
André Eduardo Pacheco Dias, Gabriela Moreira Mafra e Lucas Schmitt Seidel

*Estudo comparativo entre 3 analisadores de código C sobre as
ameaças de nível 1 do padrão CERT C 2016*

UNIVERSIDADE DO ESTADO DE SANTA CATARINA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

**André Eduardo Pacheco Dias, Gabriela Moreira Mafra e Lucas
Schmitt Seidel**

**ESTUDO COMPARATIVO ENTRE 3 ANALISADORES DE
CÓDIGO C SOBRE AS AMEAÇAS DE NÍVEL 1 DO PADRÃO
CERT C 2016**

Trabalho de conclusão de curso submetido à Universidade do Estado de Santa Catarina
como parte dos requisitos para a obtenção do grau de Bacharel em Ciência da Computação

Charles Christian Miers
Orientador

Joinville, Março de 2019

ESTUDO COMPARATIVO ENTRE 3 ANALISADORES DE CÓDIGO C SOBRE AS AMEAÇAS DE NÍVEL 1 DO PADRÃO CERT C 2016

André Eduardo Pacheco Dias, Gabriela Moreira Mafra e Lucas Schmitt
Seidel

Este Trabalho de Conclusão de Curso foi julgado adequado para a obtenção do título de Bacharel em Ciência da Computação e aprovado em sua forma final pelo Curso de Ciência da Computação Integral do CCT/UDESC.

Banca Examinadora

Charles Christian Miers - Doutor (orientador)

Resumo

Esse trabalho compara quantitativamente a capacidade de detecção do descumprimento das regras de prioridade 27, 18 e 12 (Nível 1) do padrão de código CERT C de 2016 pelas ferramentas de código aberto: Cppcheck, FlawFinder e LGTM. Ao fim, é determinada a ferramenta de código aberto mais conforme à capacidade de impôr as diretrizes do padrão de código, levando em conta às regras de maior prioridade.

Palavras-chaves: Padrão de Código, Programação Segura, Ferramenta de código livre, Analisador de Código, CERT C

Abstract

This work makes a quantitative comparison of the capacity for noncompliance detection of rules with priority 27, 18 and 12 (Level 1) from CERT C coding standard from 2016 by open source tools: Cppcheck, FlawFinder and LGTM. At the end, the open source tool that is most compliant with the coding standard directives is determined, considering the rules with most priority.

Keywords: Coding Standard, Secure Programming, Open Source Tool, Code Analyzer, CERT C

Conteúdo

Lista de Figuras	7
Lista de Tabelas	8
1 Introdução	9
1.1 Objetivos	10
1.1.1 Objetivos Específicos	10
2 Conceitos	11
2.1 Definição	11
2.2 Histórico	11
2.3 Padrão CERT C	12
2.4 Analisadores de código	12
2.5 Ferramentas escolhidas	13
2.5.1 Cppcheck	13
2.5.2 FlawFinder	13
2.5.3 LGTM	13
2.6 Método de aplicação da ferramenta	14
2.7 Tipos de regras	14
3 Regras	16
3.1 EXP33-C	16
3.1.1 Descrição	16
3.1.2 Código não conforme analisado	16
3.2 EXP34-C	17

3.2.1	Descrição	17
3.2.2	Código não conforme analisado	17
3.3	ARR38-C	18
3.3.1	Descrição	18
3.3.2	Código não conforme analisado	18
3.4	STR31-C	19
3.4.1	Descrição	19
3.4.2	Código não conforme analisado	19
3.5	STR32-C	20
3.5.1	Descrição	20
3.5.2	Código não conforme analisado	20
3.6	STR38-C	21
3.6.1	Descrição	21
3.6.2	Código não conforme analisado	21
3.7	MEM30-C	21
3.7.1	Descrição	21
3.7.2	Código não conforme analisado	22
3.8	MEM34-C	23
3.8.1	Descrição	23
3.8.2	Código não conforme analisado	23
3.9	FIO30-C	23
3.9.1	Descrição	23
3.9.2	Código não conforme analisado	24
3.10	FIO34-C	25
3.10.1	Descrição	25
3.10.2	Código não conforme analisado	26

3.11	FIO37-C	26
3.11.1	Descrição	26
3.11.2	Código não conforme analisado	26
3.12	ENV32-C	27
3.12.1	Descrição	27
3.12.2	Código não conforme analisado	27
3.13	ENV33-C	28
3.13.1	Descrição	28
3.13.2	Código não conforme analisado	29
3.14	SIG30-C	29
3.14.1	Descrição	29
3.14.2	Código não conforme analisado	30
3.15	ERR33-C	30
3.15.1	Descrição	30
3.15.2	Código não conforme analisado	30
3.16	MSC32-C	31
3.16.1	Descrição	31
3.16.2	Código não conforme analisado	32
3.17	MSC33-C	33
3.17.1	Descrição	33
3.17.2	Código não conforme analisado	33
4	Comparativo	34
4.1	Método de aplicação da ferramenta	34
4.1.1	Cppcheck	34
4.1.2	FlawFinder	34
4.1.3	LGTM	34

4.2	Resultados	35
5	Considerações Finais	38
	Referências	39

Lista de Figuras

3.1	Código não conforme para a regra EXP33-C	17
3.2	Código não conforme para a regra EXP34-C	18
3.3	Código não conforme para a regra ARR38-C	19
3.4	Código não conforme para a regra STR31-C	20
3.5	Código não conforme para a regra STR32-C	20
3.6	Código não conforme para a regra STR38-C	21
3.7	Código não conforme para a regra MEM30-C	22
3.8	Código não conforme para a regra MEM34-C	24
3.9	Código não conforme para a regra FIO30-C	25
3.10	Código não conforme para a regra FIO34-C	26
3.11	Código não conforme para a regra FIO37-C	27
3.12	Código não conforme para a regra ENV32-C	28
3.13	Código não conforme para a regra ENV33-C	29
3.14	Código não conforme para a regra SIG30-C	31
3.15	Código não conforme para a regra ERR33-C	32
3.16	Código não conforme para a regra MSC32-C	32
3.17	Código não conforme para a regra MSC33-C	33

Lista de Tabelas

2.1	Severidade de uma regra	15
2.2	Chance de uma regra	15
2.3	Custo de remediação de uma regra	15
2.4	Níveis de prioridades	15
4.1	Detecção de conformidade com cada regra por cada ferramenta	36
4.2	Matriz de confusão para detecção com Cppcheck	37
4.3	Matriz de confusão para detecção com FlawFinder	37
4.4	Matriz de confusão para detecção com LGTM	37

1 Introdução

Com o crescimento do valor em ativos resguardados por software, existe uma tendência ao crescimento de ataques a esses. Essa tendência valoriza a importância de programadores conhecerem as vulnerabilidades existentes, assim como as estratégias de mitigação adequadas (SEACORD, 2005).

Algumas vulnerabilidades de sistemas computacionais são consequências da chamada programação insegura. Programar com segurança significa mitigar as vulnerabilidades produzidas pela execução do código compilado, e necessita de um conhecimento das vulnerabilidades, assim como da linguagem e do compilador, por parte do programador.

Visando centralizar tal conhecimento sobre vulnerabilidades, o padrão CERT C para programação segura foi proposto. Este padrão se aplica para códigos nas linguagens de programação C e C++, e é dividido em regras que especificam procedimentos a serem implementados ou evitados para mitigar vulnerabilidades do código (SNAVELY, 2016).

As regras apresentadas são classificadas por prioridade, que é formada pelas suas avaliações de severidade, chance de ocorrer e custo de remediação. Nesse trabalho, serão envolvidas apenas as regras classificados no nível de maior prioridade (nível 1).

O padrão CERT C ainda indica que hajam verificações automáticas com auxílio de ferramentas analisadoras de código (CARNEIGE MELLON UNIVERSITY, 2016). Essas análises são interessantes durante o processo de desenvolvimento. Com elas, as vulnerabilidades são identificadas o mais cedo possível, permitindo que a correção seja menos custosa e tenha menos impacto no sistema. As ferramentas automáticas ainda permitem uma revisão mais consistente do que a humana, uma vez que não apresenta vícios após ser repetida diversas vezes.

É necessário, contudo, que essas ferramentas sejam capazes de identificar o maior número possível de vulnerabilidades. Para o padrão CERT C, indica-se que seja utilizada a ferramenta que detecta a maior quantidade de não conformidades colocadas no documento (CARNEIGE MELLON UNIVERSITY, 2016).

Assim, no momento de escolha de uma ferramenta de análise, é interessante

que haja um estudo de sua capacidade de detecção, para que o auxílio provido no processo de desenvolvimento seja relevante, e que a segurança do sistema não seja comprometida por falsos negativos.

1.1 Objetivos

Este trabalho tem como objetivo comparar três ferramentas analisadoras com código aberto recomendadas pela OWASP (*Open Web Application Security Project* - Projeto Aberto de Segurança em Aplicações Web) (OWASP, 2019): Cppcheck, Flawfinder e LGTM; de acordo com sua detecção de não conformidades com as regras de nível 1 do padrão CERT C 2016.

1.1.1 Objetivos Específicos

- Preparar códigos não conformes para serem analisados pelas ferramentas.
- Executar as ferramentas analisadoras sobre todos os códigos não conformes.
- Analisar os resultados obtidos a fim de obter uma quantificação por regra.
- Comparar os resultados quantificados entre as ferramentas através de matrizes de confusão.

2 Conceitos

2.1 Definição

Segundo a RedHat (REDHAT DEVELOPER, 2019), programação segura é um conjunto de tecnologias e boas práticas para tornar software tão seguro e estável quanto possível. Ela engloba conceitos desde criptografia, certificados e identidade federada até recomendações para movimentação de dados sensíveis, acesso a sistemas de arquivos e gerenciamento de memória.

Para definir o que faz um programa seguro, existem padrões de programação segura. Esses padrões estabelecem regras bem definidas para separar código conforme de não conforme. Para isso, os padrões são direcionados a uma linguagem de programação específica, e tratam de potenciais vulnerabilidades que emergem daquela linguagem.

2.2 Histórico

A ideia de um padrão CERT para programação segura surgiu nem um encontro do comitê de padrões C, em 2006 (SNAVELY, 2016). O padrão C é um documento oficial, porém mais direcionado a desenvolvedores de compiladores e é considerado obscuro pela comunidade de desenvolvedores mais geral. Um padrão de código seguro seria direcionado primariamente a programadores da linguagem C, guiando-os a programar de forma segura na linguagem.

Com essa ideia, foi criada uma *wiki* onde membros da comunidade e do próprio comitê contribuíram para, ao fim de dois anos e meio, a publicação do primeiro padrão de código seguro CERT C na forma de um livro em 2008. A wiki continuou em desenvolvimento, e gerou uma nova publicação em 2014. A última versão foi gerada em 2016, e está disponível em formato PDF.

2.3 Padrão CERT C

O Padrão SEI CERT C, edição 2016 (CARNEIGE MELLON UNIVERSITY, 2016), determina regras para programação segura na linguagem de programação C, contendo título, descrição, exemplos de não conformidade e a solução conforme. As regras são propostas com o objetivo de desenvolver sistemas seguros e confiáveis, como por meio da eliminação de comportamentos que podem levar a indefinições do programa, gerando vulnerabilidades exploráveis.

As regras determinadas por esse padrão são estabelecidas necessárias para um software que busca confiança e segurança, porém não são suficientes para tornar um programa confiável e seguro.

2.4 Analisadores de código

Ferramentas de análise de código fonte ou Testes Estáticos de Segurança da Aplicação (SAST - *Static Application Security Testing*) tem como objetivo analisar código fonte ou suas versões compiladas para encontrar falhas de segurança (OWASP, 2019). É interessante, para o ciclo de desenvolvimento de software, que essas análises possam ser feitas com alta frequência, diminuindo o tempo de feedback e detectando os problemas o quanto antes.

Os analisadores tendem a ser escaláveis, funcionando bem para códigos com muitas linhas, e a trazer informações completas para o programador, como o número da linha do código onde o problema acontece. São úteis para encontrar alguns problemas específicos com alto grau de confiança, como para erros de *Buffer Overflow*.

Muitos problemas de segurança, contudo, não podem ser encontrados de maneira automática e, portanto, não são detectáveis com esse tipo de ferramenta. Os analisadores disponíveis no momento da escrita desse trabalho são capazes de encontrar apenas uma minoria das falhas de segurança em aplicações no geral. As ferramentas ainda tendem a apresentar numerosos falsos positivos.

Apesar das análises feitas por essas ferramentas poderem ser feitas pelo próprio programador, manualmente, a automação do processo permite uma frequência maior de verificações e tende a ser menos suscetível a erros conforme o aumento da frequência de

análises.

2.5 Ferramentas escolhidas

2.5.1 Cppcheck

A ferramenta de análise de código C/C++ Cppcheck (MARJAMÄKI, 2010) busca detectar problemas não identificados por compiladores. Sendo assim, ela não detecta problemas de sintaxe, buscando alertar potenciais vazamentos de memória, alocações sem respectivas desalocações e vice-versa, *buffer overrun*, e outros problemas dessa família.

O objetivo da ferramenta é anular todos os falsos positivos. Sendo assim, há vários problemas que deixam de ser detectados - mas aqueles que são tem uma probabilidade alta de serem problemas reais. Esse objetivo ainda não foi atingido e a ferramenta está em estado de desenvolvimento.

2.5.2 FlawFinder

O FlawFinder (WHEELER, 2010) é uma ferramenta que examina código C/C++ e reporta possíveis fragilidades de segurança ordenadas por um nível de vulnerabilidade. O objetivo é permitir uma checagem rápida por potenciais problemas de segurança. Algumas das vantagens apontadas pra essa ferramenta incluem a presença de um relatório amigável e detalhado, assim como a facilidade de instalação e uso.

2.5.3 LGTM

O LGTM (MOOR, 2019) é uma plataforma de análise de variantes que checa código em busca de vulnerabilidades. Ela combina uma busca profunda na semântica do código com informações encontradas com ciência de dados para relacionar os resultados mais importantes e mostrar alertas relevantes.

O conceito por trás do funcionamento do LGTM consiste na observação de que os mesmos problemas aparecem repetidas vezes no ciclo de vida de um software e na base de código. Essas repetições podem estar apresentadas em diferentes formas, chamadas variantes. Nesse sentido, sempre que um problema é encontrado, são buscadas variantes

dele e vulnerabilidades semelhantes.

2.6 Método de aplicação da ferramenta

Para realizar os testes de detecção de falhas em códigos não conformes, o seguinte procedimento é adotado:

1. Construção de um código funcional mínimo que inclua o exemplo de não conformidade provido na especificação da regra. O código deve incluir o mínimo de estrutura para ser compilado com sucesso.
2. Execução de cada ferramenta, tendo como parâmetro o arquivo do código contendo a não conformidade.
3. Levantamento de dados sobre o relatório gerado por cada ferramenta. Esse processo deve levar em conta os quatro quadrantes de uma matriz de confusão simples, isso é: quantidade de falsos positivos, falsos negativos, verdadeiros positivos e verdadeiros negativos. Para isso, é considerada que a única vulnerabilidade presente é aquela sendo avaliada.

2.7 Tipos de regras

O padrão CERT C faz avaliações de risco e custo de remediação para cada guia associado a uma regra e recomendação. Com o uso dessas informações, é definida uma prioridade para cada regra, o que pode ser usado como classificação em análises como a feita neste trabalho.

Em (CARNEIGE MELLON UNIVERSITY, 2016), a prioridade é definida com base em três outras avaliações conforme as definições: severidade - quão sérias são as consequências da regra ser ignorada, conforme a Tabela 2.1; chance - quão provável é que uma falha introduzida por ignorar uma regra leve a uma vulnerabilidade explorável, conforme a Tabela 2.2; e custo de remediação - quão custoso é ficar conforme com a regra, conforme a Tabela 2.3.

Os valores dessas três avaliações são multiplicados para obter a prioridade de

Valor	Significado	Exemplos de Vulnerabilidades
1	Baixo	Ataque de negação de serviço
2	Médio	Violação da integridade dos dados
3	Alto	Rodar um código arbitrário

Tabela 2.1: Severidade de uma regra

Valor	Significado
1	Pouco provável
2	Provável
3	Muito provável

Tabela 2.2: Chance de uma regra

Valor	Significado	Detecção	Correção
1	Alto	Manual	Manual
2	Médio	Automático	Manual
3	Alto	Automático	Automático

Tabela 2.3: Custo de remediação de uma regra

uma regra. A Tabela 2.4 traz a classificação das possíveis prioridades em três níveis com possíveis interpretações para seus significados.

Nível	Prioridades	Possível Interpretação
L1	12, 18, 27	Alta severidade, muito provável, sem custo de reparo
L2	6, 8, 9	Média severidade, provável, custo médio de reparo
L3	1, 2, 3, 4	Baixa severidade, pouco provável, custo alto de reparo

Tabela 2.4: Níveis de prioridades

O comparativo neste trabalho trata estritamente das regras classificadas como L1, ou seja, de prioridade 12, 18 ou 27.

3 Regras

3.1 EXP33-C

3.1.1 Descrição

A regra EXP33-C diz respeito a não fazer leitura de memória não inicializada. Esse tipo de leitura é problemática porque são retornados valores indeterminados quando é feito acesso de variáveis cujo valor não foi inicializado, conforme previsto pelo padrão C em (ISO, 2011).

Variáveis locais, alocadas na pilha de execução, assumem o valor presente na pilha. Variáveis alocadas com os alocadores dinâmicos `malloc()`, `aligned_alloc()` e `realloc()` não tem valores inicializados, retornando valores indeterminados se alguma leitura for feita sobre eles.

A execução sobre um valor indeterminado pode gerar comportamentos inesperados que são potenciais vulnerabilidades, podendo servir de via para ataques.

3.1.2 Código não conforme analisado

O código analisado para essa regra é um exemplo de não conformidade apresentado na sua especificação em (CARNEIGE MELLON UNIVERSITY, 2016) e exposto na Figura 3.1. A não conformidade se encontra na linha 14, onde o `sign` é desreferenciado e pode não ter sido inicializado, já que a função `set_flag()` não atribui nenhum valor a ele se o valor do primeiro parâmetro for 0.

```
1 void set_flag(int number, int *sign_flag) {
2     if (NULL == sign_flag) {
3         return;
4     }
5     if (number > 0) {
6         *sign_flag = 1;
7     } else if (number < 0) {
8         *sign_flag = -1;
9     }
10 }
11 int is_negative(int number) {
12     int sign;
13     set_flag(number, &sign);
14     return sign < 0;
15 }
```

Figura 3.1: Código não conforme para a regra EXP33-C

3.2 EXP34-C

3.2.1 Descrição

A regra EXP34-C diz respeito a não desreferenciar ponteiros nulos, isto é, não acessar seus dados. Como ponteiros nulos apontam para uma área indeterminada da memória, desreferenciá-lo retorna valores indeterminados.

Em muitas plataformas, tentar desreferenciar um ponteiro nulo resulta em aborto da execução, mas isso não é um requisito do padrão. Sendo assim, é necessário que haja cuidado por parte do programador para que isso não aconteça.

3.2.2 Código não conforme analisado

O código analisado para essa regra é um exemplo de não conformidade apresentado na sua especificação em (CARNEIGE MELLON UNIVERSITY, 2016) e exposto na Figura 3.2. A não conformidade acontece ao desreferenciar `chunkdata` na linha 5. Como o alocador `png_malloc()` da biblioteca `png` retorna `NULL` quando recebe o parâmetro de tamanho 0, `chunkdata` será nulo se `length` for igual a -1, e tentar escrever em seu endereço resulta em comportamento indeterminado.

```
1 #include <png.h> /* From libpng */
2 #include <string.h>
3 void func(png_structp png_ptr, int length, const void *
    user_data) {
4     png_charp chunkdata;
5     chunkdata = (png_charp)png_malloc(png_ptr, length + 1);
6     /* ... */
7     memcpy(chunkdata, user_data, length);
8     /* ... */
9 }
10
11 int main() {}
```

Figura 3.2: Código não conforme para a regra EXP34-C

3.3 ARR38-C

3.3.1 Descrição

A regra ARR38-C diz respeito a garantir que funções de bibliotecas não formem ponteiros inválidos. Isso é, ao chamar funções de outras bibliotecas que manipulam a memória, é necessário especificar o tamanho correto dos dados a serem manipulados.

O padrão C (ISO, 2011) define que o comportamento é indeterminado se o vetor passado para uma função de biblioteca não tem todos os endereços válidos e acessíveis. Passar parâmetros incorretos pode resultar em um ponteiro que aponta de forma parcial para o objeto, ou que ultrapassa os seus limites, causando esse tipo de comportamento indeterminado.

3.3.2 Código não conforme analisado

O código analisado para essa regra é um exemplo de não conformidade apresentado na sua especificação em (CARNEIGE MELLON UNIVERSITY, 2016) e exposto na Figura 3.3. A não conformidade se encontra na passagem de parâmetros para `wmemcpy()` na linha 10, onde o contador (terceiro parâmetro) é passado como `sizeof(w_str)`, que retorna o tamanho em *bytes*, o que equivale a tamanho de `char` porém difere de `wchar_t`. Assim, o tamanho da memória manipulada pela função será diferente do tamanho do objeto.

```
1 #include <string.h>
2 #include <wchar.h>
3 static const char str[] = "Hello world";
4 static const wchar_t w_str[] = L"Hello world";
5 void func(void)
6 {
7     char buffer[32];
8     wchar_t w_buffer[32];
9     memcpy(buffer, str, sizeof(str));           /* Compliant */
10    wmemcpy(w_buffer, w_str, sizeof(w_str)); /* Noncompliant */
11 }
12
13 int main() {}
```

Figura 3.3: Código não conforme para a regra ARR38-C

3.4 STR31-C

3.4.1 Descrição

A regra STR31-C diz respeito a garantir que o armazenamento para *strings* tem espaço suficiente para os dados em caractere e o terminador nulo (`'\0'`). Copiar mais dados do que o espaço permite resulta em um *buffer overflow*.

Para evitar este problema que causa uma vulnerabilidade, é possível truncar os valores ou, preferencialmente, assegurar-se de que o *buffer* de destino tem espaço suficiente para receber todos os caracteres e o terminador nulo.

3.4.2 Código não conforme analisado

O código analisado para essa regra é um exemplo de não conformidade apresentado na sua especificação em (CARNEIGE MELLON UNIVERSITY, 2016) e exposto na Figura 3.4. A não conformidade se encontra na linha 9, uma vez que o laço de repetição não considera o terminador nulo e itera sobre todo o vetor alocado. Assim, a atribuição do terminador nulo será feita um *byte* após o fim de `dest`, causando um *buffer overflow*.

```
1 #include <stddef.h>
2 void copy(size_t n, char src[n], char dest[n])
3 {
4     size_t i;
5     for (i = 0; src[i] && (i < n); ++i)
6     {
7         dest[i] = src[i];
8     }
9     dest[i] = '\0';
10 }
11 int main() {}
```

Figura 3.4: Código não conforme para a regra STR31-C

3.5 STR32-C

3.5.1 Descrição

A regra STR32-C diz respeito a não passar uma sequência de caracteres que não terminada por nulo (`'\0'`) para uma função de biblioteca que espera uma *string*. Muitas bibliotecas oferecem funções que aceitam *strings* com a restrição de que elas sejam apropriadamente terminadas com nulo. Passar uma sequência de caracteres que não atende tal restrição para essas funções pode resultar em acesso de memória fora dos limites do objeto, expondo uma vulnerabilidade. O mesmo vale para funções que esperam *wide strings*.

3.5.2 Código não conforme analisado

O código analisado para essa regra é um exemplo de não conformidade apresentado na sua especificação em (CARNEIGE MELLON UNIVERSITY, 2016) e exposto na Figura 3.5. A não conformidade se encontra na passagem de `c_str` para `printf` na linha 5, uma vez que `c_str` não contém o terminador nulo.

```
1 #include <stdio.h>
2 void func(void)
3 {
4     char c_str[3] = "abc";
5     printf("%s\n", c_str);
6 }
7 int main() {}
```

Figura 3.5: Código não conforme para a regra STR32-C

3.6 STR38-C

3.6.1 Descrição

A regra STR38-C diz respeito a não confundir *strings* comuns com *wide strings* ao passar argumentos para funções que esperam parâmetros desses tipos. Passar uma *string* para uma função de biblioteca que espera uma *wide string*, assim como o inverso, pode resultar em comportamentos indefinidos e inesperados. Como os dois tipos tem tamanhos diferentes, podem haver problemas de escala. Além disso, *wide strings* tem um terminador nulo diferente e pode conter *bytes* nulos, o que pode ocasionar inconsistências de tamanho se analisado como uma *string* comum.

3.6.2 Código não conforme analisado

O código analisado para essa regra é um exemplo de não conformidade apresentado na sua especificação em (CARNEIGE MELLON UNIVERSITY, 2016) e exposto na Figura 3.6. A não conformidade se encontra na passagem de parâmetros do tipo *wide string* para a função `strncpy()` que espera parâmetros do tipo *string*, na linha 7.

```
1 #include <stddef.h>
2 #include <string.h>
3 void func(void)
4 {
5     wchar_t wide_str1[] = L"0123456789";
6     wchar_t wide_str2[] = L"0000000000";
7     strncpy(wide_str2, wide_str1, 10);
8 }
9 int main() {}
```

Figura 3.6: Código não conforme para a regra STR38-C

3.7 MEM30-C

3.7.1 Descrição

A regra MEM30-C diz respeito a não acessar memória liberada. Avaliar um ponteiro - o que inclui desreferenciá-lo, usá-lo em uma operação aritmética, fazer *cast* do seu tipo

e usá-lo no lado esquerdo de uma atribuição - que já foi liberado com uma função de gerenciamento de memória como `free()` causa comportamento indefinido. O acesso a esses ponteiros pode resultar em vulnerabilidades.

Após ser liberado, um ponteiro se torna inválido e o espaço de endereço para o qual ele aponta pode ser usado para outros fins. Assim, o resultado de uma leitura pode até parecer válido em um momento, mas mudar inesperadamente no instante seguinte.

3.7.2 Código não conforme analisado

O código analisado para essa regra é um exemplo de não conformidade apresentado na sua especificação em (CARNEIGE MELLON UNIVERSITY, 2016) e exposto na Figura 3.7. A não conformidade se encontra em no acesso a `p->next` na linha 9, uma vez que o passo do laço `for` é executado após o seu bloco, que está liberando a memória de `p`. Assim, no momento do acesso, a memória já foi liberada, e seu acesso pode resultar em comportamento indeterminado.

```
1  #include <stdlib.h>
2  struct node
3  {
4      int value;
5      struct node *next;
6  };
7  void free_list(struct node *head)
8  {
9      for (struct node *p = head; p != NULL; p = p->next)
10     {
11         free(p);
12     }
13 }
14 int main() {}
```

Figura 3.7: Código não conforme para a regra MEM30-C

3.8 MEM34-C

3.8.1 Descrição

A regra MEM34-C diz respeito a liberar apenas memória que foi alocada dinamicamente. Liberar outro tipo de memória pode resultar em corrompimento da *heap* e outros erros graves. O comportamento resultante desse tipo de operação é indeterminado.

Funções de liberação de memória como `free()` só devem ser usadas sobre ponteiros retornados por outras funções de gerenciamento de memória como `malloc()`, `calloc()`, `aligned_alloc()` e `realloc()`.

Essa regra não se aplica para ponteiros nulos, uma vez que é garantido pelo padrão C que liberar um ponteiro nulo não causa nenhuma ação.

3.8.2 Código não conforme analisado

O código analisado para essa regra é um exemplo de não conformidade apresentado na sua especificação em (CARNEIGE MELLON UNIVERSITY, 2016) e exposto na Figura 3.8. A não conformidade se encontra na chamada de `free()` para `c_str` na linha 31, uma vez que, se `argc` for diferente de 2, `c_str` não é resultado da alocação de memória por algum gerenciador, e então não pode ser liberada com `free()`.

3.9 FIO30-C

3.9.1 Descrição

A regra FIO30-C diz respeito a excluir entradas de usuário de *strings* de formatação. Em uma avaliação da função `fprintf()`, a *string* de formatação é avaliada e, se ela conter alguma entrada do usuário, permite que um atacante execute código arbitrário ao passar como entrada uma *string* de formatação. Essa execução terá as mesmas permissões do processo com a vulnerabilidade relacionada a essa regra.

Assim, toda a entrada de usuário deve ser tratada ou utilizada como *string* comum, e nunca diretamente como uma *string* de formatação.

```
1 #include <stdlib.h>
2 #include <string.h>
3 #include <stdio.h>
4 enum
5 {
6     MAX_ALLOCATION = 1000
7 };
8 int main(int argc, const char *argv[])
9 {
10     char *c_str = NULL;
11     size_t len;
12     if (argc == 2)
13     {
14         len = strlen(argv[1]) + 1;
15         if (len > MAX_ALLOCATION)
16         {
17             /* Handle error */
18         }
19         c_str = (char *)malloc(len);
20         if (c_str == NULL)
21         {
22             /* Handle error */
23         }
24         strcpy(c_str, argv[1]);
25     }
26     else
27     {
28         c_str = "usage: $>a.exe [string]";
29         printf("%s\n", c_str);
30     }
31     free(c_str);
32     return 0;
33 }
34 int main() {}
```

Figura 3.8: Código não conforme para a regra MEM34-C

3.9.2 Código não conforme analisado

O código analisado para essa regra é um exemplo de não conformidade apresentado na sua especificação em (CARNEIGE MELLON UNIVERSITY, 2016) e exposto na Figura 3.9. A não conformidade se encontra na passagem de `msg` para `fprintf()` na linha 24, já que `msg` é construída com entrada do usuário e está sendo usada como uma *string* de formatação.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 void incorrect_password(const char *user)
5 {
6     int ret;
7     /* User names are restricted to 256 or fewer characters */
8     static const char msg_format[] = "%s cannot be
9         authenticated.\n";
10    size_t len = strlen(user) + sizeof(msg_format);
11    char *msg = (char *)malloc(len);
12    if (msg == NULL)
13    {
14        /* Handle error */
15    }
16    ret = snprintf(msg, len, msg_format, user);
17    if (ret < 0)
18    {
19        /* Handle error */
20    }
21    else if (ret >= len)
22    {
23        /* Handle truncated output */
24    }
25    fprintf(stderr, msg);
26    free(msg);
27 }
28 int main() {}
```

Figura 3.9: Código não conforme para a regra FIO30-C

3.10 FIO34-C

3.10.1 Descrição

A regra FIO34-C diz respeito à distinção entre caracteres lidos de um arquivo e EOF ou WEOF. Em plataformas onde o tamanho do tipo `int` é igual ao do tipo `char`, ao fazer *cast* de tipo de um caractere lido para compará-lo com EOF ou WEOF, pode haver conflito de forma que um caractere válido possa ser igual a representação EOF ou WEOF. Assim, é necessário verificar o término de um arquivo de outra forma como com as funções `feof()` e `ferror()`.

3.10.2 Código não conforme analisado

O código analisado para essa regra é um exemplo de não conformidade apresentado na sua especificação em (CARNEIGE MELLON UNIVERSITY, 2016) e exposto na Figura 3.10. A não conformidade se encontra na comparação do caractere `c` com `EOF` na linha 8, sem outras verificações que assegurem que não é um conflito resultante do *cast* de tipo.

```
1 #include <stdio.h>
2 void func(void)
3 {
4     int c;
5     do
6     {
7         c = getchar();
8     } while (c != EOF);
9 }
10 int main() {}
```

Figura 3.10: Código não conforme para a regra FIO34-C

3.11 FIO37-C

3.11.1 Descrição

A regra FIO37-C diz respeito a não assumir que as funções `fgets()` e `fgetws()` retornam uma string não vazia quando tem sucesso. Dado que existem teclados capazes de produzir o caractere nulo, assim como a possibilidade de redirecionar a leitura para um arquivo binário (com o operador *pipe* do terminal) contendo o mesmo caractere, não é seguro assumir que uma leitura bem sucedida retorna uma string não vazia. Operações que assumem tal propriedade podem fazer execuções inapropriadas para uma string vazia, resultando em comportamentos inesperados.

3.11.2 Código não conforme analisado

O código analisado para essa regra é um exemplo de não conformidade apresentado na sua especificação em (CARNEIGE MELLON UNIVERSITY, 2016) e exposto na Figura 3.11. A não conformidade está na não verificação do conteúdo ou tamanho de `buf` antes da atribuição feita na linha 14. Caso `buf` inicie com o terminador nulo, o seu tamanho

calculado por `strlen()` será 0, e a atribuição em um valor positivo alto resultante de 0 - 1 estará fora dos limites de `buf`.

```
1 #include <stdio.h>
2 #include <string.h>
3 enum
4 {
5     BUFFER_SIZE = 1024
6 };
7 void func(void)
8 {
9     char buf[BUFFER_SIZE];
10    if (fgets(buf, sizeof(buf), stdin) == NULL)
11    {
12        /* Handle error */
13    }
14    buf[strlen(buf) - 1] = '\\0';
15 }
16 int main() {}
```

Figura 3.11: Código não conforme para a regra FIO37-C

3.12 ENV32-C

3.12.1 Descrição

A regra ENV32-C diz respeito a necessidade de todos os manipuladores de saída (*exit handlers*) retornarem normalmente. Uma chamada por `exit()` dentro de uma função registrada como `atexit()` causa comportamento indefinido. Assim, é necessário que todas as funções desse tipo terminem com `return`.

3.12.2 Código não conforme analisado

O código analisado para essa regra é um exemplo de não conformidade apresentado na sua especificação em (CARNEIGE MELLON UNIVERSITY, 2016) e exposto na Figura 3.12. A não conformidade se encontra na chamada de `exit()` na linha 13, fazendo com que a execução do código tenha comportamento indefinido caso a condição `some_condition` seja verdadeira.

```
1 #include <stdlib.h>
2 void exit1(void)
3 {
4     /* ... Cleanup code ... */
5     return;
6 }
7 void exit2(void)
8 {
9     extern int some_condition;
10    if (some_condition)
11    {
12        /* ... More cleanup code ... */
13        exit(0);
14    }
15    return;
16 }
17 int main(void)
18 {
19     if (atexit(exit1) != 0)
20     {
21         /* Handle error */
22     }
23     if (atexit(exit2) != 0)
24     {
25         /* Handle error */
26     }
27     /* ... Program code ... */
28     return 0;
29 }
```

Figura 3.12: Código não conforme para a regra ENV32-C

3.13 ENV33-C

3.13.1 Descrição

A regra ENV33-C diz respeito a não chamar a função `system()` ou equivalentes. Essa função permite a execução do comando especificado em algum processador de comandos como o shell em sistemas UNIX ou `cmd.exe` em sistemas Microsoft Windows. Chamadas como essa pode resultar em vulnerabilidades, gerando a possibilidade de execução de comandos de sistema arbitrários.

3.13.2 Código não conforme analisado

O código analisado para essa regra é um exemplo de não conformidade apresentado na sua especificação em (CARNEIGE MELLON UNIVERSITY, 2016) e exposto na Figura 3.13. A não conformidade se encontra na chamada de `system()` na linha 21. A entrada `input` pode conter comandos maliciosos que serão executados pelo sistema pela vulnerabilidade exposta.

```
1  #include <string.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4  enum
5  {
6      BUFFERSIZE = 512
7  };
8  void func(const char *input)
9  {
10     char cmdbuf[BUFFERSIZE];
11     int len_wanted = snprintf(cmdbuf, BUFFERSIZE,
12                             "any_cmd '%s'", input);
13     if (len_wanted >= BUFFERSIZE)
14     {
15         /* Handle error */
16     }
17     else if (len_wanted < 0)
18     {
19         /* Handle error */
20     }
21     else if (system(cmdbuf) == -1)
22     {
23         /* Handle error */
24     }
25 }
26 int main() {}
```

Figura 3.13: Código não conforme para a regra ENV33-C

3.14 SIG30-C

3.14.1 Descrição

A regra SIG30-C diz respeito a chamar apenas funções assíncrono-seguras (*asynchronous-safe*) dentro de manipuladores de sinais (*signal handlers*). Estritamente, apenas as funções

`abort()`, `_Exit()`, `quick_exit()`, e `signal()` podem ser chamadas dentro de um manipulador de sinal. De forma geral, é necessário consultar uma lista de funções assíncrono-seguras para todas as implementações onde o programa será executado.

3.14.2 Código não conforme analisado

O código analisado para essa regra é um exemplo de não conformidade apresentado na sua especificação em (CARNEIGE MELLON UNIVERSITY, 2016) e exposto na Figura 3.14. A não conformidade se encontra nas chamadas `log_message()` - que chama `fputs()` - e `free()` dentro do manipulador `handler()`, ambas não assíncrono-seguras.

3.15 ERR33-C

3.15.1 Descrição

A regra ERR33-C diz respeito a detectar e lidar com erros da biblioteca padrão. A maioria das funções da biblioteca padrão retorna um valor específico do tipo esperado em caso de erro, como -1 ou um ponteiro nulo. Não verificar o valor desses retornos, assumindo o sucesso da execução da função, pode levar a comportamentos inesperados. É necessário verificar o retorno pelo valor correspondente a erro de cada função utilizada.

3.15.2 Código não conforme analisado

O código analisado para essa regra é um exemplo de não conformidade apresentado na sua especificação em (CARNEIGE MELLON UNIVERSITY, 2016) e exposto na Figura 3.15. A não conformidade se encontra na linha 10, onde o retorno de `setlocale()` não é verificado. Caso ocorra algum erro nessa execução, a chamada da linha 11 pode ter comportamento inesperado.

```
1 #include <signal.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 enum
5 {
6     MAXLINE = 1024
7 };
8 char *info = NULL;
9 void log_message(void)
10 {
11     fputs(info, stderr);
12 }
13 void handler(int signum)
14 {
15     log_message();
16     free(info);
17     info = NULL;
18 }
19 int main(void)
20 {
21     if (signal(SIGINT, handler) == SIG_ERR)
22     {
23         /* Handle error */
24     }
25     info = (char *)malloc(MAXLINE);
26     if (info == NULL)
27     {
28         /* Handle Error */
29     }
30     while (1)
31     {
32         /* Main loop program code */
33         log_message();
34         /* More program code */
35     }
36     return 0;
37 }
38 int main() {}
```

Figura 3.14: Código não conforme para a regra SIG30-C

3.16 MSC32-C

3.16.1 Descrição

A regra MSC32-C diz respeito a passar sementes apropriadas para geradores de números pseudo-aleatórios. Para geradores que podem receber sementes, é necessário que sejam

```
1 #include <locale.h>
2 #include <stdlib.h>
3 int utf8_to_wcs(wchar_t *wcs, size_t n, const char *utf8,
4               size_t *size)
5 {
6     if (NULL == size)
7     {
8         return -1;
9     }
10    setlocale(LC_CTYPE, "en_US.UTF-8");
11    *size = mbstowcs(wcs, utf8, n);
12    return 0;
13 }
14 int main() {}
```

Figura 3.15: Código não conforme para a regra ERR33-C

passadas sementes ao inicializá-los, e que essas sementes sejam diferentes em cada execução. Executar o programa passando a mesma semente para o gerador mais de uma vez implica na geração da mesma sequência de números aleatórios, de forma que um atacante possa prevêê-los.

3.16.2 Código não conforme analisado

O código analisado para essa regra é um exemplo de não conformidade apresentado na sua especificação em (CARNEIGE MELLON UNIVERSITY, 2016) e exposto na Figura 3.16. A não conformidade se encontra na linha 8, onde não é passada uma semente para o inicializador do gerador `random()`, que gerará a mesma sequência de números em todas as execuções.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 void func(void)
4 {
5     for (unsigned int i = 0; i < 10; ++i)
6     {
7         /* Always generates the same sequence */
8         printf("%ld, ", random());
9     }
10 }
11 int main() {}
```

Figura 3.16: Código não conforme para a regra MSC32-C

3.17 MSC33-C

3.17.1 Descrição

A regra MSC33-C diz respeito a não passar dados inválidos para a função `asctime()`. O uso dessa função é desencorajado no geral e ela é considerada obsoleta, sendo substituída pela função `asctime_s()`. Quando for utilizada, deve receber dados válidos conforme esperado, uma vez que a passagem de algum parâmetro com tamanho maior do que o esperado causará *buffer overflow* na string resultante, já que o tamanho alocado para ela é fixo e não há verificações sobre o formato dos parâmetros.

3.17.2 Código não conforme analisado

O código analisado para essa regra é um exemplo de não conformidade apresentado na sua especificação em (CARNEIGE MELLON UNIVERSITY, 2016) e exposto na Figura 3.17. A não conformidade se encontra na chamada de `asctime()` na linha 4 sem a sanitização dos dados em `timez_tm`.

```
1 #include <time.h>
2 void func(struct tm *time_tm)
3 {
4     char *time = asctime(time_tm);
5     /* ... */
6 }
7 int main() {}
```

Figura 3.17: Código não conforme para a regra MSC33-C

4 Comparativo

Para cada ferramenta a ser comparada, executou-se um processo de testes para detecção de falhas de segurança nos códigos não conformes listados no Capítulo 3. A seguir, serão expostos os resultados obtidos, assim como comparações relevantes.

4.1 Método de aplicação da ferramenta

4.1.1 Cppcheck

A execução dos testes com a ferramenta Cppcheck demanda seu download e instalação, feito a partir da fonte oficial (MARJAMÄKI, 2010). Os códigos não conformes foram organizados em um diretório local, e a ferramenta permitiu a verificação a partir do caminho raiz do diretório criado. Assim, uma única execução foi suficiente para obter os resultados com erros apontados para todos os arquivos de código fonte. Cada alerta informa o arquivo e a linha do código potencialmente inseguro.

4.1.2 FlawFinder

O funcionamento da ferramenta FlawFinder proporciona um procedimento semelhante ao feito para o analisador Cppcheck. O formato da saída é equivalente, e ela, da mesma forma, permite a verificação de um diretório. Assim, o mesmo diretório de códigos fonte não conformes foi utilizado. Os preparativos envolvendo download e instalação foram feitas de forma semelhante, a partir da fonte oficial (WHEELER, 2010).

4.1.3 LGTM

A ferramenta LGTM apresentou demandas disparadamente diferentes. Este analisador funciona como um serviço, se conectando como uma aplicação de outros serviços como o GitHub. Assim, se tornou necessário que os códigos analisados estivessem disponíveis em tal serviço.

O serviço LGTM não permite a verificação de um diretório, e sim de um projeto. Isso se torna relevante por duas diferenças de estrutura: A necessidade de um arquivo para construção (*Makefile*) e de remoção de conflitos.

Os exemplos de não conformidade providos pela CERT-C são colocados de maneira isolada. Assim, muitos deles possuem nomes idênticos para funções, e todos implementam uma função `main`. Esses nomes foram alterados para nomes arbitrários não conflitantes, afim de permitir a compilação do projeto criado. A especificação no arquivo de construção determina que todos os arquivos de código fonte devem ser compilados.

Com o projeto sem conflitos e com arquivo de construção disponível em um repositório do GitHub configurado com a aplicação LGTM, é possível abrir um *Pull Request* e receber uma verificação automática, com os arquivos e linhas correspondentes ao alerta listados na página *web* do serviço.

4.2 Resultados

Com as informações resultantes da execução da ferramenta, executou-se um processo de análise para identificar se a falha apontada é relacionada com a determinação da regra imposta pela CERT-C. A Tabela 4.1 indica quais ferramentas apontaram falhas referentes a quais regras de prioridade 1 - representado com S nas células onde a ferramenta da coluna foi capaz de detectar não conformidade para a regra da linha, e N caso a ferramenta não tenha apontado a falha em questão.

Ainda, os resultados foram analisados em busca de falsos positivos. Nesse contexto, um falso positivo é tido como uma falha apontada que não tem uma relação direta com o problema. Em algumas ocasiões, as ferramentas apontam falhas que são consequências indiretas da não conformidade - o que foi considerado um falso positivo com a justificativa de que um programador fazendo correções referentes aos alertas tenderia a resolver o problema errado. Caso o alerta aponte uma situação que, se corrigida, torna o código conforme, ele é considerado uma detecção correta.

Com essa análise, é possível construir matrizes de confusão para as ferramentas. Uma matriz de confusão simples possui quatro quadrantes:

- Verdadeiro positivo: Falhas detectadas correspondem à não conformidade.

Regra	Detecção (S/N)		
	Cppcheck	FlawFinder	LGTM
EXP33-C	N	N	N
EXP34-C	N	N	N
ARR38-C	N	S	N
STR31-C	S	S	N
STR32-C	N	S	N
STR38-C	N	S	N
MEM30-C	N	N	N
MEM34-C	N	N	N
FIO30-C	N	S	N
FIO34-C	N	N	N
FIO37-C	N	S	N
ENV32-C	N	N	N
ENV33-C	N	S	N
SIG30-C	N	N	N
ERR33-C	N	N	N
MSC32-C	N	N	N
MSC33-C	S	N	S

Tabela 4.1: Detecção de conformidade com cada regra por cada ferramenta

- Falso negativo: A não conformidade não foi apontada como falha.
- Falso positivo: Falhas detectadas não correspondem à não conformidade.
- Verdadeiro negativo: Não foram detectadas falhas não correspondentes.

Destaca-se a diferenças entre as linhas da matriz de confusão para esse contexto: todos os códigos analisados possuem não conformidade com alguma regra, e falsos positivos e verdadeiros negativos correspondem a detecção ou não de falhas não correspondentes com a não conformidade.

	Detecção de falha	Não detecção de falha
Falha correspondente	2	15
Restante do código	0	17

Tabela 4.2: Matriz de confusão para detecção com Cppcheck

	Detecção de falha	Não detecção de falha
Falha correspondente	7	10
Restante do código	4	13

Tabela 4.3: Matriz de confusão para detecção com FlawFinder

	Detecção de falha	Não detecção de falha
Falha correspondente	1	14
Restante do código	0	17

Tabela 4.4: Matriz de confusão para detecção com LGTM

As matrizes de confusão são apresentadas por ferramenta, sendo a matriz para os resultados de Cppcheck dispostos na Tabela 4.2, de FlawFinder na Tabela 4.3 e de LGTM na Tabela 4.4.

A partir da observação desses resultados com base no critério proposto pelo padrão (CARNEIGE MELLON UNIVERSITY, 2016) de quantidade de não conformidades detectadas, a ferramenta FlawFinder se mostra mais adepta para análise de código C/C++ com objetivo de mitigar vulnerabilidades abordadas nas regras de nível 1 do padrão CERT C.

5 Considerações Finais

Na execução deste trabalho, foi possível organizar um diretório com exemplos de códigos não conformes disponibilizados pela CERT C 2016 (CARNEIGE MELLON UNIVERSITY, 2016), bem como um projeto onde todos os arquivos fontes são compilados através de um aquivo de construção. Nessa etapa, foi selecionado arbitrariamente um código de exemplo para cada regra, devido a limitações de recursos. Considera-se que seria interessante para este comparativo a inclusão de mais exemplos por regra. No entanto, se houverem números diferentes de exemplos por regra, o método de análise deve ser revisito, já que o utilizado não seria adequado, favorecendo ferramentas que detectam não conformidades de regras com mais exemplos.

Nos experimentos realizados, todas as ferramentas foram executadas com sucesso. A ferramenta LGTM apresentou maiores dificuldades de configuração por ser no formato de serviço e envolver mais fatores. O formato dos resultados de todas as ferramentas permitiu uma análise equivalente, já que se apresentavam de maneira semelhante. Algumas ferramentas apresentaram alertas para problemas de estilo de código, que foram ignoradas quando não relevantes, uma vez que não fazem parte do escopo deste trabalho.

A quantificação dos resultados foi obtida a partir da análise de códigos não conformes apenas, sem compará-la com uma análise sobre a versão conforme. Isso se dá pelo direcionamento deste trabalho, em alinhamento com as recomendações do padrão CERT C, em buscar pelo maior número de verdadeiros positivos. As matrizes de confusão apresentadas estão de acordo com esse escopo. Considera-se que uma outra comparação possível poderia envolver a minimização de alertas para códigos conformes, o que necessitaria a análise das versões conformes do código.

Concluindo o trabalho, os resultados obtidos se mostraram relevantes ao indicar uma ferramenta com capacidade de detecção de não conformidades com as regras de nível 1 do padrão CERT C 2016 mais satisfatória, com uma diferença notável quando comparada com as demais.

Referências

- CARNEIGE MELLON UNIVERSITY. *C Coding Standard*. 2016. Disponível em: <<https://resources.sei.cmu.edu/downloads/secure-coding/assets/sei-cert-c-coding-standard-2016-v01.pdf>>. Acesso em: 28 mar. 2019.
- ISO. *ISO/IEC 9899:2011 Information technology — Programming languages — C*. [s.n.], 2011. 683 (est.) p. Disponível em: <http://www.iso.org/iso/_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853>.
- MARJAMÄKI, D. *Cppcheck Design*. 2010. Disponível em: <<https://sourceforge.net/projects/cppcheck/files/Articles/cppcheck-design-2010.pdf/download>>. Acesso em: 28 mar. 2019.
- MOOR, O. de. *Open results on LGTM: The key to securing open source*. 2019. Disponível em: <https://lgtm.com/blog/open_results_on_lgtm>. Acesso em: 05 jun. 2019.
- OWASP. *Source Code Analysis Tools*. 2019. Disponível em: <https://www.owasp.org/index.php/Source_Code_Analysis_Tools>. Acesso em: 05 jun. 2019.
- REDHAT DEVELOPER. *Secure Coding*. 2019. Disponível em: <<https://developers.redhat.com/topics/secure-coding/>>. Acesso em: 05 jun. 2019.
- SEACORD, R. *Secure Coding in C and C++*. Pearson Education, 2005. ISBN 9780768685138. Disponível em: <<https://books.google.com.br/books?id=jfn1IAN3dvwC>>.
- SNAVELY, W. *SEI CERT C Coding Standard History*. 2016. Disponível em: <<https://wiki.sei.cmu.edu/confluence/display/c/History>>. Acesso em: 05 jun. 2019.
- WHEELER, D. A. *Flawfinder*. 2010. Disponível em: <<https://dwheeler.com/flawfinder/>>. Acesso em: 28 mar. 2019.