**Universidade Paulista - UNIP**

**Mateus Renato Rodrigues Spadacio**

**Análise de falhas de segurança em código**

**Limeira**

**2018**

**Universidade Paulista - UNIP**

**Mateus Renato Rodrigues Spadacio**

**Análise de falhas de segurança em código**

Trabalho de conclusão de curso apresentado à banca examinadora da Faculdade UNIP, como requisito parcial à obtenção do Bacharelado em ciência da computação sob a orientação do professor Me. Antônio Mateus Locci.

**Limeira**

**2018**

**Sergio Eduardo Nunes**

**Análise de falhas de segurança em código**

Trabalho de conclusão de curso apresentado à banca examinadora da Faculdade UNIP, como requisito parcial à obtenção do Bacharelado em ciência da Computação sob a orientação do professor Me. Antônio Mateus Locci.

Aprovada em XX de XXXXX de 201X.

**BANCA EXAMINADORA**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Prof. Dr. Nome completo

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Prof. Me. Nome completo

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Prof. Esp. Nome completo

*“A tarefa não é tanto ver aquilo que ninguém viu, mas pensar o que ninguém ainda pensou sobre aquilo que todo mundo vê”.*

(Arthur Schopenhauer)

**RESUMO**

Este artigo apresenta uma solução algorítmica que analisa falhas de segurança comuns no desenvolvimento de um sistema. O programa foi desenvolvido na linguagem de programação Java com objetivo de analisar outros códigos fontes. Java é uma linguagem muito utilizada entre os desenvolvedores. Abordaremos sobre o algoritmo e também sobre as falhas de segurança de diferentes linguagens, afim de melhorar a qualidade e segurança do código.

Palavra-Chave: Código; Segurança; Falha.

**ABSTRACT**

This article introduces an algorithmic solution which analyzes common problems when a programmer is developing a system. The program was written in Java in order to analyzing others fonts codes. Java has been used for many developers. I will discuss about this algorithm and a lot of security gaps in different program languages in order to improve code security and its quality.

Key Words: Code; Security; Java.

**LISTA DE FIGURAS**

[Figura 1 – Validação de e-mail por ER 16](#_Toc514525756)

[Figura 2 – Exemplo de SQL incorreta 18](#_Toc514525757)

[Figura 3 – Exemplo de SQL alterada por um atacante 18](#_Toc514525758)

[Figura 4 – Método que detecta concatenação de variáveis em consultas SQL 18](#_Toc514525759)

[Figura 5 – Exemplo de como inserir valores na instrução SQL corretamente 19](#_Toc514525760)

[Figura 6 – Método que detecta a falha do tipo *PasswordField* utilizando *getText*. 21](#_Toc514525761)

[Figura 7 – Método que detecta variáveis estáticas que não possui *final* 22](#_Toc514525762)

[Figura 8 – Alterando valores de atributos estáticos. 22](#_Toc514525763)

[Figura 9 – Detectando socket que não possui criptografia SSL 23](#_Toc514525764)

[Figura 10 – Exemplo de *Unboxing* 24](#_Toc514525765)

[Figura 11 – Aceitação de valores nulos 24](#_Toc514525766)

[Figura 12 – Valor nulo sendo atribuído a variável primitiva 24](#_Toc514525767)

[Figura 13 – Identificando objetos *wrapper* no código 25](#_Toc514525768)

[Figura 14 – Incluindo página recuperada por parâmetro 26](#_Toc514525769)

[Figura 15 – Identificando uma possível má manipulação de URL 27](#_Toc514525770)

**LISTA DE QUADROS**

[Quadro 1 – Metacaracteres 15](#_Toc513393013)

[Quadro 2 – Quantificadores 15](#_Toc513393014)

[Quadro 3 – Metacaracteres de Fronteira 15](#_Toc513393015)

[Quadro 4 – Agrupadores 16](#_Toc513393016)

**LISTA DE ABREVIATURAS**

ER Expressões Regulares

GET Verbo do HTTP para requisição de dados

HTTP *Hypertext Transfer Protocol* (Protocolo de Transferência de Hipertexto)

SQL *Structured Query Language* (Linguagem de Consulta Estruturada)

SSL *Secure Socket Layer* (Camada de Socket Seguro)

URL *Uniform Resource Locator* (Localizador Padrão de Recursos)

**SUMÁRIO**

[INTRODUÇÃO 11](#_Toc514522566)

[1.1 Objetivo 12](#_Toc514522567)

[1.2 Justificativa 12](#_Toc514522568)

[1.3 Metodologia 13](#_Toc514522569)

[2. O QUE SÃO FALHAS DE SEGURANÇA 13](#_Toc514522570)

[3. UM POUCO SOBRE EXPRESSÕES REGULARES 14](#_Toc514522571)

[3.1 Exemplos de uso 16](#_Toc514522576)

[4. UM POUCO SOBRE A CLASSE *STRING* DO JAVA 17](#_Toc514522577)

[4.1 Método *contains* 17](#_Toc514522578)

[4.2 Método *split* 17](#_Toc514522579)

[4.3 Método *matches* 17](#_Toc514522580)

[5. ESTRATÉGIA DE DETECÇÃO 17](#_Toc514522581)

[5.1 Detectando as linguagens 17](#_Toc514522582)

[5.2 Estratégia de análise 17](#_Toc514522583)

[5.3 Exibindo dados para o usuário 17](#_Toc514522584)

[6. VULNERABILIDADES 17](#_Toc514522585)

[6.1 Exemplos de vulnerabilidade em Java e como o algoritmo as detecta 17](#_Toc514522586)

[6.1.1 Análise de segurança em instruções SQL. 17](#_Toc514522587)

[6.1.2 Analise de segurança em campos do tipo *JPasswordField*. 20](#_Toc514522588)

[6.1.3 Analise de segurança em variáveis estáticas públicas. 21](#_Toc514522589)

[6.1.4 Análise de segurança em sockets 23](#_Toc514522590)

[6.1.5 Análise de *Unboxing* não tratado 23](#_Toc514522591)

[6.2 Exemplos de vulnerabilidade em PHP e como o algoritmo as detecta 26](#_Toc514522592)

[6.2.1 Manipulações de URL em PHP 26](#_Toc514522593)

[CONCLUSÃO 28](#_Toc514522594)

[REFERÊNCIAS BIBLIOGRÁFICAS 29](#_Toc514522595)

INTRODUÇÃO

Expor um problema cotidiano de muitos programadores, onde a falta de boas práticas de segurança na hora do desenvolvimento pode acarretar sérios problemas na manipulação dos dados do usuário.

Frequentemente nos deparamos com problemas de segurança da informação, dados sigilosos sendo acessado por hackers através de falhas de segurança dos sistemas, estes invasores não somente se aproveitam de falhas de segurança de rede, mas também de falhas de logica ou más práticas de programação que acabam acarretando esses tipos de falhas.

As falhas de segurança não estão somente nas redes de computadores, mais também nos sistemas escritos, um sistema bem escrito pode ser mais eficiente do que uma rede bem segura, mas em todo o caso é importante que tanto a rede como o sistema sejam seguros.

Existem várias falhas de segurança nos dias de hoje, como por exemplo a de SQL *Injection* que basicamente é uma falha de logica do programador em programar consultas SQL em conjunto com uma linguagem de programação, um tipo de falha bem comum e que pode ser mortal já que este tipo de falha infringe diretamente o banco de dados.

Outro tipo de falha comum é da má manipulação de dados, que é quando o sistema deixa resíduos de dados sigilosos na memória, dessa forma um usuário de nível avançado pode acabar tendo acesso a está memória e retirar dados importantes de usuários.

E também, outro tipo de ataque comum é a de falhas inesperadas não tratadas em um sistema, caso ocorra alguma falha não tratada e o sistema não consegue se recuperar dessa falha, o sistema trava, dessa forma as empresas precisam parar seus serviços para resolver tal falha, gerando transtorno para o usuário e perda de dinheiro por parte da empresa.

Tratar estes erros é prioridade em qualquer sistema, isso exige tempo por parte da equipe de programação, mas é essencial que sejam corrigidas o máximo de falhas possível para ter um sistema de respeito no mercado.

Mesmo tendo uma equipe experiente de programadores e testadores, é comum que algumas brechas de segurança passem despercebidos por parte da equipe e acabem sendo descobertas depois que acontece alguma invasão, e isso é prejudicial para a empresa pois dependendo da falha e dos dados que foram roubados a empresa pode entrar a falência.

Se o processo de análise pode-se ser automatizado por algum programa de computador, as chances de falha poderiam ser amenizadas, e os desenvolvedores iriam entregar softwares mais robustos e seguros e o custo com manutenções de segurança poderia ser diminuído.

E ainda os programadores aprenderiam sobre estas falhas com o próprio programa, pois o mesmo geraria um relatório sobre as falhas e daria algumas dicas e recomendações sobre como tratar tal falha.

* 1. Objetivo

Desenvolver um programa de computador capaz de detectar algumas falhas de segurança em códigos escritos nas diversas linguagens de programação com o intuito de avisar o programador sobre as mesmas antes que ele lance o seu projeto para os usuários finais.

O programa será desenvolvido na linguagem de programação Java utilizando recursos da mesma no auxílio a detecção destas falhas, e com uma interface fácil de ser manipulada pelo usuário.

1.2 Justificativa

Nos dias de hoje muitos profissionais em segurança da informação buscam técnicas e novas tecnologias para melhorarem seus sistemas que estão a cada dia mais vulneráveis a ataques.

Para amenizar essas falhas de segurança que muitas vezes o programador não consegue enxerga-las em código, pensei em criar este algoritmo que analisaria todo o código em busca destas falhas, afim de mostrar as mesmas para o programador corrigir antes do programa ser disponibilizado para o usuário final.

1.3 Metodologia

A primeira etapa consiste em estudar sobre algumas das falhas de segurança das linguagens de programação, a princípio, estudaremos sobre Java, C++ e PHP.

A segunda etapa consiste em assimilar as falhas de segurança, entender em que momento ela pode ser uma falha ou não, fazer um levantamento das diferentes formas que essa falha pode ocorrer, por que cada programador pode possuir uma lógica diferente.

Após o levantamento de requisitos foi pensado na arquitetura do projeto, em que linguagem será escrita, como o código será dividido, qual estratégia de programação será utilizada, como o relatório será gerado.

Em seguida, utilizaremos a linguagem de programação Java e seus recursos nativos para desenvolver um algoritmo com base nos levantamentos de requisitos descritos acima, utilizaremos da classe *String* que é uma classe nativa do Java e também utilizaremos expressões regulares para fazer a leitura do código e descobrir as falhas de segurança.

Depois de construído a sua estrutura básica genérica que pode ser utilizado para todas as linguagens, será implementado as logicas para capturar estas falhas, em módulos separados pelas suas respectivas linguagens.

Por fim, o algoritmo será testado afim de corrigir o máximo de inconsistências e falsos positivos, e para implementar mais falhas de segurança é só repetir todos os paços descritos acima.

1. O QUE SÃO FALHAS DE SEGURANÇA

Uma falha de segurança ou podemos chamar também de vulnerabilidade, é uma fraqueza existente em qualquer tipo de sistema, seja uma infraestrutura de rede, em um sistema ou em algum componente de hardware que permite a um usuário avançado acesso a determinadas partes do sistema que o mesmo não deveria ter, comprometendo a integridade da informação.

Existem vários exemplos de usuários avançados acessando regiões de memórias restritas e obtendo informações importantes. Geralmente essas falhas ocorrem em sistemas que possuem falhas de logica ou abordagem que proporcionam estas brechas nos sistemas, como por exemplo, uma falha inesperada que não é tratada no sistema, o atacante pode obter alguma vantagem sobre a mesma, afim de obter informações confidencias ou travar o sistema.

É muito difícil criar sistemas que são 100% seguros, é quase uma tarefa impossível, e isso é notável por que até grandes empresas de software que possuem os melhores profissionais são atacadas e algumas delas são levadas a falência por conta de informações sigilosas de usuários serem roubadas e a empresa não conseguir de alguma forma amparar o usuário que não tem nada a ver com a história.

Mesmo fazendo grandes investimentos na área, não há nada garantido de que seu sistema estará a salvo de qualquer ataque, pois novas abordagens de invasão de sistemas são renovadas e reinventadas todos os dias, dessa forma, a guerra nunca acaba, sendo necessário também renovar e reinventar todos os dias as formas de prevenção de ataques.

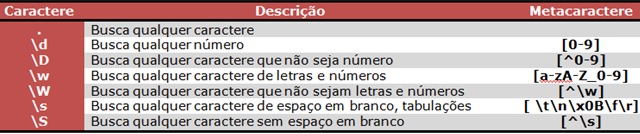
3. UM POUCO SOBRE EXPRESSÕES REGULARES

Expressões Regulares ou também podem ser abreviados como ER em português ou RE ou REGEX em inglês, é uma forma de representação de padrões de texto, o texto em si pode ser qualquer coisa, mais desde que possua um padrão para que possa ser validado.

Para que seja possível a representação de um determinado texto, a ER utiliza de símbolos representacionais que são usados em conjunto para uma determinada representação de um texto, são aquelas famosas letras e números misturados com símbolos que quase ninguém entende.

Iremos abordar estes símbolos e será dado alguns exemplos para que você possa compreender o funcionamento do mesmo e o que cada símbolo significa.

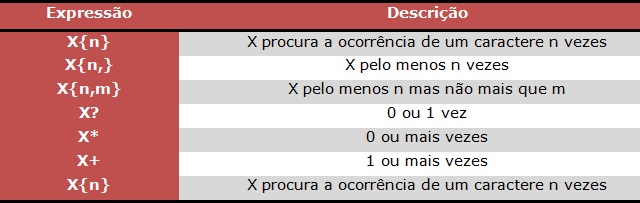
Quadro 1 – Metacaracteres



Fonte: Dev Media (2018)

Metacaracteres são utilizados para indicar ocorrências de números, letras ou qualquer outro caractere especial que o texto possua e que está representado entre os colchetes.

Quadro 2 – Quantificadores



Fonte: Dev Media (2018)

Os Quantificadores agem como controladores dos Metacaracteres, as vezes é necessário que alguns caracteres sejam repetidos afim de se obter um resultado mais diferenciado, na imagem acima o X significa qualquer Metacaracter seguido da quantidade de vezes que será repetido.

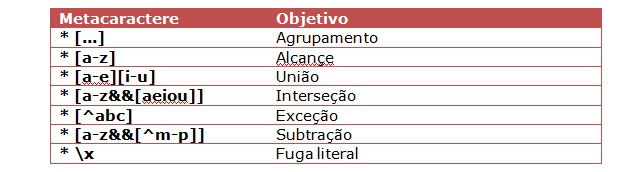
Quadro 3 – Metacaracteres de Fronteira



Fonte: Dev Media (2018)

Os Metacaracteres de fronteira são caracteres delimitadores de texto, com eles é possível dizer onde inicia a busca e onde finaliza, e também é possível analisar duas condições diferentes na mesma ER.

Quadro 4 – Agrupadores



Fonte: Dev Media (2018)

Os Agrupadores como o próprio nome diz, são responsáveis por agrupar os Metacaracteres através dos critérios descritos acima.

E por fim os modificadores, eles são responsáveis por mudar a forma de como o texto será analisada, eles são inseridos no fim da ER, os seguintes modificadores são.

* (?i) Ignora maiúsculas e minúsculas
* (?m) Trabalha com múltiplas linhas
* (?s) Faz com que o caractere possa encontrar novas linhas
* (?x) Permite a inclusão de espaços e comentários

3.1 Exemplos de uso

Existem vários exemplos de uso de expressões regulares e um deles é a validação de campos pelo usuário, como um e-mail, CPF, RG, senha e por ai vai, com expressões regulares é possível validar se tal informação fornecida pelo usuário é válida, a seguir temos um exemplo de validação de e-mail por ER.

Figura 1 – Validação de e-mail por ER

C:\Users\teus\Documents\TCC\Imagens\regexEmail.png

Fonte: Elaborada pelo autor

Nesta expressão temos como início o [\\w](file:///\\w) que é a expressão utilizada para encontrar qualquer letra de a até z e de 0 a 9, com o sinal de + significa que é necessário ter pelo menos uma ou mais ocorrências de letras ou números, em seguida o e-mail deve conter o arroba seguido de uma um conjunto de caracteres, depois o e-mail precisa ter um ponto seguido de três caracteres mais outro ponto seguido de até três caracteres.

O método *matches()* da classe *String* é responsável por interpretar esta expressão regular de acordo com a cadeia de caracteres passada, retornando *true* caso a cadeia de caracteres passada condiz com a validação, ou *false* caso a cadeia não condizer.

4. UM POUCO SOBRE A CLASSE *STRING* DO JAVA

4.1 Método *contains*

Este método é um dos mais utilizados no algoritmo, pois ele é capaz de encontrar uma determinada *String* no texto, o método basicamente retorna um booleano, se a palavra se encontrar no texto ele retornara verdadeiro, se não falso, abaixo será mostrado um exemplo de como aplicar.

Figura 2 – Utilizando método *contains*



Fonte: Elaborada pelo autor

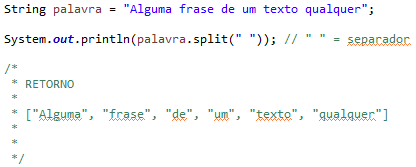
Com esse método é possível mapear todas as palavras chaves dos códigos que são lidos e através de alguma lógica é possível determinar se tal código possui uma falha de segurança ou não.

A grande flexibilidade que o método *contains* traz em relação ao uso de *regex* é que é possível tratar partes da frase isoladamente, dessa forma, se um conjunto de palavras estiver dentro da mesma é possível realizar um tratamento especial.

4.2 Método *split*

Um método interessante da classe *String*, com ele é possível dividir uma determinada frase em pedaços menores e aplicar processamentos mais específicos para cada palavra da mesma, abaixo segue um exemplo de sua utilização.

Figura 3 – Utilizando método *split*



Fonte: Elaborada pelo autor

Ele é bastante utilizado para capturar variáveis nos códigos, através de uma lógica é possível separar a variável do seu tipo primitivo e consecutivamente armazenar a mesma para que possa ser aplicado alguma outra logica para verificar alguma falha de segurança.

4.3 Método *matches*

O método *matches* possui basicamente a mesma função que o método *contains*, mas com a diferença de que este método aceita uma expressão regular ao invés de uma sequencia de caracteres, ele também retorna um booleano que caso a expressão regular der certo, o mesmo retornara verdadeiro, caso contrario falso, segue um exemplo de uso abaixo.

Figura 4 – Utilizando método *matches*



Fonte: Elaborada pelo autor

No exemplo a cima foi utilizado uma simples expressão regular para a validação de um e-mail, com essa ferramenta é possível simplificar as validações de *Strings*, dando uma flexibilidade maior e com uma lógica mais simples.

O algoritmo em si não utiliza deste método, o mesmo utiliza outras abordagens especificas para poder analisar o código de uma forma mais dinâmica e não tão estática como a validação de e-mail.

5. ESTRATÉGIA DE DETECÇÃO

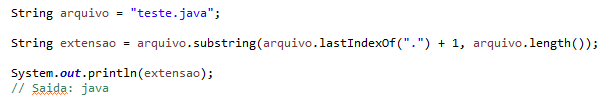
5.1 Detectando as linguagens

Para que o algoritmo tenha uma performance de analise alta, é necessário realizar uma breve verificação de qual linguagem de programação está sendo analisada para que não ocorra uma análise de dados desnecessária, dessa forma o algoritmo iria demorar muito mais tempo e a queda de performance seria visível.

A forma que o algoritmo usa para detectar a linguagem é muito simples, ele verifica a extensão do arquivo, então se o arquivo for do tipo *.java*, ele saberá que se trata de um arquivo Java, se for *.php*, ele saberá que se trata de um arquivo PHP, e segue da mesma forma para todas as linguagens.

Abaixo terá um exemplo básico de como o algoritmo extrai a extensão do arquivo para que essa extensão possa ser comparada com as já declaradas no *enumerator*.

Figura 5 – Extração de extensão de arquivo



Fonte: Elaborada pelo autor

Basicamente o código acima captura do nome do arquivo a extensão retirando o ponto, dessa forma ele retorna a extensão da linguagem de forma pura para que a mesma apontar o fluxo de dados para o modulo correto, isto será explicado no próximo tópico.

Parece ser algo tão simples mais faz uma grande diferença em termos de performance, conseguir prever o tipo de dado que será analisado e através disso realizar uma analise mais otimizada é muito importante, pois além de analisar melhor, será mais rápido e o usuário final ficará mais contente.

5.2 Estratégia de análise

Para uma análise de maior desempenho e também mais precisa, o código foi projetado de forma modular, ou seja, vários módulos conversando entre si para gerar algum dado ou por simplesmente necessitar de dados processados pelos outros módulos da aplicação.

Neste tópico será abordado os principais pontos do algoritmo, como o algoritmo consegue se comunicar com os diferentes módulos da aplicação, também será abordado a parte de otimização de pesquisa e de como o algoritmo redireciona os dados para o modulo correto afim de otimizar o tempo de análise, e como o algoritmo em si analisa as falhas de segurança no código.

Também será abordado a parte logica do algoritmo, ou seja, a parte do código onde ele analisa se há alguma falha de segurança, esta parte do código é puramente logica, em outras palavras, não é utilizado nenhuma biblioteca externa para analise, toda a construção do algoritmo é feita em cima dos recursos nativos da própria linguagem.

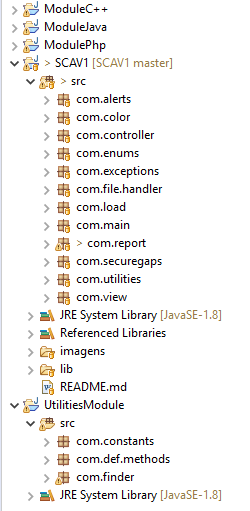
Será explicado agora a parte de módulos do algoritmo, quais são os módulos, o que cada modulo possui, e como os mesmos se comunicam entre si e trocando dados processados e não processados.

O algoritmo é dividido atualmente em cinco módulos (no futuro é possível implementar mais módulos), dentre esses módulos existem dois que são essenciais e que fazem toda a gerencia do algoritmo, que são o modulo gerenciador denominado de SCAV1 e o *UtilitiesModule* que é responsável por encapsular as interfaces e métodos reutilizáveis que são de extrema importancia para a comunicação entre os módulos.

Os demais módulos são os específicos de cada linguagem de programação, esses módulos são responsáveis por encapsular toda a logica de analise das falhas de segurança, dessa forma fica muito fácil implementar novas falhas de segurança futuras para serem analisadas pelo algoritmo.

Um modulo basicamente possui uma divisão em pacotes, essa divisão é importante para que a falha de segurança fique em um diretório único e separado, para que a manutenção e implementação sejam mais rápidas. Abaixo segue a imagem da divisão dos módulos e pacotes.

Figura 6 – Estrutura de módulos

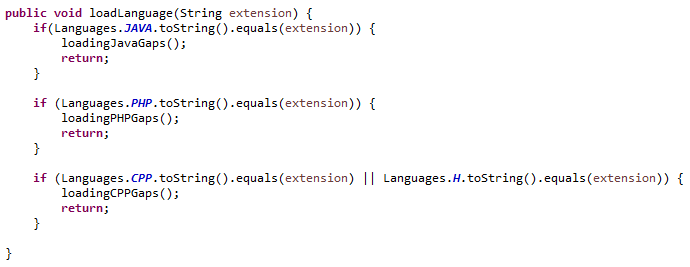


Fonte: Elaborada pelo autor

Acima estão os módulos atuais e alguns módulos abertos como exemplo para demonstrar a parte de organização em pacotes que a aplicação segue, ajudando na organização e manutenção do código.

Agora será explicado sobre como o algoritmo se comunica com outros módulos, para isso iremos mostrar uma classe importante do projeto que é chamada de *LoadClasses*, ela é a responsável por fazer o carregamento por demanda das falhas de segurança e também por redirecionar o fluxo de analise para os módulos corretos como por exemplo, se o algoritmo estiver analisando um arquivo em Java, ele ira apontar para o modulo Java e assim com os demais módulos, abaixo será mostrado o método que detecta a linguagem que será analisada e ira carregar as instancias do modulo para realizar a análise.

Figura 7 – Detectando linguagem e carregando classes



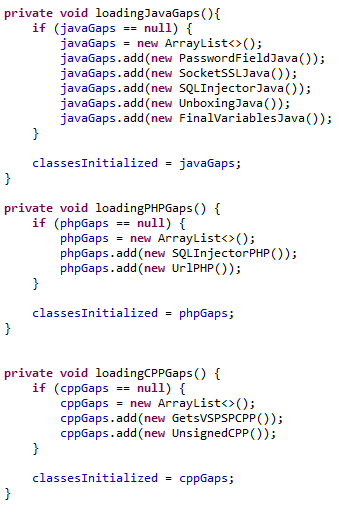
Fonte: Elaborada pelo autor

O que esse método basicamente faz é verificar a extensão do arquivo e assim ele sabe qual é a linguagem que está sendo analisada e carrega as instancias do modulo daquela linguagem invocando o método *loadingJavaGaps* que fara o carregamento por demanda das instancias.

No momento isso não traz um ganho de performance visível, mais com o passar do tempo e com o crescimento do algoritmo, isso pode ser visível e ser visto como um diferencial no desempenho do mesmo.

Abaixo será mostrado os métodos responsáveis por fazer este carregamento, a lógica e os códigos são bem simples mais fornecem o carregamento por demanda do algoritmo.

Figura 8 – Carregando instancias do modulo



Fonte: Elaborada pelo autor

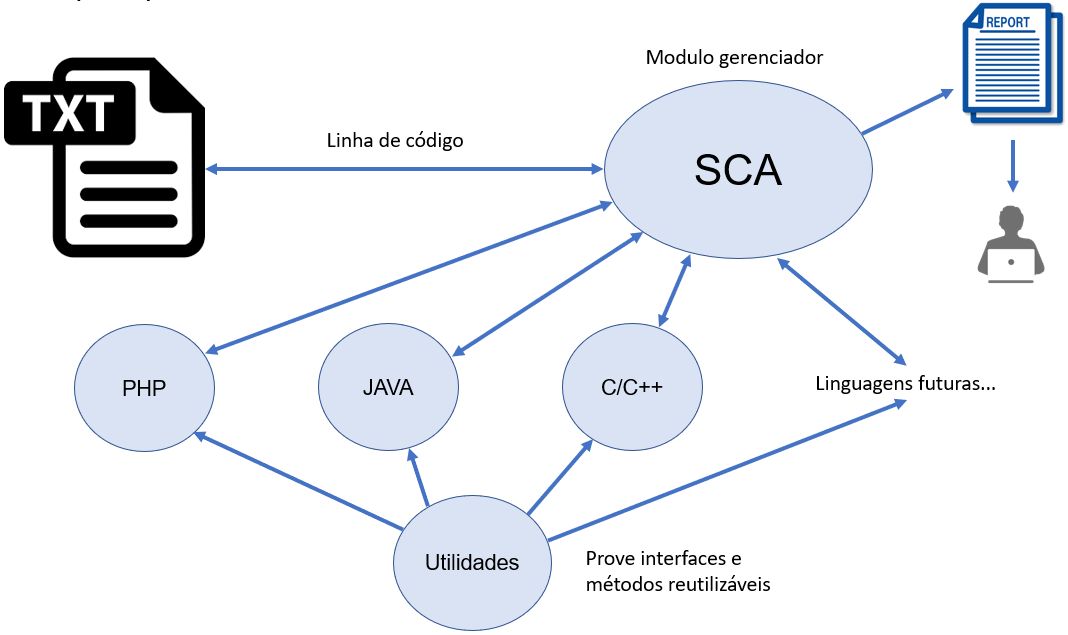
Oque basicamente cada método faz é verificar se a lista com as instancias dos objetos de cada modulo esta instanciada, se não estiver ou seja, se for igual a nulo ele instancia a lista e carrega dentro dela todas as instancias daquele modulo, e atribui em seguida para uma lista global que fara a gerencia das sub listas de cada modulo.

Dessa forma é carregado apenas o necessário, e de quebra, o fluxo de dados será redirecionado para essa lista, dessa forma, a análise é focada apenas no que é necessário e assim o desempenho aumenta.

E o algoritmo cresce de uma maneira fácil pois quando mais módulos forem adicionados, o programador simplesmente criara um novo método seguindo os mesmos padrões de programação, com isso, o algoritmo já se encarrega de realizar o carregamento automaticamente.

Agora será mostrado como a leitura e comunicação entre os módulos ocorre, antes de aprofundar na parte do código, será mostrado um desenho para que o entendimento superficial fique mais claro antes de abordarmos a parte técnica, abaixo segue o desenho.

Figura 9 – Esboço



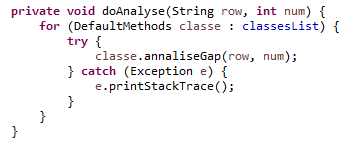
Fonte: Elaborada pelo autor

Tudo começa pelo modulo gerenciador, ele é o responsável por fazer toda a gestão do algoritmo, ele inicia a operação identificando a linguagem que esta sendo analisada, em seguida, carrega as instancias do modulo daquela linguagem, ou seja, estabelece uma comunicação entre o modulo, em seguida faz a leitura do código em texto puro (*String*), essas linhas de código que são lidas são redirecionadas para seu determinado modulo, então o mesmo faz a analise bruta destas linhas de código e armazena os dados em um objeto especifico.

Depois de toda a analise do arquivo, os dados são recolhidos e pelo modulo gerenciador para que o mesmo possa trata-los e gerar um relatório que o usuário final, no caso um programador, visualize e tome as providencias necessárias para fixar as falhas de segurança.

Depois desta rápida introdução em desenho sobre o funcionamento do algoritmo, entraremos agora na parte técnica onde será apresentado o código responsável por fazer esta comunicação com o modulo, segue o código abaixo.

Figura 10 – Método que envia as linhas de código para o modulo

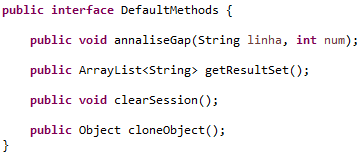


Fonte: Elaborada pelo autor

Basicamente o que este método faz é receber a linha de código e o número onde essa linha se encontra, e em seguida faz um laço chamando cada instancia daquele modulo e passando por parâmetro a linha e o numero da mesma, e dessa forma que o modulo gerenciador conversa com os outros módulos do algoritmo.

Isso tudo é possível graças a uma interface genérica que possui os métodos necessários para fazer a conexão entre os mesmos, o nome dessa interface é *DefaultMethods*, cada instancia deve obrigatoriamente implementar esta interface para que o modulo gerenciador a reconheça, segue abaixo a interface e suas assinaturas de método.

Figura 11 – Interface principal

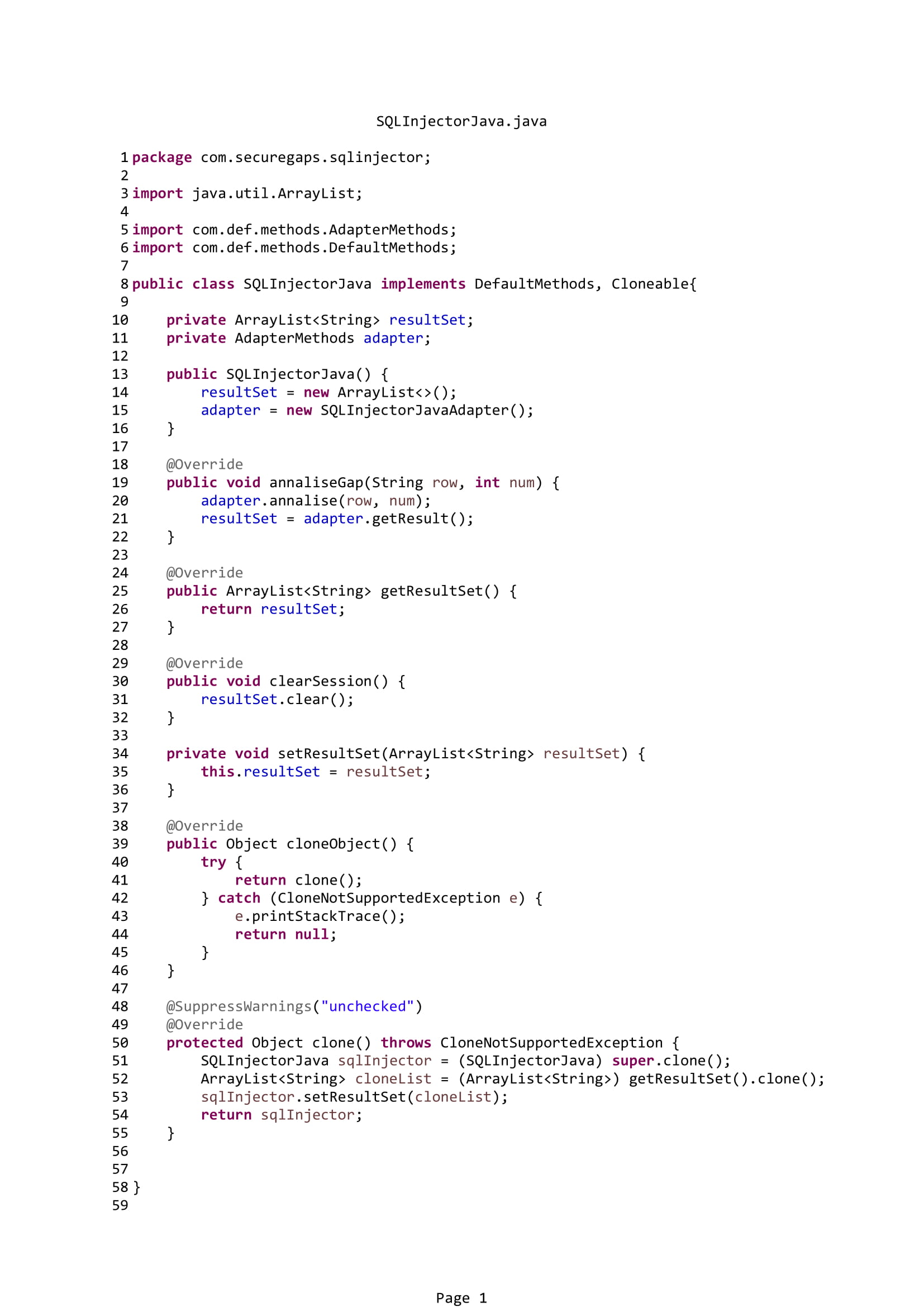


Fonte: Elaborada pelo autor

Esta é a interface essencial para que o modulo gerenciador consiga realizar a gestão dos dados, sem ela, não é possível analisar e nem recolher os dados da análise.

Agora será mostrado um exemplo da sua implementação em uma classe e como a mesma recebe os dados, clona o objeto e limpa a sessão para que o próximo arquivo seja analisado, segue o exemplo abaixo.

Figura 12 – Exemplo de implementação da interface



Fonte: Elaborada pelo autor

No construtor desta classe são instanciadas uma lista global do tipo *String* e uma instancia global da classe que é responsável por toda a logica e analise das linhas de código.

Em seguida o método *annaliseGap* é o responsável por receber linha por linha do código e passa-las para a implementação, depois que a analise termina, a lista global é atualizada recuperando o resultado da analise naquele instante, dessa forma se houver alguma alteração, o objeto é atualizado de forma instantânea, isso nada mais é que um ponteiro apontando para uma região da memória.

O método *clearSession* fara nada mais do que limpar a sessão, depois que o arquivo completo é analisado, seus dados são recolhidos e em seguida todos os objetos que armazenam esses dados são limpados para que o próximo arquivo seja analisado.

E por fim o método *clone* que é um método da interface *Cloneable*, este método da suporte a clonagem de objetos, ou seja, consigo criar uma nova instancia daquele objeto e com os mesmos valores já alocados, dessa forma, não é necessário criar um método a mão para fazer tal clonagem, o próprio Java com esta interface consegue realizar a clonagem do objeto.

5.3 Exibindo dados para o usuário

6. VULNERABILIDADES

6.1 Exemplos de vulnerabilidade em Java e como o algoritmo as detecta

6.1.1 Análise de segurança em instruções SQL.

Inicialmente o algoritmo é capaz de detectar as falhas mais comuns como a concatenação de valores diretamente na *String* da consulta SQL*,* gerando uma brecha de segurança para ataques do tipo SQL *Injector*.

Este tipo de ataque tem como objetivo alterar ou inserir comandos SQLno momento da execução, por um exemplo, o algoritmo está programado para fazer uma consulta de um usuário especifico na tabela de usuários, para que tal consulta ocorra é necessário criar uma variável para receber o e-mail ou código do usuário para que possa ser utilizado na consulta. As vezes por falta de conhecimento do programador ou pelo fato de não ter notado o erro, acaba concatenando o valor diretamente na consulta, como este exemplo.

Figura 2 – Exemplo de SQL incorreta

C:\Users\teus\Documents\TCC\Imagens\select_incorreto.png

Fonte: Elaborada pelo autor

Em termos de sintaxe, não está incorreto e a consulta funcionará normalmente, mas, para um atacante, está é uma porta aberta para acabar com o seu sistema em poucos segundos, o atacante pode simplesmente alterar o valor desta variável que está concatenada para apagar os registros de todos usuários, como o exemplo abaixo.

Figura 3 – Exemplo de SQL alterada por um atacante

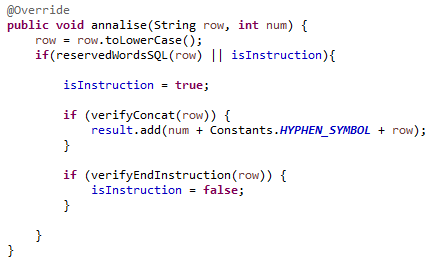
C:\Users\teus\Documents\TCC\Imagens\select_atacado.png

Fonte: Elaborada pelo autor

Como os compiladores de SQLdelimitam as linhas de comando por ponto e vírgula, não havendo necessidade de endentação, este comando seria executado com sucesso e todos os registros dos usuários seriam deletados.

Para que o algoritmo consiga detectar este erro, é necessário analisar a linha por linha do código em busca de indícios de concatenação de variáveis em código SQL*,* o método para detecção está na imagem abaixo.

Figura 4 – Método que detecta concatenação de variáveis em consultas SQL



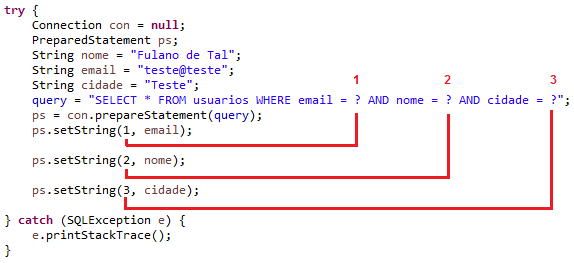
Fonte: Elaborada pelo autor

Este método funciona da seguinte forma, a linha e o número da linha do código que está sendo analisado são recebidos por parâmetro, para padronizar a analise sem afetar os dados, foi necessário alterar todas as letras da linha para letras minúsculas, feito isso é necessário verificar se a linha é uma linha de instrução SQL*,* o método ‘*palavrasReservadasSQL’* tem como objetivo verificar se a linha que está sendo analisada naquele momento é uma instrução SQL, se for, ele verificará se existe alguma variável que está concatenada com está instrução, se houver, ele adiciona a linha na lista de resultados para ser exibida mais tarde no relatório, não será possível mostrar a funcionalidade dos outros métodos para o artigo não ficar muito longo.

A mesma lógica é utilizada para detecta-las em PHP, porem existem algumas modificações de logica nos métodos por conta da sintaxe das duas linguagens serem diferentes.

Para evitar este tipo de ataque e deixar sua aplicação mais segura, é necessário realizar algumas modificações na hora de inserir os valores na instrução SQLpara que não haja possibilidade de ataques externos. Abaixo será mostrado um exemplo de como corrigir está falha utilizando alguns comandos adicionais.

Figura 5 – Exemplo de como inserir valores na instrução SQL corretamente



Fonte: Elaborada pelo autor

O exemplo acima mostra que o sinal de interrogação é o índice onde o dado será inserido, então no índice 1 o sinal de interrogação será substituído pelo valor do segundo parâmetro passado que no caso é o e-mail, e assim sucessivamente podendo conter inúmeros parâmetros.

Passando-os dessa forma os dados serão encapsulados e a segurança e integridade da instrução SQLserão maiores e evitando ataques de SQL *Injection* e também aumentando o desempenho de sua aplicação.

6.1.2 Analise de segurança em campos do tipo *JPasswordField*.

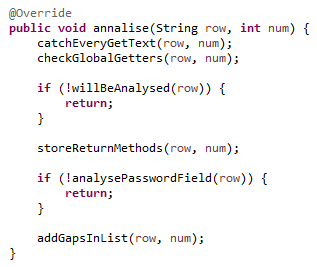
Um problema comum que muitos desenvolvedores que estão começando a programar em Java é criar um campo de texto do tipo *PasswordField* e utilizar o método *getText* para fazer a recuperação da senha do usuário, apesar deste método ter sido depreciado com a utilização de campos do tipo senha, alguns desenvolvedores desavisados ainda utilizam para recuperar informações desses campos, e isto pode ser uma grande falha de segurança.

A falha está no fato do *getText* recuperar o texto do campo no formato de *String,* mas o que tem de ruim nisto? A parte ruim está em como o Java as manipula*,* quando um campo deste tipo é recebido pela JVM, a mesma cria um objeto desta *String* e armazena este objeto na *String Pool* que é onde o Java armazena as *Strings* para servirem de referência quando for necessário efetuar uma comparação entre as mesmas.

E o problema está aí, as *Strings* ficam armazenadas nessa memoria em sua forma pura. Desta forma qualquer atacante que obtiver acesso a mesma terá os dados completos de usuário e senha da sessão, e está é a última coisa que o desenvolvedor quer que aconteça.

Pensando nisso o algoritmo possui uma implementação para detectar está falha, ele possui a mesma lógica de análise da falha que vimos anteriormente, mas agora será verificado outros valores e a sua abordagem é um pouco diferente da anterior, veja o código abaixo.

Figura 6 – Método que detecta a falha do tipo *PasswordField* utilizando *getText*.



Fonte: Elaborada pelo autor

A lógica deste método é bem simples, a linha que está sendo analisada é recebida e passa por um filtro, da mesma forma que é analisado as falhas de SQL,mas com outros parâmetros.

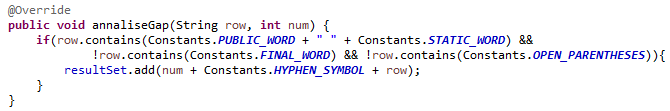
Ele irá verificar se a linha é promissora para ser analisada, caso for, ele passa pelo segundo filtro que fará a extração das variáveis de forma dinâmica, e com as variáveis desse tipo em mãos, é possível analisar todo o código dinamicamente e verificar se ela está sendo usada em conjunto com o *getText* para obter o valor.

6.1.3 Analise de segurança em variáveis estáticas públicas.

Um dos grandes fatores de um sistema é a imutabilidade, muitas das vezes alguns atributos precisão ter um valor único, imutável, de forma necessária, afim de manter o funcionamento correto de uma determinada classe.

Declarar variáveis públicas estáticas que não possuam o modificador *final* é uma má pratica de programação e ao mesmo tempo uma falha de segurança, pois este atributo pode ser facilmente modificado. Afim de detectar estas falhas foi desenvolvido o método a seguir.

Figura 7 – Método que detecta variáveis estáticas que não possui *final*



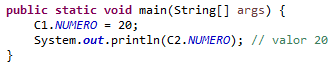
Fonte: Elaborada pelo autor

Este método identifica a sequência de palavras que declaram de forma correta uma variável estática, caso a linha possua *public static* mas não possui o *final*, ela será detectada e armazenada em uma lista para que mais tarde possa ser gerado o relatório com este erro.

O modificador *static* muda o escopo do atributo, ou seja, ao invés dele pertencer a instancia do objeto, ele pertence à classe e com isso pode ser invocado diretamente na classe.

O problema está nesta facilidade de acesso que o modificador *static* proporciona, quando este atributo não é declarado com o modificador *final* seu valor pode ser facilmente alterado e afetará todas as instâncias dessa classe.

Figura 8 – Alterando valores de atributos estáticos.



Fonte: Elaborada pelo autor

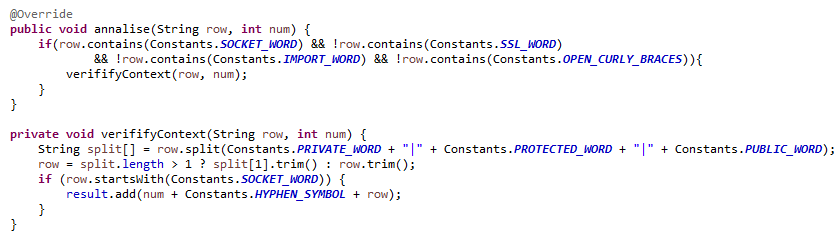
A imagem a cima é um exemplo dessa falha, a classe C1 herda da classe C2 a variável estática número que inicialmente possui o valor 0, em seguida o seu valor é alterado para 20 e todas as instâncias tanto da classe C1 como da C2 serão afetadas com esse novo valor, isto é uma má pratica pois um invasor pode ter a liberdade de manipular este dado e afetar partes do sistema.

6.1.4 Análise de segurança em sockets

Grande parte dos sistemas desenvolvidos para internet utilizam sockets para estabelecer conexões entre cliente e servidor de forma segura, para isso na maioria dos casos é utilizado o protocolo SSL (*Secure Socket Layer*).

Para detectar esta falha em código é bem simples, quando o código está sendo analisado ele verifica se possui a palavra reservada SSL que é provida da biblioteca *javax.net.sll*.

Figura 9 – Detectando socket que não possui criptografia SSL



Fonte: Elaborada pelo autor

Basicamente o que este método faz é verificar se o objeto socket possui uma declaração SSL em sua linha, se não tiver, a linha é armazena em uma lista para ser exibida no relatório.

Para corrigir este problema basta utilizar a classe *SSLSocketFactory* no lado do cliente, e no lado do servidor utilize *SSLServerSocketFactory* para que o servidor possa manda o certificado de autenticação e a conexão segura seja estabelecida.

6.1.5 Análise de *Unboxing* não tratado

*Unboxing* é o processo em que dados primitivos do Java que são *int*, *char*, *float*, *double*, *byte*, *short* e *boolean*, recebem dados de suas respectivas classes *wrappers*, que são *Integer*, *Character*, *Float*, *Double*, *Byte*, *Short* e *Boolean*.

Figura 10 – Exemplo de *Unboxing*

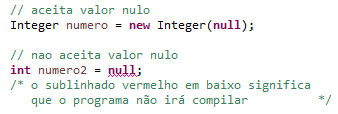
C:\Users\teus\Documents\TCC\Imagens\exemploUnboxing.png

Fonte: Elaborada pelo autor

Está é uma pratica muito comum quando se trabalha com *Lists* desses objetos para fazer alguma manipulação de dados, mas, existe um problema sério no que diz respeito a dados nulos.

No Java, classes *wrappers* aceitam valores nulos, já os atributos primitivos não aceitam esses tipos de valores, gerando um erro de interpretação como mostrado abaixo.

Figura 11 – Aceitação de valores nulos

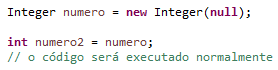


Fonte: Elaborada pelo autor

O interpretador não permitirá que o código seja compilado por conta do valor nulo sendo atribuído a variável primitiva, até aí tudo bem, mas surge um problema em relação a tudo isto.

No processo de unboxing, que seria a atribuição de um valor de um objeto *wrapper* correspondente a variável primitiva que está recebendo o valor, pode ocorrer desta variável receber um valor nulo, como mostrado no exemplo abaixo.

Figura 12 – Valor nulo sendo atribuído a variável primitiva



Fonte: Elaborada pelo autor

Como havia dito anteriormente, variáveis primitivas não aceitam valores nulos, mas, repare no trecho de código acima, o objeto “numero” é instanciado com o valor nulo e logo em seguida é atribuído este objeto para uma variável primitiva, sendo assim um processo de *unboxing*.

Mas repare que se “numero” vale nulo e este valor é atribuído a uma variável primitiva, por que o interpretador do Java não sublinhou a linha em vermelho nos avisando que a variável não pode receber o valor nulo? A resposta é por que o interpretador do Java não possui a capacidade de prever tal ação, e então o código será compilado normalmente. Quando o código é compilado e começa a ser executado, uma exceção é gerada, e se a mesma não for tratada pode afetar seu sistema.

O algoritmo consegue identificar com base em alguns cenários se a variável está sendo tratada em algum bloco de tratamento de exceções.

Primeiramente ele precisa localizar no código os objetos *wrapper* primitivos, aqueles os quais citei acima, para conseguir extrair estes objetos em código, desenvolvi a seguinte rotina.

Figura 13 – Identificando objetos *wrapper* no código



Fonte: Elaborada pelo autor

Basicamente ele verifica se a linha é promissora para ser analisada, caso for ele começa o processo de extração desses objetos e armazena os mesmos em uma lista, em seguida ele verifica através de alguns exemplos se o objeto está sendo tratado por algum bloco *try-catch* ou *throws* no método, se não estiver, o objeto é armazenado em uma lista para ser exibida no relatório.

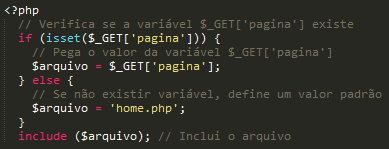
6.2 Exemplos de vulnerabilidade em PHP e como o algoritmo as detecta

6.2.1 Manipulações de URL em PHP

Uma falha comum em alguns sites é a de passar por parâmetro páginas web para serem incluídas na aplicação utilizando o método *include*.

Por exemplo, uma URL [www.site.com.br/?pagina=cadastro.php](http://www.site.com.br/?pagina=cadastro.php) está passando via GET a página que será incluída e exibida para o usuário, o código abaixo é um exemplo disto na pratica.

Figura 14 – Incluindo página recuperada por parâmetro

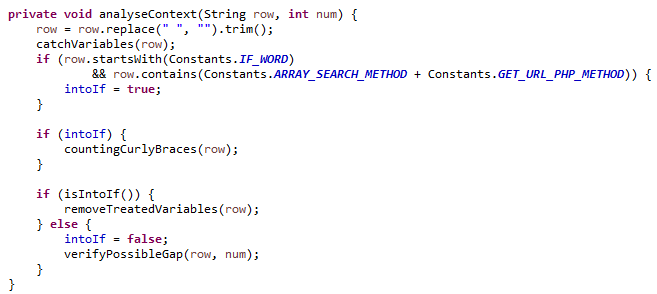


Fonte: Elaborada pelo autor

Isto é uma falha de segurança grave no sistema pois ele não valida qual é a página que está sendo recebida, mais somente se ela existe, e por conta disso, um atacante pode inserir uma página externa sem nenhum problema, como por exemplo a seguinte URL abaixo manipulada por um atacante [www.site.com.br/?pagina=www.malintencionado.com.br/deleta-usuarios.php](http://www.site.com.br/?pagina=www.malintencionado.com.br/deleta-usuarios.php) em seu sistema. Dessa forma o script externo seria executado e deletaria todos os usuários do seu sistema.

Existem algumas formas de solucionar este problema, isto depende da lógica do programador, o programa não consegue afirmar com exatidão se há de fato uma má manipulação na URL, mas com base em alguns exemplos ele pode dizer se de fato a o erro, abaixo está o método.

Figura 15 – Identificando uma possível má manipulação de URL



Fonte: Elaborada pelo autor

Basicamente o método verifica se a variável que armazena a URL está sendo tratada por uma lista de permissões, que é uma das tratativas mais utilizadas para se tratar este tipo de falha, se não estiver, ele armazena a variável em uma lista para ser exibida no relatório.

CONCLUSÃO

O trabalho permitiu...

REFERÊNCIAS BIBLIOGRÁFICAS

BELEM, Thiago. Principais falhas de segurança em PHP. Disponível em: <<http://blog.thiagobelem.net/principais-falhas-de-seguranca-no-php>>. Acessado em 16 Fevereiro 2017.

Caelum. Modificadores de acesso e atributos de classe. Disponível em: <<https://www.caelum.com.br/apostila-java-orientacao-objetos/modificadores-de-acesso-e-atributos-de-classe/>>. Acessado em 26 Julho 2017.

Dev Media. Autoboxing e unboxing em Java. Disponível em: <<https://www.devmedia.com.br/autoboxing-e-unboxing-em-java/28620>>. Acessado em 5 Novembro 2017.

LONG, Fred; DHRUV, Mohindra; C. SEACORD, Robert; F. SURTHERLAND, Dean; SVOBODA David. Java Coding Guidelines: 75 Recommendations for Reliable and Secure Programs. Disponível em: <<https://www.amazon.com/Java-Coding-Guidelines-Recommendations-Engineering/dp/032193315X>>. Acessado em 13 Março 2018.

MCGRAW, Gary, FELTEN, Edward. *Twelve Rules for Developing More Secure Java Code*. Disponível em: <<http://www.javaworld.com/article/2076837/mobile-java/twelve-rules-for-developing-more-secure-java-code.html>>. Acessado em 22 Julho 2017.

SHIFLETT, Chris. Essential PHP Security. Disponível em: <<https://www.amazon.com/exec/obidos/ASIN/059600656X/ref=nosim/chrisshiflett-20>>. Acessado em 13 Março 2018.

Sun Microsystems. *Java Security Code Guidelines for Java SE*. Disponível em: <<http://www.oracle.com/technetwork/java/seccodeguide-139067.html>>. Acessado em 20 Julho 2017.

VIEGA, John. Secure Programming Cookbook for C and C++. Disponível em: <<https://www.amazon.com/Secure-Programming-Cookbook-Cryptography-Authentication/dp/0596003943>>. Acessado em 13 Março 2018.