieira M. wsrbench: An On-Line Tool for Robustness Benchmarking. In: Proceedings of the IEEE International Conference on Services Computing. 2008.SCC '08. Honolulu, Hawaii, USA; volume 2: 7-11 July/2008.

[55] GraziaFugini M, Pernici B, Ramoni F. Quality Analysis of Composed Services through Fault Injection. In: Proceedings of the 2007 International Conference on Business process management. Springer; Berlin, Heidelberg: 3 Jul 2009.

[56] Dao TB, Shibayama E. Idea: Automatic Security Testing for Web Applications. In: Proceedings of the 1st International Symposium on Engineering Secure Software and Systems. ESSoS '09. Springer-Verlag; Berlin, Heidelberg: 2009.

[57] Zhao G, Zheng W, Zhao J; Chen H. An Heuristic Method for Web-Service Program Security Testing. In: Proceedings of the 2009 Fourth ChinaGrid Annual Conference. CHINAGRID '09. IEEE Computer Society Press; Yantai, China, 21-22 Aug 2009.

[58] Cao TD, Phan-Quang TT; Felix P, Castanet R. Automated Runtime Verification for Web Services. In: Proceedings of the 2010 IEEE International Conference on Web Services. ICWS. IEEE Computer Society Press; Miami, Florida, 5-10 July 2010.

[59] Seo J, Kim HS, Cho S, Cha S. Web Server Attack Categorization Based on Root Causes and their Locations. In: Proceedings of the International Conference on Information Technology: Coding and Computing. ITCC 2004. IEEE Computer Society Press; Las Vegas-NE, USA, 5-7 April 2004.

[60] Bartolini C, Bertolino A, Marchetti E, Polini A. WS-TAXI: A WSDL-based Testing Tool for Web Services. In: Proceedings of the International Conference on Software Testing Verification and Validation, 2009. ICST '09. IEEE Computer Society; Denver, Colorado, 1-4 April 2009.

[61] Morais A, Martins E, Cavalli A, Jimenez W. Security Protocol Testing Using Attack Trees. In: Proceedings of the International Conference on Computational Science and Engineering, 2009.CSE '09. IEEE Computer Society Press; São Paulo, Brasil, 29-31 Aug. 2009.

[62] Martins E, Morais A, Cavalli A. Generating Attack Scenarios for the Validation of Security Protocol Implementations. In: Proceedings of the II Brazilian Workshop on Systematic and Automated Software Testing. SBC; Campinas-SP, Brasil, 2008.

[63] Laranjeiro N, Vieira M, Madeira H. Improving Web Services Robustness. In: Proceedings of the IEEE International Conference on Web Services. ICWS 2009. ; IEEE Computer Society; Los Angeles, USA, 2009.

[64] Antunes N, Vieira M. Benchmarking Vulnerability Detection Tools for Web Services. In: Proceedings of the 2010 IEEE International Conference on Web Services. ICWS. IEEE Computer Society Press; Miami, Florida, 5-10 July 2010.

[65] Antunes N, Vieira M. Detecting SQL Injection Vulnerabilities in Web Services. In: Proceedings of the Fourth Latin-American Symposium on Dependable Computing, 2009. LADC '09. IEEE Computer Society Press; João Pessoa-PB, Brasil, 1-4 Sept. 2009.

[66] Antunes N, Vieira M. Comparing the Effectiveness of Penetration Testing and Static Code Analysis on the Detection of SQL Injection Vulnerabilities in Web Services. In: Proceedings of the 15th IEEE Pacific Rim International Symposium on Dependable Computing, 2009. PRDC '09. IEEE Computer Society Press; Shangai, China, 16-18 Nov 2009.

[67] Laranjeiro N, Vieira M, Madeira H. Protecting Database Centric Web Services against SQL/XPath Injection Attacks. In: Proceedings of the 20th International Conference on Database and Expert Systems Applications, 2009. DEXA '09. Springer-Verlag, Berlin, Heidelberg, 2009.

[68] Vieira M, Laranjeiro N. Comparing Web Services Performance and Recovery in the Presence of Faults. In: Proceedings of the IEEE International Conference on Web Services, 2007. ICWS 2007. IEEE Computer Society Press; Beijing, China, 9-13 July 2007.

[69] Vieira M, Laranjeiro N, Madeira H. Assessing Robustness of Web-Services Infrastructures. In: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2007. DSN '07. IEEE Computer Society Press; Edinburgh, UK, 25-28 July 2007.

[70] Dolev D, Yao A. On the Security of Public Key Protocols. In: IEEE Transactions on Information Theory. IEEE Computer Society Press: Mar 1983.

[71] Edge KS, Raines RA, Baldwin RO, Grimaila MA, Bennington R. The Use of Attack and Protection Trees to Analyze Security for an Online Banking System. In: Proceedings of the Hawaii International Conference on System Sciences-HICSS-40. IEEE Computer Society Press; Kauai, Hawaii, 3-6 Jan 2007.

[72] Paton N, Díaz O, Williams MH, Campin J, Dinn A, Jaime A. Dimensions of Active Behaviour. In: Proceedings of the 1st International Workshop on Rules in Database Systemsm 1993. RIDS'93. Springer-Verlag; Edinburgh, Scotland, September 1993.

[73] Drebes RJ, Jacques-Silva G, da Trindade JF, Weber TS. A Kernel-based Communication Fault Injector for Dependability Testing of Distributed Systems. In: Proceedings of Haifa Verification Conference. Haifa, Israel, 2005

[74] Maja WY, Martins E. Teste de Robustez de uma Infra-Estrutura Confiável para Arquiteturas Baseadas em Serviços Web. Dissertação (Mestrado em Ciências da Computação) – Instituto de Computação, Universidade Estadual de Campinas. 21 06 2011.

[75] Lakatos EM, Marconi MA. Metodologia científica. São Paulo. Atlas, 1991.

[76] Kohlert D, Arun G. The Java API for XML-Based Web Services (JAX-WS) 2.1 (relatório ténico). Mai 2007.

[77] Powers, DMW. Evaluation: from Precision, Recall and F-Measure to Roc, Informedness, Markedness & Correlation. Jounal of Machine Learning Technologies. [acesso 08 Sep 2012]. ISSN: 2229-3981 & ISSN: 2229-399X, Volume 2, Issue 1, 2011, pp-37-6. Disponível em: http://www.bioinfo.in/uploadfiles/13031311552_1_1_JMLT.pdf