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 dos professores Me. Antônio Mateus Locci e Me. Marcos Vinicius Gialdi.

Limeira
2018

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 dos professores Me. Antônio Mateus Locci e Me. Marcos Vinicius Gialdi.

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
Figura 2 – Utilizando método *contains* 17
Figura 3 – Utilizando método *split* 18
Figura 4 – Utilizando método *matches* 18
Figura 5 – Extração de extensão de arquivo 19
Figura 6 – Estrutura de módulos 21
Figura 7 – Detectando linguagem e carregando classes 22
Figura 8 – Carregando instancias do modulo 23
Figura 9 – Esboço 24
Figura 10 – Método que envia as linhas de código para o modulo 25
Figura 11 – Interface principal 25
Figura 12 – Exemplo de implementação da interface 26
Figura 13 – Exemplificando a detecção do algoritmo 28
Figura 14 – Relatório de Falhas 29
Figura 15 – Exemplo de SQL incorreta 30
Figura 16 – Exemplo de SQL alterada por um atacante 30
Figura 17 – Método que detecta concatenação de variáveis em consultas SQL 31
Figura 18 – Exemplo de como inserir valores na instrução SQL corretamente 32
Figura 19 – Método que detecta a falha do tipo *PasswordField* utilizando *getText*. 33
Figura 20 – Método que detecta variáveis estáticas que não possui *final* 34
Figura 21 – Alterando valores de atributos estáticos. 35
Figura 22 – Detectando socket que não possui criptografia SSL 35
Figura 23 – Exemplo de *Unboxing* 36
Figura 24 – Aceitação de valores nulos 36
Figura 25 – Valor nulo sendo atribuído a variável primitiva 37
Figura 26 – Identificando objetos *wrapper* no código 38
Figura 27 – Incluindo página recuperada por parâmetro 39
Figura 28 – Identificando uma possível má manipulação de URL 39
Figura 29 – Detectando métodos que indicam vazamento de memória 40
Figura 30 – Adição com variáveis *Unsigned* não tratadas 41
Figura 31 – Análise de *Cross site scripting* no *SonarQube* 42

LISTA DE QUADROS

Quadro 1 – Metacaracteres 15
Quadro 2 – Quantificadores 15
Quadro 3 – Metacaracteres de Fronteira 15
Quadro 4 – Agrupadores 16

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
1.1 Objetivo 12
1.2 Justificativa 12
1.3 Metodologia 13
2. O QUE SÃO FALHAS DE SEGURANÇA 13
3. UM POUCO SOBRE EXPRESSÕES REGULARES 14
3.1 Exemplos de uso 16
4. UM POUCO SOBRE A CLASSE *STRING* DO JAVA 17
4.1 Método *contains* 17
4.2 Método *split* 18
4.3 Método *matches* 18
5. ESTRATÉGIA DE DETECÇÃO 19
5.1 Detectando as linguagens 19
5.2 Estratégia de análise 20
5.3 Exibindo dados para o usuário 28
6. VULNERABILIDADES 30
6.1 Exemplos de vulnerabilidade em Java e como o algoritmo as detecta 30
6.1.1 Análise de segurança em instruções SQL. 30
6.1.2 Analise de segurança em campos do tipo *JPasswordField*. 32
6.1.3 Analise de segurança em variáveis estáticas públicas. 34
6.1.4 Análise de segurança em sockets 35
6.1.5 Análise de *Unboxing* não tratado 36
6.2 Exemplos de vulnerabilidade em PHP e como o algoritmo as detecta 38
6.2.1 Manipulações de URL em PHP 38
6.3 Exemplos de vulnerabilidades em C/C++ e como o algoritmo as detecta 40
6.3.1 Alguns métodos que indicam vazamento de memória 40
6.3.2 Variáveis com modificador *Unsigned* não tratadas 40
7. COMPARATIVO 42
7.1 SonarQube vs SCA 42
7.2 Resultados 43
CONCLUSÃO 44
REFERÊNCIAS BIBLIOGRÁFICAS 45

INTRODUÇÃO

Falhas de segurança em códigos são falhas de lógica que podem ser dados que não são verificados de forma correta ou uma validação mal feita, códigos mal escritos que são o uso incorreto de determinadas bibliotecas ou métodos da própria linguagem de programação que permitem usuários avançados acessarem áreas restritas do sistema afim de prejudicar ou obter informações privilegiadas, sendo uma das brechas mais exploradas pelos usuários avançados.

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.

1.1 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.

2. 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

Caractere	Descrição	Metacaractere
.	Busca qualquer caractere	
\d	Busca qualquer número	[0-9]
\D	Busca qualquer caractere que não seja número	[^0-9]
\w	Busca qualquer caractere de letras e números	[a-zA-Z_0-9]
\W	Busca qualquer caractere que não sejam letras e números	[^\w]
\s	Busca qualquer caractere de espaço em branco, tabulações	[\t\n\x0B\f\r]
\S	Busca qualquer caractere sem espaço em branco	[^\s]

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

Expressão	Descrição
X{n}	X procura a ocorrência de um caractere n vezes
X{n,}	X pelo menos n vezes
X{n,m}	X pelo menos n mas não mais que m
X?	0 ou 1 vez
X*	0 ou mais vezes
X+	1 ou mais vezes
X{n}	X procura a ocorrência de um caractere n vezes

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

Metacaractere	Objetivo
* ^	Inicia
* $	Finaliza
* |	Ou (condição)

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

Metacaractere	Objetivo
* [...]	Agrupamento
* [a-z]	Alcance
* [a-e][i-u]	União
* [a-z&&[aeiou]]	Interseção
* [^abc]	Exceção
* [a-z&&[^m-p]]	Subtração
* \x	Fuga literal

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

```
String email = "fulano@teste.com.br";
System.out.println(email.matches("\\w+@\\w+\\.\\w{2,3}\\.\\w{2,3}"));
// true
```

Fonte: Elaborada pelo autor

Nesta expressão temos como início o \\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*

```
String palavra = "Alguma frase de um texto qualquer";

System.out.println(palavra.contains("frase")); // true
```

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*

```
String palavra = "Alguma frase de um texto qualquer";

System.out.println(palavra.split(" ")); // " " = separador

/*
 * RETORNO
 *
 * ["Alguma", "frase", "de", "um", "texto", "qualquer"]
 *
 *
 */
```

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*

```
String email = "teste@email.com";

System.out.println(email.matches("\\w+@{1}[a-z]+\\.{1}[a-z]{3}"));
// true
```

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

```
String arquivo = "teste.java";

String extensao = arquivo.substring(arquivo.lastIndexOf(".") + 1, arquivo.length());

System.out.println(extensao);
// Saida: java
```

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.

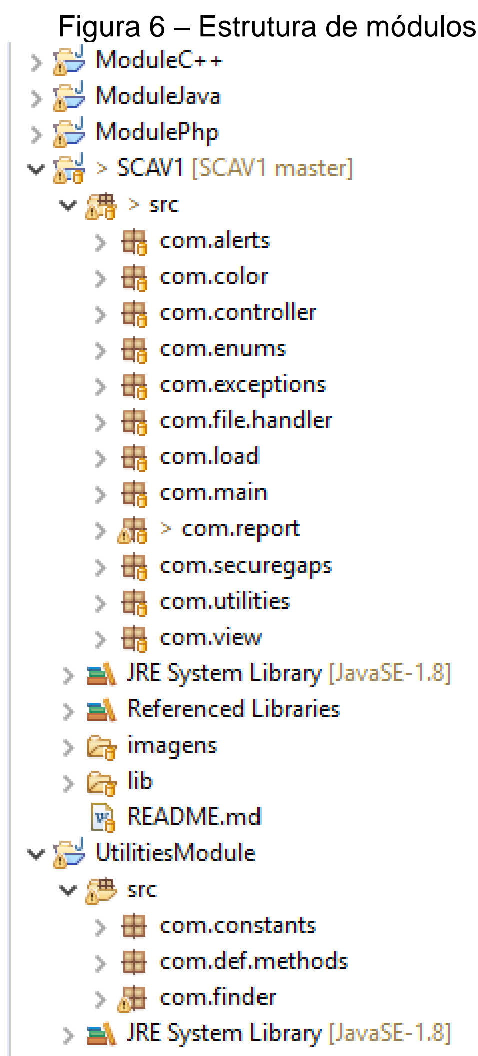
Será explicado agora a parte de módulos do algoritmo, quais são os módulos, o que cada modulo possui, e como os mesmos se comunicam entre si e trocando dados processados e não processados.

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

```
public void loadLanguage(String extension) {
    if(Languages.JAVA.toString().equals(extension)) {
        loadingJavaGaps();
        return;
    }

    if (Languages.PHP.toString().equals(extension)) {
        loadingPHPGaps();
        return;
    }

    if (Languages.CPP.toString().equals(extension) || Languages.H.toString().equals(extension)) {
        loadingCPPGaps();
        return;
    }

}
```

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

```java
private void loadingJavaGaps(){
    if (javaGaps == null) {
        javaGaps = new ArrayList<>();
        javaGaps.add(new PasswordFieldJava());
        javaGaps.add(new SocketSSLJava());
        javaGaps.add(new SQLInjectorJava());
        javaGaps.add(new UnboxingJava());
        javaGaps.add(new FinalVariablesJava());
    }

    classesInitialized = javaGaps;
}

private void loadingPHPGaps() {
    if (phpGaps == null) {
        phpGaps = new ArrayList<>();
        phpGaps.add(new SQLInjectorPHP());
        phpGaps.add(new UrlPHP());
    }

    classesInitialized = phpGaps;
}


private void loadingCPPGaps() {
    if (cppGaps == null) {
        cppGaps = new ArrayList<>();
        cppGaps.add(new GetsVSPSPCPP());
        cppGaps.add(new UnsignedCPP());
    }

    classesInitialized = cppGaps;
}
```

Fonte: Elaborada pelo autor

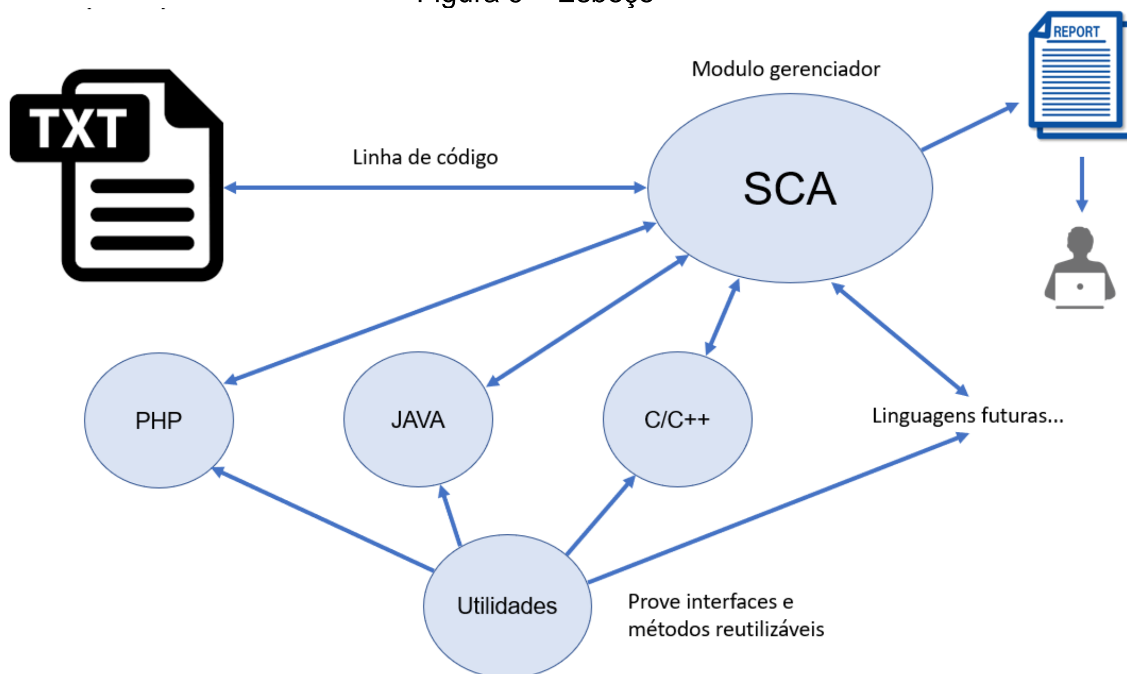
Oque basicamente cada método faz é verificar se a lista com as instancias dos objetos de cada modulo esta instanciada, se não estiver ou seja, se for igual a nulo ele instancia a lista e carrega dentro dela todas as instancias daquele modulo, e atribui em seguida para uma lista global que fara a gerencia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

```
private void doAnalyse(String row, int num) {
    for (DefaultMethods classe : classesList) {
        try {
            classe.annaliseGap(row, num);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

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

```
public interface DefaultMethods {

    public void annaliseGap(String linha, int num);

    public ArrayList<String> getResultSet();

    public void clearSession();

    public Object cloneObject();
}
```

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

SQLInjectorJava.java

```
1 package com.securegaps.sqlinjector;
2
3 import java.util.ArrayList;
4
5 import com.def.methods.AdapterMethods;
6 import com.def.methods.DefaultMethods;
7
8 public class SQLInjectorJava implements DefaultMethods, Cloneable{
9
10     private ArrayList<String> resultSet;
11     private AdapterMethods adapter;
12
13     public SQLInjectorJava() {
14         resultSet = new ArrayList<>();
15         adapter = new SQLInjectorJavaAdapter();
16     }
17
18     @Override
19     public void annaliseGap(String row, int num) {
20         adapter.annalise(row, num);
21         resultSet = adapter.getResult();
22     }
23
24     @Override
25     public ArrayList<String> getResultSet() {
26         return resultSet;
27     }
28
29     @Override
30     public void clearSession() {
31         resultSet.clear();
32     }
33
34     private void setResultSet(ArrayList<String> resultSet) {
35         this.resultSet = resultSet;
36     }
37
38     @Override
39     public Object cloneObject() {
40         try {
41             return clone();
42         } catch (CloneNotSupportedException e) {
43             e.printStackTrace();
44             return null;
45         }
46     }
47
48     @SuppressWarnings("unchecked")
49     @Override
50     protected Object clone() throws CloneNotSupportedException {
51         SQLInjectorJava sqlInjector = (SQLInjectorJava) super.clone();
52         ArrayList<String> cloneList = (ArrayList<String>) getResultSet().clone();
53         sqlInjector.setResultSet(cloneList);
54         return sqlInjector;
55     }
56
57
58 }
59
```

Page 1

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 dá 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.

Basicamente este é o objeto que armazena todos os dados de analise daquela falha descrita como o nome da classe, está é uma classe de exemplo da demais que estão declaradas em outros módulos do algoritmo, mais é basicamente desta forma que o objeto é estruturado.

Agora será abordado a parte logica do algoritmo, onde a analise bruta da falha de segurança é realizada, iremos abordar como é feita, e o fluxo das linhas de código para que o algoritmo consiga analisar o código por completo.

Será mostrado alguns exemplos de detecção de algumas falhas de segurança para exemplificar como o algoritmo trabalha e por que ele trabalha desta forma, será explicado de uma forma mais abstrata e logo em seguida será mostrado as linhas de código que executam tal função, a fim de atrelar teoria a pratica.

Vamos utilizar como exemplo a falha de *SQLInjection* por ser uma falha mais genérica, o entendimento será mais fácil compreender como o algoritmo trabalha na detecção.

Figura 13 – Exemplificando a detecção do algoritmo

```
@Override
public void annalise(String row, int num) {
    row = row.toLowerCase();
    if(Constants.reservedWordsSQL(row) || isInstruction){

        isInstruction = true;

        if (Constants.verifyConcat(row)) {
            result.add(num + Constants.HYPHEN_SYMBOL + row);
        }

        if (verifyEndInstruction(row)) {
            isInstruction = false;
        }

    }
}

private boolean verifyEndInstruction(String row) {
    return row.endsWith(Constants.SIMICOLON_SYMBOL);
}
```

"SELECT * FROM user WHERE name = " + name

Primeiro ele verifica se a linha de código que está vindo é uma String de SQL, se caso for, ele verifica se esta String possui uma concatenação, que no caso seria "+ [concatenação], se ele tiver este simbolo significa que há uma falha de segurança e a linha é capturada.

Fonte: Elaborada pelo autor

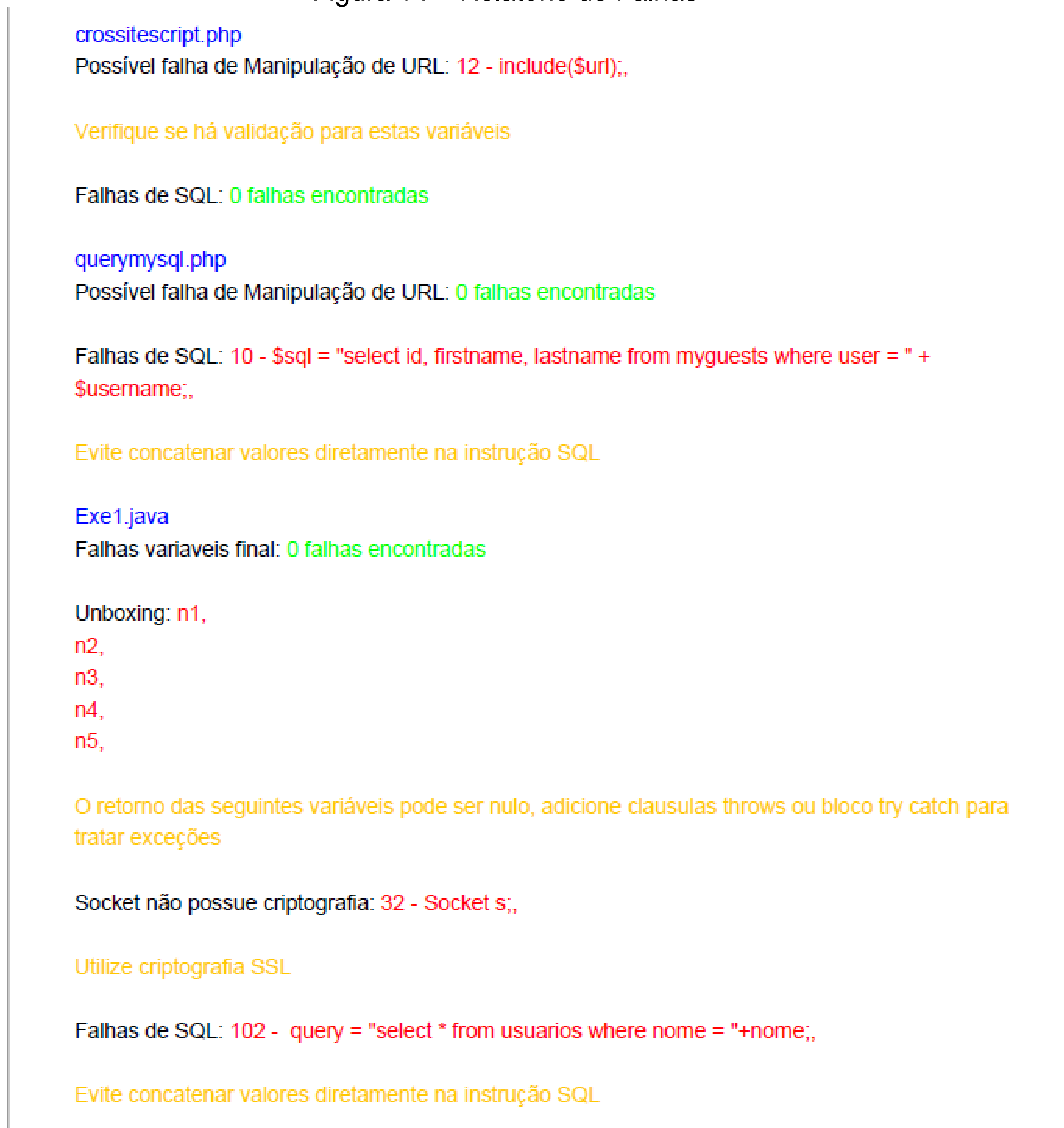
A logica não foge muito das outras falhas de segurança, ele faz um pré-processamento do dado para verificar se aquele dado realmente vale a pena analisar ou não, nesse caso ele verifica se é uma instrução SQL, se caso não for, ele não aprofunda a analise e vai para a próxima linha de código, se caso fosse ele verificaria se a mesma possui concatenação, se tivesse ele capturaria a linha de código para exibir no relatório.

5.3 Exibindo dados para o usuário

As linhas de código capturadas pelo algoritmo precisão de alguma forma serem exibidas para o usuário de uma forma fácil, ou seja, em qual arquivo o mesmo se encontra para que o usuário consiga localiza-la de uma forma rápida e aplicar as devidas correções.

O relatório também fornece uma explicação breve da falha para que o programador tenha ciência do por que ela foi capturada pelo algoritmo, e também para ajudar o programador a encontrar uma solução rápida para a falha.

Figura 14 – Relatório de Falhas



crossitescript.php
Possível falha de Manipulação de URL: 12 - include($url);,

Verifique se há validação para estas variáveis

Falhas de SQL: 0 falhas encontradas

querymysql.php
Possível falha de Manipulação de URL: 0 falhas encontradas

Falhas de SQL: 10 - $sql = "select id, firstname, lastname from myguests where user = " + $username;,

Evite concatenar valores diretamente na instrução SQL

Exe1.java
Falhas variaveis final: 0 falhas encontradas

Unboxing: n1,
n2,
n3,
n4,
n5,

O retorno das seguintes variáveis pode ser nulo, adicione clausulas throws ou bloco try catch para tratar exceções

Socket não possue criptografia: 32 - Socket s;,

Utilize criptografia SSL

Falhas de SQL: 102 - query = "select * from usuarios where nome = "+nome;,

Evite concatenar valores diretamente na instrução SQL

Fonte: Elaborada pelo autor

O relatório acima foi gerado através de um código aleatório de exemplo com várias falhas de segurança para exemplificar a montagem dos dados para o usuário final.

Em azul está o nome do arquivo que foi analisado, e abaixo estão o nome das falhas e quais as linhas de código que estão com a mesma, para facilitar o entendimento para o usuário final.

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 SQL no 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 15 – Exemplo de SQL incorreta

```
email = "teste@teste";
query = "SELECT * FROM usuarios WHERE email ="+email;
```

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 16 – Exemplo de SQL alterada por um atacante

```
email = "''; DROP TABLE usuarios;";
query = "SELECT * FROM usuarios WHERE email ="+email;
//Seria: SELECT * FROM usuarios WHERE email = ''; DROP TABLE usuarios;
```

Fonte: Elaborada pelo autor

Como os compiladores de SQL delimitam 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 17 – Método que detecta concatenação de variáveis em consultas SQL

```
@Override
public void annalise(String row, int num) {
    row = row.toLowerCase();
    if(reservedWordsSQL(row) || isInstruction){

        isInstruction = true;

        if (verifyConcat(row)) {
            result.add(num + Constants.HYPHEN_SYMBOL + row);
        }

        if (verifyEndInstruction(row)) {
            isInstruction = false;
        }

    }
}
```

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 SQL para que não haja possibilidade de ataques externos. Abaixo

será mostrado um exemplo de como corrigir está falha utilizando alguns comandos adicionais.

Figura 18 – Exemplo de como inserir valores na instrução SQL corretamente

```
try {
    Connection con = null;
    PreparedStatement ps;
    String nome = "Fulano de Tal";
    String email = "teste@teste";
    String cidade = "Teste";                          1              2                3
    query = "SELECT * FROM usuarios WHERE email = ? AND nome = ? AND cidade = ?";
    ps = con.prepareStatement(query);
    ps.setString(1, email);

    ps.setString(2, nome);

    ps.setString(3, cidade);

} catch (SQLException e) {
    e.printStackTrace();
}
```

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 SQL serão 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 19 – Método que detecta a falha do tipo *PasswordField* utilizando *getText.*

```
@Override
public void annalise(String row, int num) {
    catchEveryGetText(row, num);
    checkGlobalGetters(row, num);

    if (!willBeAnalysed(row)) {
        return;
    }

    storeReturnMethods(row, num);

    if (!analysePasswordField(row)) {
        return;
    }

    addGapsInList(row, num);
}
```

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 20 – Método que detecta variáveis estáticas que não possui *final*

```
@Override
public void annaliseGap(String row, int num) {
    if(row.contains(Constants.PUBLIC_WORD + " " + Constants.STATIC_WORD) &&
            !row.contains(Constants.FINAL_WORD) && !row.contains(Constants.OPEN_PARENTHESES)){
        resultSet.add(num + Constants.HYPHEN_SYMBOL + row);
    }
}
```

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 21 – Alterando valores de atributos estáticos.

```
public static void main(String[] args) {
    C1.NUMERO = 20;
    System.out.println(C2.NUMERO); // valor 20
}
```

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 22 – Detectando socket que não possui criptografia SSL

```
@Override
public void annalise(String row, int num) {
    if(row.contains(Constants.SOCKET_WORD) && !row.contains(Constants.SSL_WORD)
            && !row.contains(Constants.IMPORT_WORD) && !row.contains(Constants.OPEN_CURLY_BRACES)){
        verififyContext(row, num);
    }
}

private void verififyContext(String row, int num) {
    String split[] = row.split(Constants.PRIVATE_WORD + "|" + Constants.PROTECTED_WORD + "|" + Constants.PUBLIC_WORD);
    row = split.length > 1 ? split[1].trim() : row.trim();
    if (row.startsWith(Constants.SOCKET_WORD)) {
        result.add(num + Constants.HYPHEN_SYMBOL + row);
    }
}
```

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 23 – Exemplo de *Unboxing*

```
Integer numero = new Integer(5);

int numero2 = numero;
```

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 24 – Aceitação de valores nulos

```
// aceita valor nulo
Integer numero = new Integer(null);

// nao aceita valor nulo
int numero2 = null;
/* o sublinhado vermelho em baixo significa
   que o programa não irá compilar          */
```

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 25 – Valor nulo sendo atribuído a variável primitiva

```
Integer numero = new Integer(null);

int numero2 = numero;
// o código será executado normalmente
```

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 26 – Identificando objetos *wrapper* no código

```
@Override
public void annalise(String row, int num) {
    if (!analyseUnboxing(row)) {
        return;
    }

    row = row.replace(row.subSequence(0, row.indexOf(Constants.OPEN_PARENTHESES)),
            Constants.VOID_SYMBOL);
    String[] variables = row.split(Constants.SPLIT_CLASSES_UNBOXING);
    for (String value : variables) {
        value = value.replace(Constants.CLOSE_PARENTHESES + Constants.OPEN_CURLY_BRACES,
                Constants.VOID_SYMBOL);
        value = value.replace(Constants.OPEN_PARENTHESES, Constants.VOID_SYMBOL);

        if (value.contains(Constants.CLOSE_PARENTHESES)) {
            value = value.substring(0, value.indexOf(Constants.CLOSE_PARENTHESES));
        }

        if (value.contains(Constants.COMMA_SYMBOL)) {
            value = value.replace(value.subSequence(value.indexOf(Constants.COMMA_SYMBOL), value.length()),
                    Constants.VOID_SYMBOL);
        }

        if (!value.isEmpty()) {
            result.add(value);
        }

    }
    analyseTreatedUnboxing(row);
}
```

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 está passando via GET a página que será incluída e exibida para o usuário, o código abaixo é um exemplo disto na pratica.

Figura 27 – Incluindo página recuperada por parâmetro

```php
<?php
  // Verifica se a variável $_GET['pagina'] existe
  if (isset($_GET['pagina'])) {
    // Pega o valor da variável $_GET['pagina']
    $arquivo = $_GET['pagina'];
  } else {
    // Se não existir variável, define um valor padrão
    $arquivo = 'home.php';
  }
  include ($arquivo); // Inclui o arquivo
```

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 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 28 – Identificando uma possível má manipulação de URL

```java
private void analyseContext(String row, int num) {
    row = row.replace(" ", "").trim();
    catchVariables(row);
    if (row.startsWith(Constants.IF_WORD)
            && row.contains(Constants.ARRAY_SEARCH_METHOD + Constants.GET_URL_PHP_METHOD)) {
        intoIf = true;
    }

    if (intoIf) {
        countingCurlyBraces(row);
    }

    if (isIntoIf()) {
        removeTreatedVariables(row);
    } else {
        intoIf = false;
        verifyPossibleGap(row, num);
    }
}
```

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.

6.3 Exemplos de vulnerabilidades em C/C++ e como o algoritmo as detecta

6.3.1 Alguns métodos que indicam vazamento de memória

Na linguagem de programação C é de extrema importância manter a aplicação concisa e evitar ao máximo funções que podem fornecer esta vulnerabilidade a aplicação, existem algumas funções que não são aconselháveis de se utilizar como as funções *gets, vsprintf e sprintf,* estas funções não possuem um limite de memória para alocar seus dados.

Por exemplo, o usuário deseja armazenar seu nome em uma variável, o programa utiliza da função *gets* para fazer a captura do dado, um vetor de caracteres de 30 posições por exemplo, a função *gets* conseguira fazer esta captura mas ela terá um estoque de memória infinito (no sentido de poder usar toda a memória do computador) e dessa forma ocorre o vazamento de memória, se um atacante tiver acesso a esta variável, ele poderá inserir códigos maliciosos e prejudicar sua aplicação.

Com uma simples expressão regular, é possível detectar se o código fonte está utilizando desses métodos, abaixo se encontra a expressão responsável por essa verificação.

Figura 29 – Detectando métodos que indicam vazamento de memória

```
public static boolean stackOverflowMethods(String row) {
    return row.matches("(.*)(sprintf|vsprintf|gets)\\s*\\((.*)");
}
```

Fonte: Elaborada pelo autor

O método a cima recebe a linha que está sendo analisada, em seguida o teste é feito, caso a linha conter um desses métodos, o mesmo retorna verdadeiro e a linha é adicionada ao relatório de falhas.

6.3.2 Variáveis com modificador *Unsigned* não tratadas

Em C, uma variável que possui o modificador *unsigned* pode somente aceitar valores positivos, por exemplo, a capacidade de uma variável do tipo inteiro é cerca de -2147483648 e 2147483647, se eu atribuir a esta variável o modificador *unsigned* sua capacidade é alterada para 4294967296 ou seja a

parte negativa vira positiva e é somado com a outra parte positiva, dessa forma a variável só aceita números positivos.

Quando uma variável desse tipo recebe um valor menor do que zero e passa pela conversão, o número que é gerado é exorbitante, fazendo uma subtração do range de 4294967296 e o valor passado, por um exemplo, se eu armazenar um valor -6 em uma variável desse tipo, no momento de conversão ele fara 4294967296 – 6 que seria 4294967290 e armazenara este valor na variável.

Isto é uma falha de segurança grave pois este valor pode ser de um estoque de uma empresa, onde o funcionário está fazendo uma baixa de estoque e caso não haja estoque suficiente pode ser agregado valor negativo a mesma e o valor acabar sendo gravado na base de dados.

O algoritmo para detectar esse tipo de falha é complexo e ainda pode ser um pouco impreciso pois ainda depende muito de cenário de como o programador está manipulando estas variáveis, abaixo segue um exemplo de adições não tratadas com essas variáveis.

Figura 30 – Adição com variáveis *Unsigned* não tratadas

```
private void detectAdditionNotTreated(String row, int num) {
    for (String variable : variablesNotTreated) {

        if (!startWith(variable, Constants.PLUS_SYMBOL)) {
            continue;
        }

        variable = variable.replace(Constants.PLUS_SYMBOL, "");

        if (row.contains(Constants.IF_WORD) && (row.contains(variable + Constants.LESS_THAN_SYMBOL)
                || row.contains(Constants.MORE_THAN_SYMBOL + variable))) {
            removeVariable(Constants.PLUS_SYMBOL + variable);
            break;
        }
    }

}
```

Fonte: Elaborada pelo autor

O método acima detecta se a variável recebera um valor positivo depois de uma operação de soma, isso é possível mapeando uma logica simples de que se a variável depois do cálculo for negativo, ele entra na logica para tratar isso, caso contrario a falha é configurada.

O método ainda é impreciso mais é possível detectar alguns cenários simples, contudo, ainda necessita de melhorias para uma melhor detecção.

7. COMPARATIVO

7.1 SonarQube vs SCA

SonarQube é um dos softwares mais conhecidos e utilizados em grandes empresas afim de se analisar códigos malcheirosos, *bugs*, e vulnerabilidades de código.

Contudo o mesmo se baseia em cenários, dessa forma, acaba deixando de encontrar algumas vulnerabilidades que podem ser prejudiciais ao código, sendo esse seu ponto fraco.

De fato, ela é uma ferramenta poderosa que é utilizada em grandes empresas, mas o SCA vem com uma maneira de analisar um pouco diferente, não focada nos cenários mais sim em pontos relevantes do código que podem se configurar uma falha de segurança.

Abaixo segue um exemplo explicito de uma falha de *Cross site scripting* onde a variável no qual recebe o conteúdo não é verificado sua origem e diretamente é incluída no site e o SonarQube não conseguiu detectar esta falha por se basear em cenários.

Figura 31 – Análise de *Cross site scripting* no *SonarQube*

My project > crossitescript.php

My project
/crossitescript.php

14 Lines | 0 Issues | 0.0% Coverage

```php
<?php
        $param = 'redirect_url';
        $url = $_GET[$param];

        // a variavel $url teria que entrar neste if para que a falha não ocorra
        if (array_search($_GET[$param], [/*as url's autorizadas*/])) {
                // .... alguma logica
        }

        // como não entrou e foi injetada diretamente, a falha ocorre
        include ($url);

?>
```

Fonte: Elaborada pelo autor

As linhas vermelhas que estão a frente da linha significa somente que o código não foi testado por nenhum teste unitário, isso não afeta a análise, o que deve ser visto é que a quantidade de *Issues* que está zero, ou seja, o

programa não conseguiu detectar nenhuma vulnerabilidade neste código, e o mesmo é um código vulnerável.

Já o SCA conseguiu detectar a vulnerabilidade por meio de pontos importantes do código que aos poucos foram configurando-se uma falha de segurança, ele identificou que a variável não sofreu nenhum tipo de tratativa ao longo do processamento e logo foi incluída, caracterizando a falha.

Este é um novo conceito de análise que o SCA vem trazendo, não se prendendo exclusivamente a cenários mais também em pontos importantes do código.

Outro fator importante é o desempenho, o SonarQube por ser um software mais completo onde o mesmo detecta possíveis *bugs, code smells* e vulnerabilidades, demorou cerca de dez segundos para analisar o código, já o SCA que é mais focado em vulnerabilidades demorou cerca de 0,450 segundos para analisar o mesmo, menos de meio segundo, com essa velocidade é possível analisar grandes projetos de forma instantânea.

7.2 Resultados

SonarQube é um software completo que abrange varias ferramentas de análise, contudo, algumas empresas necessitam de programas mais enxutos e rápidos para a suas necessidades, no caso o SCA somente analisa vulnerabilidades de código.

Por se tratar de uma análise, o SCA consegue interceptar falhas de segurança independente de cenário e de forma rápida, atendendo assim, grandes projetos com milhares de linhas de código.

Dessa forma o SCA vem como uma alternativa mais especifica que combate somente vulnerabilidades de código afim de se ter um código altamente seguro independente se o mesmo está em risco ou não, até porque um código de qualidade é aquele que se protege mesmo quando não precisa.

CONCLUSÃO

O seguinte trabalho permitiu ampliar meus conhecimentos na área de programação e segurança da informação com ênfase em programação segura, ampliando assim a possibilidade de que uma falha de segurança não necessariamente está em um *firewall* mal configurado ou uma rede não criptografada mais sim em um código fonte mal escrito com brechas de segurança.

Este algoritmo foi construído com o intuito de analisar falhas de segurança em códigos de uma forma rápida e eficiente, em grandes projetos com milhares de linhas de código onde fica difícil a análise e prevenção dessas falhas periodicamente, e retirar o fator humano que é falho e não consegue enxergar todas as vulnerabilidades do código.

O mesmo também possui o intuito de ensinar outros desenvolvedores sobre estas falhas e de como elas podem afetar de forma séria um sistema de informação e assim comprometer a integridade dos dados.

Tendo em mente que o mercado de tecnologia vem crescendo a cada ano e novas abordagens para a área de segurança da informação tem sido bem recebidas para suprir a necessidade do usuário final e dos desenvolvedores se sentirem seguros com o produto que possuem em mãos e dessa forma a indústria continuar crescendo e inovando.

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.