**Análise de falhas de segurança em códigos**

Mateus Renato Rodrigues Spadacio

Universidade Paulista – UNIP

Rua Miguel Guidote, 405 - Egisto Ragazzo, Limeira - SP, 13480-290

Orientadores: Profº Me. Antônio Mateus Locci, Prof° Me. Sergio Eduardo Nunes

**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.

**Palavras-Chave:** Código; Segurança; Java.

**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 is used for many developers. We will discuss about this algorithm and a lot of security gaps in different program languages, in order to improve code security and its quality.

**Keywords:** Code; Security; Java.

**1. INTRODUÇÂO**

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.

As falhas de segurança que serão abordadas em Java são falhas de declarações *SQL*, capturar dados de *JPasswordField* de forma incorreta, não utilização de escopo *final* em variáveis públicas estáticas e a falta de utilização de criptografia SSL em sockets e Unboxing. Já em PHP também serão tratadas falhas de declarações SQL e alguns cenários de url’s sendo manipuladas de forma impropria. E em C++ é possível detectar alguns métodos que podem caracterizar um vazamento de memória e também detectar indícios de variáveis *unsigned* que não possui seu valor verificado, por se tratarem de variáveis que somente aceitam valores positivos.

Pensando nisso foi proposto uma solução algorítmica escrito na linguagem de programação Java utilizando as bibliotecas gráficas da linguagem para interação com o usuário e de métodos da classe *String* que é uma classe nativa do Java.

O algoritmo consiste na análise semântica do código escrito em Java, ele busca por palavras chave da linguagem para identificar elementos inseguros ou maneiras incorretas de se utilizar determinado elemento apenas analisando o código semanticamente.

Em muitos casos onde o fonte possui milhares de linhas de código, seria uma tarefa árdua e que despenderia muito tempo para analisar e buscar estas falhas.

O objetivo do programa é de automatizar esta tarefa para o programador, ele analisaria o código e traria os resultados através de um relatório que é gerado no fim da análise, apontando quais são as falhas e o número da linha onde elas se encontram, dessa forma ficaria muito mais fácil de corrigi-las.

O algoritmo não consegue analisar todas as falhas existentes das linguagens, mais sim algumas das mais relevantes e relatadas entre diversos fóruns de programação na internet, tomando como referência artigos estrangeiros, e a própria documentação das linguagens.

Algumas das falhas em termos de análise são muito complexas e que podem variar de programador para programador, dessa forma algumas das falhas são analisadas através de critérios genéricos, em outras palavras, seria a lógica mais óbvia que um programador faria aquela aplicação.

Dessa forma as chances de se detectar determinada falha é bem maior pois o código é analisado utilizando exemplos genéricos, e que são comumente utilizados pelos programadores.

É possível futuramente implementar mais exemplos e diferentes estilos de programação, pois o próprio algoritmo fornece essa flexibilidade por se tratar de um algoritmo modulado, onde as responsabilidades são divididas e isoladas do sistema, dessa forma cada modulo possui sua própria responsabilidade dentro do sistema e assim fica muito mais fácil dar manutenção e implementar novas detecções.

O algoritmo possui a capacidade de detectar a linguagem que está analisando de forma automática, sem a necessidade do usuário declarar qual linguagem ele quer analisar.

Dessa forma o algoritmo se torna mais independente pois é muito comum nos dias de hoje um projeto possuir mais de uma linguagem de programação, dessa forma, todas as linguagens de programação suportadas serão analisadas.

**2. DESENVOLVIMENTO**

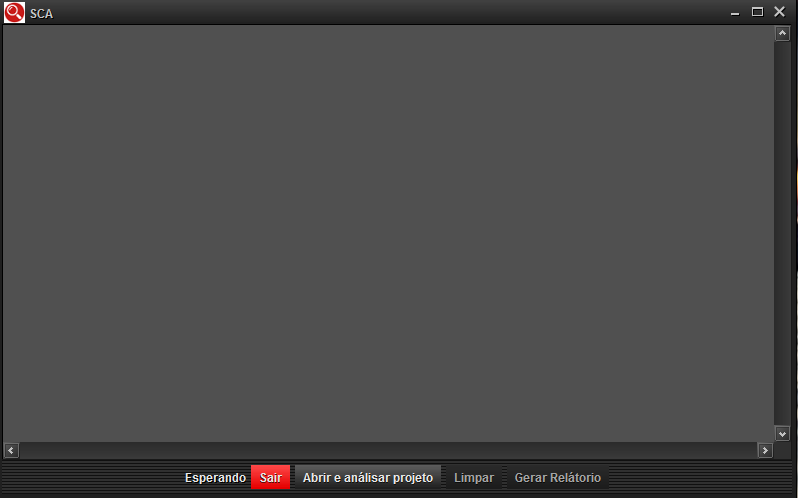
**2.1 Interface gráfica para iteração com o usuário**

Elemento importante onde o usuário terá contato com o algoritmo de uma forma gráfica, foi pensado em uma interface simples, similar aos editores de texto que são uma das ferramentas utilizadas pelos programadores para o desenvolvimento de seus sistemas.

Com poucos botões para que o usuário não se perca com tanta informação, pois trata-se de um programa que somente verificará falhas de segurança de código e que retornará estas informações através de um relatório, para que o usuário localize as falhas mais facilmente.

E por fim, caso o usuário que está utilizando o programa queria reportar as falhas encontradas, o programa possui a funcionalidade de gerar um relatório com todas as falhas encontradas e suas respectivas localizações no código, ou seja, o número da linha onde a falha se encontra.

Figura 1: Interface gráfica do algoritmo.



Fonte: Elaborada pelo autor

**2.2 Analise e dicas de correções**

**2.2.1 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 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

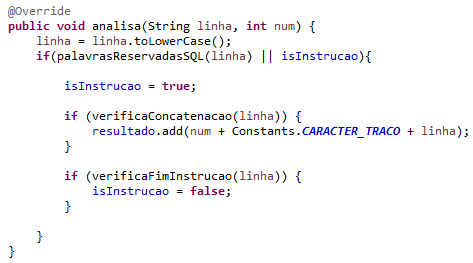
C:\Users\teus\Documents\TCC\Imagens\select_atacado.png

Fonte: Elaborada pelo autor

Como os compiladores de *SQL* delimitam as linhas de comando por ponto e vírgula, não havendo necessidade de indentação, 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*



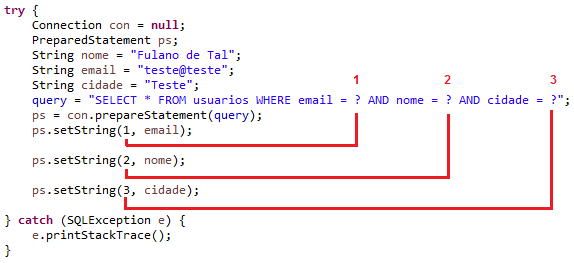
Fonte: Elaborada pelo autor

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 *SQL* para 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 *SQL* serão maiores e evitando ataques de *SQL Injection* e também aumentando o desempenho de sua aplicação.

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

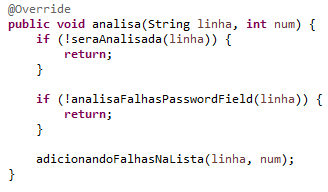
Um problema comum que muitos desenvolvedores que estão começando a programar em Java é criar um campo de texto do tipo *PasswordField* e utilizar o método *getText* para fazer a recuperação da senha do usuário, apesar deste método ter sido depreciado com a utilização de campos do tipo senha, alguns desenvolvedores 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 ai, 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.

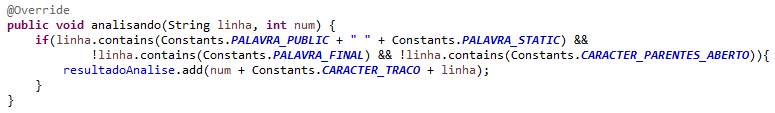
Ele irá verificar se a linha é promissora para ser analisada, caso for, ele passa pelo segundo filtro que fará a extração das variáveis de forma dinâmica, e com as variáveis desse tipo em mãos, é possível analisar todo o código dinamicamente e verificar se ela está sendo usada em conjunto com o *getText*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

**2.2.3 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*



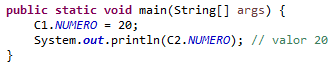
Fonte: Elaborada pelo autor

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

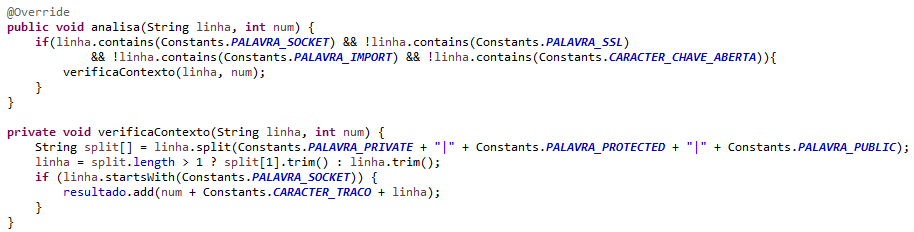
A imagem a cima é um exemplo dessa falha, a classe C1 herda da classe C2 a variável estática número que inicialmente possui o valor 0, em seguida o seu valor é alterado para 20 e todas as instâncias tanto da classe C1 como da C2 serão afetadas com esse novo valor, isto é uma má pratica pois um invasor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

**2.2.4 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.

**2.2.5 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

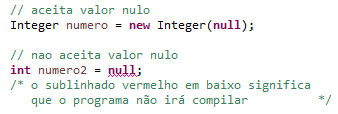
C:\Users\teus\Documents\TCC\Imagens\exemploUnboxing.png

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

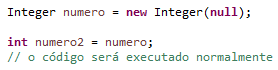


Fonte: Elaborada pelo autor

O interpretador não permitirá que o código seja compilado por conta do valor nulo sendo atribuído a variável primitiva, até ai 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

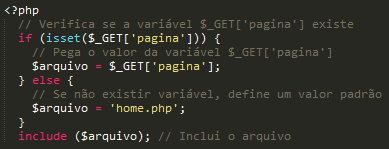
Basicamente ele verifica se a linha é promissora para ser analisada, caso for ele começa o processo de extração desses objetos e armazena os mesmos em uma lista, em seguida ele verifica através de alguns exemplos se o objeto está sendo tratado por algum bloco *try-catch* ou *throws* no método, se não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

**2.2.6 Manipulações de URL em PHP**

Um 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

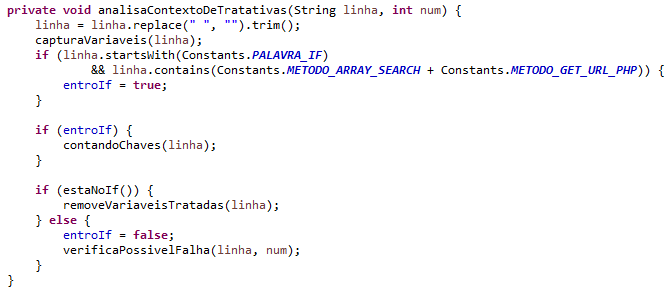


Fonte: Elaborada pelo autor

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 [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.

**2.2.7 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 seu 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.

**2.2.8 Variáveis *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.

**3. CONCLUSÂO**

O presente artigo tem como objetivo auxiliar os programadores a desenvolverem softwares mais seguros, implementando algumas das dicas que foram citadas no discorrer do mesmo.

E para ajudar neste desenvolvimento foi apresentado uma ferramenta que faz a análise de todo o código em busca dessas falhas de forma automática, não tendo a necessidade do programador verificar essas falhas em código já que isto seria uma tarefa mais demorada e menos produtiva e com uma precisão maior já que o algoritmo analisará linha por linha evitando o fator humano que poder ser falho.

Existem muitas técnicas para se programar uma aplicação de forma segura, o algoritmo não consegue detectar todas as falhas, ele detecta apenas as demonstradas acima, para saber mais sobre falhas de segurança e de fatoração, os links da biografia te darão mais informações sobre os temas discorridos aqui.

Quem sabe no futuro o algoritmo possua suporte para analisar todas as falhas e até falhas mais complexas que envolveriam duas ou mais classes Java na análise. Por enquanto, este é apenas um protótipo que necessita de mais estudo e desenvolvimento para poder enfim se tornar um software profissional.

Mas, o importante é que o programador tenha esses conceitos em mente na hora de desenvolver aplicações para web, com o mundo cada vez mais conectado e cada vez mais ataques acontecendo, todo o cuidado é pouco, e até as pequenas brechas podem causar danos significativos em seu sistema, mas tendo um código robusto e seguro, isto pode ser evitado ou atrasado, já que nenhuma aplicação é 100% segura.

**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.