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

ROGÉRIO APARECIDO CAMPANARI XAVIER

SISTEMA DE RASTREAMENTO DE VULNERABILIDADES EM APLICAÇÕES WEB: FERRAMENTA OPENTRACKER

MARÍLIA
2010

ROGÉRIO APARECIDO CAMPANARI XAVIER

SISTEMA DE RASTREAMENTO DE VULNERABILIDADES EM APLICAÇÕES WEB: FERRAMENTA OPENTRACKER.

Trabalho de Curso apresentado ao Curso de 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. FABIO DACÊNCIO PEREIRA

MARÍLIA
2010

Xavier, Rogério Aparecido Campanari
Sistema de rastreamento de vulnerabilidades em aplicações WEB: OpenTracker/ Rogério Aparecido Campanari Xavier; Orientador: Fabio Dacêncio Pereira. Marília, SP: [s.n], 2010.
58 f.
Trabalho de Curso (Graduação em Ciência da Computação) – Curso de Ciência da Computação, Fundação de Ensino "Eurípides Soares da Rocha", mantenedora do Centro Universitário Eurípides de Marília – UNIVEM, Marília, 2010.

1. Segurança em Aplicações WEB 2. JavaBeans 3. Análise Estática

CDD:004.0684

CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

TRABALHO DE CONCLUSÃO DE CURSO – AVALIAÇÃO FINAL

Rogério Aparecido Campanari Xavier

SISTEMAS DE RASTREAMENTO DE VULNERABILIDADES EM APLICAÇÕES WEB: FERRAMENTA OPENTRACKER

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: 10 (Dez)

Orientador: Fábio Dacêncio Pereira

1°. Examinador: Jose Eduardo Santarem Segundo

2°. Examinador: Rodolfo Barros Chiaramonte

Marília, 01 de dezembro de 2010.

Dedicatória

Dedico este trabalho primeiramente a Deus, por mais está vitória em minha vida.

Dedico-o também a minha família, meus amigos e professores que sempre estiveram ao meu lado, em todos os momentos.

AGRADECIMENTOS

Agradeço as manifestações de carinho e apreço, recebidas de todos os colegas, os quais foram os artifícios e a luz inspiradora, para o sucesso deste trabalho.

Agradeço de modo particular:

A minha Mãe e ao meu Pai, por sempre estarem ao meu lado nos momentos de dificuldade, e pela educação carinho e afeto que me deram em todos esses anos.

Aos profs. Fabio Dacêncio, Leonardo Botega e Maurício Duarte, por sempre estarem presentes e me "aturarem" a maioria das tardes na faculdade, o que foi imprescindível para o desenvolvimento e conclusão deste trabalho.

Obrigado por tudo.

XAVIER, Rogério Aparecido Campanari. **Sistema de rastreamento de vulnerabilidades em aplicações WEB: OpenTracker**. 2010. 58 f. Trabalho de Curso (Bacharelado em Ciência da Computação) – Centro Universitário Eurípides de Marília, Fundação de Ensino "Eurípides Soares da Rocha", Marília, 2010.

RESUMO

Este projeto foi concebido para auxiliar na análise de segurança de aplicações WEB, na forma de ferramenta, onde o programador possa verificar o nível de segurança do seu código. A ferramenta denominada OpenTracker é capaz de analisar vários formatos de códigos como: Java (concluído atualmente), ASP.NET e PHP. Esta permite a interação do desenvolvedor durante o processo de análise, diminuindo o número de falsos positivos, além possibilitar a integração de módulos, criados para o tratamento de vulnerabilidades de outras linguagens de programação. Por fim, é possível verificar por meio de relatórios os problemas encontrados para uma determinada aplicação WEB em Java. O projeto em questão apresenta o desenvolvimento do Sistema de Rastreamento de Vulnerabilidades (SRV) que compõe a ferramenta OpenTracker, responsável pela identificação das linhas de códigos relevantes utilizadas no processo de análise das vulnerabilidades.

Palavras-Chave: Segurança em Aplicações WEB. JavaBeans. Análise Estática.

XAVIER, Rogério Aparecido Campanari. **Sistema de rastreamento de vulnerabilidades em aplicações WEB: OpenTracker**. 2010. 58 f. Trabalho de Curso (Bacharelado em Ciência da Computação) – Centro Universitário Eurípides de Marília, Fundação de Ensino "Eurípides Soares da Rocha", Marília, 2010.

ABSTRACT

This project is designed to assist in analyzing the security of web applications in the form of tool, where the programmer can check the security level of your code. A tool called OpenTracker is capable of analyzing various code formats like Java (now completed), ASP.NET and PHP. This allows the interaction of the developer during the analysis process, reducing the number of false positives, it also allows the integration of modules created for the treatment of vulnerabilities of other programming languages. Finally, you can check by reporting problems to a specific web application in Java. The project in question presents the development of the Tracking System Vulnerabilities (SRV) which comprises the OpenTracker tool, responsible for identifying the relevant lines of codes used in the analysis of vulnerabilities.

Keywords: Web Applications Security. JavaBeans. Static Analysis.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API: Application Programming Interface

ASP: Application Service Provider

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

CG: Controle Geral

GUI: Graphical User Interface

HTTP Hyper Text Transfer Protocol

HTTPS Hyper Text Transfer Protocol Secure

HTML: HyperText Markup Language

IDE: Integrated Development Environment

OWASP: Open Web Application Security Project

PCI-DSS: Security Standards Council Data Security Standard

PCI PA-DSS: Security Standards Council Payment Application Data Security Standard

PHP: Hypertext Preprocessor

RATS: Rough Auditing Tool for Security

SQL: Structured Query Language

SRV: Sistema de Rastreamento de Vulnerabilidades

URL: Universal Resource Locator

XSRF: Scross-Site Request Forgery

XSS: Cross-Site Scripting

XSSI: Cross-Site Script Inclusion

SUMÁRIO

INTRODUÇÃO

A sociedade depende cada vez mais de sistemas confiáveis, conectados e onicientes, em consequência acabam por gerar sistemas mais complexos, onde essa complexidade causa situações inesperadas (SEVESTRE, 2009).

A evolução das plataformas e tecnologias de desenvolvimento de aplicações WEB colaborou para a migração de aplicações desktop *stand-alone* (normalmente não compartilham seu modo de execução), para aplicações WEB (COELHO, 2007, p.1), aditando diversas características como: processamento ao lado do servidor, interface gráfica limitada à capacidade do browser do usuário, facilidade de instalação e atualização entre vários clientes. Em contrapartida, as aplicações desktop não são muito eficientes quando se trata de instalações e atualizações em diversos clientes, cuja compatibilidade entre sistemas e hardwares do usuário é relevante para o funcionamento da aplicação na maioria das vezes. Devido ao aumento do número de usuários acessando a Internet e aplicações WEB, se torna fundamental navegar, distribuir e acessar informações de maneira segura.

Segundo Chmielewski, et al (2007) para obtenção de um código seguro é fundamental a preocupação com a codificação no aspecto de segurança, proceder as revisões manuais de códigos é dispendiosa e difícil, dependendo unicamente da experiência do desenvolvedor, dessa forma é possível realizar um trabalho em conjunto com uma ferramenta, capaz de identificar grandes números de vulnerabilidades de códigos, fazendo assim da revisão, uma oportunidade de documentação da cobertura do código, a segurança em áreas específicas do código, bem como, enriquecer o conhecimento organizacional e aumentar a conscientização da segurança.

Muitas vezes por falta de tempo no prazo de entrega, necessidade de otimização de custo, falta de conhecimento e por carência de uma boa ferramenta de auxílio, os programadores acabam por gerarem códigos com acúmulos de vulnerabilidades, permitindo que as aplicações sejam expostas a ataques com maior facilidade (TEIXEIRA, 2007, p.1).

Atualmente nos cursos de graduação não é abordado suficientemente este assunto, fazendo com que muitos programadores por falta de conhecimento ou simples erro humano, às vezes gerem códigos vulneráveis. Esse é o cenário que motiva a realização deste trabalho.

PROBLEMÁTICA

Preocupar-se com a segurança de aplicações WEB não é um objetivo fácil de cumprir, devido à complexidade dos sistemas atuais que alcançam milhares de linhas de códigos, contendo uma significativa quantidade de vulnerabilidades. Uma vez que as vulnerabilidades do software são exploradas pelos atacantes com o intuito de obtenção de informações confidenciais e invasão de redes coorporativas, podem causar desde a indisponibilidade do sistema até o controle total do computador. Este cenário piora quando não é adotado um ciclo de desenvolvimento de software seguro, gerando especificações inseguras e configurações vulneráveis das plataformas ocultas (UTO; MELO, 2009, p.238).

Uma pesquisa feita pelo grupo OWASP (2010) aponta a injeção de falha (*SQL Injection*) onde os dados hostis do atacante trapaceiam o interpretador levando-o a executar comandos que não entende ou a alterar dados e o *Cross-Site Scripting* (XSS) que permite aos atacantes executarem *scripts* (programa ou uma sequência de instruções) no *browser* da vítima, como os principais ataques que afetam basicamente sistemas WEB. Essas vulnerabilidades podem ser evitadas aumentando a preocupação com a segurança no desenvolvimento do código fonte, mas como fazer isso visando o tempo do prazo de entrega, a otimização de custo e a falta de conhecimento dos desenvolvedores e o simples erro humano?

JUSTIFICATIVA

Para resolver o problema citado anteriormente, é permitido ao programador o auxílio na análise de possíveis falhas de segurança em sua aplicação, com visualização dessas vulnerabilidades encontradas, propondo correções na forma de relatório. Dessa forma é esperado a geração de códigos mais robustos, otimizados e seguros.

Para este feito a ferramenta denominada OpenTracker, será capaz de identificar grande quantidade de vulnerabilidades contidas no código do desenvolvedor, através de uma análise estática, apresentando seu código, analisado adjunto de uma tabela contendo as linhas com possíveis vulnerabilidades.

Diferente das demais ferramentas de análise estática encontradas no mercado atualmente, como o “RATS” (Rough Auditing Tool for Security) caracterizada por analisar códigos C, C++, Perl, PHP e Python ou a ferramenta “Pixy” que possui um analisador online

e para download da linguagem PHP, a OpenTracker possui uma estrutura capaz de integrar diferentes módulos, cada um tratando das vulnerabilidades de uma linguagem de programação diferente, beneficiado por um banco de dados que armazenará uma vasta quantidade de dados de tratamentos para vulnerabilidades relacionadas àquela linguagem, necessitando unicamente seguir os padrões pré-definidos para o desenvolvimento do módulo. O Controle Geral (CG) da ferramenta OpenTracker, é quem será responsável pela comunicação e a utilização dos serviços disponibilizados pelos módulos.

OBJETIVOS

Implementar o Sistema de Rastreamento de Vulnerabilidades (SRV) da ferramenta OpenTracker, que identifique as linhas de códigos importantes, utilizadas nas análises sobre um determinado tipo de vulnerabilidade. Esta deve permitir a interação do usuário durante a execução e também apresentar um relatório com as linhas relevantes encontradas, identificando-as no código fonte.

A seguir os principais objetivos específicos do projeto:

- Criar recursos para relatórios de vulnerabilidades: o sistema deve identificar as posições e o conteúdo das ocorrências encontradas e destacá-las no próprio código fonte;
- Estatística de falsos positivos da análise: deve permitir através da interação do usuário determinar a relevância dos métodos encontrados pela análise no sentido de aprimorar o tratamento de segurança, resultando em uma análise com redução de falsos positivos;
- Estruturas baseadas em componentes: as listas geradas pelo SRV como, por exemplo: variáveis, métodos, ocorrências encontradas, hierarquia de escopos, devem estar disponíveis sempre que requisitadas;
- Documentação para evolução da ferramenta *Open Source*: desenvolver a documentação para a criação de uma comunidade para evolução da OpenTracker (em andamento).

METODOLOGIA

Para o projeto e desenvolvimento da ferramenta OpenTracker, foram organizadas algumas tarefas metodicamente para concepção da ferramenta proposta.

Inicialmente foi realizado um estudo teórico de conceitos e tecnologias correlatas ao projeto, destacando as seguintes frentes: compiladores (análise léxica, sintática e semântica), componentes JavaBeans, segurança de sistemas e aplicações WEB, tipos de ataques mais frequentes nas aplicações WEB e ferramentas correlatas de análises estáticas para detecção de vulnerabilidades.

Uma vez finalizada a etapa de estudos e revisão bibliográfica, foi realizado a análise de requisitos da ferramenta proposta. Neste trabalho destaca-se o tratamento de código de aplicações WEB em Java.

A ferramenta foi desenvolvida utilizando a linguagem Java por ser uma linguagem portável, sendo completamente orientado a objetos com forte suporte a técnicas adequadas de engenharia de software e permitindo através das API`s que são bibliotecas, composta por um conjunto de classes (DEITEL; DEITEL, 2003), o desenvolvimento de componentes.

Para o desenvolvimento e a integração dos componentes (*Bean*s) optou-se a utilização da API JavaBeans (ENGLANDER, 1997), por ser capaz de definir um modelo de componente de software para Java, onde é possível integrar esses componentes em uma única aplicação.

O projeto proposto foi desenvolvido através do IDE NetBeans. Após o desenvolvimento foram realizados testes funcionais utilizando um conjunto de programas artificiais e na sequência aplicados a programas reais desenvolvidos em Java. Por fim, será criada uma documentação formal para a ferramenta que será disponibilizada como *Open Source*.

CAPÍTULO 1 – VULNERABILIDADES EM APLICAÇÕES WEB

Atualmente nenhuma aplicação está imune a *bugs* ou falhas de segurança incluindo aplicações WEB, uma vez que a Internet tem assumido um papel importante na vida das pessoas ultrapassando da utilização da troca de emails e busca de textos, a downloads de vídeos de alta definição e arquivos de grande tamanho. Consequentemente, acompanhar essa evolução e aprender a se defender dos riscos que trazem essas novas tecnologias, se torna um fator preocupante considerando que, diferentes tipos de ameaças vindas da internet afetam casas e empresas, com objetivos específicos desde roubo de informações sigilosas a ataques aos servidores WEB.

Outro fator importante a se abordar, é a falta de preocupação com requisitos de segurança (CERON at al, 2008), muitas vezes considerado um dos primeiros requisitos não funcionais a ser descartado pelo projeto de uma empresa, devido a problemas no desenvolvimento da engenharia de software, necessidade de entrega de um sistema atrasado, corte de custos entre outros fatores. Agravando ainda mais esse cenário, com a utilização de mecanismos de buscas como ferramentas utilizadas pelos atacantes para encontrar sites vulneráveis, tornou-se um risco ainda maior, no qual se torna trivial acrescer prioridade e atenção nesse requisito não funcional que é a segurança de uma aplicação (CERON at al, 2008).

No Brasil, o Centro de Estudos, Resposta e Tratamento de Incidentes de Segurança no Brasil (CERT.BR, 2010) 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.

O primeiro ataque a internet ocorreu por volta de 1988, um *Worm* (programa auto-replicante) que explorava um erro de programação no *fingerd* (comando UNIX que permite descobrir informações sobre outro usuário com uma conta UNIX), levando aos programadores a verificarem e eliminarem funções *gets()* dos servidores afetados por ele, fazendo com que os demais fornecedores de UNIX repetissem a ação, alertando os programadores e cientistas sobre outras possíveis funções vulneráveis como a *strcpy()* (copia a String origem, para a String destino) (PEREIRA, 2000).

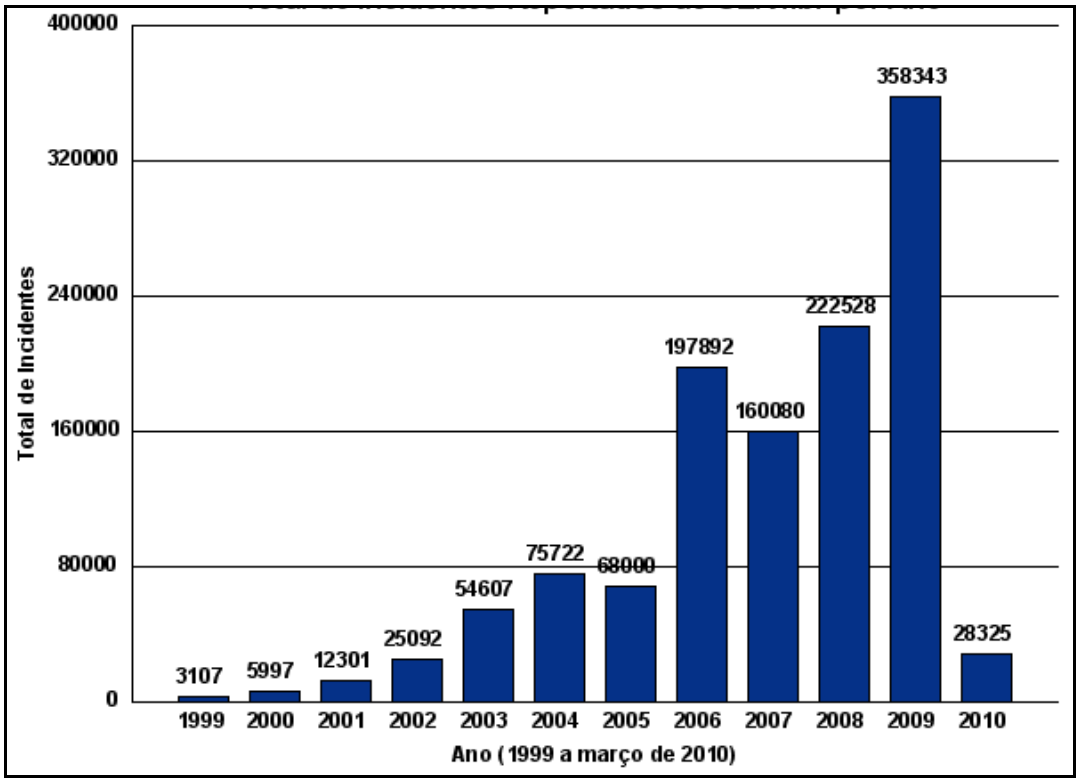

Figura 1 – Incidentes Reportados ao CERT.br por Ano (CERT.br, 2010).

1.1 Ferramentas de Aprendizado de Vulnerabilidades

Para que fosse possível aprender na prática a encontrar, explorar e principalmente a se defender de vulnerabilidades *SQL Injection* e *XSS*, foram utilizadas de ferramentas de aprendizado como *WebGoat* desenvolvida pelo grupo OWASP e de uma *codlab* denominada Web *Application Exploits and Defenses* que são discriminadas a seguir.

1.1.1 OWASP

O grupo *Open Web Application Security Project* é uma organização mundial, sem fins lucrativos, que visa divulgar aspectos de segurança em aplicações WEB, para que pessoas e empresas possam devidamente avaliar e tomar decisões, com informações sobre os verdadeiros riscos encontrados nesses ambientes (OWASP, 2010).

A entidade contribui com mais de 150 capítulos locais, separados entre os cinco continentes, todos abertos e gratuitos, promovem conferências internacionais e encontros sobre o tema, divulgando também diversos projetos, que variam desde guias à implementação

segura até ferramentas (OWASP, 2010). Sobre o grupo OWASP é importante citar os projetos descritos a seguir:

- *Top Ten* – é uma lista das 10 vulnerabilidades mais críticas em aplicações WEB, escrita por membros do grupo, compostos de especialistas de segurança de todo o mundo, que utilizam suas experiências práticas. O *Top Ten* aborda o gerenciamento de riscos e não somente como evitar vulnerabilidades, por essa razão, foi adotado os padrões PCI DSS e PCI PA-DSS, para configurar como os itens mínimos a serem considerados na codificação segura de aplicações WEB (UTO; MELO, 2009, p.241).
- *Code Review Project* – é um livro que ensina a identificar vulnerabilidades de aplicações WEB, através da revisão manual do código fonte, contendo diversos exemplos nas linguagens, Java, C, C++ e ASP (OWASP, 2008).
- *WebGoat* – é uma aplicação desenvolvida em J2EE intencionalmente vulnerável, programada para ensinar lições de segurança de aplicações WEB e testes de invasão (OWASP, 2010).
- *WebScarab* – é um framework para análise de aplicações WEB que se comunicam com protocolos HTTP e HTTPS, desenvolvido em Java, sua utilização mais comum é funcionar como um interceptador de Proxy, permitindo ao operador analisar e modificar solicitações do browser, e alterar respostas retornadas do servidor antes de serem recebidas pelo browser (OWASP, 2010).

1.1.2 Web Application Exploits and Defenses

"*De acordo com o ditado popular: é preciso um hacker para pegar um hacker"* (Leban Bruce, 2010), a Google desenvolveu recentemente (2010) uma *codlab* intitulado "*Web Application Exploits and Defenses*", cujo objetivo é similar ao da ferramenta *WebGoat* descrita anteriormente, demonstrando que esse ditado faz sentido, pois para se desenvolver uma aplicação segura é necessário descobrir quais são as possíveis falhas ao qual ela está sujeita.

Desenvolvida em torno da *Gruyere* (aplicação WEB que permite publicar textos e armazenar arquivos), essa *codlab* possui uma variedade de vulnerabilidades propositais, permitindo descobrir através de uma série de dicas, a forma como os hackers encontram

vulnerabilidades, como as exploram e principalmente ensinando aos desenvolvedores como se defender desses ataques.

As vulnerabilidades tratadas por ela envolvem *cross-site scripting* (XSS), *cross-site request forgery* (XSRF), *cross-site script inclusion* (XSSI), entre outras e apresentam de forma simples erros que podem levar a divulgação de informações, negações de serviços e execuções remotas de códigos (LEBAN, 2010).

1.2 Principais Técnicas de Ataques

Nesta seção, encontram-se algumas vulnerabilidades em ordem do maior ao menor risco, considerado pelo grupo OWASP no projeto *Top Ten* 2010, as vulnerabilidades aqui apresentadas seguem a seguinte abordagem: uma breve descrição da vulnerabilidade, as causas da ocorrência, seguido de uma ilustração do ataque e possíveis contramedidas que podem ser tomadas para evitar essa vulnerabilidade na aplicação. As informações abordadas a seguir baseiam-se às referências de (OWASP, 2010) e (UTO; MELO, 2009).

1.2.1 Injeções de Falhas

Injeções de falhas tais como, SQL, OS e LDAP (WASC, 2010), ocorrem quando são enviados dados ao interpretador, como parte de comandos ou consultas, onde não são feitas as devidas validações das entradas de dados. Essas informações maliciosas confundem o interpretador que irá executar os comandos manipulados e enviar dados modificados, permitindo ao atacante modificar, ler e até mesmo excluir dados disponíveis na aplicação, podendo comprometer toda a aplicação e os sistemas relacionados.

Dado o seguinte comando SQL presente na tabela 1:

Tabela 1 – Exemplo de Injeção de Falha.

sComando= "Select * from user_data where last_name='"+inputLastName+"';",	
Comando SQL presente no código fonte.	
Select * from user_data where last_name='Smith';	Select * from user_data where last_name=''or 1=1;--'
Comando Esperado	**Injeção de Falha**

Fonte: (Próprio Autor, 2010).

É esperado como valor da variável *inputLastName* um nome como por exemplo "*Smith"*, contudo uma aplicação vulnerável com *SQL Injection* permite executar *"'or1=1;--'"*, fazendo com que a condição da clausura *where* da SQL seja verdadeira, retornando por exemplo informações sigilosas como é mostrado na figura 3 ao invés de retornar os valores esperados apresentado pela figura 2.

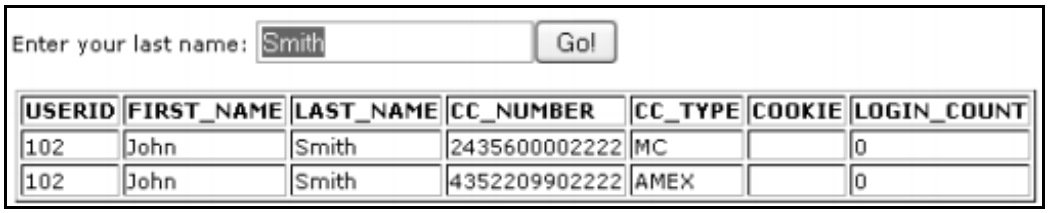

Figura 2 – Execução Normal de uma Aplicação (UTO; MELO, 2009).

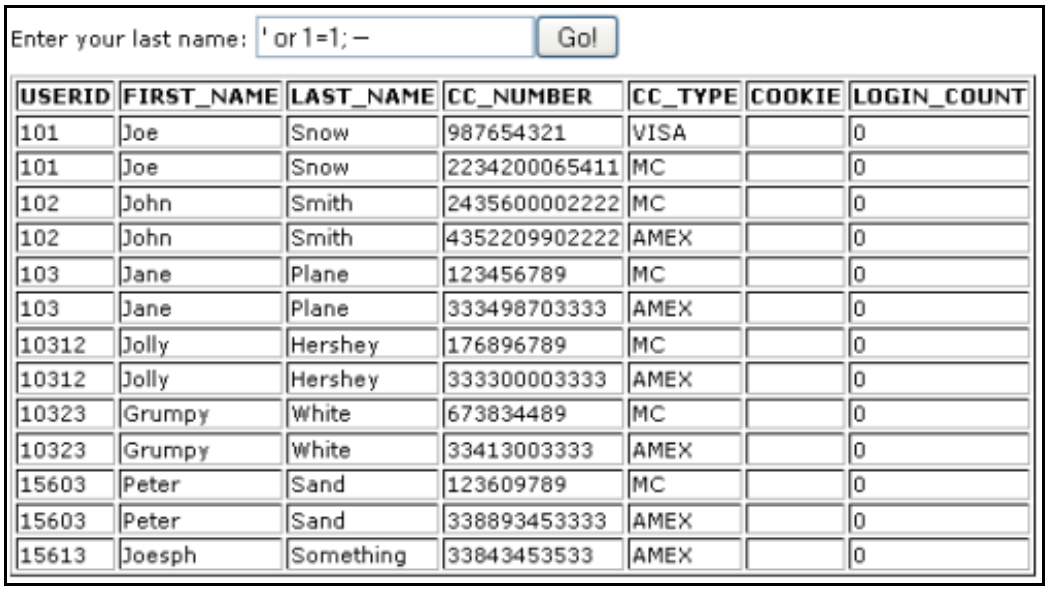

Figura 3 – Dados Revelados pela Injeção de Falha (UTO; MELO, 2009).

A prevenção pode ser feita de diversas maneiras como, por exemplo, realizando uma validação na entrada de dados, impedindo caracteres especiais nos campos de dados ou através da utilização de uma API`s de segurança como, por exemplo, a JDBC (FARIAS, 2009).

1.2.2 Cross-Site Scripting (XSS)

Essa vulnerabilidade está presente em uma aplicação quando se obtém informações do usuário e as envia de volta ao navegador, sem executar validação ou codificação da informação, sendo um conjunto de inserções HTML que afetam os Frameworks de aplicações

vulneráveis, o XSS permite aos atacantes executarem scripts maliciosos dentro do contexto de um site confiável, permitindo sequestrar sessões de usuários, inserirem conteúdos hostis, desfigurarem WEB sites, obterem controle do navegador do usuário e conduzirem ataques de roubo de informações. A figura 4 apresenta um exemplo de ataque XSS.

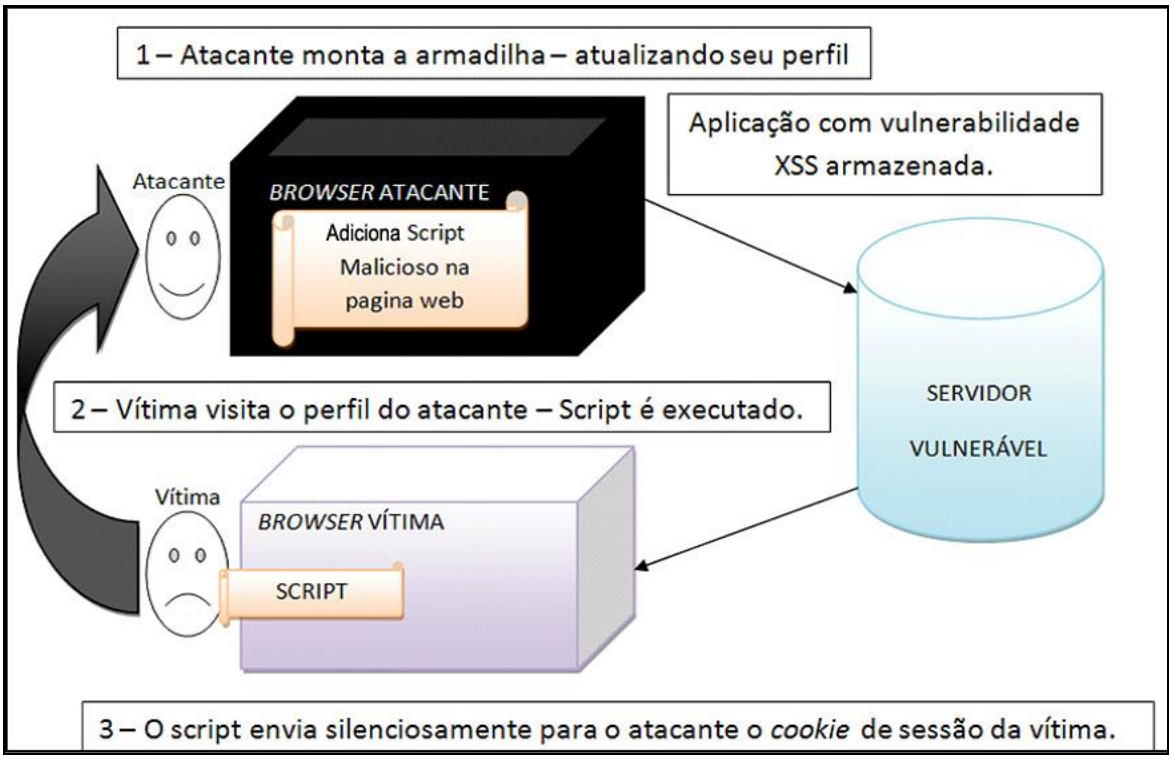

Figura 4 – Funcionamento de um Ataque XSS (Próprio Autor, 2010).

A forma mais eficaz de se prevenir contra esse tipo de ataque é incluir escapes de dados com base no contexto HTML como: *body*, *attribute*, *JavaScript*, CSS, ou URL, não necessário se a aplicação já utilizar um Framework para interface de usuário.

1.2.3 Quebra de Autenticação e Gerenciamento de Sessão

Está vulnerabilidade ocorre em aplicações onde funções relacionadas à autenticação e gerenciamento de sessão são mal implementadas, permitindo aos atacantes comprometer senhas, chaves, índices, *tokens* de sessão ou explorar brechas de implementação para assumir a identidade de outros usuários, conforme é visualizado na figura 5.

Figura 5 – Exemplo de Aplicação de Timeout não Definida (Próprio Autor, 2010).

Para evitar essa vulnerabilidade, deve-se utilizar um conjunto único de autenticação e controles de gerenciamento de sessão afim de: cumprir requisitos de autenticação, obter uma interface simples para os desenvolvedores e evitar falhas XSS.

1.2.4 Referência Insegura Direta a Objetos

Acontece quando o desenvolvedor expõe referências a um objeto implementado internamente como: arquivos, diretórios, registros do banco de dados ou chaves, na forma de uma URL ou parâmetro de formulário, permitindo ao atacante acessar outros objetos sem autorização através destas referências, a não ser que, exista um mecanismo de controle de acesso. Em uma aplicação que utiliza dados não verificados em chamadas SQL para acessar uma determinada conta, é possível realizar o ataque demonstrado pela figura 6.

Para impedir a existência desse tipo de ataque, é necessário verificar se a aplicação não permite referências diretas a objetos que sejam manipulados por atacantes, como por exemplo, usar referência de objetos indiretos através do usuário ou sessão.

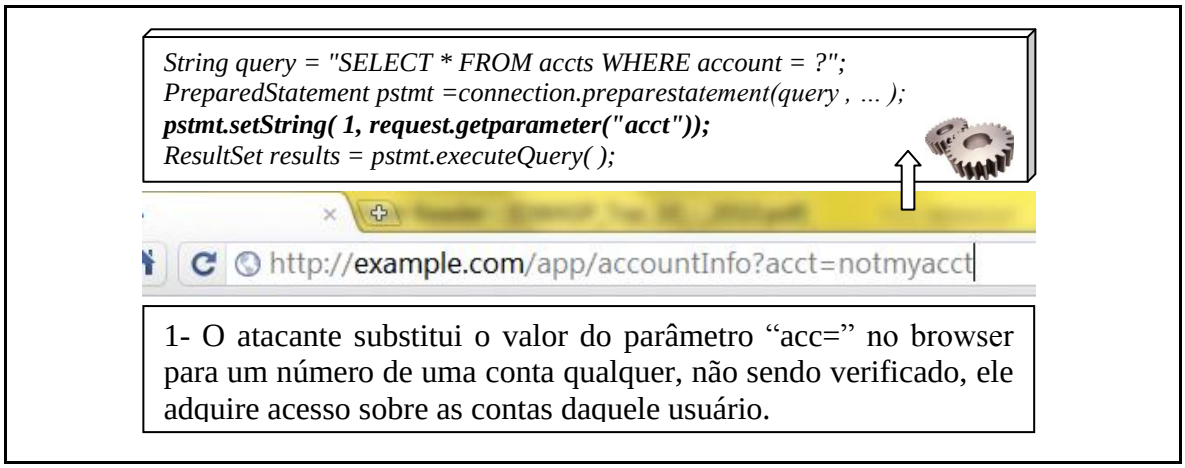

Figura 6 – Ataque de Referência Insegura Direta a Objetos (Próprio Autor, 2010).

1.2.5 Cross-Site Request Forgery (CSRF)

Também conhecido como *Session Riding*, ataque *One-Click* e *Automation Attack*, este ataque força o navegador autenticado da vítima a realizar ações automatizadas sem o conhecimento e consentimento da mesma. A figura 7 apresenta um ataque CSRF.

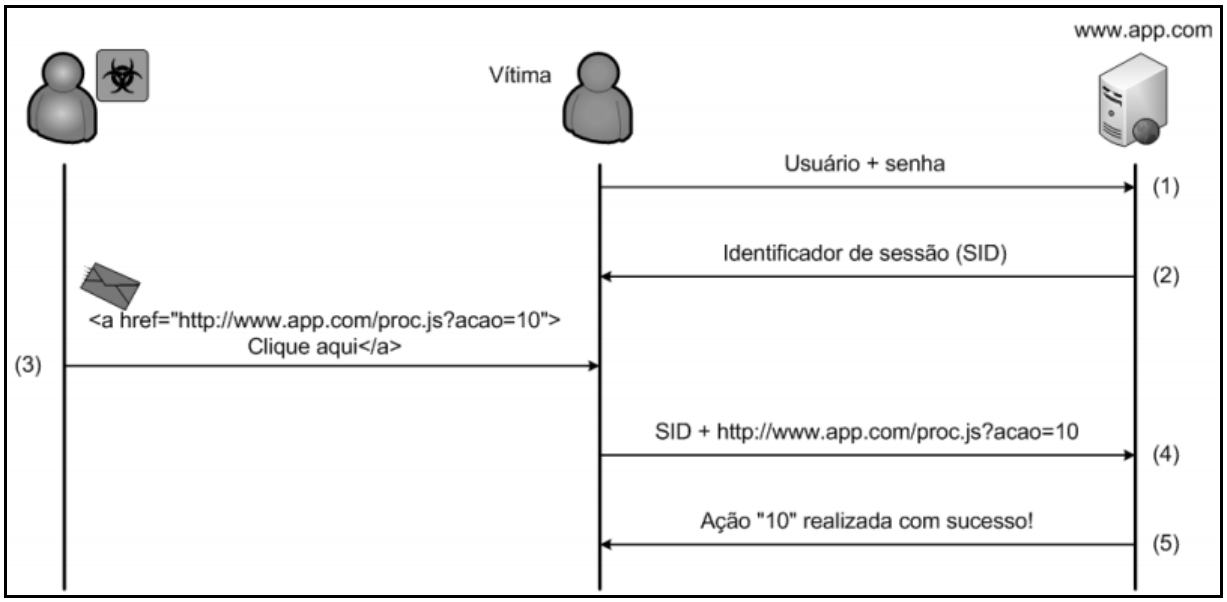

Figura 7 – Exemplo de um Ataque CSRF (UTO; MELO, 2009).

Na primeira etapa o usuário autentica-se em uma página para ter acesso a recursos protegidos do sistema, na segunda, é feita a validação do usuário e é atribuído um identificador único à sessão daquele usuário. Na terceira etapa, um atacante envia o *link* contendo uma ação que será executada pelo usuário induzido a clicar no *link*, dessa forma, na

quarta etapa o usuário acessa o link que aceita a requisição, enviando suas credenciais, concluindo a quinta e última etapa que executa a requisição (UTO; MELO, 2009).

Para se proteger dessa ameaça, deve-se utilizar um *token* que seja único por sessão ou pedido de usuário no *body* ou URL de cada solicitação HTTP através de um campo oculto, evitando a inclusão da ação na URL (OWASP, 2010).

1.3 Análises de Código

Análises de código permitem criar códigos mais seguros, devido à identificação e localização de falhas e *bugs.* Ferramentas de análise tais como: análise estática, análise binária e engenharias reversas, aprimoram a segurança ainda no processo de desenvolvimento. Priorizada por esse projeto, a análise estática é importante, devido a varias linguagens de programação não serem projetadas com recursos de segurança necessários, tornando os sistemas ainda mais vulneráveis (MENEZES, 2009)(OWASP, 2008).

Ferramentas de análise estáticas são parecidas com análises manuais com a diferença que elas não necessitam de pessoas especializadas, elas diminuem a propagação de erros para outras fases por serem utilizadas no processo de concretização do projeto, contribuindo para diminuição dos custos. São rápidas e fornecem um maior impacto no ciclo de desenvolvimento de um produto final. Contudo, este tipo de ferramenta possui um problema devido à emissão de falsos positivos (ocorrências encontradas pela ferramenta que na verdade não representam vulnerabilidades), devendo ser configuradas corretamente de acordo com as regras de negocio maximizando os resultados (TEIXEIRA, 2007, p.26). A origem desse tipo de ferramenta iniciou-se na época do advento do UNIX quando eram utilizadas para encontrar padrões no conteúdo de um arquivo (MENEZES, 2009).

CAPÍTULO 2 – INTRODUÇÃO A COMPILADORES, COMPONENTIZAÇÃO JAVABEANS E TRABALHOS CORRELATOS

Neste capítulo é apresentado um estudo sucinto de compiladores envolvendo análise léxica, análise sintática e análise semântica. Também são abordados conceitos relevantes estudados sobre a componentização JavaBeans, onde ambos os estudos tiveram participação significativa no desenvolvimento do Sistema de Rastreamento de Vulnerabilidades (SRV). Concluindo este capítulo serão tratados os trabalhos correlatos funcionais encontrados na WEB atualmente (2010).

2.1 Compiladores

O estudo realizado teve papel fundamental no desenvolvimento do SRV, pois permitiu a identificação de informações importantes através da análise léxica e sintática, bem como, o tratamento de sobrecarga que será abordado na seção 3.4.6, com o uso da análise semântica.

2.1.1 História e Definição de Compiladores

Antes da existência dos compiladores havia um analisador sintático que avaliava a sintática de um programa, de uma linguagem de programação. A partir da década de 50, surgiu o primeiro compilador para a linguagem Fortran, desenvolvido pela equipe de John Bacus, dando início a uma série de estudos sobre os compiladores (TEIXEIRA, 2007, p.12).

Segundo TEIXEIRA, 2007, p.12-13 compiladores: *"É um programa que traduz uma representação de alto nível para uma representação de baixo nível."*, no sentido de transformar uma linguagem de programação em uma linguagem de máquina, e gerando um executável. Dessa forma, compiladores executam um conjunto de fases definidas, analisam, transformam o código do programa ou cedem seus resultados a uma fase posterior.

2.1.2 Análise Léxica

Análise Léxica é responsável por ler o arquivo fonte em busca de unidades significativas chamadas de *tokens* (padrões de caracteres com um significado específico em um código fonte) instanciadas por lexemas (ocorrências de um *token* em um código fonte,

também são chamadas de átomos). Ela também “varre” o arquivo de entrada, eliminando comentários e caracteres indesejáveis ao agrupar caracteres com um papel bem definido (EDUARDO, 2004).

2.1.3 Análise Sintática

O analisador léxico não se preocupa em verificar se a ordem dos *tokens* encontrados é válida ou não. Essa função é executada pelo analisador sintático, que verifica a sequência de símbolos contidos no programa (EDUARDO, 2004), transformando *tokens* recebidos em uma árvore abstrata baseada na estrutura sintática do programa, que são construídas através das regras gramaticais da linguagem (TEIXEIRA, 2007, p.13).

2.1.4 Análise Semântica

Essa fase é responsável por verificar se existem incoerências quanto ao significado das construções utilizadas pelo programador, conhecida através de verificação dos aspectos semânticos do programa (EDUARDO, 2004) como, por exemplo: se as variáveis são declaradas antes de serem usadas ou a consistência entre tipos, etc (TEIXEIRA, 2007, p.13). Análise semântica não utiliza mais o código fonte para realizar a verificação, ela é feita através da árvore gerada pela análise sintática.

Todas as fases descritas são executadas sequencialmente (TEIXEIRA, 2007, p.13), considerando que não foram encontrados erros sintáticos ou semânticos, o compilador então executa a fase da criação do programa objeto (refletem as instruções de baixo nível, os comandos do programa fonte) (EDUARDO, 2004).

2.2 JavaBeans

Para que fosse possível a criação dos módulos da ferramenta OpenTracker, optou-se pelo desenvolvimento baseados em componentes, por permitir interoperabilidade (capacidade de se comunicar de forma transparente) entre eles e o sistema e também a futura criação de novas técnicas e métodos para detecção de vulnerabilidades, sem que seja necessário modificar a estrutura da OpenTracker. A seguir é abordado o conceito de JavaBeans, optado para o desenvolvimento dos componentes.

2.2.1 Conceitos e Apresentação ao JavaBeans

Antes de iniciar o conceito de JavaBeans é necessário abordar componentes que são elementos autossuficientes de software que podem ser controlados de forma dinâmica e utilizados para montar aplicações, também devem ser capazes de interagir de acordo com o conjunto de regras estabelecidas, se comportando de maneira esperada. Ex: Os trabalhadores de uma empresa (componentes) é a parte funcional já a direção da empresa que é quem cuida da ordem e a estrutura (ORACLE, 2010).

JavaBeans é uma plataforma independente de criação de componentes escrito em linguagem Java. Sua origem é dada através dos esforços colaborativos da indústria, com o objetivo de permitir aos desenvolvedores criar componentes reutilizáveis em linguagem de programação Java. A API JavaBeans permite criar esses componentes independente dos componentes da plataforma, sendo possível combiná-los em *applets* (aplicativos).

Um componente JavaBeans é conhecido como *Bean* e pode ser considerado como uma unidade de software, dinâmico no sentido de ser possível alterar, personalizar e salvar (persistir) sua nova manipulação visual através do designer de uma ferramenta de construtor, utilizando a janela de propriedade do *Bean* (ENGLANDER, 1997).

2.2.2 Características

A ferramenta de construção (*Builder tools*) permite revelar as características do *Bean* como propriedades, métodos e eventos através de um processo chamado introspecção (“olhar dentro do *Bean*”) que pode ser feito de duas maneiras: através da classe *Introspector* que obtêm as características do *Bean* pela análise de acordo com as regras específicas baseada no padrão de projeto, ou, através das informações da classe do *Bean* que implementa a interface chamada *BeanInfo*, que explicitamente enumera as características do *Bean* que deverão ser expostas a ferramenta de construção (ENGLANDER, 1997).

A seguir um breve resumo sobre as características de componentes JavaBeans.

2.2.3 Propriedades.

As propriedades representam as aparências e características do comportamento do *Bean* que podem ser mudadas em tempo de design. A ferramenta de construção de

introspecção de um *Bean* descobre e expõem essas propriedades para manipulação, permitindo realizar personalizações de duas maneiras: usando o editor de propriedade ou usando algum tipo de customização mais sofisticado.

Alguns exemplos de propriedades dos *Beans* envolvem a cor, rótulo, tipo de letra, tamanho da fonte e tamanho do display, onde as propriedades podem ser definidas nos seguintes aspectos.

- *Simple*: a propriedade possui um único valor onde alterações independem das mudanças de qualquer outra propriedade.
- *Indexed*: caracterizado por suportar diversos valores ao invés de somente um como acontece no *simple*.
- *Bound*: propriedade que trata quando acontece alguma mudança de propriedade que resultará em uma notificação a ser enviada algum outro *Bean*.
- *Constrained*: cada mudança de propriedade será avaliada por outro *Bean* dizendo se a mudança é apropriada ou não podendo rejeitar a mudança.
- *Writable*: as propriedades podem ser alteradas definidas em três modelos: *standard*, *expert* e *preferred*.
- *Read Only*: a propriedade não pode ser modificada.
- *Hidden*: as propriedades podem ser alteradas, porem elas não são divulgadas na classe do BeanInfo (ORACLE, 2010).

2.2.4 Eventos

A comunicação entre *Beans* é feita através de eventos, onde o *Bean* a receber eventos (*listener*) registra seu interesse com o *Bean* que dispara o evento (*source*). Os componentes normalmente executam suas ações baseados nas notificações que eles recebem através dos eventos. O modelo de evento foi projetado para acomodar a arquitetura do JavaBeans, onde ela utiliza o modelo de delegação que é composto de três partes: *events*, *listeners*, e *sources*. Um *source* (fonte) gera um evento e o envia para um ou mais *listeners* (ouvintes), os *listeners* precisam ser previamente registrados junto à fonte do evento para receber a notificação desta fonte (ORACLE, 2010).

2.2.5 Persistência

Persistência permite ao *Bean* salvar e restaurar seu estado, na arquitetura JavaBeans é utilizado serialização de objetos Java para que possa suportar essa persistência. Os métodos de um *Bean* não são diferentes dos métodos de Java, e podem ser chamados a partir de outros *Beans*. Por padrão todos os métodos públicos são exportados.

Os *Beans* variam em termos de funcionalidade e finalidade. Alguns exemplos de *Beans* conhecidos são: GUI (*Graphical User Interface*), *Beans* não visuais como verificador ortográfico, *applet* de animação e aplicação de planilhas (ENGLANDER, 1997).

Existem três tipos de JavaBeans :

- Visuais: utilizados no desenvolvimento de interfaces;
- Dados: que fornecem um padrão para acesso a valores;
- Serviços: usados para cálculos, acesso a tabelas e algoritmos específicos.

2.2.6 Criação de um *Bean*

Para ser considerado como um JavaBean, as classes precisam obedecer as convenções que definem que a classe deve:

- Implementar a interface *java.io.Serializable* (que possibilita a persistência e restauração do estado do objeto da classe);
- Possua um construtor sem argumentos;
- As propriedades dever ser acessíveis através de métodos "*get*" e "*set*", seguindo um padrão de nomenclatura;
- Possa conter qualquer método de tratamento de eventos (ORACLE, 2010).

Na figura 8 é apresentado um exemplo do desenvolvimento de um componente simples utilizando JavaBeans.

Seguindo estas regras, é determinado um padrão que possibilita a utilização de *Beans* como componentes em ferramentas de desenvolvimento. Este tipo de componente permitiu o desenvolvimento dos módulos para que o SRV obtivesse as informações necessárias sobre a linguagem ao qual ele irá analisar como por exemplos: listas de palavras reservadas, identificadores da linguagem e etc.

```
import java.io.Serializable;

01 public class MyBean implements Serializable {
02
03      private int property1;
04      private boolean property2;
05
06        public MyBean() {
07
08        }
09
10      public void setProperty1(int property1) {
11           this.property1 = property1;
12      }
13
14      public void setProperty2(boolean property2) {
15           this.property2 = property2;
16      }
17
18      public int getProperty1() {
19           return property1;
20      }
21
22      public boolean isProperty2() {
23           return property2;
24      }
25  }
```

Figura 8 – Exemplo de um *Bean* Simples (Próprio Autor, 2010).

2.3 Trabalhos Correlatos

Existem diversas ferramentas para análise estática, ferramentas gratuitas ou comerciais, todas distintas, que abordam diferentes vulnerabilidades de acordo com a técnica de análise empregada por cada uma delas, podendo detectar as mesmas vulnerabilidades utilizando técnicas diferentes (TEIXEIRA, 2007, p.36).

Nesta seção são descritas algumas ferramentas de caráter similar à proposta por esse projeto onde todas possuem código fonte aberto.

2.3.1 Flawfinder

Ferramenta com a funcionalidade de analisar código fonte C e C++ gerando relatórios de possíveis falhas de segurança classificadas por níveis de risco. Ela trabalha em conjunto com um banco de dados contendo vulnerabilidades conhecidas. Desenvolvida em Python, possui interface através de linhas de comandos onde sua instalação pode ser feita independente do SO (WHELLER, 2004).

2.3.2 Pixy

Feita em Java, executa “varreduras” automáticas de códigos fonte PHP4, visando à detecção de XSS e vulnerabilidade de injeção SQL. Ela tem como entrada um programa PHP e cria um relatório de possíveis vulnerabilidades encontradas com informações sobre cada uma delas. Com versões tanto para download como WEB, algumas características diferem uma da outra segundo a tabela 2.

Tabela 2 – Características da Ferramenta Pixy da Versão WEB/Desktop.

Interface Web	Versão para Download
Análise XSS	XSS e análise SQL Injection
Não incluem a resolução de arquivo.	Completa inclui a resolução de arquivo
Tempo de execução limitado há um minuto.	Termina a execução quando a análise for concluída.

Fonte: Site Oficial da ferramenta Pixy.

A ferramenta Pixy trabalha em cima de fluxo de minúsculas (*flow-sensitive*), interprocedimento (*interprocedural*), fluxo de dados sensíveis ao contexto (*context-sensitive data flow*) para descobrir vulnerabilidades de um programa. Sua desvantagem é que ela não suporta recursos orientados a objetos do PHP, pois o tratamento do uso de variáveis de objeto e métodos de membros é tratado de uma forma otimista, impedindo que dados maliciosos surjam de tais construções, acrescentado também que arquivos incluídos com palavras do tipo “include” ou similares não são escaneados pela ferramenta (JOVANOVIC, 2006).

2.3.3 RATS: Rough Auditing Tool for Security

Com a vantagem de analisar múltiplas linguagens como C, C++, Perl, PHP e Python, a ferramenta RATS desenvolvida Secure Software que foi recentemente adquirida pela Fortify Software, Inc, ela analisa o código fonte, identificando os pontos de conflito em potencial, juntamente com a descrição do problema sugerindo soluções. Baseada em uma análise aproximada do código fonte, está ferramenta gera sinalizações de segurança comuns relacionadas a erros de programação como buffer overflows e condições de corrida (FORTIFY).

2.3.4 Findbugs

Caracterizada pela forma de análise, Findbugs utiliza o *byteCode* (arquivos compilados da classe) para realizar análises de códigos fonte, não necessário que a aplicação seja executada. Flexível no sentido de não ser limitada ao conjunto de funcionalidades, Findbug permite à criação e utilização de *plug-ins* adicionando novos detectores a ferramenta. A desvantagem dessa ferramenta é que ela analisa apenas linguagem Java, contudo, pode ser instalada em qualquer máquina virtual compatível com Sun JDK 1.5 (PUGH, 2009).

2.3.5 Contribuições Oferecidas.

Uma vez conhecida algumas ferramentas de segurança de análise estática. A ferramenta OpenTracker, virá a contribuir de maneira significativa o processo de análise se segurança, onde o programador poderá verificar através de uma análise estática, o nível de segurança do seu código, além de ser permitido a ele, a interação com o sistema durante o processo de análise, diminuindo o número de falsos positivos.

Por ser uma ferramenta baseada em componentes, será possível desenvolver módulos que analisem diferentes linguagens de programação, aumentando o domínio de análise da ferramenta, e criando a esses módulos, novas possíveis técnicas de detecções de vulnerabilidades.

CAPÍTULO 3 – SISTEMA DE RASTREAMENTO DE VULNERABILIDADES

Neste capítulo será abordado o processo de desenvolvimento do Sistema de Rastreamento de Vulnerabilidades, denominado SRV.

Como descrito inicialmente no texto, uma maneira eficaz para o desenvolvedor gerar um código seguro, é expor a sua aplicação a ferramentas que permitam a identificação de possíveis vulnerabilidades, ferramentas de análise estática como a proposta por esse projeto.

Dessa forma, para tornar possível a criação do SRV da ferramenta OpenTracker, foi desenvolvido um protótipo através da linguagem Java, no qual é descrito a seguir.

3.1 Estrutura Inicial do Protótipo do SRV

Para uma implementação inicial do protótipo da ferramenta SRV, foi desenvolvido um sistema que permitisse a identificação de todas as palavras reservadas da linguagem Java e a realização de buscas no código fonte para identificação de ocorrências de expressões impostas pelo usuário. Depois de feita a identificação, o sistema gera uma apresentação visual, destacando através do pacote "*Java.awt.Color*" todas as ocorrências encontradas.

3.1.1 Arquitetura do Protótipo

Desenvolvido em Java, através da utilização da IDE NetBeans 6.8. A figura 9 apresenta a arquitetura do protótipo.

Este analisador é composto basicamente por três classes principais: "BancoDeDados.java", "ColorPane.java" e "Funcoes.java". Sendo apresentada a seguinte descrição em relação a cada uma dessas classes:

- *BancoDeDados.java:* essa classe foi desenvolvida para que o analisador trabalhasse sobre qualquer tipo de linguagem de programação. Ela obtém através de um banco de dados *MySQL* todas as palavras reservadas daquela linguagem, que são armazenadas em listas disponibilizadas ao sistema sempre que necessário.
- *Funcoes.java:* identifica as ocorrências de palavras reservadas e expressões. Contém métodos de validação de *tokens* e trabalha em conjunto com outros métodos Java como: *indexOf()* e *charAt()*.

- *ColorPane.java:* implementa métodos e funções, responsáveis pela representação visual do código fonte, tanto para busca de palavras reservadas quanto para busca de expressões impostas pelo usuário.

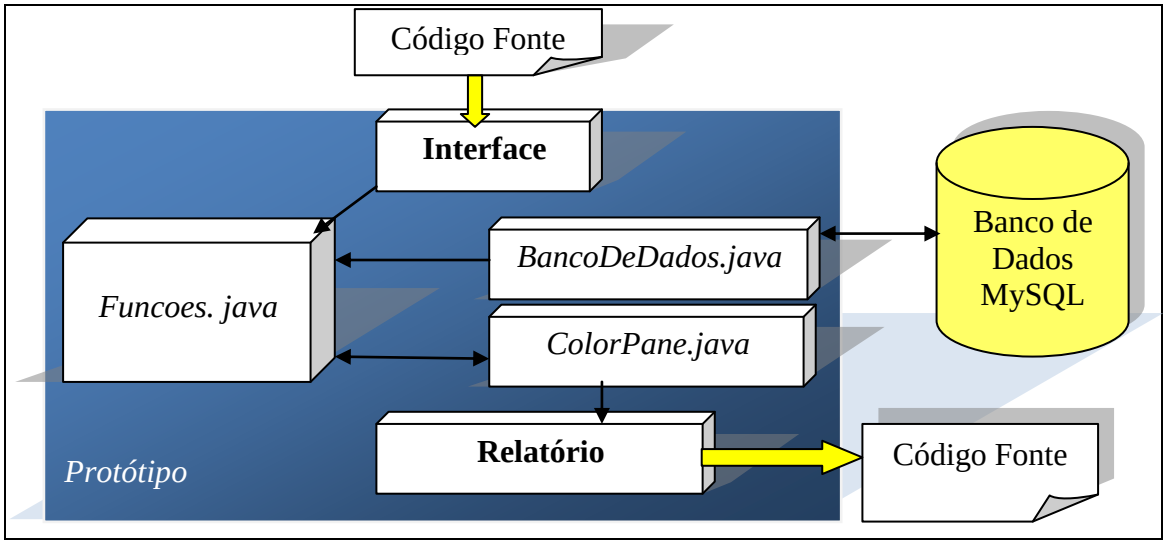

Figura 9 – Arquitetura do Protótipo (Próprio Autor, 2010).

3.1.2 Exemplo do Funcionamento do Protótipo

Na figura 10, é possível ver uma demonstração do sistema em funcionamento.

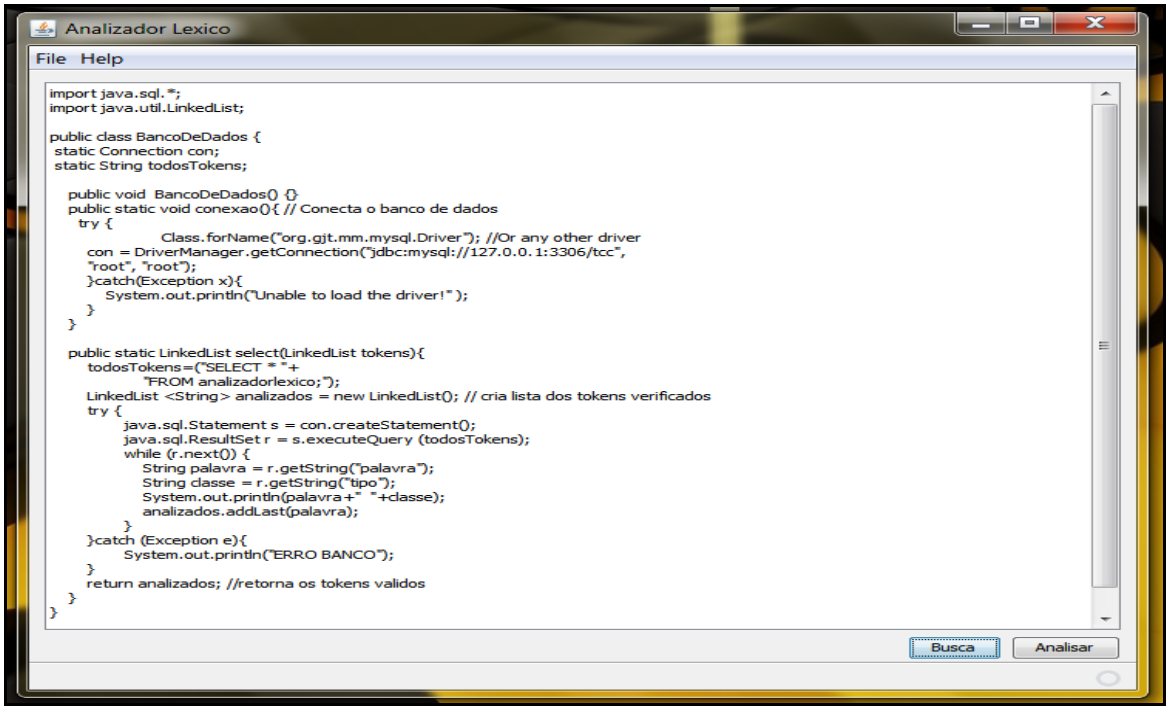

Figura 10 – Analisador Léxico em Java (Próprio Autor, 2010).

De acordo com a arquitetura do sistema apresentada anteriormente, o protótipo possui uma interface que recebe como entrada um código fonte, onde são executadas uma das seguintes funcionalidades: “buscar uma expressão” ou “identificar as palavras reservadas”.

O botão “Analisar” executa o método que identifica todas as palavras reservadas da linguagem Java, e o “Buscar” recebe como entrada uma expressão definida pelo usuário, onde é feita uma análise no código fonte, identificando todas as ocorrências daquela expressão e retratadas através de um relatório, indicando o número e um seguimento da linha em que as ocorrências foram localizadas, conforme pode ser visualizado na figura 11.

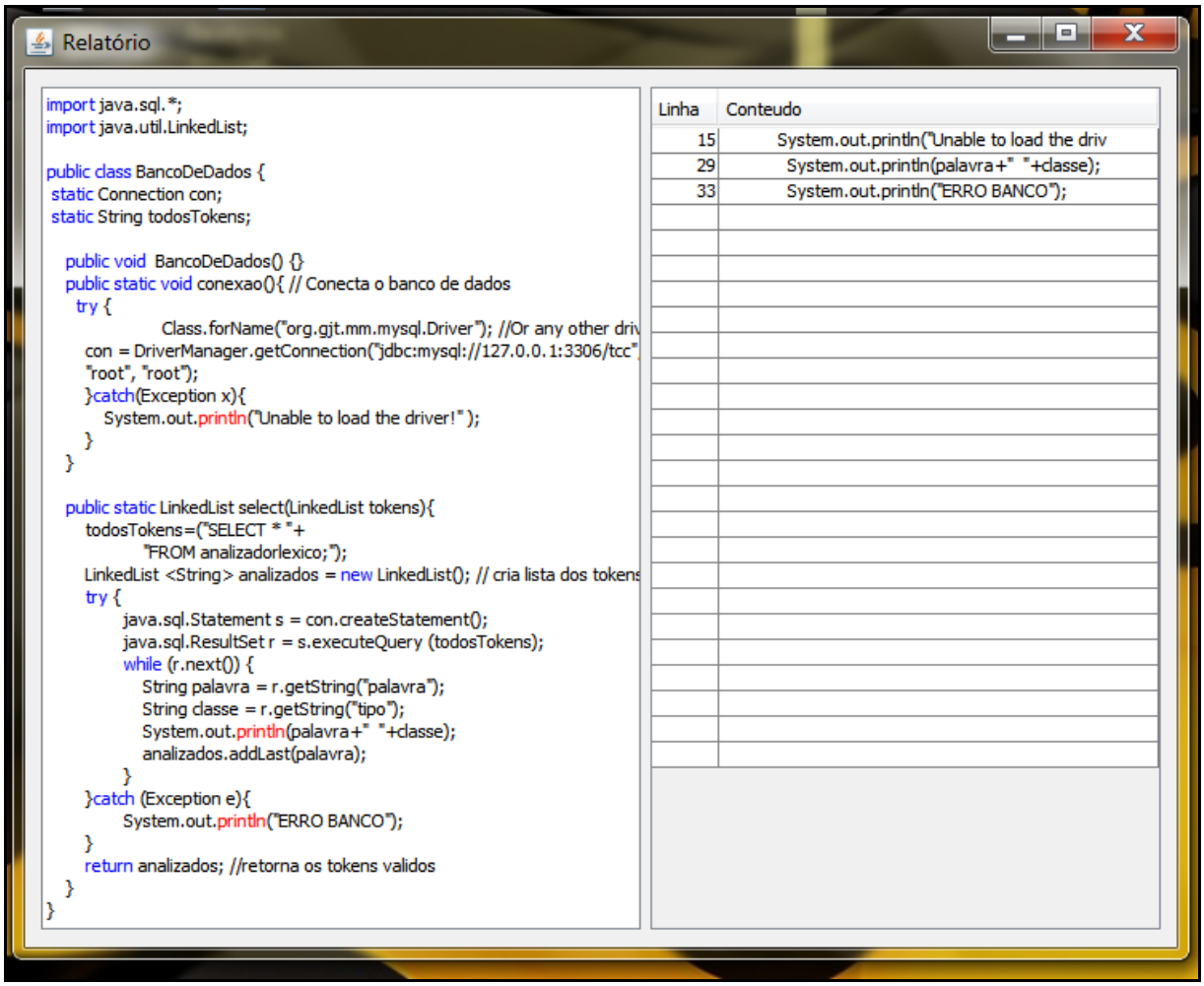

Figura 11– Relatório de Ocorrências Encontradas (Próprio Autor, 2010).

A figura 11 contempla um exemplo do relatório gerado, realizando a busca da palavra “*println*”, sendo localizadas todas as ocorrências representadas pela tabela acima. Este protótipo serviu como base para a criação dos relatórios gerados pelo SRV.

3.2 Arquitetura Completa do SRV

A partir da implementação do protótipo, foi possível identificar determinados *tokens* de um código fonte, esses métodos e funções desenvolvidos serviram de base ao desenvolvimento do SRV auxiliando na busca de assinaturas conhecidas das vulnerabilidades.

A figura 12, demonstra a arquitetura geral da ferramenta OpenTracker, onde é possível entender melhor a participação do SRV.

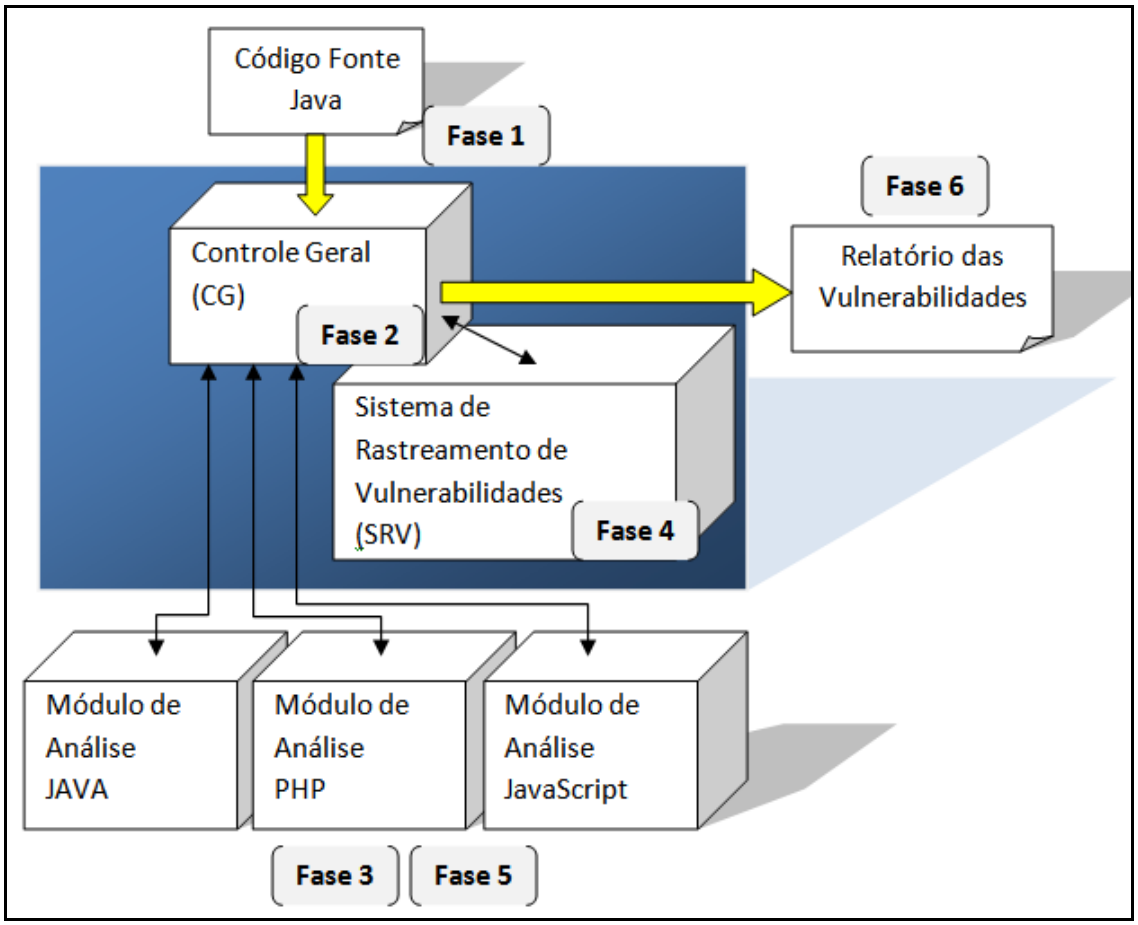

Figura 12 – Arquitetura da Ferramenta OpenTracker (Próprio Autor, 2010).

Segundo a representação da arquitetura da OpenTracker, pode-se observar que a ferramenta encontra-se organizada por meio de componentes (Beans), divididos em dois módulos principais.

O primeiro módulo divide-se em dois componentes independentes. A primeira fase trata da recepção do código fonte Java possivelmente vulnerável, e o entrega ao Controle Geral (CG) para análise. Depois de recebido, o CG é responsável por ordenar a sequência de

atividades e tomada de decisões para a verificação de cada tipo de vulnerabilidade, através das informações obtidas pelo SRV (Fase 2).

Na terceira fase, são requisitadas ao módulo de análise, todas as informações relevantes da linguagem de programação ao qual está sendo analisado, como estruturas de métodos, tipo de identificadores (modificadores de acesso, de classes, variáveis ou métodos, controle de fluxo dentro de um bloco de código e demais informações), estruturas de comentários, contexto de escopos, todo tipo de informação que possa caracterizar a linguagem e ser útil ao CG (Fase 3). Cada módulo de análise é especializado em apenas uma linguagem de programação.

Em seguida, após a obtenção dessas informações são especificados quais tratamentos serão executados pelo CG (*SQL Injection*, XSS, etc.), dessa forma caso necessite, o CG requisita ao SRV, uma pré-análise do código fonte, separando somente as linhas de códigos relevantes a serem processadas pelo módulo de análise (Fase 4).

Uma característica interessante sobre o SRV é a flexibilidade adquirida, sendo possível utilizar a interação com o usuário, permitindo que ele interaja de maneira direta na obtenção dos resultados. Essa interação é descrita na seção 3.5.4.

Concluindo o processo de rastreamento, o Controle Geral realiza a execução de métodos desenvolvidos pelos módulos, para obter informações relevantes que o auxilie durante a análise de cada vulnerabilidade, como por exemplo: identificações de possíveis tratamentos realizados na entrada e saída de dados de uma variável durante o tratamento de *SQL Injection* (Fase 5).

Finalizada a quinta fase, o CG possui a relação de todas as possíveis vulnerabilidades encontradas no código fonte, sejam adquiridas com ou sem a interação do usuário, gerando um relatório final com os possíveis tratamentos (Fase 6).

Está arquitetura acaba por ser modular e portável, uma vez que os módulos possuem interfaces bem definidas onde cada componente trabalha de forma independente permitindo a integração de novas funcionalidades a cada um deles.

3.3 Estrutura da Implementação do SRV

Inicialmente foram estudadas quais as classes que deveriam ser implementadas para organizar a estrutura do SRV, uma vez que, uma das características principais do sistema é permitir a análise de diversas linguagens de programação. Para isso, foi necessário que as

classes do SRV implementassem métodos padrões. A figura 13 apresenta a estrutura das implementações dos pacotes do SRV.

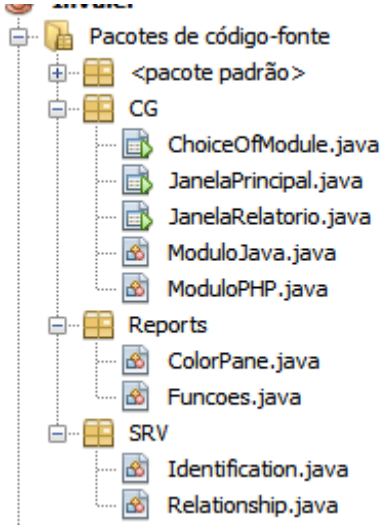

Figura 13 – Estrutura dos Pacotes do SRV (Próprio Autor, 2010).

O desenvolvimento do SRV pode ser dividido em quatro partes principais. A primeira apresenta o conjunto de classes da *interface* gráfica (pacote CG), simbolizando o funcionamento do Controle Geral (CG), responsável pelo controle do SRV. Para isso, foram implementadas funcionalidades básicas em que o CG terá que abordar como: comunicação entre o CG e os módulos (componentes JavaBeans), chamadas de funções do SRV e a geração de relatórios (figura 14).

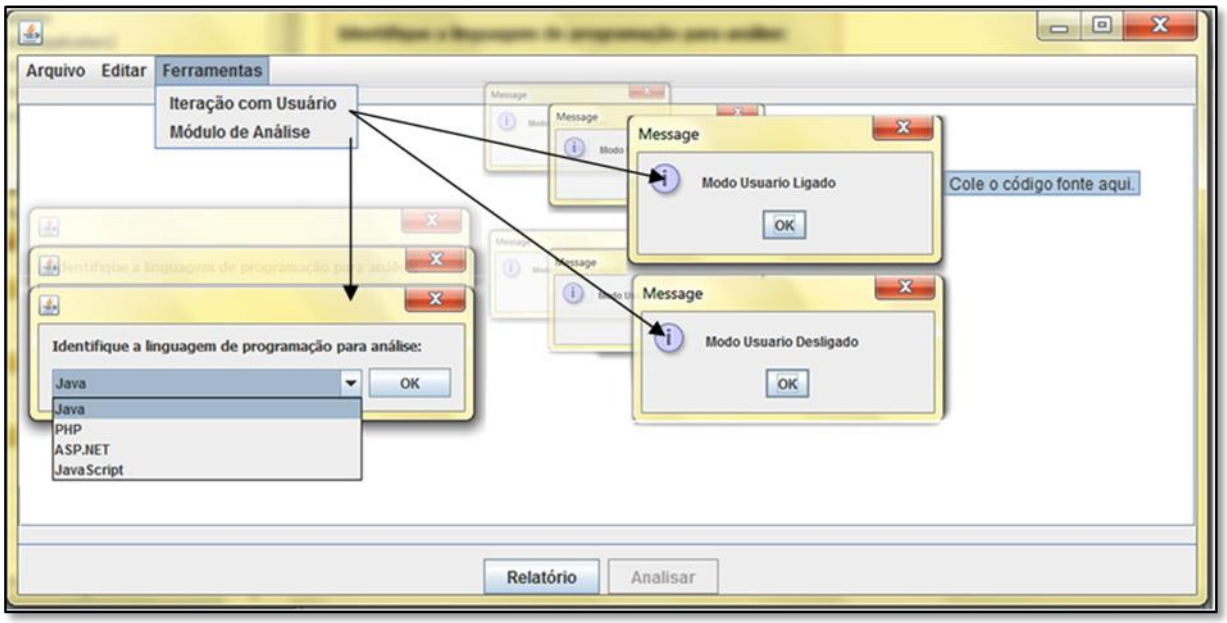

Figura 14 – Interface que Representa o Controle Geral (Próprio Autor, 2010).

A *interface* apresenta quatro funcionalidades:

- *Interação com Usuário:* definição da interação ou não do usuário durante o processo de execução do SRV, influenciando na obtenção dos resultados e diminuindo o número de falsos positivos encontrados no código fonte;
- *Módulo de Análise:* escolha da linguagem de programação que será realizada a análise (implementado o módulo para linguagem Java);
- *Relatório:* visualização das ocorrências encontradas pelo SRV;
- *Analisar:* execução do SRV.

A segunda parte é responsável pela criação de um dos módulos desenvolvido em JavaBeans, contendo listas de todas as informações relevantes (necessárias para a execução do SRV), a respeito da linguagem de programação em que ele analisa como: palavras reservadas, tipos primitivos, identificadores de *SQL`s* e os respectivos valores codificados (*hashCode()*) da lista de identificadores especiais que serão utilizados pelo SRV. Por ser desenvolvido como um componente (*Bean*), ele permite que novas funcionalidades sejam desenvolvidas sem que seja necessário realizar mudanças na estrutura do CG.

A terceira parte apresenta o pacote *Reports,* que contém as classes responsáveis pela implementação dos relatórios gerados pelo CG.

Na quarta parte, foi desenvolvido o pacote do SRV propriamente dito, compostos de duas classes principais como mostra a figura 15, que contém uma variedade de métodos utilizados na identificação das linhas relevantes de código.

Em relação à figura 15, ambas as classes não possuem relacionamento entre si, sendo que, a relação é feita através da classe "*JanelaPrincipal.java*" do CG, em que ela instancia um objeto de cada uma das classes acima e administra esses objetos de maneira a obter os resultados desejados.

A seguir são apresentados os métodos principais que compõe cada uma das classes desenvolvidas no pacote SRV, sendo que a classe *Identification.java* possui:

- *ignoreComments():* remove os comentários do código e codifica os *tokens* relevantes contidos em aspas simples e aspas duplas, permitindo otimizar a análise e evitar erros durante o processo de análise sintática;
- *makeIdentification():* identifica os escopos do código fonte, permitindo ao SRV o controle de fluxo de todo o código;

- *markingIdentifiers():* através destas marcações são localizados de maneira direta os identificadores encontrados no código fonte, utilizadas na análise sintática;
- *relocateScope():* este método é responsável por validar os escopos encontrados na análise;
- *variablesFound():* obtém todas as variáveis declaradas no código fonte, incluindo o tipo, nome, escopo a que pertence e se ela foi declarada como parâmetro de um método ou não (utilizado pelo módulo);
- *identifyListOfMethods():* armazena uma lista com todos os métodos encontrados no código fonte.

O SRV analisa inicialmente vulnerabilidades *SQL Injection* e apresenta os seguintes métodos principais contidos na classe *Relationship.java*:

- *locatedSQL():* registra todas as *SQL* encontradas no código fonte;
- *storageHierarchyOccurrences():* identificam quais são os escopos em que as variáveis relacionadas às *SQL*, possam ser utilizadas;
- locatesVariablesInSql(): armazena todas as variáveis e métodos utilizados nas definições das SQL`s.
- *locatesOccurrencesSql()*: armazenam listas contendo todas as variáveis que tiveram participação na definição da SQL, utilizando o auxílio da hierarquia de ocorrência citado anteriormente;
- *validatesOverload():* quando é inserido um novo método na análise, ele é identificado através de uma análise semântica realizando um tratamento de sobrecarga;
- *questionAboutTheMethods():* questiona o usuário em relação aos métodos envolvidos com a *SQL*, se eles realizam algum tipo de tratamento de segurança como, por exemplo: validação de entrada e saída de dados. Em caso afirmativo, aqueles métodos são descartados da análise, bem como todas as variáveis relacionadas a ele.

Ao final do desenvolvimento das classes descritas (“*Identification.java” e “Relationship.java”)* constata-se que vários métodos desenvolvidos em ambas as classes são parecidos, principalmente os métodos de filtragem, o que futuramente serão modificados, afim de trabalhar de maneira mais adequada o conceito de orientação a objetos e efetuando melhorias de otimização no SRV.

Figura 15 – Diagrama de Classe do Pacote SRV (Próprio Autor, 2010).

Por fim, a última parte desenvolvida é voltada a geração de relatórios em que é apresentado o código fonte com as ocorrências encontradas, e uma tabela indicando os números e trechos das linhas de código fonte identificadas. Para o desenvolvimento dessa interface gráfica, foram utilizadas classes do protótipo desenvolvido descrito anteriormente, o

que permitiu a identificação das palavras reservadas e a identificação no código fonte das linhas relevantes identificadas pelo SRV.

3.4 Funcionamento dos Processos Desenvolvidos

Abaixo serão demonstrados alguns exemplos dos métodos executados pelo SRV.

3.4.1 Remoção de Comentários e Codificação de *Tokens*

A tabela 3 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.

Tabela 3 – Remoção de Comentários e Codificação de *Tokens*.

ENTRADA	SAÍDA

`/* Comentário */`
`// Comentário`
`String variável = "{ (";`
	`0##`
	`1##`
	`2## String variável = "@123@ @41@";`

Fonte: (Próprio Autor, 2010).

3.4.2 Identificação de Escopos

Para obter a identificação e o controle dos escopos contidos em um código fonte (tabela 4), 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.

Tabela 4 – Identificação de Escopos (Próprio Autor, 2010).

ENTRADA	SAÍDA
`public class Teste { }`	`public class Teste H1E1T1 H1F1T1`

Fonte: (Próprio Autor, 2010).

3.4.3 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 5).

Este método impede que ocorrências do tipo "String *privateVariavel*;" representada pela tabela 5, 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.

Tabela 5 – Marcação de Identificadores (Próprio Autor, 2010).

private String
metodoTeste(){
}
String *privateVariavel;*
	%-314497661% String metodoTeste ()
	H1E1T1
	H1F1T1
	String *@Fail@Variavel;*

ENTRADA	**SAÍDA**

Fonte: (Próprio Autor, 2010).

3.4.4 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 6 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.

Tabela 6 – Representação de Variáveis no SRV (Próprio Autor, 2010).

String variavel = "string"; public void metodo(int variavel){}	String:|variavel@Escopo: %H1E1T1#Type: default int:|variavel@Escopo: %H1E9T9#Type: method
ENTRADA	**SAÍDA**

Fonte: (Próprio Autor, 2010).

3.4.5 Hierarquia de Escopos

A lista presente na tabela 7, 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 inicio a análise do escopo "*HxEy+1Tz+1"*.

Tabela 7 – 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: (Próprio Autor, 2010).

3.4.6 Tratamento de Sobrecarga

O SRV utiliza o tratamento de sobrecarga para identificar a localização dos métodos que foram inseridos na análise (figura 16).

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.

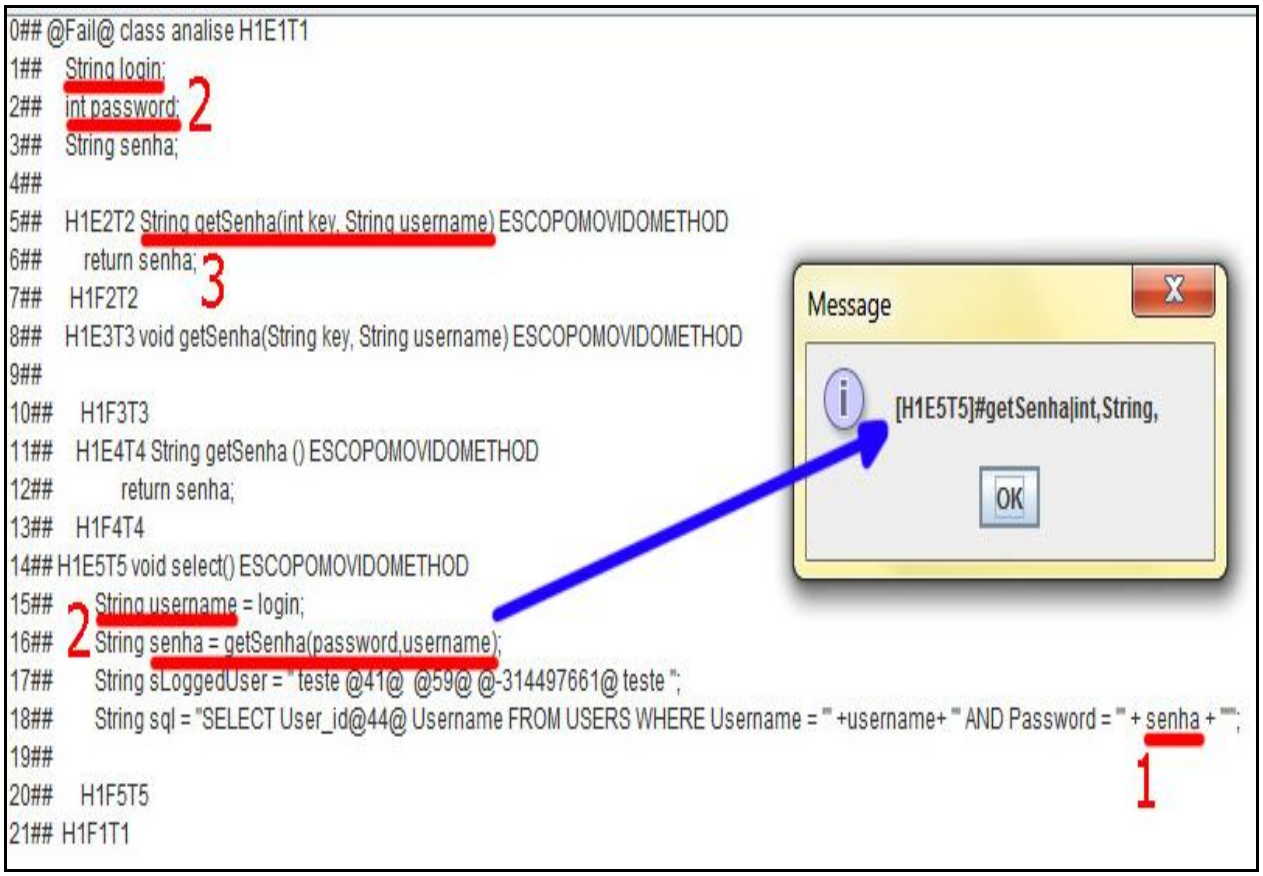

Figura 16 – Exemplo da Utilização do Tratamento de Sobrecarga (Próprio Autor, 2010).

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.

3.5 Utilizações do SRV pelo Controle Geral

Para entender a ordem da chamada dos métodos do SRV pelo CG, uma vez que é conhecido como os métodos fundamentais do SRV foram desenvolvidos e a funcionalidade de cada um deles, serão abordadas nesta seção a ordem destas chamadas e a sequência de atividades executadas pelo usuário no CG (figura 17).

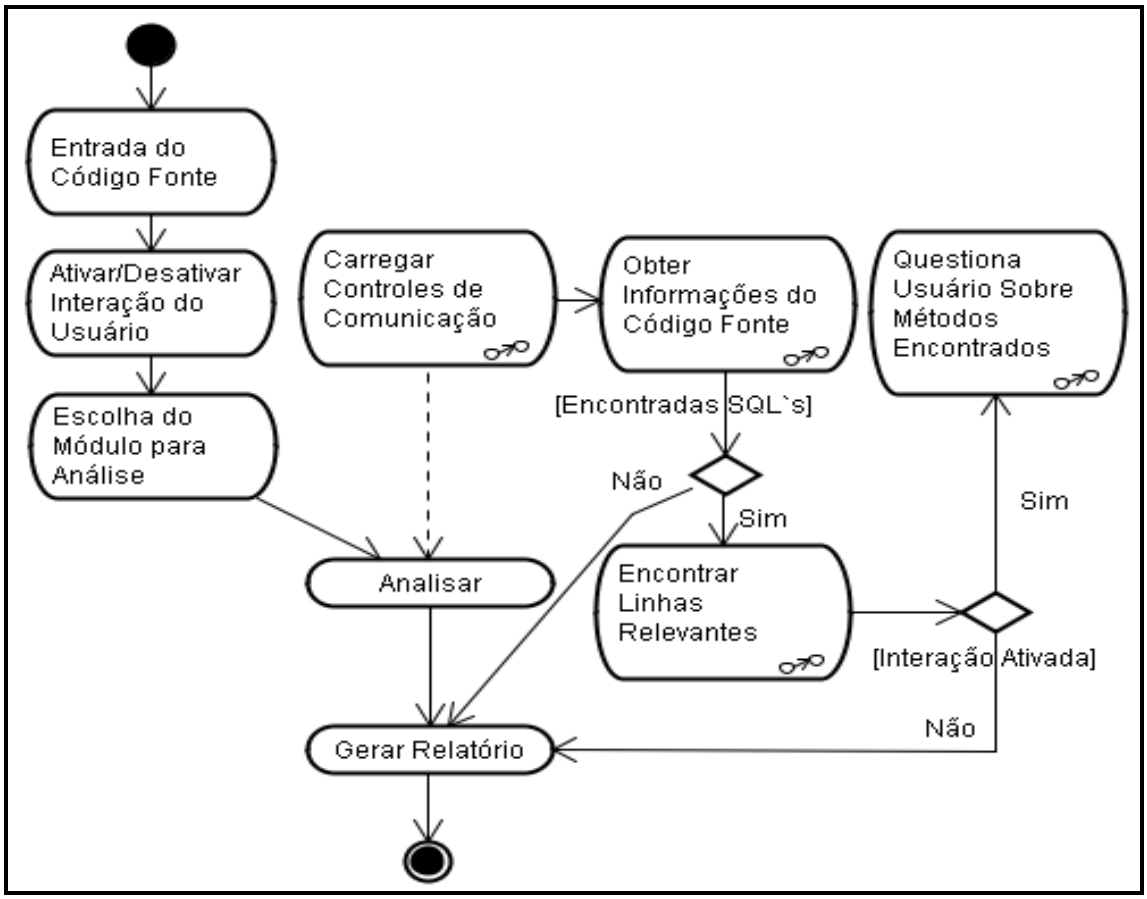

Figura 17 – Diagrama de Blocos do Controle Geral (Próprio Autor, 2010).

Primeiramente, o CG recebe como entrada do usuário o código fonte que será analisado, então é especificado a ativação da interação ou não do usuário e feita à escolha do módulo da linguagem de programação. Em seguida é iniciado o processo de análise ao qual serão executados alguns métodos contidos no pacote de classes do CG como: "Carregar controles de Comunição" (*contr_LoadsCommunications()*), "Obter Informações do Código Fonte" (*contr_Informations()*), "Encontrar Linhas Relevantes" (*exec_isVulnerable()*) e "Questiona Usuário Sobre Métodos Encontrados" (*contr_questionAboutTheMethods()*). Terminado a execução é gerado o relatório.

3.5.1 Carregar Controles de Comunicação

Após entender o fluxo de atividades executadas pelo usuário no CG é possível visualizar na figura 17 a ordem das chamadas dos métodos que o CG realiza ao ser executado pelo botão "Analisar". Inicialmente o CG instancia um objeto do módulo especificado pelo usuário e carrega as informações relevantes da linguagem a ser analisada através da chamada

de um método em comum entre os módulos denominado *reset().* Entre as informações que são carregadas encontram-se para a linguagem Java (implementada): *primitiveTypes* (String, int...), *identifiersDisplacement* (public, for...), *sqlIdentifiers* (SELECT, INSERT...), *reservedWords* (private, public, do...). Também são carregados os valores codificados destes atributos, obtidos através da função *hashCode()* do Java, sendo que, esses valores serão utilizados pelo SRV para localização de ocorrências.

3.5.2 Obter Informações do Código Fonte

Após obter as informações relevantes da linguagem de programação, o CG inicia o processo de reconhecimento do código fonte, adquirindo uma grande quantidade de informações que serão armazenadas em listas e estarão disponíveis ao CG sempre que necessário. A figura 18 apresenta a ordem das chamadas dos métodos.

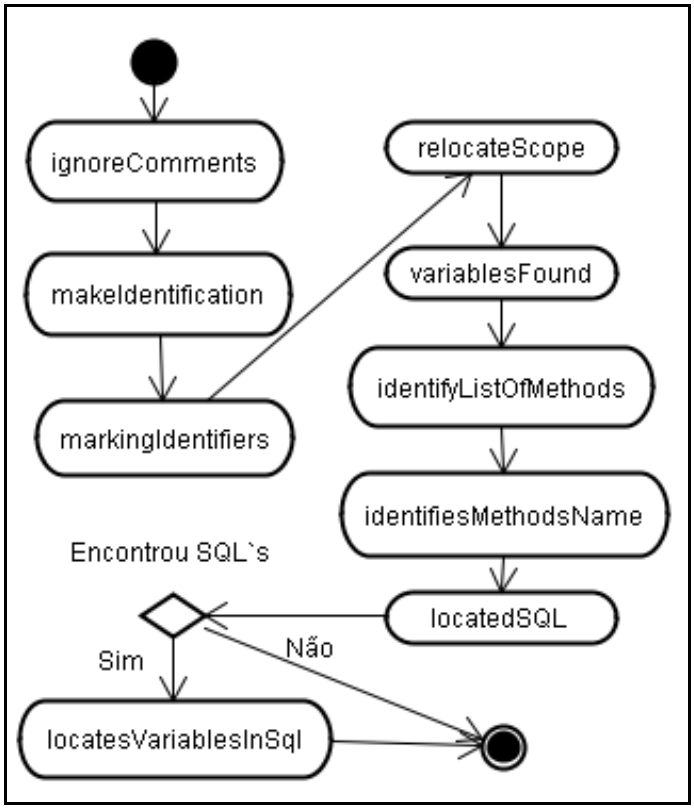

Figura 18 – Expansão do Método "Obter Informações do Código Fonte" (Próprio Autor, 2010).

Os métodos da figura 18 pertencem as classes "*Identification.java" e "Relationship.java"* ao qual já foram descritos na seção 3.4, o método *locatedSQL()* armazena a lista de SQL`s encontradas no código fonte que será verificada para determinar a

chamada ou não do método “Encontrar Linhas Relevantes”, em caso positivo é obtido previamente a localização e o armazenamento das variáveis encontradas nas *SQL`s* (*locatesVariablesInSql()*).

3.5.3 Encontrar Linhas Relevantes

A figura 19 mostra a expansão da chamada do método “Encontrar linhas Relevantes” presente na figura 17.

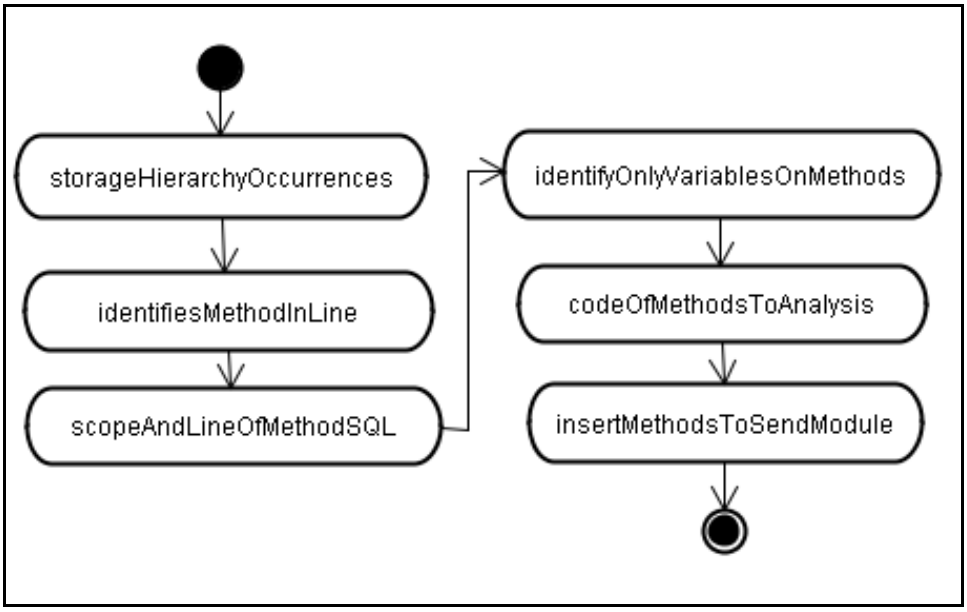

Figura 19 – Expansão do Método “Encontrar Linhas Relevantes” (Próprio Autor, 2010).

Primeiramente, o método *storageHierarchyOccurrences(),* obtém a relação da hierarquia de escopos em que as variáveis presentes nas SQL`s possam ser utilizadas, sendo que, é neste método que é obtida a lista de todas as variáveis que tiveram participação na definição da SQL através da chamada do método *locatesOccurrencesSql()*.

O método *identifiesMethodInLine(),* armazena uma lista com os nomes dos métodos e o conteúdo das linhas de código onde estes foram encontrados, para que eles possam ser inseridos na análise como por exemplo, se uma variável identificada como relevante durante a análise receber uma atribuição de retorno de um método qualquer, então é necessário que aquele método seja analisado.

Em seguida o método *scopeAndLineOfMethodSQL()* identifica o numero da linha e o escopo de cada método inserido na análise, o que facilita a sua localização considerando que

um mesmo método pode ser utilizado varias vezes durante a execução de uma aplicação, adquirindo melhor controle sobre os já analisados.

Dando continuidade a ordem das chamadas, o próximo método *identifyOnlyVariablesOnMethods()* é o único representado pela figura 19, implementado pela classe "*Identification.java",* responsável por identificar as variáveis contidas nos parâmetros dos métodos analisados, e realizar através da análise semântica um preparo inicial para o tratamento de sobrecarga explicado na seção 3.4.6.

Por fim, são executados os métodos *codeOfMethodsToAnalysis()* e *insertMethodsToSendModule,* em que o primeiro executa a validação de sobrecarga dos métodos analisados e o segundo é responsável por obter todo o código fonte de cada um, encontrado na análise, após a validação de sobrecarga armazenando o código desses métodos em uma lista que será utilizada para diminuição de falsos positivos.

3.5.4 Questiona Usuário Sobre Métodos Encontrados

Concluída a execução dos métodos descritos anteriormente, o último "Questiona Usuário Sobre Métodos Encontrados" é executado caso a opção de interação do usuário esteja ativada. Ele questiona o usuário sobre os métodos encontrados durante a análise, se eles realizam ou não algum tipo de tratamento voltado à questão da segurança, sendo os que tenham respostas positivas é removido da lista de métodos relevantes.

Essa interação é importante porque ela diminui o número de falsos positivos (detecções que na realidade não apresentam ameaças), uma vez que com o auxílio da interação do usuário no processo de análise, o número de linhas relevantes encontradas pelo SRV ao qual o CG terá que analisar poderá ser reduzido.

CAPÍTULO 4 – RESULTADOS OBTIDOS

Após entender o funcionamento e a maneira como o SRV obtém informações a partir de um código fonte, serão abordados nesta seção os resultados obtidos de acordo com os objetivos propostos pelo projeto.

4.1 Linhas Relevantes para Detecção de *SQL Injection*

Depois de realizada a análise o Sistema de Rastreamento de Vulnerabilidades (SRV) obtém as linhas de código relevantes que serão repassadas ao controle geral e trabalharão em conjunto com o módulo da linguagem específica para detecção das vulnerabilidades encontradas.

Para o tratamento de *SQL Injection* o SRV identifica todas as *SQL`s* presente no código fonte, bem como todas as variáveis e métodos que tiveram participação na definição destas *SQL`s*.

Todo método incluído na análise, passa por uma validação de parâmetros através de um tratamento de sobrecarga, armazenando o conteúdo do código fonte daqueles métodos em uma lista de métodos para análise.

Dessa forma, durante a execução o desenvolvedor pode interagir com o sistema, informando se os métodos encontrados nas *SQL`s* ou durante a análise recursiva do código fonte, efetuam algum tipo de tratamento de vulnerabilidades. São removidos da análise aqueles com respostas afirmativas e enviados ao CG os códigos fonte dos que o desenvolvedor determinou que não efetuasse algum tipo de tratamento.

Realizada a identificação, é apresentado um relatório destacando as linhas de código que serão repassadas ao controle geral, bem como um relatório contendo o número e a linha identificada pela análise. A seguir na figura 20, um exemplo de um resultado obtido a partir de um código fonte.

A linha 48 indicada no relatório, repassa ao CG todo código fonte do método *getSenha()* e não somente sua declaração. Foi definido esse tipo de representação para melhorar a visualização do relatório demonstrando os métodos que serão analisados pelo CG.

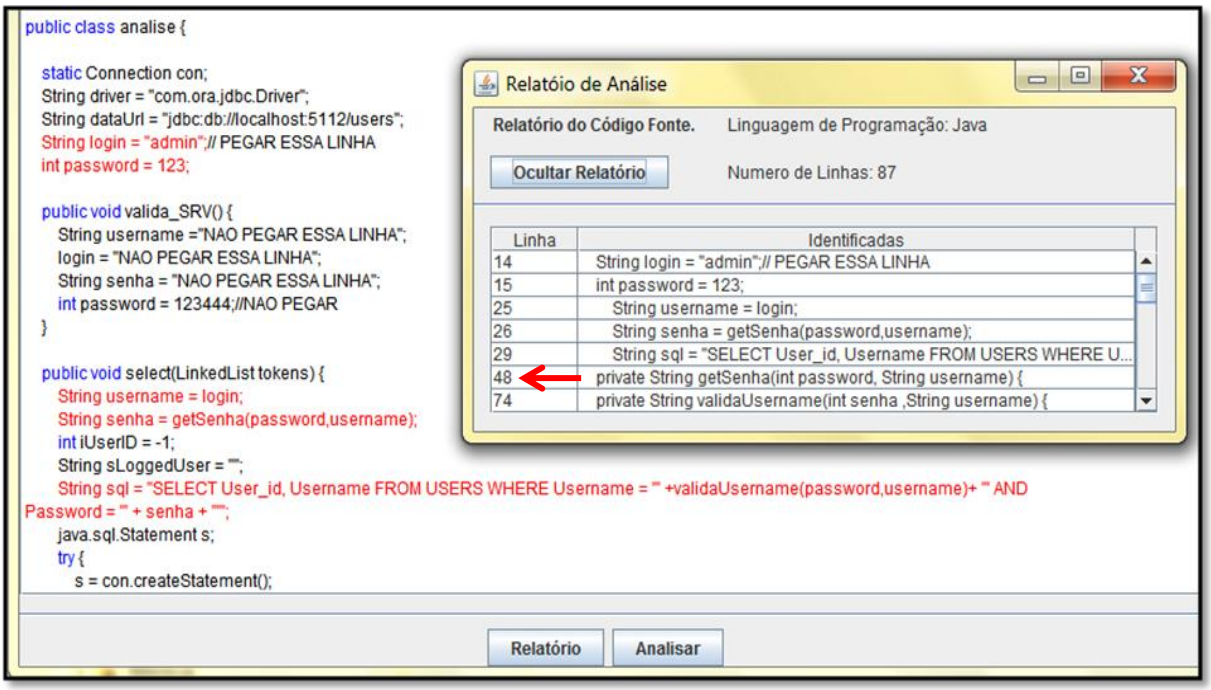

Figura 20 – Identificação das Linhas Relevantes para Análise *SQL Injection* (Próprio Autor, 2010).

4.2 Informações Relevantes Disponibilizadas ao CG

Durante o desenvolvimento do SRV houve uma grande preocupação em relação às informações que deveriam ser armazenadas e disponibilizadas ao CG, como por exemplo, listas de variáveis obtidas, os métodos presentes do código fonte, a hierarquia de escopos, entre outras, sendo que, grande parte dessas informações poderá ser atribuída aos módulos para que eles efetuem suas análises de maneira mais adequada.

De fato, a ferramenta OpenTracker, consegue adquirir um maior domínio sobre o código que está sendo analisado, uma vez que, varias informações do código já estão disponíveis é facilitado assim, o desenvolvimento dos demais componentes.

4.3 Estatísticas de Falsos Positivos das Análises

Uma vez ativada a opção de interação do usuário e removido da análise os métodos que realizam algum tipo de tratamento voltado à questão da segurança, diminuindo assim o número de falsos positivos. Será desenvolvido posteriormente um mecanismo para remover todas as variáveis e métodos utilizados, que tiveram algum tipo de relação com aqueles que foram indicados pelo usuário, como sendo métodos para o tratamento de segurança.

Dessa forma, possibilitará ao SRV diminuir consideravelmente a relação das variáveis e métodos encontrados durante a análise, que serão enviados ao controle geral para que sejam executadas as análises de detecções das vulnerabilidades.

4.4 Documentação para Evolução da Ferramenta *Open Source*

A ferramenta OpenTracker está em processo de desenvolvimento tendo concluído a implementação do componente SRV, que precisa ainda de algumas melhorias, como por exemplo: a organização do diagrama de classes, melhoria no processo de diminuição de falsos positivos, otimização dos métodos implementados e o tratamento de sobrecarga para métodos genéricos.

Contudo, o desenvolvimento dos demais componentes estão sendo realizados por um grupo de iniciação científica, ao qual, é esperado a construção de uma versão da ferramenta OpenTracker com todos os módulos já desenvolvidos, para que seja posteriormente desenvolvido a documentação para a criação da comunidade *Open Source*.

CONCLUSÕES

Realizado o desenvolvimento do SRV, são identificadas as linhas importantes do código fonte, necessárias para efetuar a análise de *SQL Injection*, que são visualizadas através de um relatório, indicando as posições, o conteúdo das ocorrências encontradas e uma representação visual no próprio código fonte. Para a identificação será implementado em trabalhos futuros, um mecanismo que possa guardar as informações obtidas pela análise, no qual permitira uma otimização ainda maior na análise do SRV, devido ao fato, que o analisador irá verificar previamente se houveram mudanças ou não no código fonte, antes de realizar a busca, dessa forma será permitido o controle de análise mais robusto e otimizado.

A eficiência adquirida pelo tratamento de falsos positivos com a interação do usuário, não atingiu um nível ideal, no qual não foi possível concluir a implementação da remoção das variáveis e métodos encontrados pela análise, que tiveram relação a um método indicado pelo usuário, como sendo um método para o tratamento relacionado à segurança da aplicação. Este será um dos problemas a ser resolvido em trabalhos futuro.

O SRV dispõe de todas suas listas ao CG, sendo que essas informações serão de grande utilidade ao desenvolvimento dos módulos, uma vez que durante o desenvolvimento,

houve uma grande preocupação em caracterizar quais informações deveriam ser armazenadas e disponibilizadas. Dessa forma foi atingido de maneira significativa o objetivo proposto inicialmente pelo projeto.

A documentação para a comunidade *Open Source* está em andamento, pois é esperada uma versão funcional da ferramenta OpenTracker, sendo que a documentação *javadoc* do SRV desenvolvido já se encontra disponível no cd entregue junto a está monografia.

Com o desenvolvimento do Sistema de Rastreamento de Vulnerabilidades da ferramenta OpenTracker, será possível realizar análises de vulnerabilidades específicas, sendo que, os módulos analisaram somente as linhas relevantes e será admitido a estes e a todos os demais componentes da ferramenta OpenTracker, a criação de novas funcionalidades, devido a flexibilidade que a arquitetura baseada em componentes impõe.

Por se tratar de uma ferramenta *Open Source*, a OpenTracker dependerá do envolvimento de toda a comunidade para que possam ser realizadas as atualizações. Dessa forma, a OpenTracker tenderá a crescer gradativamente, e com isso possibilitará aos desenvolvedores uma visão diferenciada na maneira de programar, permitindo o desenvolvimento de códigos mais seguros, uma vez que, a ferramenta OpenTracker irá detectar essas vulnerabilidades e apresentará possíveis soluções.

REFERÊNCIAS

SEVESTRE, P. (2009) "**Programação Segura utilizando Análise Estática**". Disponível em: <http://www.owasp.org/images/a/aa/AppSec_Brasil_2009SecureProgrammingWithStaticAnalysis.pdf>. Acessado em: 29 Abril 2010.

COELHO, F.(2007) **"Desenvolvimento de aplicações distribuídas para a internet/intranet "**. Disponível em: <http://paginas.fe.up.pt/~aas/pub/Aulas/GestEst/Resumos/_ResPT/PT_18.pdf>. Acessado em: 19 Nov 2009.

CHMIELEWSKI, M. et al. (2007) "**Find and Fix Vulnerabilities Before Your Application Ships**". Disponibilizado em: <http://msdn.microsoft.com/en-us/magazine/cc163312.aspx>. Acessado em: 18 Abril 2010.

UTO, N., MELO, S. P. (2009) **"Vulnerabilidades em Aplicações Web e Mecanismos de Proteção".** Cap.6, p.238.

OWASP. "Top 10 -2010: **The Tem Most Critical Web Application Security Risks**". Disponível em: <http://owasptop10.googlecode.com/files/OWASP%20Top%2010%20-%202010.pdf>. Acesso em: 07, junho 2010.

DEITEL, H. M., DEITEL P.J. (2003) "**Java Como Programar 4 edição**", Publicado por Bookman Companhia Editora, Brasil.

ENGLANDER, R. (1997) "**Developing Java Beans**", Publicado por O`Reilly & Associates.

CERON, J. M. et al. (2008) "**Vulnerabilidades em Aplicações Web:** uma Análise Baseada nos Dados Coletados em Honeypots". Rio grande do Sul: Porto Alegre. Disponível em:<http://www.pop-rs.rnp.br/arquivos/2008/honeypot_web_sbseg.pdf>. Acesso em: 21 setembro 2009.

CERT.BR. "**Estatísticas dos Incidentes Reportados ao CERT.br"**. Disponível em: <http://www.cert.br/stats/incidentes/#2010>. Acesso em: 02 agosto 2010.

PEREIRA, F. "**Programas Seguros:** Vulnerabilidades comuns e cuidados no desenvolvimento". Disponível em:<http://www.las.ic.unicamp.br/paulo/papers/2000-SSI-felipe.pereira-programas.seguros.pdf>. Acesso em: 07 janeiro 2010.

LEBAN, B. at al. (2010) “Do Know Evil: **WEB application vulnerabilities**”. Disponível em: <http://googleonlinesecurity.blogspot.com/2010/05/do-know-evil-web-application.html>. Acesso em: 16 setembro 2010.

WASC, Threat Classification. “**WEB Application Security Consortium**”. Disponível em: <https://files.pbworks.com/download/H3hIH3aEnX/webappsec/13247059>. Acesso em: 22, outubro, 2010.

OWASP. (2008) “**Code Review Guide, V1.1**”. Disponível em: <http://www.lulu.com/items/volume_64/5678000/5678680/13/print/5678680.pdf>. Acesso em: 07, setembro 2010.

FARIAS, M. B. (2009) “Injeção de SQL em aplicações WEB: **Causas e Prevenção**” Disponível em: <http://www.lume.ufrgs.br/bitstream/handle/10183/18573/000730977.pdf?sequence=1>. Acesso em: 11, julho 2010.

MENEZES, A. F. (2009) “**Análise Estática do Framework Demoiselle do Governo Federal com Ênfase em Segurança de Aplicações WEB**”. Disponível em: <http://www.scribd.com/doc/24042296/Análise-Estática-do-Framework-Demoiselle-com-Enfase-em-Seguranca-de-Aplicacoes-Web>. Acesso em: 25, agosto, 2010.

TEIXEIRA, E. (2007) “**Ferramenta de Analise de Código para Detecção de Vulnerabilidades**”, Disponibilizado em: <http://docs.di.fc.ul.pt/jspui/bitstream/10455/3090/1/07-29.pdf>. Acesso em: 25 abril 2010.

EDUARDO, D. M. (2004) “Como Construir um Compilador: **Utilizando Ferramentas Java**”. Publicado por NovaTec.

ORACLE Java Techonology “Java SE Documentation: **JavaBeans Component API**”. Disponível em: <http://download.oracle.com/javase/6/docs/technotes/guides/beans/>. Acesso em: 25, setembro, 2009.

WHELLER, D. A. (2004) “Flawfinder: **A Source Code Scanner**” Disponível em: <http://www.dwheeler.com/flawfinder/#othertools>. Acesso em: 12, março, 2010.

JOVANOVIC, N. at al (2006) “Pixy: **A Static Analysis Tool for Detecting Web Application Vulnerabilities**”. Disponível em: <http://www.seclab.tuwien.ac.at/papers/pixy.pdf>. Acesso em: 15, abril, 2010.

FORTIFY, An HP Company. “**RATS: Rough Auditing Tool for Security**”. Disponível em: <https://www.fortify.com/ssa-elements/threat-intelligence/rats.html>. Acesso em: 21, março, 2010.

PUGH, B. at al (2009) “Findbugs: **Because it`s Easy**”. Disponível em: <http://findbugs. sourceforge.net/factSheet.html >. Acesso em: 24, março, 2010.

ROCHA, H (2005) “Programação em Java com J2SE: **Tipos Genéricos**”. Disponível em: <htt p://www.argonavis.com.br/cursos/java/j100/java-5-generics.pdf>. Acesso em: 3, novembro, 2010.