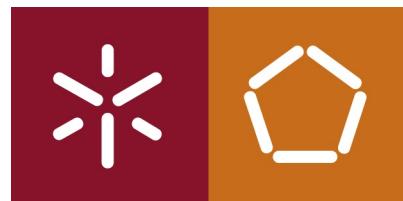


# Universidade do Minho



Mestrado Integrado em Engenharia Informática  
Engenharia de Segurança

---

## Ferramentas e técnicas de *Security*

---

Grupo 8



João de Macedo  
A76268



João Aloísio  
A77953



Nelson Gonçalves  
A78173

11 de Maio de 2020

# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>2</b>
<b>2</b>	<b>Indicador de qualidade de software <i>Segurança</i></b>	<b>3</b>
2.1	O que é? . . . . .	3
2.2	Para que serve? . . . . .	3
<b>3</b>	<b>Ferramentas utilizadas</b>	<b>4</b>
3.1	SAST . . . . .	4
3.2	DAST . . . . .	4
3.3	SCA . . . . .	5
<b>4</b>	<b>Exemplos de ferramentas</b>	<b>6</b>
4.1	Commercial Tools . . . . .	6
4.1.1	Checkmarx SAST (CxSAST) . . . . .	6
4.1.2	Veracode . . . . .	7
4.2	Open Source or Free Tools . . . . .	7
4.2.1	Flawfinder . . . . .	7
4.2.2	Exemplos: . . . . .	8
4.2.3	Brakeman . . . . .	10
4.2.4	Exemplo de um report . . . . .	11
<b>5</b>	<b>Conclusão</b>	<b>12</b>

# 1 Introdução

A verdadeira qualidade de um produto de *software* vê-se após o seu lançamento. Esta qualidade de *software* é definida por 3 pontos:

- número de defeitos encontrados após o lançamento
- a gravidade desses defeitos
- o esforço necessário para resolver esses mesmos defeitos

Os custos de reparação aumentam exponencialmente se forem encontrados posteriormente no processo de desenvolvimento de *software*. Quanto mais tarde uma falha for encontrada, muito maior será o seu custo de reparação. Portanto, a maneira de poupar recursos e dinheiro, é, se possível, arranjar uma maneira de medir a qualidade do *software* de um sistema antes do seu lançamento.

Ao longo deste documento será abordado um dos 8 atributos principais de qualidade de *software* definidos pela *International Organization for Standardization*, sendo este atributo, a **segurança**. O foco está na qualidade do código. Será explicado o que é o indicador de qualidade de **segurança**, para que serve e quais as ferramentas utilizadas consoante as linguagens de programação escolhidas, detalhando ferramentas open-source e apresentando exemplos básicos e avançados da sua utilização.

## 2 Indicador de qualidade de software *Segurança*

### 2.1 O que é?

Segurança, em contexto de qualidade de software, reflete a probabilidade de um atacante conseguir invadir um sistema de software, podendo interromper a sua atividade ou ganhar acesso a informações confidenciais, devido a más práticas de código e de arquitectura. O principal conceito na segurança são as vulnerabilidades do software, ou seja, erros ou falhas que pode resultar na falha ou intrusão do software. Sendo assim, o número e a gravidade de vulnerabilidades descobertas no software é um importante indicador do seu nível de segurança.

### 2.2 Para que serve?

Cada vez mais, os utilizadores confiam nos produtos software para proceder a operações com informações confidenciais relativas à sua vida pessoal e profissional. Dado isto, quanto menos vulnerável a falhas já conhecidas o software for, melhor é. Para isso é medido a segurança de um software através de:

- **Número de vulnerabilidades:** É possível identificar as vulnerabilidades conhecidas até ao momento de uma aplicação software, sendo que dependendo do número de vulnerabilidades encontradas pode ser um bom ou mau indicador.
- **Tempo de resolução:** Qual é o tempo de arranjar uma solução a uma vulnerabilidade que tenha sido encontrada no software.
- **Número de incidentes, a sua gravidade:** O número de vezes que o software sofreu ataques, a gravidade dos ataques e se estes poderiam afetar utilizadores na fase de testes, são também um indicador tanto para o lado positivo ou negativo de um software.

### 3 Ferramentas utilizadas

As ferramentas utilizadas atualmente podem se destinguir em três tipos, SAST, DAST e mais recente SCA, sendo que estas são aplicadas em diferentes fases do software.

#### 3.1 SAST

Para reconhecer as vulnerabilidades de uma aplicação software, atualmente existe diversas ferramentas de análise de código-fonte que podem ser usadas, as Static Application Security Testing (SAST). Estas tem o objetivo de identificar vulnerabilidades antes da sua compilação, ou seja, na fase de desenvolvimento. As principais vantagens destas ferramentas é que permitem ao implementar o software, seja monitorizado o código constantemente e identificar problemas antecipadamente, providenciando assim também de uma forma de mitigar e uma maior integridade do código.

Apesar de estas ferramentas sejam úteis para identificar as vulnerabilidades mais conhecidas tais como, *Buffer Overflow*, não consegue identificar vulnerabilidades que são mais difíceis automaticamente tais como, erros de configuração ou problemas de autenticação.

É importante então escolher as ferramentas certas tendo em conta o tipo de linguagem, as vulnerabilidades e o tipo de código que será analisado.

#### 3.2 DAST

Outro tipo de ferramentas, Dynamic Analysis Security (DAST), que podem ser usadas já na fase de testes da aplicação, sendo que estas são executadas fora do aplicativo software e iniciam solicitações maliciosas contra o mesmo para descobrir vulnerabilidades, analisando as respostas recebidas. Nesta fase, outros tipos de vulnerabilidades que não foram detetadas com o SAST serão detetadas tais como, configurações incorretas. No entanto, ao contrário da análise estática que pode ser usada imediatamente, nesta análise é necessário definir as regras contra esses cenários possíveis, de acordo com a aplicação analisada.

### 3.3 SCA

Quanto às ferramentas do tipo Software Composition Analysis (SCA), surgiram devido ao aumento de uso de códigos open-source por parte da indústria o que dificultou identificar as componentes open-source manualmente usando spreadsheets. Tendo isso como uma necessidade foram desenvolvidas com o objetivo de gerar um inventário com todos os componentes de open-source no software assim como todas as suas dependências. Isto é necessário porque é preciso ter controlo sobre o uso do open-source de maneira a que seja adequado, pois se não sabe ao certo o que é utilizado não é possível proteger. Após a identificação de todos os componentes open-source, é fornecida as informações sobre cada componente, sendo que estas incluem a licença e se há ~~uma~~ vulnerabilidade associada. Ferramentas mais avançadas são capazes de indicar ao implementador a localização preciso de um componente vulnerável, reduzindo assim o esforço usado na correção, a capacidade de alertar sobre vulnerabilidades em ~~um componente que ainda se encontra na~~ página web, sendo evitado que o componente entre no sistema.

## 4 Exemplos de ferramentas

### 4.1 Commercial Tools

#### 4.1.1 Checkmarx SAST (CxSAST)

CxSAST é uma ferramenta, tal como o próprio nome indica, SAST que faz parte da plataforma de exposição de software da empresa conhecida Checkmarx, sendo ela desenvolvida com o objetivo de lidar com os riscos de segurança de software durante a fase de desenvolvimento do mesmo. Esta ferramenta consegue identificar centenas de vulnerabilidades de segurança seja em componentes open source ou personalizados, suportando cerca de 25 linguagens de programação tais como, Java, Python e C++.

O CxSAST fornece resultados obtidos através do scan tais como, relatórios estáticos ou uma interface interativa que permite saber o comportamento do tempo de execução por vulnerabilidade através do código e fornece ferramentas e orientações de forma a corrigir as mesmas. Nestes resultados é também possível eliminar os falsos positivos.

Este scan é feito diretamente no código-fonte, ou seja, não é necessária nenhuma construção ou compilação do próprio, nem uma biblioteca precisa de estar disponível. Com isto, o código nem precisa de ser capaz de compilar corretamente, logo o CxSAST pode executar scan e gerar relatórios de segurança a qualquer momento da fase de desenvolvimento de um projeto de software.

Com isto, CxSAST aponta alguns pontos únicos que a ferramenta contém:

- **Encontra as vulnerabilidades antecipadamente:** Visto que não necessita que o código esteja corretamente para ser compilado nem construído, não necessita de configurações de dependências e na alteração de linguagem não precisa da curva de aprendizagem, esta consegue encontrar antecipadamente as vulnerabilidades;
- **Rápida correção:** Permite aos implementadores corrigir várias vulnerabilidades num único ponto de código usando um exclusivo algoritmo disponível que indica o melhor ponto de correção;
- **Facilidade de automação:** Integrado facilmente em todos os IDEs, ferramentas de que rastreiam bugs e repositórios de origem para aplicar automaticamente uma política de segurança;

#### 4.1.2 Veracode

Veracode é uma ferramenta SAST, tal como a referida acima, mas também apresenta soluções para os outros tipos de ferramentas, DAST e SCA, e ainda teste manual de intrusão. Esta disponibiliza de uma dashboard onde se pode observar o estado da aplicação através de todos os testes. A aplicação é desenvolvida para implementadores e inclui uma API que pode ser adaptada para o software. Fornece também dicas para solucionar as vulnerabilidades que encontra, sendo que, segundo Veracode, os implementadores que trabalham em ambientes DevSecOps, conseguem corrigir erros 11 vezes mais rápido com as suas soluções do que os outros implementadores. Tal como a ferramenta acima, esta suporta diversas linguagens como, Java, Python, C, entre outras.

Assim, Veracode apresenta alguns aspectos sobre a aplicação:

- **Security Feedback While Coding:** Na escrita de código, o scan do IDE providencia feed-back da segurança e ajuda os implementadores rapidamente a remediar, oferecendo exemplos de código, guias e links diretos para tutorias da aplicação;
- **High Accuracy Without Tuning:** A aplicação tem uma taxa menor que 1,1% de falsos-positivos, graças à aprendizagem continua dos mecanismos baseados em SaaS (software como serviço);
- **Focus On Fixing:** Segundo Veracode, para além de encontrar, apresenta soluções em que os utilizadores têm uma taxa maior que 70% de solucionar os problemas que são encontrados, providenciando dicas, guias e tutoriais.

## 4.2 Open Source or Free Tools

### 4.2.1 Flawfinder

Flawfinder é uma ferramenta open-source que permite encontrar algumas das vulnerabilidades mais conhecidas tais como, *buffer overflow*, em linguagens C/C++. Para usufruir desta ferramenta, basta lhe fornecer uma directória ou ficheiro, sendo que dado uma directória, todos os ficheiros com extensão C/C++ irão ser examinados. Após serem examinados vai produzir uma lista que contém possíveis falhas de segurança, que variam entre 0 (pouco risco) até 5 (grande risco), ordenada da vulnerabilidade com maior risco para a de menor risco. Esta ferramenta serve como uma ajuda e pode não encontrar todas as vulnerabilidades do código, sendo que para as que são encontradas apresenta algumas soluções alternativas. Esta lista pode ter o formato de html para melhor compreensão.

#### 4.2.2 Exemplos:

Neste exemplo foi testado num ficheiro de linguagem C, trabalhado na Aula9, *overflowHeap1.c* como podemos ver na imagem:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    char *dummy = (char *) malloc (sizeof(char) * 10);
    char *readonly = (char *) malloc (sizeof(char) * 10);

    strcpy(readonly, "laranjas");
    strcpy(dummy, argv[1]);
    printf("%s\n", readonly);
}
```

Para utilizar a ferramenta, foi utilizado o seguinte comando, **flawfinder -html -context overflowHeap.1.c > results.html**, que vai examinar o código e fornecer uma lista com as possíveis falhas em formato html (-html) e a linha onde existe a falha (-context). Este formato é guardado no ficheiro results.html. Abrindo assim o ficheiro results.html é possível obter a seguinte lista:

## Flawfinder Results

Here are the security scan results from [Flawfinder version 2.0.11](#), (C) 2001-2019 [David A. Wheeler](#). Number of rules (primarily dangerous function names) in C/C++ ruleset: 223

- overflowHeap.1.c:10: [4] (buffer) *strcpy: Does not check for buffer overflows when copying to destination [MS-banned]* ([CWE-120](#)). Consider using *snprintf*, *strcpy\_s*, or *strlcpy* (warning: *strncpy* easily misused).
- ```
strcpy(dummy, argv[1]);
```
- overflowHeap.1.c:9: [2] (buffer) *strcpy: Does not check for buffer overflows when copying to destination [MS-banned]* ([CWE-120](#)). Consider using *snprintf*, *strcpy\_s*, or *strlcpy* (warning: *strncpy* easily misused). Risk is low because the source is a constant string.
- ```
strcpy(readonly, "laranjas");
```

## Analysis Summary

```
Hits = 2
Lines analyzed = 12 in approximately 0.01 seconds (1491 lines/second)
Physical Source Lines of Code (SLOC) = 10
Hits@level = [0] 1 [1] 0 [2] 1 [3] 0 [4] 1 [5] 0
Hits@level+ = [0+] 3 [1+] 2 [2+] 2 [3+] 1 [4+] 1 [5+] 0
Hits/KSLOC@level+ = [0+] 300 [1+] 200 [2+] 200 [3+] 100 [4+] 100 [5+] 0
Minimum risk level = 1
Not every hit is necessarily a security vulnerability.
There may be other security vulnerabilities; review your code!
See 'Secure Programming HOWTO' (https://dwheelera.com/secure-programs) for more information.
```

Como podemos observar nesta imagem, existem duas falhas no código analisado sendo uma de nível 4 e outra de nível 2. Ambas devem-se ao uso da função `strcpy()` que não verifica se o tamanho alocado na variável destino, tem espaço suficiente para a *string* fornecida, que em caso de ultrapassar o limite pode escrever em pedaços de memória indevidos e alterar dados.

Estas diferem no nível de risco porque uma, a nível 2, é fornecido uma *string* constante, tendo assim mais controlo do que lhe é fornecido, mas a de nível 4, é dado como input sem qualquer verificação do tamanho da string. Podemos observar também que é fornecido algumas funções alternativas mais seguras do que `strcpy()`.

Outro exemplo, foi analisado um ficheiro disponível no site da ferramenta Flawfinder, test.c, sendo que este já possui mais linhas de código e apresenta mais riscos de segurança.

Para examinar o ficheiro foi usado o mesmo comando indicado em cima, obtendo assim o output para o ficheiro, result2.html. A seguinte imagem mostra um sumário da quantidade de riscos por níveis:

## **Analysis Summary**

```
Hits = 36
Lines analyzed = 117 in approximately 0.02 seconds (6559 lines/second)
Physical Source Lines of Code (SLOC) = 80
Hits@level = [0] 16 [1] 9 [2] 7 [3] 3 [4] 10 [5] 7
Hits@level+ = [0+] 52 [1+] 36 [2+] 27 [3+] 20 [4+] 17 [5+] 7
Hits/KSLOC@level+ = [0+] 650 [1+] 450 [2+] 337.5 [3+] 250 [4+] 212.5 [5+] 87.5
Suppressed hits = 2 (use --neverignore to show them)
Minimum risk level = 1
Not every hit is necessarily a security vulnerability.
There may be other security vulnerabilities; review your code!
See 'Secure Programming HOWTO' (https://dwheelers.com/secure-programs) for more
information.
```

Podemos observar que neste código de 117 linhas existe muitas mais falhas, 36, sendo que 7 são de nível 5 e 10 são de nível 4. Na imagem em baixo temos alguns riscos de nível 5 e as suas possíveis funções alternativas de modo a mitigar esse risco:

- test.c:32: [5] (buffer) *gets*: Does not check for buffer overflows ([CWE-120](#), [CWE-20](#)). Use *fgets()* instead.  

```
gets(f);
```
- test.c:56: [5] (buffer) *strncat*: Easily used incorrectly (e.g., incorrectly computing the correct maximum size to add) [MS-banned] ([CWE-120](#)). Consider *strcat\_s*, *strlcat*, *snprintf*, or automatically resizing strings. Risk is high; the length parameter appears to be a constant, instead of computing the number of characters left.  

```
strncat(d,s,sizeof(d)); /* Misuse - this should be flagged as riskier. */
```
- test.c:57: [5] (buffer) *\_tcsncat*: Easily used incorrectly (e.g., incorrectly computing the correct maximum size to add) [MS-banned] ([CWE-120](#)). Consider *strcat\_s*, *strlcat*, or automatically resizing strings. Risk is high; the length parameter appears to be a constant, instead of computing the number of characters left.  

```
_tcsncat(d,s,sizeof(d)); /* Misuse - flag as riskier */
```
- test.c:60: [5] (buffer) *MultiByteToWideChar*: Requires maximum length in CHARACTERS, not bytes ([CWE-120](#)). Risk is high, it appears that the size is given as bytes, but the function requires size as characters.  

```
MultiByteToWideChar(CP_ACP,0,szName,-1,wszUserName,sizeof(wszUserName));
```

#### 4.2.3 Brakeman

Brakeman é um scanner de segurança para aplicações Ruby on Rails. Ao contrário de muitos scanners de segurança da web, o Brakeman analisa o código fonte do seu aplicativo. Depois de Brakeman verificar o código da aplicação, ele produz um relatório de todos os problemas de segurança encontrados.

##### Vantagens:

- Não requer instalação ou configuração, basta correr.
- Como os requesitos do Brakeman é o código-fonte, o Brakeman pode ser executado em qualquer fase do desenvolvimento.
- Como o Brakeman não depende de terceiros para, ele pode fornecer uma cobertura mais completa de uma aplicação. Isto inclui páginas que podem não estar "live" ainda. Em teoria, Brakeman pode encontrar vulnerabilidades de segurança antes destas poderem ser exploráveis.
- O Brakeman foi desenvolvido especificamente para aplicações Ruby on Rails, logo pode verificar facilmente as definições de configuração para obter as melhores práticas

##### Limitações:

- Apenas quem desenvolve a aplicação pode entender se determinados valores são perigosos ou não. Por default, Brakeman é extremamente suspeito, isto pode levar a muitos "falsos positivos". **Omissão**

- Os scanners de vulnerabilidade dinâmicos executados num site ativo podem testar o servidor da web e a base de dados ao contrário do Brakeman que não relata se um servidor da web ou outro software tem problemas de segurança.

#### 4.2.4 Exemplo de um report

```
= Brakeman Report =

Application Path: /Users/jcollin/work/brakeman/test/apps/rails5
Rails Version: 5.0.0
Brakeman Version: 4.3.1
Scan Date: 2018-09-24 12:13:04 -0700
Duration: 0.6386 seconds
Checks Run: BasicAuth, BasicAuthTimingAttack, ContentTag, CreateWith, CrossSiteScripting, DefaultRoutes, Deserializing, ForgerySetting, HeaderDoS, I18nXSS, JRubyXML, JSONEncoding, JSONParsing, LinkTo, LinkToHref, MailTo, MassAssignment, PermitAttributes, QuoteTableName, Redirect, RegexDoS, Render, RenderDoS, RenderInline, ResponseSplit, File, SessionManipulation, SessionSettings, SimpleFormat, SingleQuotes, SkipBeforeFilter, SprocketsPathTraversal, XSS

= Overview =

Controllers: 5
Models: 3
Templates: 17
Errors: 0
Security Warnings: 27

= Warning Types =

Cross-Site Scripting: 17
Dangerous Eval: 2
Dangerous Send: 1
Mass Assignment: 2
Path Traversal: 1
Redirect: 1
Remote Code Execution: 1
SQL Injection: 2

= Warnings =

Confidence: High
Category: Cross-Site Scripting
Check: LinkToHref
Message: Unsafe parameter value in `link_to` href
Code: link_to("xss", url_for(params[:bad]))
File: app/views/users/show.html.erb
Line: 7

Confidence: High
Category: Cross-Site Scripting
Check: CrossSiteScripting
Message: Unescaped parameter value
Code: strip_tags(params[:x])
File: app/views/users/sanitizing.html.erb
Line: 3

Confidence: High
Category: Cross-Site Scripting
Check: CrossSiteScripting
```

## 5 Conclusão

Em suma, este trabalho permitiu ao grupo obter mais informações sobre a qualidade de um produto de software, mais especificamente no indicador de segurança.

Como sabemos, a segurança de um produto software é essencial atualmente dado que o utilizador fornece informações confidenciais suas para usufruir do produto. Quanto mais cedo uma falha de segurança for detetada, mais rapidamente esta irá ser tratada, sendo assim podemos recorrer ao uso de ferramentas existentes que nos previne dessas falhas, apresentando-as e fornecendo uma solução para a mesma, em diferentes etapas de desenvolvimento do software.

É importante aplicar boas práticas de desenvolvimento seguro, e ter em mente que nenhuma ferramenta é infalível mas sim mais uma ajuda para obter um código mais seguro.