

FUNDAÇÃO DE ENSINO “EURÍPIDES SOARES DA ROCHA”  
CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA – UNIVEM  
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

**HELDER JEFFERSON FERREIRA DA LUZ**

**ANÁLISE DE VULNERABILIDADES EM JAVA WEB APPLICATIONS**

MARÍLIA  
2011

HELDER JEFFERSON FERREIRA DA LUZ

## **ANÁLISE DE VULNERABILIDADES EM JAVA WEB APPLICATIONS**

Trabalho de Conclusão de Curso apoiado pela FAPESP, número do processo: 2010/07828-6, apresentado ao Curso de Bacharelado em Ciência da Computação da Fundação de Ensino “Eurípides Soares da Rocha”, mantenedora do Centro Universitário Eurípides de Marília – UNIVEM, como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador:  
Prof. Dr. Fábio Dacêncio Pereira

MARÍLIA  
2011



CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA  
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

**TRABALHO DE CONCLUSÃO DE CURSO – AVALIAÇÃO FINAL**

---

Helder Jefferson Ferreira da Luz

**ANÁLISE DA VULNERABILIDADE EM JAVA WEB APPLICATIONS**

Banca examinadora da monografia apresentada ao Curso de Bacharelado em Ciência da Computação do UNIVEM/F.E.E.S.R., para obtenção do Título de Bacharel em Ciência da Computação.

Nota: 9,5 (NOTA 10,0)

Orientador: Fábio Dacêncio Pereira

1º. Examinador: Elvis Fusco

2º. Examinador: Fabio Lucio Meira

Marília, 29 de novembro de 2011.

## **AGRADECIMENTOS**

*A família por terem me encorajado nos momentos necessários.*

*Aos colegas por terem me ajudado a superar as dificuldades encontradas durante o curso.*

*Ao professor Fábio Dacêncio por ter me ajudado a conseguir mais essa conquista e também  
ao professor Leonardo Botega por ter me ajudado nos momentos de ausência do meu  
orientador.*

*“Bizarre is good! Common has hundreds of explanations. Bizarre has hardly any.”*

Gregory House.

LUZ, H. J. F. **Análise de Vulnerabilidades em Java Web Applications**. 2011. 64 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Centro Universitário Eurípides de Marília, Fundação de Ensino “Eurípides Soares da Rocha”, Marília, 2011.

## RESUMO

Ataques a aplicações web com vulnerabilidades são reportados diariamente e acumulam estatísticas que não podem ser desprezadas. Devido frequentemente ao despreparo dos desenvolvedores que não conhecem as vulnerabilidades e somando-se a isso o custo e tempo necessário para a implementação de métodos eficazes de segurança, estes índices são incrementados anualmente.

Neste contexto, este trabalho visou à criação de um software para analisar o código fonte de sistemas feitos para a web ou migrados para a web em busca de falhas de segurança, auxiliando o desenvolvedor a corrigi-las.

Com base na documentação e relatórios de algumas empresas, foram definidos as vulnerabilidades de SQL Injection e XSS (Cross Site Scripting) como foco de implementação neste trabalho de conclusão, uma vez que, seu nível de periculosidade e quantidade de ataques efetuados nos últimos anos aumentou significativamente.

Igualmente, foram encontradas diversas ferramentas correlatas, no entanto grande parte haviam sido descontinuadas há algum tempo ou eram softwares pagos.

Uma das ferramentas encontradas, o Websecurify, software gratuito e que continua a ser atualizado foi usado para comparação, ele engloba uma maior quantidade de vulnerabilidades do que este trabalho inicialmente verifica, além dos testes serem de caixa preta, ou seja, não é utilizado o código fonte da aplicação no mecanismo de análise.

Para o teste e comparação com o Websecurify, foi desenvolvida uma aplicação web funcional, onde o Websecurify não conseguiu detectar nenhuma vulnerabilidade na aplicação, com isso foram testadas mais duas aplicações, também de análise de caixa preta, onde o resultado se repetiu, ambos não conseguiram detectar nenhuma vulnerabilidade. Quanto ao programa desenvolvido neste trabalho, o mesmo conseguiu detectar as vulnerabilidades criadas na aplicação ao fazer a análise diretamente no código.

**Palavras-chave:** Injeção SQL. XSS. Segurança da informação.

LUZ, H. J. F. **Análise de Vulnerabilidades em Java Web Applications**. 2011. 64 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Centro Universitário Eurípides de Marília, Fundação de Ensino “Eurípides Soares da Rocha”, Marília, 2011.

## **ABSTRACT**

Attacks to web applications with vulnerabilities are reported daily and accumulates statistics that can't be neglected. Frequently due to the unpreparedness of the developers that don't know the vulnerabilities and adding to this the cost and time required for the implementation of security effective methods, these indices are increased annually.

In this context, this work aimed the creation of a software that analyses the source code of web systems or migrated to the web in search of security flaws, helping the developer to fix them.

Based on the documentation and report of some companies, were defined the vulnerabilities of SQL Injection and XSS (Cross Site Scripting) as the implementation focus on this work, since their level of dangerousness and number of attacks in recent years increased significantly.

Also, were found many related tools, however the majority had been discontinued some time ago or were proprietary software.

One of the tools found, the Websecurify, a free software that continues to be updated were used for comparison, it covers a bigger number of vulnerabilities then that work initially verify, besides the tests been of blackbox, in other words, the source code of the application is not utilized at the analysis mechanism.

For the comparison test with the Websecurify, were developed a functional web application, where Websecurify couldn't detect any vulnerability at the application, because of that were tested two more applications, that is also blackbox analysis, where the result repeated, both couldn't detect any vulnerability. About the program developed at this work, it was able to detect the vulnerabilities created at the application to do the analysis directly on code.

**Keywords:** SQL Injection. XSS. Information Security.

## LISTA DE ILUSTRAÇÕES

Figura 1 – Gráfico de incidentes reportados no Brasil por ano .....	14
Figura 2 – Média de ataques web por dia, por mês, 2009-2010.....	18
Figura 3 – Janela de exposição por Indústria.. .....	19
Figura 4 – Detalhes sobre os fatores de Risco.....	21
Figura 5 – Vulnerabilidades de aplicações web por tipo de ataque.....	22
Figura 6 – Ilustração de ataque XSS.. .....	33
Figura 7 – Diagrama de caso de uso da ferramenta OpenTracker.....	36
Figura 8 – Diagrama de classe da ferramenta OpenTracker.. .....	38
Figura 9 – Diagrama de sequência da ferramenta OpenTrack .....	39
Figura 10 – Exemplo da utilização do tratamento de sobrecarga.....	43
Figura 11 – Modelo conceitual do software proposto .....	44
Figura 12 – Diagrama de atividades do Módulo de SQL Injection.....	45
Figura 13 – Cadastro normal na aplicação de teste.. .....	48
Figura 14 – lista de registros após cadastro normal na aplicação de teste.....	48
Figura 15 – Busca com SQL Injection na aplicação de teste .....	49
Figura 16 – Lista de registros após busca com SQL Injection na aplicação de teste....	50
Figura 17 – Cadastro com SQL Injection na aplicação de teste.....	51
Figura 18 – Lista de registros após cadastro com SQL Injection na aplicação de teste.51	
Figura 19 – Cadastro com ataque XSS na aplicação de teste .....	52
Figura 20 – Lista de registros após cadastro com XSS na aplicação de teste.. .....	53
Figura 21 – Lista de registros após ataque de SQL Injection em comando SQL com Prepared Statement .....	54
Figura 22 – Falha encontra na aplicação pelo Websecurify .....	55
Figura 23 – Relatório do OpenTracker em aplicação vulnerável.....	56
Figura 24 – Relatório do OpenTracker em aplicação vulnerável com Filtro Pattern.....	57
Figura 25 – Relatório do OpenTracker em aplicação com Prepared Statement .....	57



## **LISTA DE TABELAS**

Tabela 1 – Resumo de 2010, classificado por indústria.....	20
Tabela 2 – Remoção de comentários e codificação de tokens .....	40
Tabela 3 – Identificação de escopos.....	40
Tabela 4 – Marcação de identificadores .....	41
Tabela 5 – Representação de variáveis no SRV .....	41
Tabela 6 – Exemplo de uma lista de hierarquia de escopos .....	42

## **LISTA DE ABREVIATURAS E SIGLAS**

API: Application Programming Interface

CERT.br: Centro de Estudos, Resposta e Tratamento de Incidentes de Segurança no Brasil

CSRF: Cross Site Request Forgery

DoS: Denial of Service

DDoS: Distributed Denial of Service

DOM: Document Object Model

IDE: Integrated Development Environment

OWASP: Open Web Application Security Project

OWVH: Open Web Vulnerabilities Hunter

SQL: Structured Query Language

SRV: Sistema de Rastreamento de Vulnerabilidades

XSS: Cross-Site Scripting

WASC: Web Application Security Consortium

# SUMÁRIO

INTRODUÇÃO.....	12
Segurança na web.....	13
Objetivos .....	15
Metodologia .....	16
1. PRINCIPAIS ATAQUES A SERVIÇOS ON-LINE .....	18
1.1. Impacto das vulnerabilidades em aplicações web.....	18
1.2. Pesquisas de incidentes da Internet .....	22
1.3. OWASP Top Ten .....	23
2. SQL INJECTION E SOLUÇÕES .....	25
2.1. Impacto do SQL Injection .....	25
2.2. Tipos de SQL Injection .....	26
2.3. Prevenções contra SQL Injection .....	30
2.4. Considerações finais sobre SQL Injection .....	31
3. CROSS SITE SCRIPTING (XSS) .....	33
3.1. Impacto do XSS .....	33
3.2. Tipos de XSS.....	34
3.3. Prevenções contra XSS .....	35
3.4. Considerações finais sobre XSS.....	35
4. FERRAMENTA OPENTRACKER.....	36
4.1. Descrição funcional.....	36
4.2. Diagrama de use case .....	36
4.3. Diagrama de classe.....	37
4.4. Diagrama de sequência.....	39
4.5. Funcionamento da Implementação do OpenTracker.....	40
5. MÓDULO DE DETECÇÃO DE VULNERABILIDADE EM JAVA .....	44
6. TESTES E VALIDAÇÃO.....	46
6.1. Ferramentas Correlatas .....	46
6.2. Aplicação Web de Teste.....	47
6.3. Teste para Validação.....	54
7. CONCLUSÕES .....	58
REFERÊNCIAS .....	59

## INTRODUÇÃO

Nos primórdios, o conhecimento era guardado pelas próprias pessoas, caso o indivíduo falecer, seu conhecimento tornava-se inacessível a menos que fosse passado para outra pessoa, devido a essa limitação, manter segura uma informação era uma tarefa simples, uma informação só era passada para uma pessoa considerada confiável e digna daquela informação.

Junto da facilidade de proteger uma informação, também vem à facilidade de perdê-la, estando ela contida apenas na mente da pessoa, a morte ou problemas relacionados à memória podem acarretar na perda total da informação. Posteriormente, com o desenvolvimento do papiro e em seguida do papel, parte do conhecimento passou a ser guardado neles, amenizando o problema da perda de informação, tornando mais fácil deixar o conhecimento como o legado da pessoa, no entanto tornando um pouco mais difícil sua segurança, ainda assim, feita de forma simples, bastando apenas manter protegida a folha contendo a informação de pessoas não autorizadas.

Com a evolução tecnológica, os computadores foram então desenvolvidos, sendo inicialmente utilizados para resolver problemas (Gugik, 2009), e não armazenar informação. O computador progrediu e passou então a também armazenar informação, foi então que o conteúdo do papel começou a passar para o meio digital, sendo uma forma considerada segura para guardá-la, mesmo com a existência das redes de computadores, pois para invadi-las era necessário invadir por meio físico.

O problema para manter segura a informação veio à tona com o advento da Internet, onde se tornou muito mais fácil ter acesso a outro computador. Se manter seguro se tornou mais complicado. Inicialmente foi explorado pelos crackers, pessoas que utilizam seu conhecimento para quebrar/invadir sistemas, etc. (Morimoto, 2005), a ingenuidade das pessoas, para que instalassem softwares maliciosos em seus computadores, posteriormente também viram os próprios sites como uma oportunidade de ataque, onde eles podem atacar o próprio site, ou os seus usuários sem que eles tenham ciência disso.

A Segurança da Informação busca proteger a informação de diversos tipos de ameaças, minimizar os danos e maximizar o retorno dos investimentos e das oportunidades. (Ferreira, 2008). Ela é caracterizada pelos seguintes atributos:

- **Confidencialidade:** garantia de que a informação é acessível somente por pessoas autorizadas.
- **Integridade:** preservação da exatidão da informação e dos métodos de processamento.
- **Disponibilidade:** garantia de que os usuários autorizados obterão acesso à informação e aos ativos correspondentes sempre que necessário. (Ferreira, pág. 2, 2008).
- **Confiabilidade:** continuidade do serviço correto.
- **Inocuidade:** ausência de consequências catastróficas para os usuários ou ambientes.
- **Manutenibilidade:** capacidade de sofrer modificações e reparos. (Basso, 2010).

## **Segurança na web**

Problemas envolvendo segurança na Internet são um fato. Parte destas ocorrências são consequências de aplicações web que contém pontos vulneráveis que podem ser explorados por ataques cada vez mais sofisticados. Muitas vezes os critérios tempo e custo são determinantes em um projeto, podendo fazer de requisitos não funcionais como a segurança, um acessório.

O projeto e desenvolvimento de aplicações web atendem a requisitos funcionais e não funcionais específicos para a criação de sistemas com suporte a múltiplos acessos, interatividade, compartilhamento e reuso de informações, entre outros. No entanto, algumas vezes um requisito não funcional não é tratado com devida importância, o requisito segurança.

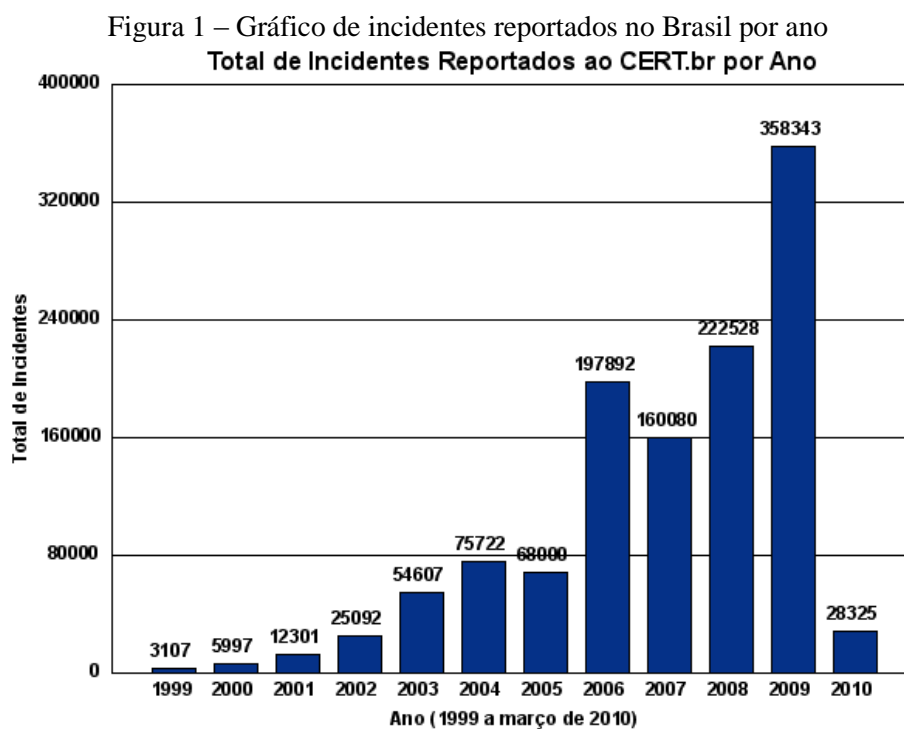
As consequências são evidentes, acarretando em diversas falhas de segurança que possibilita a um indivíduo mal intencionado invadir o sistema, modificar valores, roubar informações, interceptar comunicações, derruir o sistema impossibilitando o seu acesso e utilização, entre outras ações maliciosas. (Forristal, 2007)

Existem diversos métodos de ataque conhecidos. No entanto, muitas aplicações web não são desenvolvidas com as devidas medidas de segurança, mesmo que, para torná-las mais seguras seja necessário se precaver contra poucas dessas ameaças (IBM, 2011).

Corrigir essas falhas não só promove maior segurança ao provedor daquela aplicação, como também para o usuário que a utiliza. Contudo, frequentemente a segurança de aplicações web é tratada como um acessório e não um requisito desejado.

Algumas vezes todo recurso voltado para a segurança da aplicação não é tratado como um investimento, mas como um gasto financeiro e de tempo. Muitas vezes desenvolvedores concluem de forma equivocada que, porque o código está funcionando este também está seguro (Forristal, 2007). Muitos desenvolvedores justificam-se devido ao custo necessário para melhorar a segurança da aplicação, tornando-se inacessível para as pequenas e médias empresas fazer tal investimento. Por fim, é uma realidade que muitas empresas não desenvolvem aplicações seguras porque não contemplam o *know how* para isto.

No Brasil, o Centro de Estudos, Resposta e Tratamento de Incidentes de Segurança no Brasil (CERT.br) mantém um gráfico atualizado (figura 1) de incidentes reportados envolvendo redes conectadas à Internet no Brasil, em que mostra que ocorreu um aumento de 61% de incidentes em 2009 em relação a 2008, em 10 anos o aumento foi de 11433,4%. Isto mostra que esse tipo de ameaça tende a crescer ainda mais, tornando mais importante desenvolver aplicações web seguras.



Fonte: Cert.br, 2010.

Neste contexto, justifica-se a criação de um analisador de código fonte que busca falhas de segurança em aplicações web para amenizar esses efeitos, ajudando o desenvolvedor a encontrar e corrigir estas falhas.

O analisador, denominado Open Web Vulnerabilities Hunter (OWVH) será composto por duas partes, o OpenTracker, que busca informações sobre o código, e o módulo de detecção de vulnerabilidades, onde vários podem ser integrados ao OpenTracker, sendo que cada módulo tem a finalidade de trabalhar com determinada linguagem. O módulo através das informações enviadas pelo OpenTracker, identifica as vulnerabilidades contidas no código, no caso específico deste trabalho, as vulnerabilidades de SQL Injection e XSS em código Java.

## Objetivos

Este trabalho tem como objetivo criar um software para analisar o código fonte de sistemas feitos para a web ou migrados para a web em busca de falhas de segurança. Inicialmente serão analisados códigos fonte de aplicações web escritas na linguagem Java. No processo de concepção do software têm-se os seguintes objetivos específicos:

- Pesquisar softwares correlatos e relacioná-los para futura comparação.
- Estudar as principais anomalias relacionadas no OWASP Top Ten.
- Relacionar anomalias a Linguagem Java.
- Especificar e documentar software obedecendo à aplicação correta da engenharia de software.
- Criar método de pesquisa de anomalias em códigos Java.
- Criar mecanismos para padronizar a entrada de novas anomalias e tratamentos.
- Criar exemplos de teste e validação.
- Comparar com softwares correlatos.
- Criar uma distribuição *Open Source*.
- Desenvolver módulo de detecção de vulnerabilidades em código Java.
- Integrar módulo ao software OpenTracker.

O software proposto também tem como objetivo diminuir o custo e tempo para o melhoramento de segurança de aplicações web, facilitando para as empresas que pretendem investir em segurança por necessidade ou diferencial.

## Metodologia

O trabalho foi dividido em três fases principais que contemplam a (i) pesquisa de trabalhos correlatos e tecnologias, (ii) o projeto e desenvolvimento do software e (iii) avaliação dos resultados.

### i. Pesquisa de trabalhos correlatos e tecnologias

O OWASP Top Ten é a referência para classificação das anomalias mais comuns, no entanto publicações divulgadas pelo Cert.br, IBM, entre outras empresas foram considerados na prioridade de vulnerabilidades que serão tratadas pelo software.

Ferramentas correlatas apontam o tratamento de um conjunto reduzido de vulnerabilidades, sendo a maioria proprietárias. O software desenvolvido terá como característica ser uma ferramenta *open source*, que pode ser atualizada e mantida por uma comunidade.

### ii. Desenvolvimento do Projeto

O desenvolvimento do software esta dividido nas seguintes etapas:

- Criar um documento de requisitos do software.
- Definir a especificação do software
- Definir a arquitetura do software
- Codificar software em linguagem Java (IDE Netbeans e APIs de apoio ao desenvolvimento)

### iii. Teste e Análise dos Resultados

Será feito através de testes utilizando aplicações web funcionais que possuam vulnerabilidades, para que se possa observar a efetiva localização de falhas e o grau de precisão da solução proposta para a vulnerabilidade encontrada. Serão anotados falsos positivos onde apesar de existir a vulnerabilidade ele não o encontra, e positivos falsos em que o programa diz ser uma vulnerabilidade quando na realidade não é, para uma análise estatística da eficiência do software desenvolvido. Os casos de testes serão



artificiais na primeira etapa e em seguida serão disponibilizados para testes com aplicações web reais. Esta fase encontra-se em andamento, com previsão para finalizar ao final do período da bolsa.

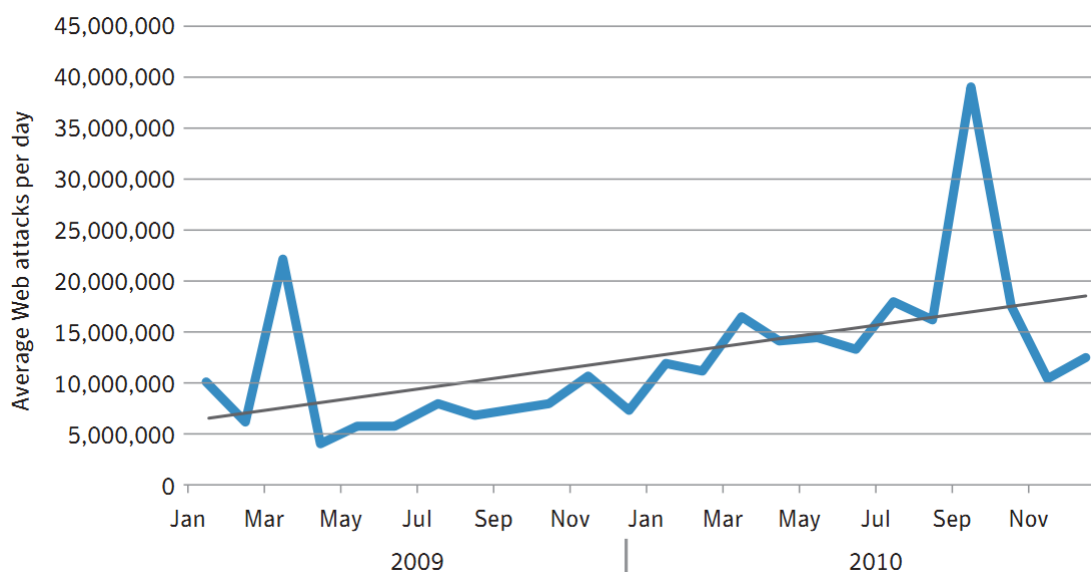
## 1. PRINCIPAIS ATAQUES A SERVIÇOS ON-LINE

Assim como surgem inovações para aplicações web são criados ataques para explorar pontos vulneráveis. Nesta seção são apresentados os principais ataques a serviços on-line e seu impacto em diferentes setores.

### 1.1. Impacto das vulnerabilidades em aplicações web

É comum que um software, que foi desenvolvido para *Desktop*, seja portado para a web, sem se preocupar com a sua segurança, ou mesmo um sistema desenvolvido especificamente para Web, porém não contempla requisitos de segurança. Com isso a quantidade de ataques feitos contra aplicações web estão aumentando de acordo com a Symantec (figura 2). É importante observar que apesar da variação dos índices de ataques mensais, a média de ataques a aplicações web apresenta-se de forma crescente.

Figura 2 – Média de ataques web por dia, por mês, 2009-2010.



Fonte: Symantec Corporation, 2011.

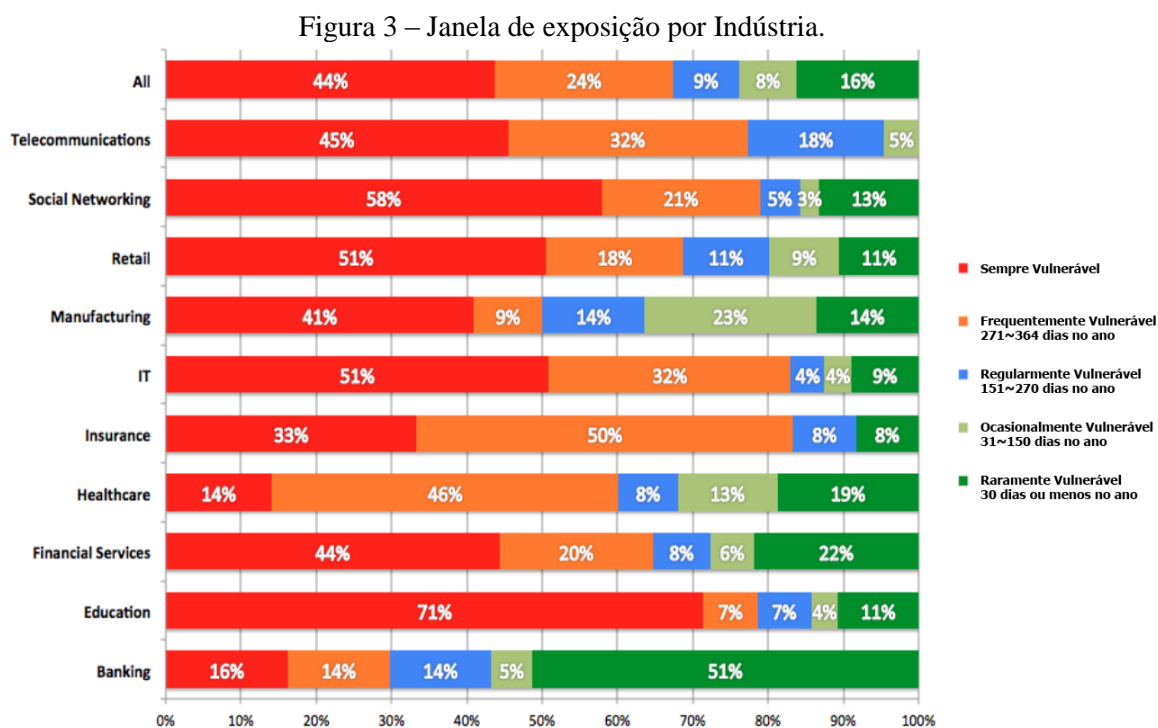
Vários estudos foram feitos por diversas empresas com relação às vulnerabilidades encontradas em aplicações web, mostrando a relevância do tema.

De acordo com a IBM, de todas as vulnerabilidades, 49% estão nas aplicações web, e é nesta área que eles consideram que está a verdadeira ameaça. A Sophos (2011b) encontra 30 mil sites perigosos todos os dias, o que equivale a um site entre 2 a

3 segundos. Desses sites, 70% são sites legítimos que foi comprometido através de modificações não autorizadas.

A WhiteHat Security (2008) em um estudo feito entre janeiro de 2006 a dezembro de 2008 mostrou que 82% dos sites possuem pelo menos uma vulnerabilidade, sendo que 63% possuem vulnerabilidades críticas. O estudo ainda revelou que o tempo para a correção de vulnerabilidades está melhorando lentamente, mas que ainda é possível melhorar muito devido ao fato de que a correção costuma levar semanas ou até meses para ser feita.

Em um novo estudo, a WhiteHat Security (2011b) mostrou que mesmo os sites de bancos são muito vulneráveis, sendo que 16% ficaram vulneráveis durante todo o ano de 2010, e metade ficaram até 150 dias vulneráveis, como pode-se observar na figura 3.



Fonte: WhiteHat Security, 2011b.

Como mostra na tabela 1, o estudo ainda revela que o número médio de vulnerabilidades encontrada em um site é de 230, e que somente 53% foram corrigidos. Além disso, a média de dias exposto a uma vulnerabilidade crítica foi de 233 dias.

Tabela 1 – Resumo de 2010, classificado por indústria.

<b>Indústria</b>	<b>Nº de Vulnerabilidades</b>	<b>Porcentagem Corrigida</b>	<b>Tempo exposto</b>
Média	230	53%	233
Banco	30	71%	74
Educação	80	40%	164
Finanças	266	41%	184
Saúde	33	48%	133
Seguro	80	46%	236
TI	111	50%	221
Fabricação	35	47%	123
Varejo	404	66%	328
Redes Sociais	71	47%	159
Telecomunicações	215	63%	260

Fonte: WhiteHat Security - WhiteHat Website Security Statistic Report, 2011b.

Um recente estudo feito pela WASC (*Web Application Security Consortium*) mostrou que 96.85% das aplicações corporativas web possuem sérias vulnerabilidades devido a falhas no código, ocasionando problemas como o XSS, SQL Injection, Information Leakage (vazamento de informação) e Predictable Resource Location (localização de recurso previsível), sendo que, a maior parte das vulnerabilidades é XSS de acordo com o MITRE (Christey, 2007), com isso, encontrando as falhas de XSS e as corrigindo já torna a aplicação consideravelmente mais segura. No entanto, para obter nível maior de segurança é necessário estender o tratamento a outros tipos de vulnerabilidades.

Há uma grande diversidade de vulnerabilidades, no entanto, de acordo com os dados do MITRE (Christey, 2007), os ataques se concentram em poucos tipos de vulnerabilidades e somando-se o fato que há muitas formas de explorar determinadas vulnerabilidades com base em um único tipo de ataque, o desenvolvimento do analisador está restrito a poucas vulnerabilidades devido ao tempo necessário para englobar uma grande quantidade de vulnerabilidades. Segue lista dos tipos de ataques contidos na documentação da OWASP, Top Ten 2007 e Top Ten 2010:

- 1) XSS (Cross Site Scripting)
- 2) Falhas de Injeção (Flaw Injection)
- 3) Execução Maliciosa de Arquivos

- 4)Referência Insegura Direta a Objetos
- 5)CSRF (Cross Site Request Forgery)
- 6)Vazamento de Informações e Tratamento de Erros Inapropriados
- 7)Falha de Autenticação e Gerenciamento de Sessão
- 8)Armazenamento Criptográfico Inseguro
- 9)Comunicações Inseguras
- 10)Falha de Restrição de Acesso a URL
- 11)Security Misconfiguration
- 12)Unvalidated Redirects and Forwards

Como há muitos meios de utilizar cada tipo de ataque, dificultando assim a detecção de todas as suas variações, o desenvolvimento deste trabalho proposto foi restrito aos dois principais tipos de ataque, o XSS e Falhas de Injeção (SQL Injection em específico). Para reforçar a justificativa, segue na figura 4 uma tabela mostrando que o ataque XSS é o mais comum, e a falha de injeção é de fácil exploração e pode ter um impacto severo em banco de dados.

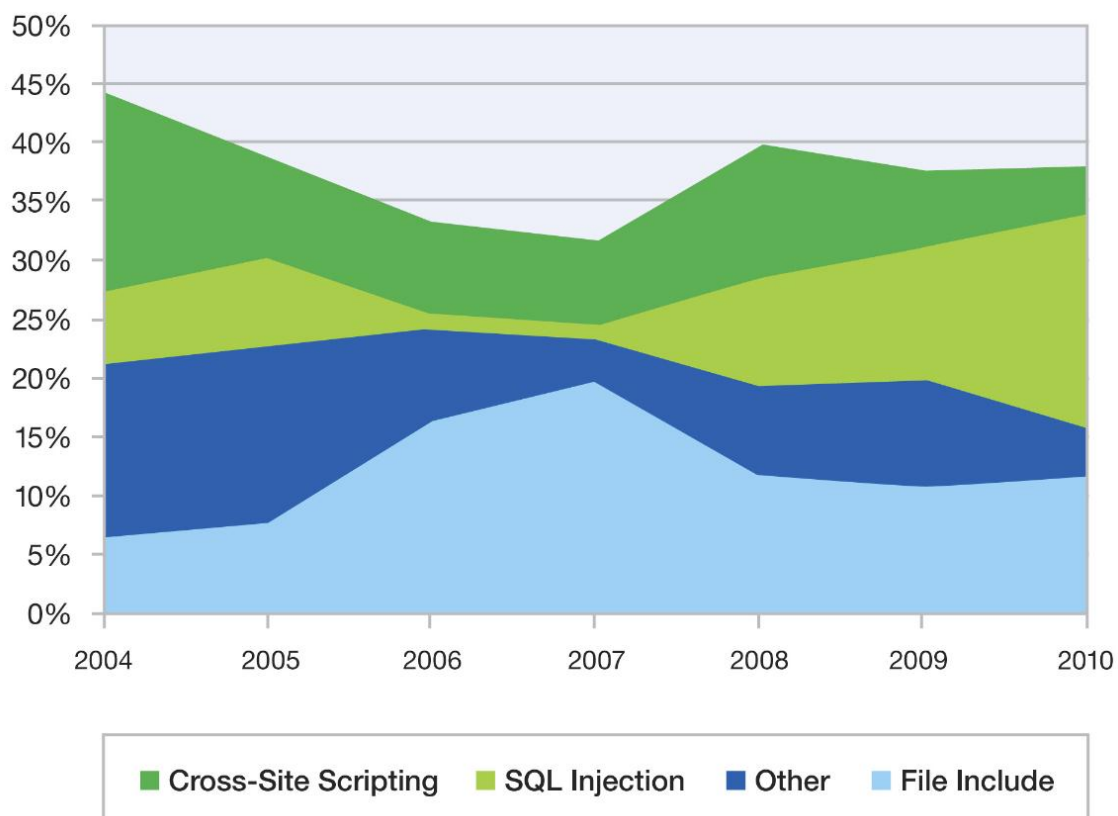
Figura 4 – Detalhes sobre os fatores de Risco

RISK	Threat Agents	Attack Vectors	Security Weakness	Technical Impacts	Business Impacts
		Exploitability	Prevalence	Detectability	Impact
A1-Injection		EASY	COMMON	AVERAGE	SEVERE
A2-XSS		AVERAGE	VERY WIDESPREAD	EASY	MODERATE
A3-Auth'n		AVERAGE	COMMON	AVERAGE	SEVERE
A4-DOR		EASY	COMMON	EASY	MODERATE
A5-CSRF		AVERAGE	WIDESPREAD	EASY	MODERATE
A6-Config		EASY	COMMON	EASY	MODERATE
A7-Crypto		DIFFICULT	UNCOMMON	DIFFICULT	SEVERE
A8-URL Access		EASY	UNCOMMON	AVERAGE	MODERATE
A9-Transport		DIFFICULT	COMMON	EASY	MODERATE
A10-Redirects		AVERAGE	UNCOMMON	EASY	MODERATE

Fonte: OWASP Top Ten, 2010.

Acrescenta-se a isso o fato de que tanto o ataque XSS quanto de Falha de Injeção estão em constante crescimento, tornando-se ainda mais predominante de acordo com o gráfico da IBM contido no relatório “IBM X-Force 2010 Trend and Risk Report”, mostrado na figura 5.

Figura 5 – Vulnerabilidades de aplicações web por tipo de ataque.



Fonte: IBM - IBM X-Force 2010 Trend and Risk Report, 2011.

## 1.2. Pesquisas de incidentes da Internet

Neste tópico destacam-se alguns casos noticiados de vulnerabilidade que geraram algum tipo de prejuízo.

Devido ao grande número de vulnerabilidades reportadas ou encontradas por diversas empresas, são muito comuns ataques utilizando dessas vulnerabilidades, sendo que os ataques de *SQL Injection*, devido a sua capacidade de prejudicar tende a aparecer mais em notícias, em especial quando ocorrem em sites conhecidos. Seguem alguns exemplos:

- O Site MySQL.com sofreu um ataque de *SQL Injection*, onde com isso conseguiram o login e senha dos usuários do site, além disso, o site da

Sun/Oracle, proprietária do MySQL.com, também foi atacado. De acordo com a Sophos, a vulnerabilidade não aparenta ser no software MySQL, mas sim na implementação do site. (Sophos - Naked Security, 2011)

- A Sony BMG grega teve seu site atacado, onde dados como o nome de usuário, verdadeiro nome e e-mail foram roubados. Aparentemente foi usada uma ferramenta automática de ataques de *SQL Injection* para encontrar a falha no site. (Sophos - Naked Security, 2011)
- O Wordpress.com, site de criação de blogs, foi alvo de ataque de DDoS (Distributed Denial of Service), o que deixou o serviço prestado por eles indisponível (Sophos – Naked Security, 2011). Apesar do ataque DDoS ter a finalidade de incapacitar um serviço, ele pode se tornar uma ferramenta para tornar possível a inserção e execução de código para acessar informações críticas de acordo com a OWASP.
- O site ATP (Association of Tennis Professionals) sofreu um ataque de SQL Injection durante o torneio de Wimbledon, onde foi inserido no site um script chamado Mal/Badsrc, que faz o download de outros scripts maliciosos, iniciando um processo de infecção no computador do usuário do site. (Sophos, 2008)
- Uma falha no site do Bradesco permitiu que um ataque de XSS fosse feito, onde os crackers através da falha requisitavam o recadastramento das “chaves de segurança” usadas nas transações através da Internet. (Assolini, 2008)
- Um ataque de XSS foi feito no site Youtube, onde através da falha, era requisitado ao usuário a instalação de um “objeto ActiveX”, que na realidade era um cavalo-de-tróia. (Rohr, 2007).

### 1.3. OWASP Top Ten

A Open Web Application Security Project é uma organização sem fins lucrativos que busca melhorar a segurança de aplicações web, aumentando a visibilidade da segurança em aplicações, para que as pessoas e organizações possam tomar melhores decisões sobre os riscos de segurança de suas aplicações.

Os resultados apresentados pela OWASP foram focos da fase de estudo deste trabalho.

A OWASP produz uma grande variedade de materiais de modo colaborativo, sendo que todas as ferramentas, documentos, fóruns e capítulos são livres e abertos para

que qualquer um possa utilizar ou melhorar. O material desenvolvido é utilizado, recomendado e referenciado por diversas organizações, como exemplo:

- World Wide Web Consortium (W3C), de âmbito mundial.
- Center for Internet Security (CIS) dos Estados Unidos.
- Cloud Security Alliance (CSA), de âmbito mundial.
- Information-Technology Promotion Agency (IPA), do Japão.
- Institute of Electrical and Electronics Engineers (IEEE), de âmbito mundial.
- Banco Central do Brasil.

Ela recebe apoio financeiro de empresas como a Adobe, Amazon, IBM e Oracle.

Essa vasta documentação desenvolvida busca auxiliar na criação de aplicações seguras, utilizando aplicações próprias para ajudar no entendimento de como certos ataques funcionam e como resolvê-los, como é o caso do WebGoat, ou auxiliar no teste de falhas de segurança na aplicação alvo no caso do documento “Testing Guide” (Meucci, 2008) por exemplo.

Somando a isso, a cada três anos a fundação OWASP lança uma documentação chamada OWASP Top Ten contendo a classificação das dez principais vulnerabilidades das aplicações web, essa documentação é desenvolvida com base nos dados de outras empresas, como o MITRE no caso do Top Ten 2007, mas também em critérios próprios da OWASP, sendo atualmente a frequência e o risco representado pela vulnerabilidade. Com isso, boa parte das vulnerabilidades críticas encontradas no OWASP Top Ten também está entre as mais críticas que propiciam perdas financeiras e de informação.



## 2. SQL INJECTION E SOLUÇÕES

Nesta seção é apresentada uma das principais falhas de segurança encontradas em aplicações web e que o software desenvolvido neste trabalho trata, o *SQL Injection*.

### 2.1. Impacto do SQL Injection

Ocorre nas aplicações que recebem dados do usuário e os envia a um interpretador sem validar ou codificar os dados. Isto permite ao atacante fazer diversas modificações na aplicação, nos sistemas relacionados a ele (em que há troca de informações) e nos dados a ele disponíveis, inclusive apagar ou criar novos dados.

Para facilitar o entendimento de como funciona uma injeção SQL, segue um exemplo simples em que o seguinte comando SQL, deveria em seu resultado retornar os dados de apenas uma conta.

```
String query = "SELECT * FROM accounts WHERE custID='" + request.getParameter("id") + "'";
```

No entanto um criminoso virtual pode modificar o parâmetro id no browser e enviar um id qualquer mais o comando “or’1’=‘1”’. Isso muda o significado do comando e faz com que a aplicação retorne todos os dados das contas do banco de dados, ao invés de apenas um.

Outro exemplo do que pode ser feito utilizando *SQL Injection* é ignorar parte do comando SQL.

```
String query = "SELECT * FROM users WHERE username = '" + username + "' AND password = '" + password + "'";
```

Na consulta acima, sua finalidade é retornar registros que contenham o nome de usuário e senha especificados, no entanto em um ataque pode-se ignorar a senha sem que haja erro de sintaxe.

```
"SELECT * FROM users WHERE username = ' João '; -- ' AND password = '" + password + "'";
```

Devido ao --, comando para transformar em comentário os caracteres subsequentes em bancos de dados Oracle, faz com que o comando a direita seja

completamente ignorado, com o restante da consulta sendo executada normalmente, sem que haja qualquer erro de sintaxe, com isso, neste caso haverá o retorno de todos os registros com nome João.

Para se prevenir de *SQL Injection* há diversas formas de acordo com a OWASP no AppSec 2009, como a não utilização do interpretador, com a aceitação de somente certos tipos de caracteres, substituição dos caracteres perigosos por códigos não nocivos, codificação de toda a entrada de dados oriunda do usuário, minimização dos privilégios do banco de dados para reduzir o impacto de uma falha, entre outros, limitando assim os dados que o usuário pode enviar.

## 2.2. Tipos de SQL Injection

A Oracle (2011b) classifica o *SQL Injection* em três categorias:

### i. Ataque de primeira ordem

O atacante entra com a String maliciosa e causa uma modificação no código que é executado imediatamente. O exemplo a seguir feito pela Oracle mostra de forma clara o funcionamento do *SQL Injection* de primeira ordem.

Exemplo: Tendo o seguinte procedimento:

```
PROCEDURE GET_PHONE (p_email VARCHAR2 DEFAULT NULL)
AS
TYPE cv_empty IS REF CURSOR;
cv cv_empty;
v_phone employees.phone_number%TYPE;
v_stmt VARCHAR2(400);
BEGIN
v_stmt := 'SELECT phone_number FROM employees WHERE email = '''
|| p_email || ''';
DBMS_OUTPUT.PUT_LINE('SQL statement: ' || v_stmt);
OPEN cv FOR v_stmt;
LOOP
FETCH cv INTO v_phone;
EXIT WHEN cv%NOTFOUND;
DBMS_OUTPUT.PUT_LINE('Phone: ' || v_phone);
END LOOP;
```

```

CLOSE cv;
EXCEPTION WHEN OTHERS THEN
    dbms_output.PUT_LINE(sqlerrm);
    dbms_output.PUT_LINE('SQL statement: ' || v_stmt);
END;
```

Ao executar o procedimento de forma esperada, utilizando, por exemplo,

**exec get\_phone('PFAY')**

Caso exista o registro ele irá retornar o valor esperado, no entanto, inserindo a string 'x" union select username from all\_users where "x"="x', no lugar de 'PFAY', devido à "x = x" ser verdadeiro, ele irá retornar todos os registros.

## ii. Ataque de segunda ordem

O atacante injeta o comando para ser armazenado, que é considerado com uma fonte segura. O ataque é executado posteriormente através de outra atividade.

Um exemplo simples mostrado pela Oracle é no caso de uma página que busca certas informações do usuário através de um cookie, diretamente não se envia a informação para o comando SQL, mas isso pode ser feito indiretamente, no exemplo de ter os seguintes comandos:

```

execute immediate 'SELECT username FROM sessiontable WHERE session
= ''||sessionid||'' into username;
```

```

execute immediate 'SELECT ssn FROM users WHERE
username= ''||username||'' into ssn;
```

Se o atacante tiver efetuado um cadastro anteriormente com um nome de usuário como este: *XXX' OR username='JANE*, devido a este ser o nome de usuário no cookie de sessão, ao executar o segundo comando, ele será modificado, ficando desta forma:

```

SELECT ssn FROM users WHERE
username= 'XXX' OR username='JANE '
```

Mesmo que o usuário XXX não exista, ele irá retornar todos os SSN (Social Security Number) dos usuários contendo Jane.

### iii. Injeção lateral

É a transformação da representação de um tipo de dados para outros. Segue exemplo feito pela Oracle, tendo algumas partes do código omitidas para não estender muito o exemplo:

Para este exemplo serão utilizados dois usuários, o usuário que será a vítima da injeção lateral de SQL, o *testuser*, e o usuário que fará o ataque, o *eviluser*.

Com o *testuser* é criada uma tabela com a seguinte informação:

```
CREATE TABLE t (a NUMBER, d DATE)
```

Nessa tabela são inseridos alguns valores para fins de testes da vulnerabilidade.

```
DECLARE
    Fmt CONSTANT VARCHAR2(80) := 'yyyy-mm-dd hh24:mi:ss';
BEGIN
    INSERT INTO t(a, d) VALUES(1, To_Date('2008-09-23 17:30:00', Fmt));
    INSERT INTO t(a, d) VALUES(2, To_Date('2008-09-21 17:30:00', Fmt));
    COMMIT;
END;
```

Por motivos de teste da injeção lateral de SQL, é criado um procedimento para contar o número de registros que a data seja maior que 2008-09-22.

```
CREATE OR REPLACE PROCEDURE p AUTHID DEFINER AS
    q CONSTANT VARCHAR2(1) := '';
    d CONSTANT DATE :=
    TO_DATE('2008-09-22 17:30:00', 'yyyy-mm-dd hh24:mi:ss');
    stmt CONSTANT VARCHAR2(32767) :=
    'SELECT COUNT(*) FROM t WHERE t.d > '||q||d||q;
    n NUMBER;
BEGIN
    EXECUTE IMMEDIATE stmt INTO n;
    DBMS_OUTPUT.PUT_LINE(n||' rows');
END p;
```

Como é possível visualizar, não há entrada de dados por parte do usuário nesse procedimento, sendo teoricamente imune a *SQL Injection*. Ao executar o procedimento, ele irá retornar o número de registros que tenha a data maior que o especificado, que no caso é 1.

Agora com o *eviluser*, executando o comando abaixo, é retornado o valor esperado, 1.

```
BEGIN testuser.p(); END;
```

No entanto, ao executar o seguinte comando:

```
ALTER SESSION SET NLS_Date_Format='yyyy'
```

E novamente executar o comando anterior, ao invés de retornar 1, ele irá retornar 2, pois com isso o procedimento conta todos os registros com o ano 'yyyy' dos dados da coluna maior que 2008, como ambos os registros inseridos anteriormente satisfazem a condição, ele retorna o valor 2, demonstrando que há uma falha no procedimento.

Para então mostrar que ele é vulnerável a Injeção lateral de SQL, cria-se com o *eviluser* a seguinte função:

```
FUNCTION EVIL RETURN NUMBER AUTHID CURRENT_USER IS
pragma Autonomous_Transaction;
BEGIN
BEGIN
DBMS_OUTPUT.PUT_LINE('In Evil()!');
EXECUTE IMMEDIATE 'delete from testuser.t';
COMMIT;
EXCEPTION
WHEN OTHERS THEN NULL;
END;
RETURN 1;
END;
```

Ao executar o comando “ALTER SESSION SET NLS\_Date\_Format='' AND eviluser.evil()=1--''” e posteriormente o comando “BEGIN testuser.p(); END;” ele irá retornar

*In Evil()!*

*0 rows.*

Mostrando que a Injeção lateral de SQL funcionou.

## 2.3. Prevenções contra SQL Injection

A OWASP (2011b) cita algumas formas para se proteger contra *SQL Injection*, dentre elas o *Prepared Statement*, *Stored Procedure* e *White List Input Validation*.

### i.Prepared Statement

No *Prepared Statement*, é primeiro definido o código SQL, posteriormente é passado os parâmetros, dessa forma é possível distinguir código de dado, não importando a entrada de dados do usuário.

Devido a isso, se um usuário entrar com "1" or "1"='1", ele não será mais interpretado como parte da requisição SQL, mas sim como um dado comum.

A utilização de *Prepared Statement* é feito da seguinte forma em Java:

```
PreparedStatement prep;
try {
    prep = con.prepareStatement("insert into cliente values (?, ?, ?)");
    prep.setInt(1, cod);
    prep.setString(2, nome);
    prep.setInt(3, idade);
    prep.executeQuery();
} catch (SQLException ex) {
    Logger.getLogger(Conexao.class.getName()).log(Level.SEVERE, null, ex);
}
```

### ii.Stored Procedure

No código ele é semelhante ao *Prepared Statement*, no entanto o comando SQL não fica explícito no código, mas sim armazenado no banco de dados. Ele oferece a mesma segurança que o *Prepared Statement* se implementado corretamente, ou seja, se ele não incluir nenhuma geração dinâmica de SQL não segura, caso isso seja necessário, é importante fazer a validação da entrada de dados.

A implementação do *Stored Procedure* em Java é feito da seguinte forma:

```
String custname = request.getParameter("customerName");
try {
    CallableStatement cs = connection.prepareCall("{call sp_getAccBalan(?)}");
    cs.setString(1, custname);
    ResultSet results = cs.executeQuery();
    // ... result set handling
}
```

```

} catch (SQLException se) {
    // ... logging and error handling
}

```

Onde o “call sp\_getAccBalan” é a chamada do comando SQL no banco, o ponto de interrogação (?) define onde irá o dado da variável determinado pelo comando “cs.setString(1, custname)”.

### iii. White List Input Validation

É utilizado para detectar se na variável há algum elemento não autorizado para concatenar com o comando SQL. Ele define quais caracteres estão autorizados a serem utilizadas nas entradas de dados, com isso, todo o resto não está autorizado. Com ele é possível definir os caracteres permitidos, mas também quais estruturas de dados são permitidas, como na utilização de estruturas bem definidas, como datas, CEP, e-mail, etc.

Em Java, um exemplo para a criação de uma *White List* é a utilização de *Pattern*, que permite criar o padrão de dados permitidos, possuindo uma vasta gama de possibilidades quanto ao filtro que pode ser criado.

Sua utilização pode ser feita dessa forma:

```

Pattern p = Pattern.compile("(\\w|\\s)*(\\w|\\s)*");
Matcher m = p.matcher(regex);
boolean b = m.matches();

```

O filtro descrito no exemplo permite apenas letras, números e espaços, onde o `\\w` se refere às letras e números, e `\\s` se refere ao espaço. (Oracle, 2011a).

## 2.4. Considerações finais sobre SQL Injection

Ao efetuar ataques de injeção SQL, algumas vezes mensagens de erro aparecem contendo o comando SQL que sofreu do erro, facilitando assim um ataque de injeção SQL. Em outras vezes ao invés de aparecer onde ocorreu o erro, aparece uma mensagem genérica feita pelo desenvolvedor, o que torna mais difícil efetuar o ataque. (OWASP, 2009a)

Ainda é possível uma variação conhecida como *Blind SQL Injection* que é um ataque SQL categorizado anteriormente, mas que aproveita do vazamento de informação da página.

O *SQL Injection* está se tornando ainda mais predominante, combinado com sua capacidade de causar danos ao Banco de Dados, torna a segurança contra esse tipo de ataque cada vez mais importante. Os seus diversos tipos torna mais complexo para se proteger, devido a necessidade de proteger não somente a entrada direta de dados do usuário, mas as indiretas também, que podem levar um dado do usuário para um comando SQL.



### 3. CROSS SITE SCRIPTING (XSS)

Nesta seção é apresentada uma a falha de segurança mais comum encontrada em aplicações web e que o software desenvolvido neste trabalho trata, o XSS.

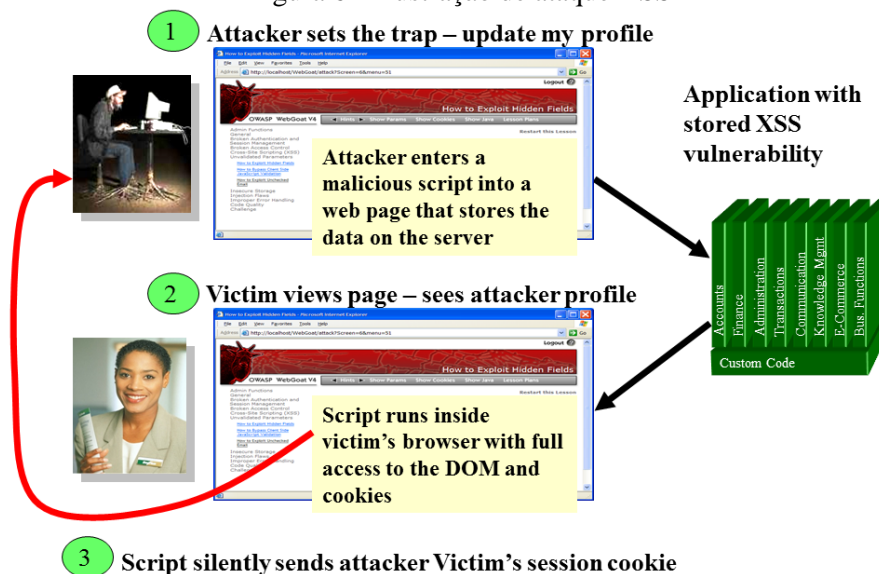
#### 3.1. Impacto do XSS

Ocorre nas aplicações que recebem dados do usuário e o envia ao navegador sem validação ou codificação. Possibilita roubo da sessão do usuário, roubo de dados, reescrita da página, redirecionamento do usuário a outra página.

Ataques XSS são muito comuns nas redes sociais, como exemplo recentemente houve um ataque de XSS no site Twitter, site em que ocorrem ataques de XSS todos os meses, feito por um brasileiro, em que ele se aproveitou dos encurtadores de url (endereço) para colocar um código maior que o permitido pelo site e esconder o código dos usuários (Abril, 2010). Quando um usuário qualquer clicava no link em questão, algumas ações na aplicação do usuário eram feitas, ações essas em que nesta ocasião não passou de uma mera brincadeira, mas que, no entanto poderia ter sido usada para roubar informações do usuário por exemplo.

Para ilustrar um ataque XSS, segue ilustração usada no Appsec 2009 da OWASP (figura 6), em que o código malicioso é colocado em uma página que grava os dados no servidor. A vítima então acessa essa página, onde ele é executado e com isso envia o cookie de sessão da vítima ao atacante.

Figura 6 – Ilustração de ataque XSS



Fonte: Wichers, 2009.

Para se prevenir contra XSS, há algumas formas de acordo com a OWASP, como validar a entrada quanto ao tamanho, tipo, sintaxe e regras de negócio, rejeitar entradas inválidas ao invés de tentar corrigir dados potencialmente hostis, codificar a saída, codificando todos os caracteres com exceção de um subconjunto muito limitado, entre outras formas, incluindo algumas específicas de cada linguagem.

### **3.2. Tipos de XSS**

A OWASP classifica o ataque de XSS em três categorias:

#### **i. Ataque XSS armazenado (ou persistente)**

É o ataque onde o código injetado é permanentemente armazenado, no banco de dados por exemplo, com isso, quando a vítima recupera a informação do banco, o código é executado.

#### **ii. Ataque XSS refletido (ou não persistente)**

O código injetado é refletido no servidor, como em uma mensagem de erro, resultado de pesquisa, ou qualquer resposta que inclua alguma entrada de dados enviada para o servidor como parte da requisição.

O ataque refletido é entregue a vítima por e-mail, mensagens pela Internet, onde o atacante persuade a vítima a clicar-lo, e com isso o código é executado no browser do usuário.

#### **iii. Ataque XSS baseado em DOM (Document Object Model)**

É o ataque onde ele é executado como resultado da modificação do “ambiente” DOM no browser da vítima, a página em si não muda, mas o código do lado do cliente contido na página executa de forma diferente devido à modificação no DOM. Diferente dos tipos anteriores, ele se aproveita de uma falha do lado do cliente ao invés do servidor.

### **3.3. Prevenções contra XSS**

Assim como no *SQL Injection*, a OWASP recomenda a validação dos dados, com a utilização de *White list*, *Black List*, entre outros. A diferença do *Black List* é que, ao invés de definir o padrão permitido, define-se o que não é permitido, apesar de ser uma boa solução, ele normalmente é incompleto, exigindo a atualização periódica da lista, devido aos novos métodos de ataque XSS usados no decorrer do tempo, o que torna o *White List* mais interessante. (OWASP, 2009b).

### **3.4. Considerações finais sobre XSS**

O ataque XSS é atualmente o mais comum, ocorre frequentemente em redes sociais como o Orkut e Twitter, e por isso é considerado um dos principais ataques, apesar dele não trazer prejuízo diretamente para a empresa, por ele não almejar a empresa, ele pode lesar o usuário, o que pode acarretar em futura perda para a empresa.

## 4. FERRAMENTA OPENTRACKER

### 4.1. Descrição funcional

Este trabalho envolve o desenvolvimento do módulo para análise das vulnerabilidades, no entanto antes disso é necessário preparar o código e extrair o maior número de informações possíveis para que o módulo de análise execute os testes e verificações. Neste contexto, foi paralelamente desenvolvido um software denominado OpenTracker por um aluno de graduação do UNIVEM, que faz o tratamento do código.

O tratamento de código implica em:

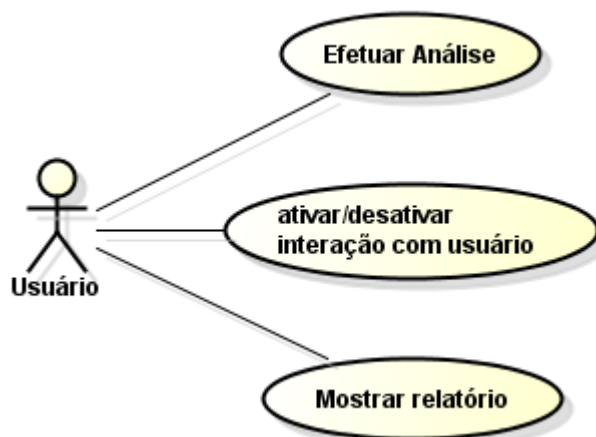
- Remover os comentários do código (para que estes não sejam tratados como código, que podem ser interpretados como uma vulnerabilidade).
- Identificar os escopos (necessário para diferenciar variáveis).
- Identificar a hierarquia de escopos
- Fazer tratamento de sobrecarga
- Entre outras funções

Posteriormente ele envia ao módulo as linhas relevantes para serem analisados, linhas essas que são selecionadas de acordo com as especificações do módulo. A junção do OpenTracker com os módulos de detecção de vulnerabilidades é chamado de Open Web Vulnerabilities Hunter (OWVH).

Seguem os diagramas referente à ferramenta OpenTracker.

### 4.2. Diagrama de use case

Figura 7 – Diagrama de caso de uso da ferramenta OpenTracker.



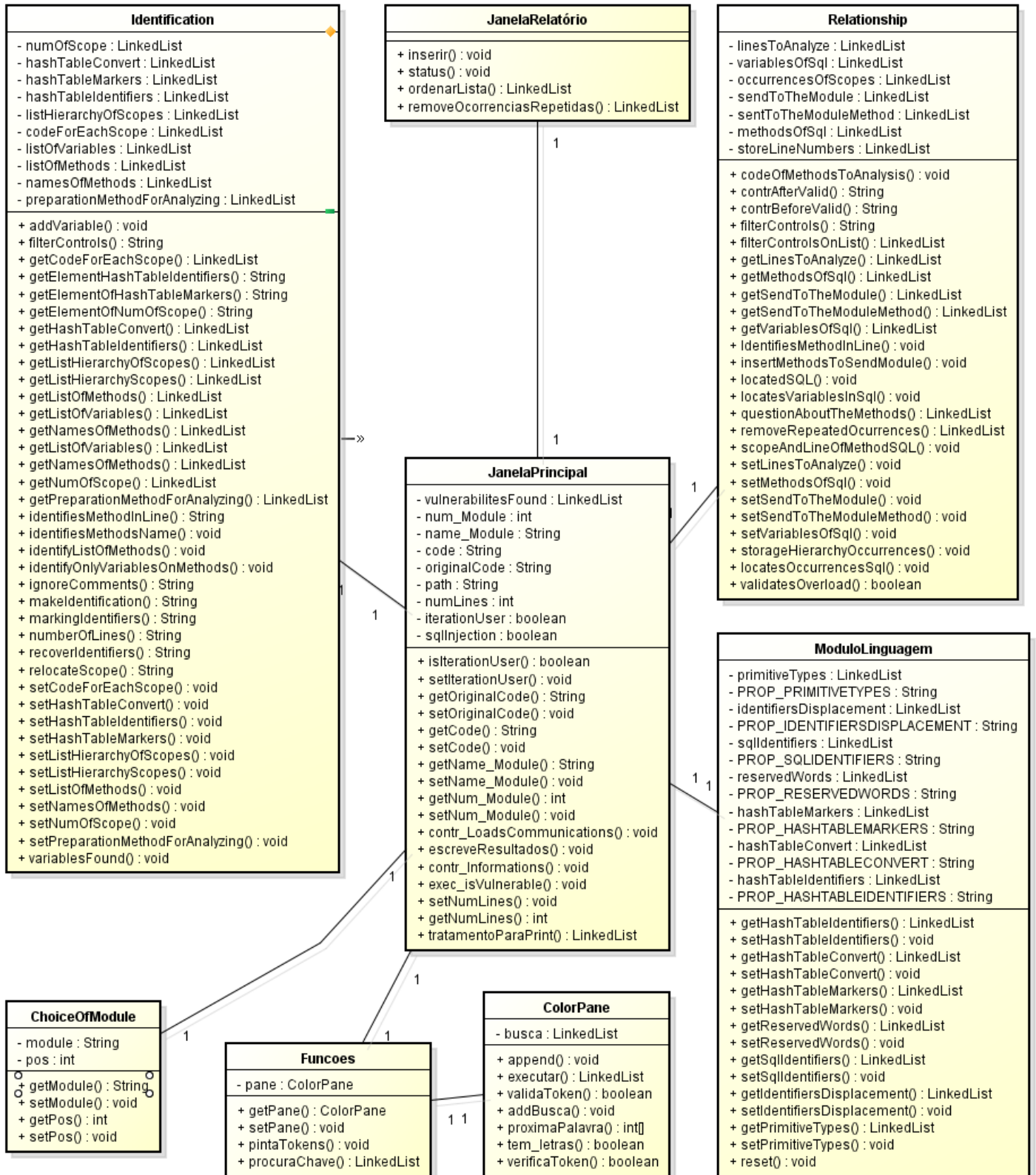
Fonte: Próprio autor, 2011.

### 4.3. Diagrama de classe

Segue o diagrama de classe (figura 8) referente às classes abaixo.

- i. ChoiceOfModule**  
Exibe a janela para a escolha da linguagem a ser analisada.
- ii. Funcoes**  
Busca as palavras reservadas da linguagem para colori-las.
- iii. Identification**  
Identifica informações referentes ao código como os escopos e variáveis, além de remover os comentários do código.
- iv. Janela Principal**  
A interface principal do software, onde o usuário interage com suas funções. Faz a ligação entre as classes para a realização da análise do código.
- v. Janela Relatório**  
Janela para a exibição dos resultados obtidos através da análise.
- vi. ModuloLinguagem**  
Possui as informações referente à linguagem, como as palavras reservadas, marcadores, entre outras informações.
- vii. Relationship**  
Identifica os comandos SQL no código, as variáveis e os métodos utilizados nesse comando, entre outras funções.

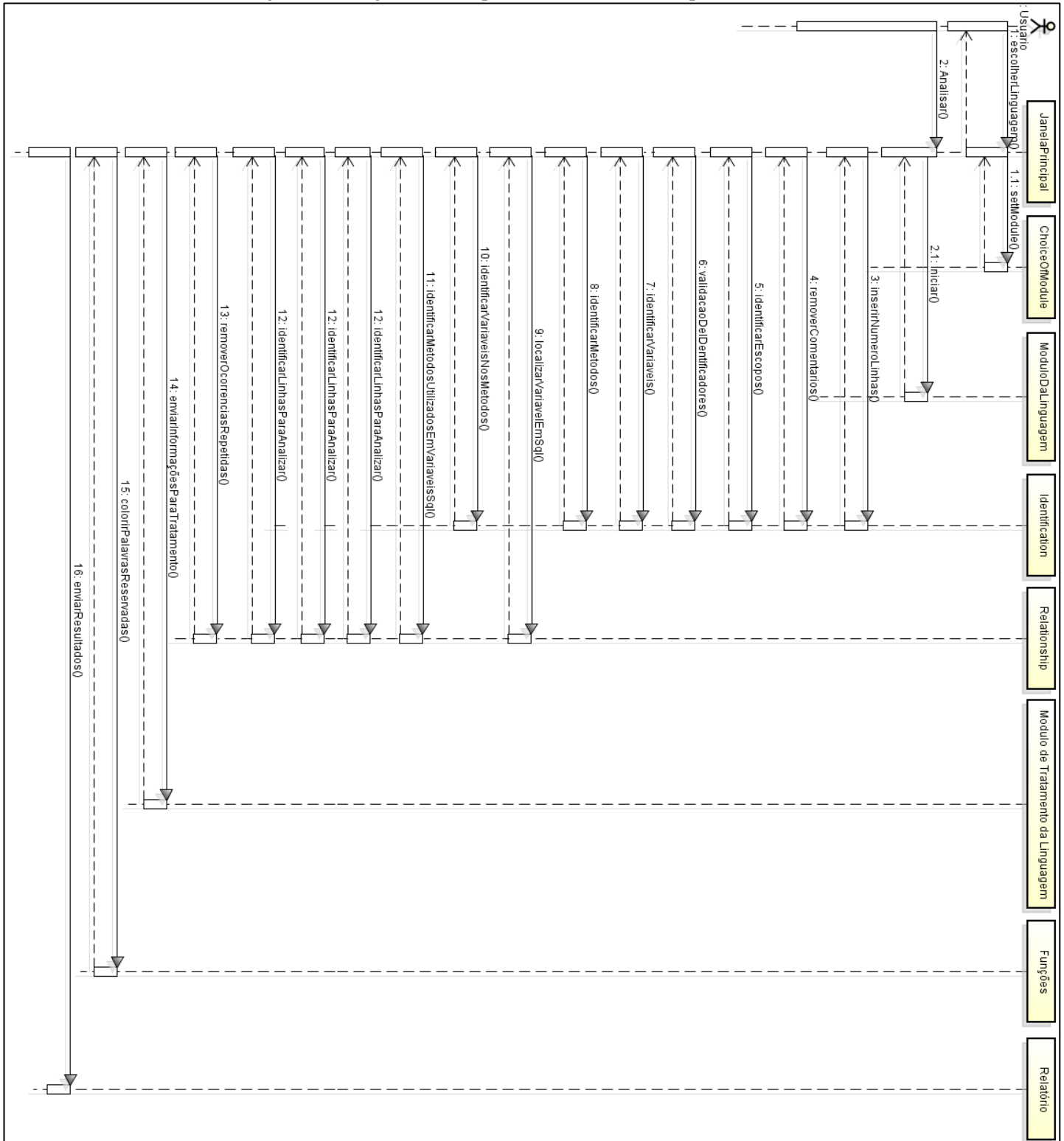
Figura 8 – Diagrama de classe da ferramenta OpenTracker.



Fonte: Próprio autor, 2011.

#### 4.4. Diagrama de seqüência

Figura 9 – Diagrama de seqüência da ferramenta OpenTracker.



Fonte: Próprio autor, 2011.

## 4.5. Funcionamento da Implementação do OpenTracker

A seguir serão demonstrados alguns exemplos dos métodos do SRV (Sistema de Rastreamento de Vulnerabilidades) da ferramenta OpenTracker, que tem a finalidade de identificar as linhas de códigos importantes, utilizadas nas análises de vulnerabilidades. (Xavier, 2010)

### i. Remoção de Comentários e Codificação de Tokens

A tabela 2 apresenta a saída obtida a partir da entrada de um código fonte, em que é realizada uma pré-análise do código, removendo os comentários da linguagem Java como “//” e “/\* \*/” e também codificando os caracteres relevantes utilizados nas análises do SRV. (Xavier, 2010)

Tabela 2 – Remoção de comentários e codificação de tokens.

/* Comentário */	0##
// Comentário	1##
String variável = "{	2## String variável = "@123@
(";	@41@";
<b>ENTRADA</b>	<b>SAÍDA</b>

Fonte: Xavier, 2010.

### ii. Identificação de Escopos

Para obter a identificação e o controle dos escopos contidos em um código fonte (tabela 3), foi desenvolvido um mecanismo que localiza através de uma análise sintática, os identificadores de início e término de escopos “{” e “}” (linguagem Java) em que são substituídos por uma expressão do tipo “H1E1T1” e “H1F1T1”. Onde: “H” indica o número da classe que pertence o escopo, “E” o número do novo escopo, “F” indica o término do escopo e “T” posição total em relação a todos os escopos do código fonte. (Xavier, 2010)

Tabela 3 – Identificação de escopos.

public class Teste {	public class Teste H1E1T1
}	H1F1T1
<b>ENTRADA</b>	<b>SAÍDA</b>

Fonte: Xavier, 2010.



### iii. Marcação de Identificadores

Durante o processo de análise sintática, foram utilizados métodos Java como: *indexOf()*, *replace()* e *charAt()*. Em que as palavras reservadas com característica de marcador ou identificador como o “private” contido na declaração do método *metodoTeste()*, fosse atribuído a ele, um valor codificado através da função *hashCode()*, permitindo que outras funções e métodos o identificasse através desse valor (tabela 4).

Este método impede que ocorrências do tipo “String privateVariavel;” representada pela tabela 4, identificasse a palavra “private” do nome da variável como um modificador de acesso como por exemplo, evitando possíveis erros da execução de métodos seguintes e efetuando dessa forma, uma validação do código fonte no processo de análise léxica. (Xavier, 2010)

Tabela 4 – Marcação de identificadores.

<code>private String metodoTeste() { } String privateVariavel;</code>	<code>%-314497661% String metodoTeste () H1E1T1 H1F1T1 String @Fail@Variavel;</code>
<b>ENTRADA</b>	<b>SAÍDA</b>

Fonte: Xavier, 2010.

### iv. Lista de Variáveis

O controle de variáveis é feito por uma lista que contém a relação de todas as variáveis que compõem o código fonte. Cada variável apresenta a seguinte estrutura: tipo da variável, declaração de nome, o escopo em que ela foi instanciada e o tipo de variável a que ela pertence. Na tabela 5 esse último parâmetro é indicado por “Type” que pode assumir dois valores: default (se ela foi criada dentro do bloco de código) ou method (instanciada como parâmetro de um método, onde seu valor inicial é conhecido por outra variável já existente). Os caracteres “|,@,%,#,” são utilizados para realizar buscas diretas na obtenção de informações específicas sobre cada variável. (Xavier, 2010)

Tabela 5 – Representação de variáveis no SRV.

<code>String variavel = "string"; public void metodo(int variavel){}</code>	<code>String: variavel@Escopo: %H1E1T1#Type: default int: variavel@Escopo: %H1E9T9#Type: method</code>
<b>ENTRADA</b>	<b>SAÍDA</b>

Fonte: Xavier, 2010.

## v. Hierarquia de Escopos

A lista presente na tabela 6 representa a árvore de escopos do código fonte, que possibilita, por exemplo, identificar o fluxo de uma variável ou separar trechos de códigos específicos para análises. A hierarquia de escopos foi obtida, através da utilização de uma pilha, e de uma lista contendo todos os escopos identificados. Dessa forma, percorrendo essa lista era identificado o escopo corrente indicado por “[HxEyTz]”, e a cada posição percorrida da lista, era inserido na hierarquia do escopo separados por “,” aqueles identificados por “HxEwTz”, concluindo quando encontrado o escopo contendo uma expressão do tipo “HxFyTz”, dando início a análise do escopo “HxEy+1Tz+1”. (Xavier, 2010)

Tabela 6 – Exemplo de uma lista de hierarquia de escopos.

[H1E1T1], H1E1T1
[H1E2T2], H1E2T2, H1E1T1
[H1E3T3], H1E3T3, H1E1T1
[H1E4T4], H1E4T4, H1E3T3, H1E1T1
[H1E5T5], H1E5T5, H1E4T4, H1E3T3, H1E1T1
[H1E6T6], H1E6T6, H1E4T4, H1E3T3, H1E1T1
[H1E7T7], H1E7T7, H1E4T4, H1E3T3, H1E1T1

Fonte: Xavier, 2010.

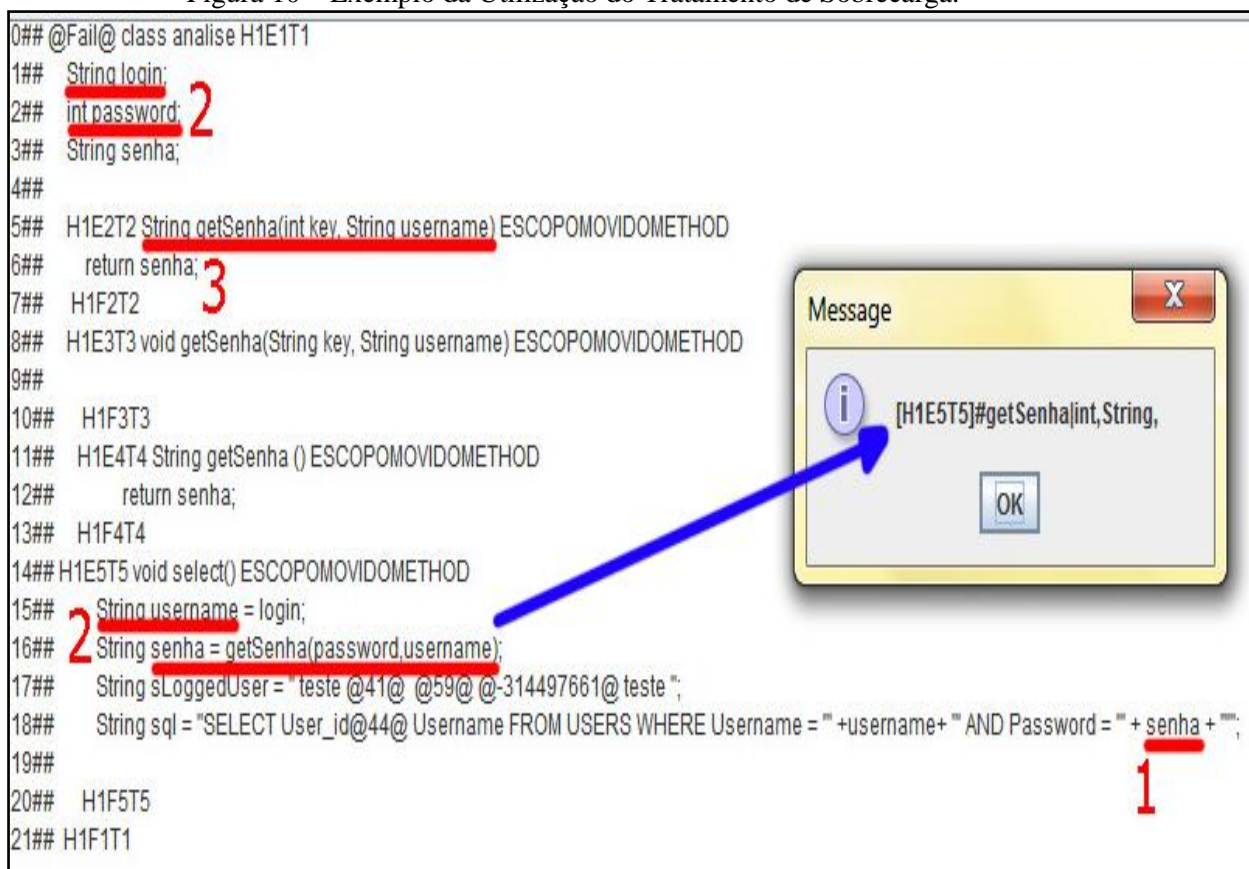
## vi. Tratamento de Sobrecarga

O SRV utiliza o tratamento de sobrecarga para identificar a localização dos métodos que foram inseridos na análise (figura 10).

Considere os números indicados como etapas. A primeira etapa apresenta a execução da análise feita pelo SRV no tratamento de SQL Injection, note que ao identificar a variável “senha” na definição na SQL, o analisador irá inserir na lista de variáveis a serem analisadas essa variável, que por sua vez, durante a análise recursiva é encontrada uma atribuição com um retorno do método getSenha(password, username) (Etapa 2) no qual deverá ser inserido na análise. O método para tratamento de sobrecarga obtém os parâmetros do método a ser inserido na análise e cria uma estrutura conforme é possível visualizar na imagem indicada pela seta, retornando o escopo de onde o método foi utilizado e os tipos de cada um de seus parâmetros. Dessa forma, na

Etapa 3 é localizado o método no código fonte realizando uma validação através da comparação dos tipos e quantidade de parâmetros. (Xavier, 2010)

Figura 10 – Exemplo da Utilização do Tratamento de Sobrecarga.



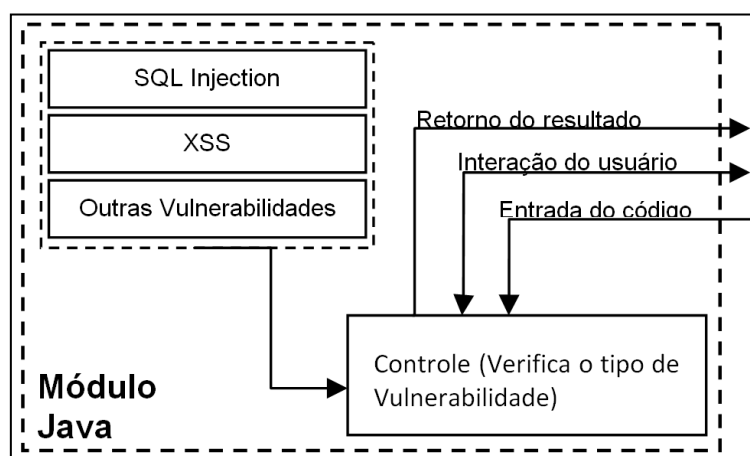
Fonte: Xavier, 2010.

É importante citar, que existe uma exceção para utilização deste tipo de tratamento na linguagem Java, que é a utilização de métodos genéricos (ROCHA, 2005) onde este tratamento não é adequado, considerando que os tipos dos parâmetros são definidos em tempo de compilação. Dessa forma estão sendo estudados, possíveis algoritmos que consigam efetuar o tratamento desejado. (Xavier, 2010)

## 5. MÓDULO DE DETECÇÃO DE VULNERABILIDADE EM JAVA

O modelo conceitual do módulo foi desenvolvido, facilitando o entendimento de como é o seu funcionamento. Como mostrado na figura 11, o código a ser analisado vai para o “Controle”, onde são verificados quais tipos de ataques às vulnerabilidades encontradas propiciam. Para isso é utilizado os módulos de cada tipo de ataque para fazer a verificação. Os resultados então são retornados para o desenvolvedor, apresentando as vulnerabilidades encontradas e os possíveis modos de elimina-las.

Figura 11 – Modelo conceitual do software proposto.

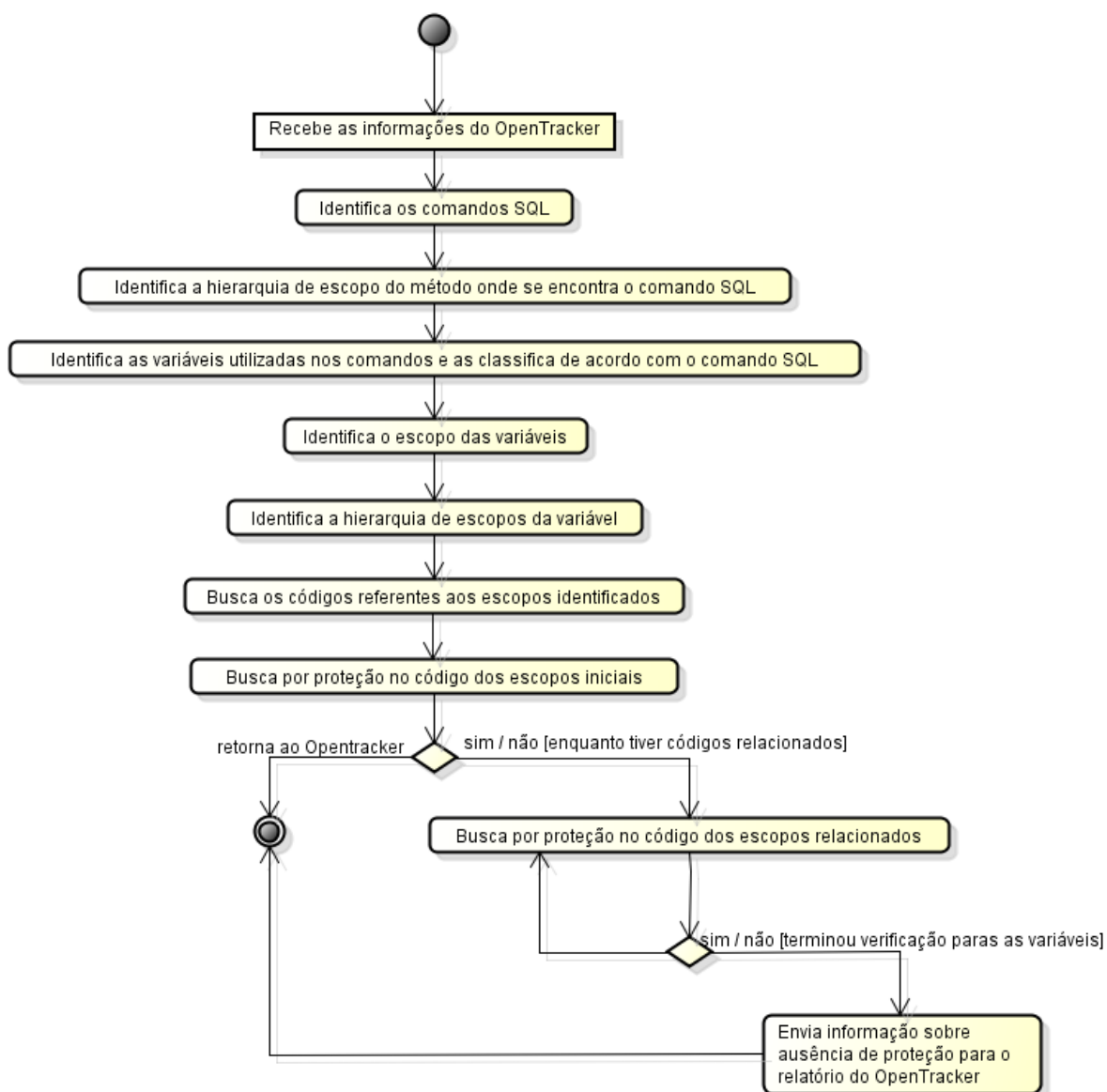


Fonte: próprio autor, 2011.

Para exemplificar o funcionamento do módulo, segue na figura 12 o diagrama de atividades do módulo de *SQL Injection*, onde, através do módulo de Controle, o OpenTracker disponibiliza ao módulo de *SQL Injection* diversas informações relevantes como a hierarquia dos escopos, o código de cada escopo, a lista de variáveis, lista de métodos, entre outras informações, para que se possa encontrar todas as ligações de determinadas informações, como o caminho que determinada variável utilizada em um comando SQL percorreu no código, para possibilitar verificar se a variável foi modificada, e com isso verificar se foram utilizadas algumas formas de proteção para *SQL Injection*, definindo assim se está protegido, ou potencialmente vulnerável.

Nos casos de estar potencialmente vulnerável, onde não é possível definir se está protegido, a interação com o usuário descrita na figura 11, ao final do processo de busca, permite o usuário definir se os casos em questão possuem alguma proteção, para isso ele terá que fazer uma busca manual no código. No final é retornado o resultado contendo as vulnerabilidades do código.

Figura 12 – Diagrama de atividades do Módulo de Detecção de Vulnerabilidade.



Fonte: próprio autor, 2011.

## 6. TESTES E VALIDAÇÃO

Foi desenvolvida uma aplicação com funcionalidade real, baseado na aplicação feita por Bianchi (2011), para o teste e comparação com ferramentas correlatas, onde a aplicação web possui vulnerabilidade de *SQL Injection* e XSS.

### 6.1. Ferramentas Correlatas

Foram encontrados alguns trabalhos correlatos, soluções pagas como o Sentinel da Whitehat Security, Qualysguard WAS (Web Application Scanning) da Qualys, Acunetix Web Vulnerability Scanner da Acunetix e Rational AppScan da IBM. Soluções gratuitas são escassas, pois boa parte foram descontinuadas. Dentre as opções gratuitas, há o software HP Scrawl (Peterson, 2008) e o Websecurify.

Com isso, foram feitos testes na aplicação web utilizando os programas correlatos Websecurify, HP Scrawl e dentre as opções pagas, o Acunetix Web Vulnerability Scanner 7 por disponibilizar uma versão trial (para teste) do seu programa.

#### i. HP Scrawl

Lançado em 2008 pela HP em parceria com a Microsoft Security Response Center (MSRC), somente por vulnerabilidade de *SQL Injection*, entre outras limitações e características (Peterson, 2008), sendo elas:

Características:

- Nunca foi atualizado desde o seu lançamento.
- Identifica *SQL Injection* em parâmetros em URL (endereço).
- Identifica o tipo de servidor SQL em uso.
- Extrai nome de tabelas para garantir que não há falsos positivos.

Limitações

- Limite de detecção para até 1500 páginas.
- Não suporta páginas que requerem autenticação.
- Não realiza *Blind SQL Injection*.
- Não recupera conteúdo do banco de dados.

- Não suporta análise de Javascript ou Flash.
- Não testa formulários por *SQL Injection*.

Para ter um produto mais completo e atualizado da HP, é necessário aderir à solução paga oferecida por eles, a HP ASC (Application Security Center).

## ii. Websecurify

O Websecurify se destaca por continuar a ser desenvolvido e analisar diversos tipos de vulnerabilidades, incluindo *SQL Injection* e *XSS*.

Suas características incluem:

- Tirar fotos automáticas de vulnerabilidades encontradas.
- Independente de plataforma.
- Possibilita o uso de extensões.
- Suporta as vulnerabilidades do OWASP Top Ten.
- Suporte a dispositivos móveis.

## iii. Acunetix Web Vulnerability Scanner 7 Trial

O Acunetix Web Vulnerability Scanner 7 em sua versão trial é limitado comparado com a versão paga.

Suas características incluem:

- Limitado a busca de *XSS*.
- Analisa sites com conteúdo em Flash, SOAP e AJAX.
- Detecta tipo de servidor e linguagem da aplicação.
- Entre outras características inclusas na versão paga do software.

## 6.2. Aplicação Web de Teste

A aplicação possui um sistema simples de cadastro, busca e listagem de dados, onde os ataques podem ser feitos e/ou observados por eles.

Para exemplificar, seu funcionamento normal ocorre da seguinte forma, como ilustrados nas figuras 13 e 14, onde são inseridos os dados “nome”, “idade” e “código”

do usuário, ao clicar em cadastrar, é feita a listagem dos registros já existentes mais o registro feito, demonstrando então o status de que o registro foi feito com sucesso.

Figura 13 – Cadastro normal na aplicação de teste.

Fonte: próprio autor, 2011.

Figura 14 – lista de registros após cadastro normal na aplicação de teste.

#### Tabela Pessoa

Nome	Idade	Codigo
Helder J.	21	1
Rafael	38	2
Andressa	17	3
Antonio	31	4
Indeberto	56	5
Humberto	21	6
Genova	44	7
Veyron	65	8
Ettore Bugatti	65	9
Mantega	28	10
Fabio	31	11

[voltar](#)

**Status:** Fabio foi cadastrado com sucesso

Fonte: próprio autor, 2011.



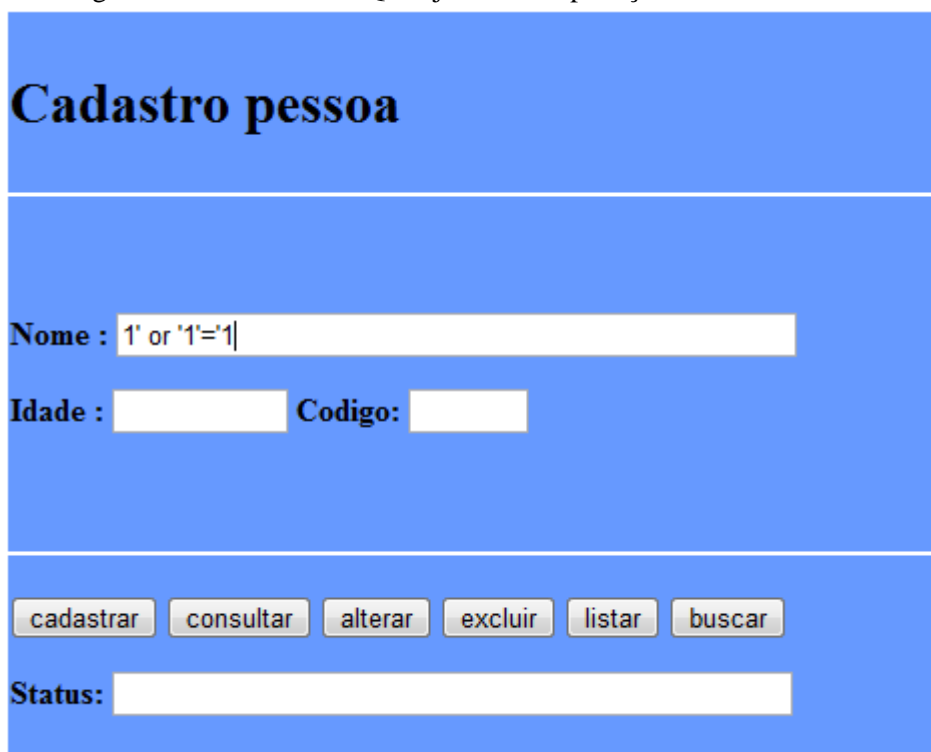
O sistema de busca funciona de forma similar ao de cadastro, no entanto só é informado o nome, com isso, se ao invés de colocar o nome ao qual se pretende fazer a busca, colocar o seguinte código:

```
1' or '1'='1
```

Ao invés de retornar apenas o registro do usuário, irá retornar todos os registros contidos no banco como visto na figura 16, pois a expressão 1 acaba não tendo relevância, e considera o or '1'='1, que é uma expressão verdadeira.

A busca, mostrado na figura 15 e 16, funciona apenas como um exemplo, já que pode-se ter acesso naturalmente a todos os registros fazendo a busca de cada elemento, ou usando a função listar, mas um sistema parecido poderia estar sendo usado para um sistema de login para buscar todas as informações de determinada conta.

Figura 15 – Busca com SQL Injection na aplicação de teste.



The image shows a web application interface with a blue background. At the top, the title "Cadastro pessoa" is displayed in a large, bold, black serif font. Below the title, there are three input fields: "Nome:", "Idade:", and "Codigo:". The "Nome:" field contains the text "1' or '1'='1|". Below the input fields, there is a row of six buttons: "cadastrar", "consultar", "alterar", "excluir", "listar", and "buscar". At the bottom, there is a "Status:" label followed by an empty input field.

Fonte: próprio autor, 2011.

Figura 16 – Lista de registros após busca com SQL Injection na aplicação de teste.

**Tabela Pessoa**

Nome	Idade	Codigo
Helder J.	21	1
Humberto	21	6
Rafael	38	2
Andressa	17	3
Antonio	31	4
Indeberto	56	5
Genova	44	7
Veyron	65	8
Ettore Bugatti	65	9
Mantega	28	10

Fonte: próprio autor, 2011.

Além disso, é possível fazer um ataque de *SQL Injection* que cause maior estrago, onde ao invés de apenas retornar todos os elementos de determinada tabela, é feito a sua total exclusão. Para isso, no cadastro por exemplo, ao invés de informar o nome que será cadastrado, pode-se colocar o seguinte comando

```
1'; drop table cliente; drop table cliente;
```

O que acarretará na exclusão da tabela cliente, como mostrado nas figuras 17 e 18. O comando “drop table cliente” neste caso é repetido duas vezes, pois o último comando ocasiona erro, o que faz com que ele não seja de fato executado. Utilizando três comandos (o último comando não tem importância), garante que o segundo será executado, sendo que o primeiro completa o requisito de informação do comando SQL original.

Figura 17 – Cadastro com SQL Injection na aplicação de teste.

**Cadastro pessoa**

Nome : 1'; drop table cliente; drop table cliente;

Idade : 21      Codigo: 21

Status:

Fonte: próprio autor, 2011.

Figura 18 – Lista de registros após cadastro com SQL Injection na aplicação de teste.

### Tabela Pessoa

Nome	Idade	Codigo

Fonte: próprio autor, 2011.

No caso do ataque XSS, como dito anteriormente, o alvo não é o banco de dados, mas sim o usuário, o ataque pode ser feito em diversas linguagens, um exemplo feito na aplicação, durante o cadastro é colocar o seguinte código:

```
<br><br>Por favor, logue no formulário abaixo antes de proceder:<form
action=\"destination.asp\"><table><tr><td>Login:</td><td><input
length=20      name=login></td></tr><tr><td>Senha:</td><td><input
length=20      name=password></td></tr></table><input
value=LOGIN></form>
```

Esse código irá criar um campo para login e senha, com um botão para efetuar a ação (embora no exemplo ele não tenha real utilidade).

Figura 19 – Cadastro com ataque XSS na aplicação de teste.



**Cadastro pessoa**

**Nome :**

**Idade :**  **Codigo:**

**Status:**

Fonte: próprio autor, 2011.

Figura 20 – Lista de registros após cadastro com XSS na aplicação de teste.

**Tabela Pessoa**

Nome	Idade	Codigo
Helder J.	21	1
Rafael	38	2
Andressa	17	3
Antonio	31	4
Indeberto	56	5
Humberto	21	6
Genova	44	7
Veyron	65	8
Ettore Bugatti	65	9
Mantega	28	10
Fabio	31	11
Por favor, logue no formulário abaixo antes de proceder: Login: <input type="text"/> Senha: <input type="text"/> <input type="button" value="LOGIN"/>	31	12

**Status:** <br><br>Por favor, logue no formulário abaixo antes de proceder:<form action=\

Login:

Senha:

foi cadastrado com sucesso">

Fonte: próprio autor, 2011.

Os comandos SQL vulneráveis no código foram feitos da seguinte forma:

```
String sql = "insert into cliente values (" + cod + ", '"+nome+"',
"+idade+)";
```

Com isso, após modifica-lo para utilizar uma das proteções citadas anteriormente, no caso, *Prepared Statement*, como mostrado no seguinte código:

```

PreparedStatement prep;
try {
    prep = con.prepareStatement("insert into cliente values(?, ?, ?)");
    prep.setInt(1, cod);
    prep.setString(2, nome);
    prep.setInt(3, idade);
    prep.executeQuery();
} catch (SQLException ex) {
    Logger.getLogger(Conexao.class.getName()).log(Level.SEVERE, null,
    ex);
}

```

Ao utilizar um dos ataques de *SQL Injection*, o mesmo não funcionou, armazenando corretamente a informação no banco de dados, como mostrado na figura 21.

Figura 21 – Lista de registros após ataque de SQL Injection em comando SQL com Prepared Statement.

### Tabela Pessoa

Nome	Idade	Codigo
Eddard Stark	41	1
Lyanna Stark	19	2
1'; drop table cliente; drop table cliente;	21	3

[voltar](#)

### Status:

1'; drop table cliente; drop table cliente; foi cadastrado com sucesso

Fonte: Próprio autor, 2011.

## 6.3. Teste para Validação

Após relatado que de fato existem vulnerabilidades de *SQL Injection* e XSS na aplicação web desenvolvida, foi feito o teste de busca de vulnerabilidades com a ferramenta Websecurify, no entanto, tirando uma informação exposta, mostrado na

figura 22, que pode ser usado para melhor definir o tipo de ataque, ele não encontrou nenhuma vulnerabilidade na aplicação.

Figura 22 – Falha encontra na aplicação pelo Websecurify.

---

#### Banner Disclosure

The server or application disclosed its type and version. This information could be used by attackers to make an educated guess about the application environment and any inherited weaknesses that may come with it.

**solution:** It is recommended to prevent the application from disclosing its type and version.

```
banner: Server: Apache-Coyote/1.1
```

```
request:
```

```
GET http://localhost:8084/WebApplicationTest/ HTTP/1.1
```

---

Fonte: próprio autor, 2011.

Devido a isso, foi feito o mesmo teste com a ferramenta HP Scrawl que busca por falha de SQL Injection, mas assim como o Websecurify, também não encontrou nenhuma falha, provavelmente por ele não detectar vulnerabilidade em formulários.

O mesmo se repetiu com a versão trial (de teste por tempo limitado) da ferramenta Web Vulnerability Scanner 7 da Acunetix, que na versão trial busca apenas por vulnerabilidade de XSS, mas não encontrou nenhuma vulnerabilidade.

No caso da ferramenta OpenTracker, o resultado foi diferente, onde foi localizado diversas vulnerabilidades, como mostrado na figura 23, que contém o relatório da ferramenta ao analisar a aplicação web.

Figura 23 – Relatório do OpenTracker em aplicação vulnerável.

Relatório de Análise

Relatório do Código Fonte. Linguagem de Programação: Java

Ocultar Relatório Numero de Linhas: 155

linha	variável	código SQL
73	cod	String sql = "insert into cliente values (" + cod + ", "+nome+", "+idade+)";
73	nome	String sql = "insert into cliente values (" + cod + ", "+nome+", "+idade+)";
73	idade	String sql = "insert into cliente values (" + cod + ", "+nome+", "+idade+)";
89	cod	String sql = "update cliente set nome = "+nome+" where codi = "+cod+";";
89	nome	String sql = "update cliente set nome = "+nome+" where codi = "+cod+";";
94	cod	String sql = "delete from cliente where codi = "+cod+";";
120	nome	String query = "select * from cliente where nome=" + nome+"";

Não foi encontrado nenhuma proteção na variável nome utilizada no comando SQL, tornando-o vulnerável a ataque de injeção SQL e XSS.

Fonte: Próprio autor, 2011.

Ao colocar um filtro na variável nome no comando sql de *insert* utilizando *Pattern*, como esperado, ele não mais o trata como vulnerável a *SQL Injection* e *XSS*, como mostra a figura 24 abaixo.

Figura 24 – Relatório do OpenTracker em aplicação vulnerável com Filtro Pattern.

Relatório de Análise

Relatório do Código Fonte. Linguagem de Programação: Java

Ocultar Relatório Numero de Linhas: 187

linha	variável	código SQL
78	cod	String sql = "insert into cliente values (" + cod + ", "+nome+", "+idade+)";
78	idade	String sql = "insert into cliente values (" + cod + ", "+nome+", "+idade+)";
96	cod	String sql = "update cliente set nome = "+nome+" where codi = "+cod+";";
96	nome	String sql = "update cliente set nome = "+nome+" where codi = "+cod+";";
111	cod	String sql = "delete from cliente where codi = "+cod+";";
152	nome	String query = "select * from cliente where nome=" + nome+"";

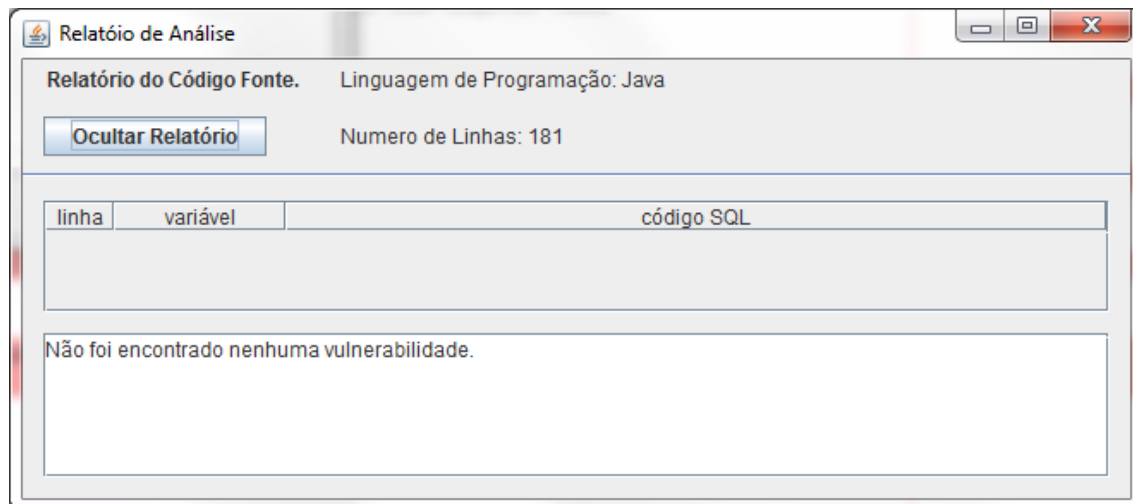
Não foi encontrado nenhuma proteção na variável nome utilizada no comando SQL, tornando-o vulnerável a ataque de injeção SQL.

Fonte: Próprio autor, 2011.



Utilizando *Prepared Statement* em todos os comandos SQL vulneráveis no código o relatório segue da seguinte forma, como mostrado na figura 25.

Figura 25 – Relatório do OpenTracker em aplicação com Prepared Statement.



Fonte: Próprio autor, 2011.

## 7. CONCLUSÕES

Com o aumento do número de ataques a aplicações web que apresentam vulnerabilidades, é importante que ocorra investimentos na melhoria da segurança dessas aplicações. Um programa que auxilia o desenvolvedor a encontrar e corrigir torna mais barato, fácil e rápido a implementação de medidas de segurança.

Apesar da grande quantidade de vulnerabilidades existentes atualmente, o documento OWASP Top Ten ajudou a entender e definir quais vulnerabilidades deveriam ser tratadas pelo módulo de detecção de vulnerabilidades. Além disso, diversas outras documentações da OWASP e de outras empresas auxiliou a ter uma melhor concepção quanto a atual proporção de ataques feitos em aplicações web.

Com base na documentação e relatórios da OWASP, IBM, entre outros, foram selecionadas as vulnerabilidades de SQL Injection e XSS devido ao seu nível de periculosidade e quantidade de ataques feitos nos últimos anos. Posteriormente foi feita a relação dessas vulnerabilidades com a linguagem Java para verificar se a vulnerabilidade está contida nas regras de negócio da aplicação.

Durante a pesquisa foram encontradas diversas ferramentas correlatas, no entanto grande parte haviam sido descontinuadas há algum tempo ou eram softwares pagos. Uma das ferramentas encontradas, chamada Websecurify, um software gratuito e que continua a ser atualizado foi usado para comparação, ele engloba uma maior quantidade de vulnerabilidades do que este trabalho verifica, além dos testes serem de caixa preta, ou seja, não é utilizado o código fonte da aplicação no mecanismo de análise.

Foram feitos testes em duas aplicações criadas, uma não funcional e outra funcional, onde o módulo de detecção de vulnerabilidades desenvolvido e integrado ao OpenTracker, conseguiu fazer a correta identificação das vulnerabilidades, mostrando sua eficácia quanto a situação provida pela aplicação testada, onde neste trabalho foi apresentado o teste na aplicação funcional.

O Websecurify não localizou nenhuma vulnerabilidade na aplicação de teste. Desta forma, foram testados o HP Scrawl e o Acunetix Web Vulnerability Scanner 7 em sua versão trial, onde foi obtido o mesmo resultado, nenhuma vulnerabilidade foi encontrada.

## REFERÊNCIAS

ABRIL. **Vírus usa nome de membro do grupo "Restart" para se espalhar pelo Twitter.** Disponível em:<<http://www.abril.com.br/noticias/tecnologia/virus-usa-nome-membro-grupo-restart-se-espalhar-pelo-twitter-594238.shtml>> Acesso em: 01 novembro 2010.

ACUNETIX. **Web Vulnerability Scanner.** Disponível em:<<http://www.acunetix.com/vulnerability-scanner/>>. Acesso em: 28 outubro 2011.

ASSOLINI, Fabio. **Falha no site Bradesco permite ataque XSS.** Julho 2008. Disponível em: <<http://www.linhadefensiva.org/2008/07/bradesco-pesquisa-inst-xss/>>. Acesso em: 23 maio 2011.

BASSO, Tania; Jino. M; Moraes R. L. O. **Uma Abordagem para Avaliação da Eficácia de Scanners de Vulnerabilidades em Aplicações Web.** Campinas; Unicamp, 2010. Disponível em: <<http://www.bibliotecadigital.unicamp.br/document/?code=000772795/>>. Acesso em: 15 maio 2011.

BIANCHI, F. R. **Aplicação em JSP e Javabeans utilizando Banco de Dados.** Outubro 2011. Disponível em:<<http://javafree.uol.com.br/artigo/10956/Aplicacao-em-JSP-e-Javabeans-utilizando-Banco-de-Dados.html>>. Acesso em: 05 outubro 2011.

CERT.BR. **Estatísticas dos Incidentes Reportados ao CERT.br.** Disponível em:<<http://www.cert.br/stats/incidentes/#2010>>. Acesso em: 22 abril 2010.

CHRISTEY, Steve; MARTIN, Robert A. **Vulnerability Type Distributions in CVE.** Maio 2007. Disponível em:<<http://cve.mitre.org/docs/vuln-trends/index.html>>. Acesso em: 13 novembro 2010.

ENDRES, R. et al. **OWASP Top 10 2007: As 10 vulnerabilidades em segurança mais críticas em aplicações Web.** 2007.

ENDRES, R. et al. **OWASP Top 10 2010: The Ten Most Critical Web Application Security Risks**. 2010.

FERREIRA, Fernando Nicolau Freitas. **Segurança da Informação**. Rio de Janeiro: Ciência Moderna, 2003.

FORRISTAL, Jeff; TRAXLER, Julie. **Hack Proofing Your Web Applications**. Rockland: Syngress, 2001.

FRANZINI, Fernando. **Vulnerabilidades de Aplicativos Web**. março 2009. Disponível em:<  
[http://imasters.uol.com.br/artigo/12132/seguranca/vulnerabilidades\\_de\\_aplicativos\\_web/](http://imasters.uol.com.br/artigo/12132/seguranca/vulnerabilidades_de_aplicativos_web/)>. Acesso em: 17 novembro 2009.

GUGIK, Gabriel. **A História dos Computadores e da Computação**. março 2009. Disponível em:< <http://www.tecmundo.com.br/1697-a-historia-dos-computadores-e-da-computacao.htm#topo>>. Acesso em: 16 maio 2011.

HP ASC. **HP Application Security Center**. Disponível em:<  
[https://www.fortify.com/products/HP\\_ASC/index.html](https://www.fortify.com/products/HP_ASC/index.html)>. Acesso em: 08 novembro 2011.

IBM. **IBM X-Force 2010 Trend and Risk Report**. Março 2011. Somers, NY.

IBM. **Rational Appscan Product Line**. Disponível em:< <http://www-01.ibm.com/software/awdtools/appscan/>>. Acesso em: 28 outubro 2011.

MEUCCI, Matteo (Ed.); KEARY, Eoin (Ed.); CUTHBERT, Daniel (Ed.). **OWASP Testing Guide**. [S.l.: s.n.], Dezembro 2008.

MORIMOTO, Carlos E. **Cracker**. junho 2005. Disponível em:  
<<http://www.hardware.com.br/termos/cracker/>>. Acesso em: 16 maio 2011.

ORACLE. **Class Pattern.** Disponível em: <<http://download.oracle.com/javase/1.4.2/docs/api/java/util/regex/Pattern.html>>. Acesso em: 06 de novembro de 2011.

ORACLE. **Welcome to the Tutorial on Defending Against SQL Injection Attacks!** Disponível em: <<http://download.oracle.com/oll/tutorials/SQLInjection/index.htm/>>. Acesso em: 25 maio 2011.

OWASP. **Blind SQL Injection.** Setembro 2009. Disponível em: <[https://www.owasp.org/index.php/Blind\\_SQL\\_Injection/](https://www.owasp.org/index.php/Blind_SQL_Injection/)>. Acesso em: 27 maio 2011.

OWASP. **Cross-site Scripting (XSS).** Outubro 2010. Disponível em: <[https://www.owasp.org/index.php/Cross-site\\_Scripting\\_%28XSS%29](https://www.owasp.org/index.php/Cross-site_Scripting_%28XSS%29)>. Acesso em: 31 maio 2011.

OWASP. **Data Validation.** Maio 2009. Disponível em: <[https://www.owasp.org/index.php/Data\\_Validation#Sanitize\\_with\\_Whitelist](https://www.owasp.org/index.php/Data_Validation#Sanitize_with_Whitelist)>. Acesso em: 06 novembro 2011.

OWASP. **Denial of Service.** Março 2010. Disponível em: <[https://www.owasp.org/index.php/Denial\\_of\\_Service/](https://www.owasp.org/index.php/Denial_of_Service/)>. Acesso em: 16 maio 2011.

OWASP. **DOM Based XSS.** Outubro 2010. Disponível em: <[https://www.owasp.org/index.php/DOM\\_Based\\_XSS/](https://www.owasp.org/index.php/DOM_Based_XSS/)>. Acesso em: 31 maio 2011.

OWASP. **Industry: Citations.** Disponível em: <<https://www.owasp.org/index.php/Industry:Citations/>>. Acesso em: 23 maio 2011.

OWASP. **SQL Injection Prevention Cheat Sheet.** Setembro 2011. Disponível em: <[https://www.owasp.org/index.php/SQL\\_Injection\\_Prevention\\_Cheat\\_Sheet](https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet)>. Acesso em: 06 novembro 2011.

PETERSON, Erik. **Finding SQL Injection with Scrawlr.** Junho 2008. Disponível em: <<http://h30499.www3.hp.com/t5/The-HP-Security-Laboratory-Blog/Finding-SQL-Injection-with-Scrawlr/ba-p/2408262>>. Acesso em: 08 novembro 2011.

ROHR, Altieres. **Fraude usa XSS para “alterar” Youtube e instalar trojan.** Outubro 2007. Disponível em: < <http://www.linhadefensiva.org/2007/10/youtube-xss-activex//>>. Acesso em: 23 maio 2011.

SCHONFELD, Erick. **Microsoft Agrees With Apple And Google: “The Future Of The Web Is HTML5”.** Abril 2010. Disponível em: <<http://techcrunch.com/2010/04/30/microsoft-html5//>>. Acesso em: 16 maio 2011.

SOPHOS. **Popular tennis websites struck in latest malware attack, Sophos warns.** Junho 2008. Disponível em:< <http://nakedsecurity.sophos.com/2011/05/22/sony-bmg-greece-the-latest-hacked-sony-site//>>. Acesso em: 22 maio 2011.

SOPHOS. **Security Threat Report 2011.** Janeiro 2011.

SOPHOS - NAKED SECURITY. **MySQL.com and Sun hacked through SQL injection.** Março 2011. Disponível em:< <http://nakedsecurity.sophos.com/2011/03/27/mysql-com-and-sun-hacked-through-sql-injection//>>. Acesso em: 22 maio 2011.

SOPHOS - NAKED SECURITY. **Sony BMG Greece the latest hacked Sony site.** Maio 2011. Disponível em:< <http://nakedsecurity.sophos.com/2011/05/22/sony-bmg-greece-the-latest-hacked-sony-site//>>. Acesso em: 22 maio 2011.

SOPHOS - NAKED SECURITY. **WordPress.com targeted by DDoS attack.** Março 2011. Disponível em:< <http://nakedsecurity.sophos.com/2011/03/03/wordpress-com-targeted-by-ddos-attack//>>. Acesso em: 22 maio 2011.

SYMANTEC CORPORATION. **Symantec Internet Security Threat Report: Trends for 2010.** Abril 2011. Mountain View, CA.

WAS. **QualysGuard Web Application Scanning.** Disponível em:< [http://www.qualys.com/products/qg\\_suite/was/>](http://www.qualys.com/products/qg_suite/was/>). Acesso em: 6 maio 2010.

WASC. **Vulnerabilidade de aplicativos Web é ‘pesadelo’ para gestores de TI.** Junho 2009. Disponível em:

<http://www.convergenciadigital.com.br/cgi/cgilua.exe/sys/start.htm?inoid=19059&sid=18>>. Acesso em: 28 novembro 2009.

WEBSECURIFY. **Web and web2.0 security.** Disponível em:<<http://www.websecurify.com>>. Acesso em: 6 maio 2010.

WEBSECURITY. **Features.** Disponível em:<<http://www.websecurify.com/features>>. Acesso em: 08 novembro 2011.

WHITEHAT SECURITY. **WhiteHat Security Quarterly Website Security Report Shows Eight out of Ten Websites Vulnerable to Attack.** Dezembro 2008. Disponível em: <[https://www.whitehatsec.com/home/news/08presssarchives/NR\\_120908stats.html/](https://www.whitehatsec.com/home/news/08presssarchives/NR_120908stats.html/)>. Acesso em: 17 maio 2011.

WHITEHAT SECURITY. **WhiteHat Security Winter Website Security Statistics Report Finds Average Website Has Serious Vulnerabilites Almost Every Day of the Year.** Março 2011. Disponível em: <[https://www.whitehatsec.com/home/news/11pressarchives/PR\\_030711statsreport.html/](https://www.whitehatsec.com/home/news/11pressarchives/PR_030711statsreport.html/)>. Acesso em: 17 maio 2011.

WHITEHAT SECURITY. **WhiteHat Website Security Statistic Report.** Março 2011. Santa Clara, CA.

WHITEHAT SECURITY SENTINEL. **Web Application Security: Sentinel Standard Edition.** Disponível em: <[https://www.whitehatsec.com/sentinel\\_services/sentinelse.html](https://www.whitehatsec.com/sentinel_services/sentinelse.html)>. Acesso em: 17 maio 2011.

WICHERS, D. et al. **OWASP AppSec DC 2009, OWASP Top10 2010rc1: The Ten Most Critical Web Application Security Risks.** 2009.

XAVIER, R. A. C. **Sistema de Rastreamento de Vulnerabilidades em Aplicações Web: Ferramenta OpenTracker**. Marília, SP: UNIVEM, 2010.