Universidade Federal de Ouro Preto
Instituto de Ciências Exatas e Aplicadas
Departamento de Computação e Sistemas

Análise de implementação de código contra o Buffer Overflow e o impacto em dispositivos de IoT

Vitor de Siqueira Cotta

TRABALHO DE CONCLUSÃO DE CURSO

ORIENTAÇÃO:
Marlon Paolo Lima

COORIENTAÇÃO:
Erik de Britto e Silva

Dezembro, 2018
João Monlevade–MG

Vitor de Siqueira Cotta

Análise de implementação de código contra o Buffer Overflow e o impacto em dispositivos de IoT

Orientador: Marlon Paolo Lima

Coorientador: Erik de Britto e Silva

Monografia apresentada ao curso de Sistemas de Informação do Instituto de Ciências Exatas e Aplicadas, da Universidade Federal de Ouro Preto, como requisito parcial para aprovação na Disciplina "Trabalho de Conclusão de Curso II".

Universidade Federal de Ouro Preto

João Monlevade

Dezembro de 2018

C846a Cotta, Vitor de Siqueira.
Análise de implementação de código contra o Buffer Overflow e o impacto em dispositivos de IoT [manuscrito] / Vitor de Siqueira Cotta. - 2018.

54f.: il.: color; tabs.

Orientador: Prof. Dr. Marlon Paolo Lima.
Coorientador: Prof. MSc. Erik de Britto e Silva.

Monografia (Graduação). Universidade Federal de Ouro Preto. Instituto de Ciências Exatas e Aplicadas. Departamento de Computação e Sistemas de Informação.

1. C (Linguagem de programação de computador). 2. Computadores digitais - Confiabilidade. 3. Internet . I. Lima, Marlon Paolo. II. Silva, Erik de Britto e. III. Universidade Federal de Ouro Preto. IV. Titulo.

CDU: 004.43

Catalogação: ficha.sisbin@ufop.edu.br

FOLHA DE APROVAÇÃO DA BANCA EXAMINADORA

Análise de implementação de código contra o Buffer Overflow e o impacto em dispositivos de IoT

Vitor de Siqueira Cotta

Monografia apresentada ao Instituto de Ciências Exatas e Aplicadas da Universidade Federal de Ouro Preto como requisito parcial da disciplina CSI499 – Trabalho de Conclusão de Curso II do curso de Bacharelado em Sistemas de Informação e aprovada pela Banca Examinadora abaixo assinada:

Marlon Paolo Lima
Doutor em Engenharia Elétrica
DECSI – UFOP

Erik de Britto e Silva
Mestre em Ciência da Computação
DCC – UFMG

Theo Silva Lins
Mestre em Ciência da Computação
Examinador
DECSI – UFOP

Carla Rodrigues Figueiredo Lara
Mestre em Ciência da Computação
Examinador
DECSI – UFOP

João Monlevade, 20 de dezembro de 2018

"Não importa quanto a vida possa ser ruim, sempre existe algo que você pode fazer, e triunfar. Enquanto há vida, há esperança."

— Stephen William Hawking (1942 – 2018)

Resumo

O *Buffer Overflow* (BOF) é uma antiga e conhecida ameaça e possui imenso potencial de causar danos e obter acesso privilegiado não autorizado em recursos computacionais. Apesar de existir na literatura um vasto material sobre o tema específico, esta vulnerabilidade se tornou mais evidente com os dispositivos chamados de *Internet of Things* (IoT). Estes dispositivos proporcionam novo patamar de interação do usuário com o meio, graças a suas funcionalidades, miniaturização e automação. Entretanto, as aplicações de IoT são comumente escritas em linguagem C, que por sua vez é vulnerável ao BOF. Esta limitação pode comprometer a integridade de uma aplicação, o que pode elevar seu custo de manutenção, além de ocasionar perda de desempenho. O objetivo deste trabalho é analisar as falhas de execução ocasionadas pelo *Buffer Overflow* na linguagem C. Adicionalmente, são apresentadas alternativas para mitigar vulnerabilidades de BOF e uma análise de desempenho destas soluções.

Palavras-chaves: *Buffer Overflow*. Linguagem C. Confiabilidade. Internet das Coisas.

Abstract

The Buffer Overflow (BOF) is a well-known old threat and has huge potential to cause damages and to obtain unauthorized privileged access in computational resources. Although there is a vast material on the subject in the literature, this vulnerability has become more evident with the Internet of Things (IoT) devices. Such devices provide new levels of user interaction with the medium through its functionality, miniaturization, and automation. However, IoT applications are commonly written in C language, which in turn is vulnerable to BOF. This limitation may compromise the integrity of an application, which can increase its maintenance cost and cause performance loss. The objective of this work is to analyze the execution failures caused by Buffer Overflow in C language. In addition, it is presented alternatives to mitigate BOF vulnerabilities and a performance analysis of these solutions.

Key-words: Buffer Overflow. Language C. Confiability. Internet of Things.

Lista de algoritmos

Lista de ilustrações

Lista de quadros

Lista de tabelas

Lista de abreviaturas e siglas

API Application Programming Interface

ARM Advanced RISC Machine

ASCII American Standard Code for Information Interchange

ASLR Address Space Layout Randomization

BLE Bluetooth Low Energy

BOF Buffer Overflow

CNES Centre National d'ÉtudesSpatiales

CPU Central Processing Unit

DBI Dynamic Binary Instrumentation

ESA European Space Agency

GPU Graphics Processing Unit

GCC GNU Compiler Collection

GPIO General Purpose Input/Output

GPL GNU General Public License

IOF Integer Overflow

IoT Internet of Things

MIT Massachusetts Institute of Technology

OSMC Open Source Media Center

SBI Static Binary Instrumentation

RAM Random Access Memory

SDRAM Synchronous Dynamic Random Access Memory

SoC System on a Chip

USB Universal Serial Bus

Sumário

1 Introdução

A *Internet* está consolidada e acessível para milhares de pessoas, entretanto, os dispositivos de IoT ampliaram o alcance da *Internet* em ambientes que no passado não era possível de se utilizar. Graças a estes dispositivos que são objetos comuns do dia a dia (telefone móvel, relógio, sensores, televisão, etc.) com acesso à *Internet* tornou possível esta realidade. A conectividade com a rede incrementa as funcionalidades dos objetos, como controle remoto, aquisição e envio de dados, monitoramento e automatização. A tecnologia IoT foi resultado da complexa contribuição das diferentes áreas do conhecimento, em especial à telecomunicação, eletrônica, ciência social e informática, sendo um instrumento valioso para comunidade acadêmica e simultaneamente tem imenso potencial nas indústrias. Estes dispositivos revolucionaram a interação do usuário com o meio e é esperado alcançar a marca de 20,4 bilhões de dispositivos inteligentes conectados em 2020 (SIMPSON; ROESNER; KOHNO, 2017), comprovando o seu sucesso.

As aplicações IoT comumente são escritas na linguagem C e esta linguagem de programação está próximo ao *hardware*, oferecendo desempenho compatível com as necessidades dos dispositivos de IoT. Entretanto, C possui várias funções vulneráveis ao BOF quando a entrada de dados excede a capacidade da memória, ocasionando falhas críticas e *exploits*. Com isso, a tecnologia IoT pode sofrer desatenção dos desenvolvedores em relação à integridade do *software*, pois, além do trabalho adicional para prevenir esta falha o desempenho do *hardware* pode ser comprometido e em muitos casos é uma prioridade nos projetos.

1.1 Justificativa

BOF é uma vulnerabilidade bem conhecida e estudada que pode provocar falhas catastróficas, mesmo seguindo as rigorosas recomendações para contê-la, dificilmente o *software* estará imune da ameaça (ZHIVICH, 2005).

Os dispositivos de IoT utilizam extensivamente a linguagem C vulnerável ao BOF por ter funções não confiáveis. A missão de desenvolver uma solução do problema nos dispositivos é mais complicado, pois, estes apresentam arquitetura modular e cada módulo deverá receber uma abordagem específica. Implementar uma solução confiável que atenda todos os módulos sem causar incompatibilidade é uma tarefa desafiadora. As alternativas confiáveis contra o BOF poderão consumir processamento (SHAW; DOGGETT; HAFIZ, 2014), sendo que em muitos projetos de IoT há privação de recursos e o desempenho se torna o elemento crítico; neste contexto a confiabilidade da aplicação será um componente secundário.

1.2 Objetivo

O presente trabalho visa esclarecer a vulnerabilidade BOF da linguagem C empregada nos dispositivos de IoT, demonstrando a importância e a magnitude dos danos ocasionados. O objetivo é avaliar os impactos de desempenho na implementação confiável de código e corrigir uma aplicação de dispositivo de IoT vulnerável ao BOF.

1.3 Estrutura do trabalho

O trabalho começa com o Referencial Teórico representado pelo Capítulo 2 abordando tópicos essenciais para a compreensão do trabalho dissertado, corroborando a motivação e compreensão do estudo através de uma breve explicação sobre os dispositivos de IoT e as ameaças do BOF. Adiante será apresentado o Capítulo 3 descrevendo as soluções escolhidas e em seguida a Metodologia no Capítulo 4. No Capítulo 5 teremos os resultados e finalmente no Capítulo 6 as Considerações Finais.

2 Referencial Teórico

Este capítulo é uma etapa que dará suporte à obtenção e análise dos dados, esclarecendo os principais aspectos que serão essenciais para a compreensão do trabalho. O conteúdo apresentado irá se desenvolver à partir de uma breve descrição dos dispositivos de IoT que utilizam extensivamente a linguagem C, passível de sofrer BOF. Adiante será apontado os riscos presentes do BOF e os tipos de corrupção de memória em C. No final será apontado algumas técnicas que podem ser empregados para atenuar os riscos do BOF.

2.1 IoT

Não há um consenso para determinar com exatidão o que são dispositivos de IoT, mas para Waher (2015) os dispositivos de IoT são objetos comuns do dia a dia capazes de interagir entre si por intermédio de uma rede, como a *Internet*, de forma ativa com o ambiente sem a intervenção humana direta. No presente momento, o maior consumidor de largura de banda (*bandwidth*) são os usuários, todavia é esperado alcançar a marca de 20,4 bilhões de dispositivos de IoT em 2020 (SIMPSON; ROESNER; KOHNO, 2017) conectados, que futuramente serão o maior consumidor de largura de banda.

É desejável que os dispositivos de IoT tenham alta interoperabilidade, baixo custo de produção e que possam ser empregados no controle da automação; Raspberry Pi tem estes atributos que será abordada na Seção 4.1. Os dispositivos de IoT tem vários aspectos em comum com os computadores tradicionais, pois compartilham as mesmas linguagens de programação em seu desenvolvimento, e, em especial a linguagem C; a Seção 2.2 irá descrever as características desta linguagem.

2.2 Linguagem C

A linguagem C foi criada em 1972 por Dennis Ritchie em *AT&T Bell Labs* com objetivo de reescrever o sistema operacional Unix, que era escrito em *Assembly*, uma linguagem difícil de manter e inflexível nas diferentes arquiteturas de computadores (RITCHIE, 1993). Na linguagem C, o programador ordena quais instruções, blocos de códigos e rotinas que o *software* deverá executar, normalmente associado com a estrutura de controle de fluxo como: *if, else, while, for, switch*, este tipo de paradigma é conhecido como programação imperativa (procedural) e estruturada. A linguagem C tem sistema de tipificação forte, pois as variáveis tem um tipo específico definido em tempo compilação (tipificação estática), ou seja, uma vez que a variável é declarada com o tipo, termina com mesmo tipo atribuído na inicialização.

A linguagem C é empregada para desenvolver os aplicações dos dispositivos de IoT, pois além de ser multiplataforma, produz um código eficiente que exige menos do *hardware* (TEIXEIRA et al., 2014), uma necessidade para sistemas embarcados. C não realiza verificações de limites de memória (muito presente em Java), que apesar de poupar os recursos computacionais, possibilita o surgimento do BOF; que será discutida na Seção 2.3.

2.3 *Buffer Overflow*

BOF é a vulnerabilidade de *software* mais perigosa na linguagem C (NAKAMURA; MARINO; MURASE, 2005). A primeira documentação a retratá-la foi: *"The Computer Security Technology Planning Study 1972"* (ANDERSON, 1972), um levantamento de requisitos de segurança computacional da força aérea americana.

De forma sucinta, BOF é uma anomalia da memória que armazena uma quantidade maior de informação do que previamente calculada, sendo resultado de códigos que não realizam a conferência correta de origem e destino dos dados, provocando sobrescrita nas regiões de memórias reservadas. A anomalia é uma abertura de falhas de execução ou invasão por injeção de códigos maliciosos (ANDERSON, 1972). Normalmente, o BOF é causado por descuido dos desenvolvedores, que poderiam corrigir a falha se realizassem testes elementares para atenuar os riscos (COWAN et al., 1998).

Um caso notório, Ariane 5 foi um foguete lançador de satélite, construído sob encomenda em 1996, foi inspecionada pela *European Space Agency* (ESA) e *Centre National d'ÉtudesSpatiales* (CNES). Durante 40 segundos do voo inaugural, o foguete explodiu e a auditoria apontou que a falha foi causada no módulo de sistema de referência inercial por um *Integer Overflow* (IOF), durante a conversão de *float* de 64 *bits* para inteiro de 16 *bits* (JAZEQUEL; MEYER, 1997). Esta vulnerabilidade custou 500 milhões de dólares e danos na reputação para ESA.

Em 2 de novembro de 1988, um quinto dos computadores conectados na *Internet* foram afetados pelo *Morris Worm*. Este *malware* foi um trabalho do Robert Tappan Morris, estudante graduando em Ciência da Computação em Cornell. Considerado como o primeiro bem-sucedido agente infeccioso a se alastrar pela *Internet*, aproveitando a falha gerada do BOF, sobrescreveu a *flag* autentificada pelo serviço Unix *fingerd* (protocolo de obtenção de dados em máquinas remotas) para poder replicar pela rede, causando prejuízos e indisponibilidades dos equipamentos (ORMAN, 2003); com o incidente, o BOF ganhou notoriedade (COWAN et al., 1998).

Atualmente há uma vasta literatura que aborda o tema, contendo soluções e métodos capazes de mitigar a ameaça, mas continua sendo excepcionalmente difícil de detectar por ser um problema indecidível para implementar as correções. Mesmo *softwares* que recebem grandes somas de recursos para detectar e corrigir a ameaça, não há garantias

de que estarão imunes ao BOF (ZHIVICH, 2005).

2.3.1 *Layout* da memória

Ao compilar as aplicações na linguagem C, normalmente são alocadas quatro regiões exclusivas na memória *Random Access Memory* (RAM), a representação varia conforme a arquitetura do processador (SCHILDT; COMPLETO, 1997). A Figura 1 é uma representação do *layout* de memória.

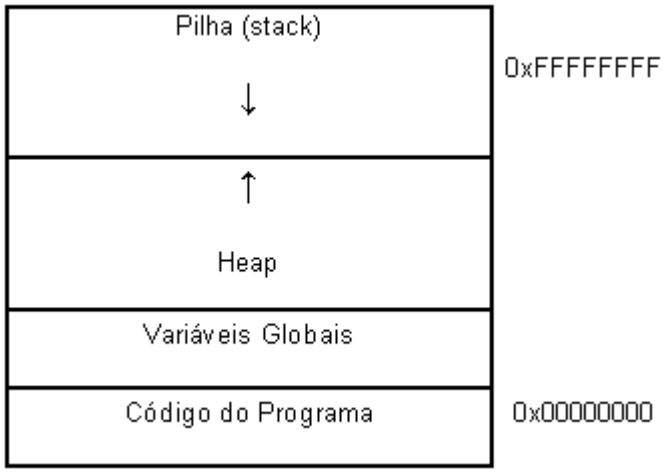


Figura 1 – *Layout* de memória

Fonte: Adaptado de Schildt e Completo (1997).

O menor endereço de memória descrita como primeira região, reserva o código do programa, adiante, a segunda região armazena as variáveis globais do programa. A terceira região conhecida como *heap* recebe a memória livre em que o programa usa para suas funções de alocação dinâmicas, como listas encadeadas e árvores binárias (SCHILDT; COMPLETO, 1997). A região de *heap* existe por não ser possível prever em tempo de compilação a quantidade de memória a ser utilizada, e após seu uso, a memória reservada é devolvida. Por último, o maior endereço de memória é a pilha (*stack*), responsável por guardar o endereço de retorno das funções, os argumentos das funções, variáveis locais e o estado atual da *Central Processing Unit* (CPU).

2.4 *Shellcode*

Resumidamente, *Shellcode* são *bytes* legíveis para o processador, sendo elemento comum para explorar BOF (GUPTA, 2012); para escrever *shellcode* é preciso ter conhecimento profundo da linguagem *Assembly* e da arquitetura (*hardware* e sistema operacional)

a ser utilizada, por esta razão é difícil reutilizar o *shellcode* em diferentes arquiteturas (DECKARD, 2005).

Shellcode é inserido no *buffer* de entrada do programa vulnerável para induzir uma execução maliciosa, mas antes é preciso ser compilado e montado antes da inserção do *buffer*, também requer uma codificação de caracteres, pois o programa vulnerável espera receber caracteres em codificação binária, comumente em *American Standard Code for Information Interchange* (ASCII). Outro fator crucial para o êxito do *shellcode* é superar o problema do endereçamento e o *byte* nulo (DECKARD, 2005).

Problema do endereçamento: as aplicações normalmente referenciam suas funções e variáveis com ponteiros, sendo definidas pelo compilador ou funções de alocação de memória como malloc(). Ao injetar o *shellcode* em uma *string*, por exemplo, o atacante precisa identificar, em tempo de execução, os endereços que estão sendo utilizados (DECKARD, 2005).

Byte nulo: *shellcode* pode ser injetado na memória do código vulnerável através das funções que manipulam *strings* de forma não-confiáveis, tais como: read(), sprintf() e strcpy(). Funções precisam terminar o processo com o *byte* nulo, quando o *shellcode* possui *byte* nulo, este *byte* será interpretado como fim de uma *string*, o conteúdo antes deste *byte* é recebido, mas o restante é descartado. Este é o maior obstáculo para executar o *shellcode*, pois o código deve estar livre de carácteres nulos (DECKARD, 2005).

Se o programa atacado for um *daemon* do servidor (programas executados em *background* do sistema sem estar sob controle de usuário) em modo superusuário, o programa *shell* terá sua execução como superusuário, através disso, o invasor terá acesso irrestrito à máquina atingida. Uma vez invadido, a única limitação é a criatividade do atacante (GUPTA, 2012).

2.5 Tipos de corrupção de memória em C

Após a breve explicação do *layout* de memória na Seção 2.3.1, é possível entender que as falhas de execução sobrescrevem os dados das regiões definidas de forma indiscriminada através do BOF, e podem ser feitos a partir das funções descritas no Quadro 1. A exploração de uma falha, normalmente envolve a manipulação de *strings* de maneira não-confiável, sobrescrevendo as regiões reservadas, como o endereço de retorno (*Change Return Address*) com o intuito de acionar o código malicioso ou injetar um código malicioso na própria memória (*Code Injection*); usualmente o código injetado é escrito em *shellcode*.

Quadro 1 – Funções com risco de *Buffer Overflow* em C

Risco Extremo	**Risco Alto**	**Risco Médio**	**Risco Baixo**
gets()	strcpy()	getchar()	fgets()
	strcat()	fgetc()	memcpy()
	sprintf()	getc()	snprintf()
	scanf()	read()	strccpy()
	sscanf()	bcopy()	strcadd()
	fscanf()		strncpy()
	vfscanf()		strncat()
	vsscanf()		vsnprintf()
	streadd()		
	strecpy()		
	strtrns()		
	realpath()		
	syslog()		
	getenv()		
	getopt()		
	getopt_long()		
	getpass()		

Fonte: Viega e McGraw (2001).

2.5.1 Stack Overflow

Stack Overflow é o mais simples e comum de BOF (COWAN et al., 1998); esta modalidade viabiliza o ataque através de um processo conhecido como *stacks mashing*. As memórias são alocadas na execução do programa em uma área crítica e os seus dados não deverão ser modificados pelos usuários, entretanto, é possível sobrescrever a mesma através da inserção comum de dados, quando não há presença de controle de limite. Neste cenário o atacante pode alterar os dados críticos, de modo a assumir a execução do programa. O Algoritmo 1 demonstra o quão fácil de reproduzir a vulnerabilidade, se o usuário inserir uma *string* com mais de 9 caracteres na variável **sobrenome**, haverá sobreposição de conteúdo na área alocado para a variável **nome**.

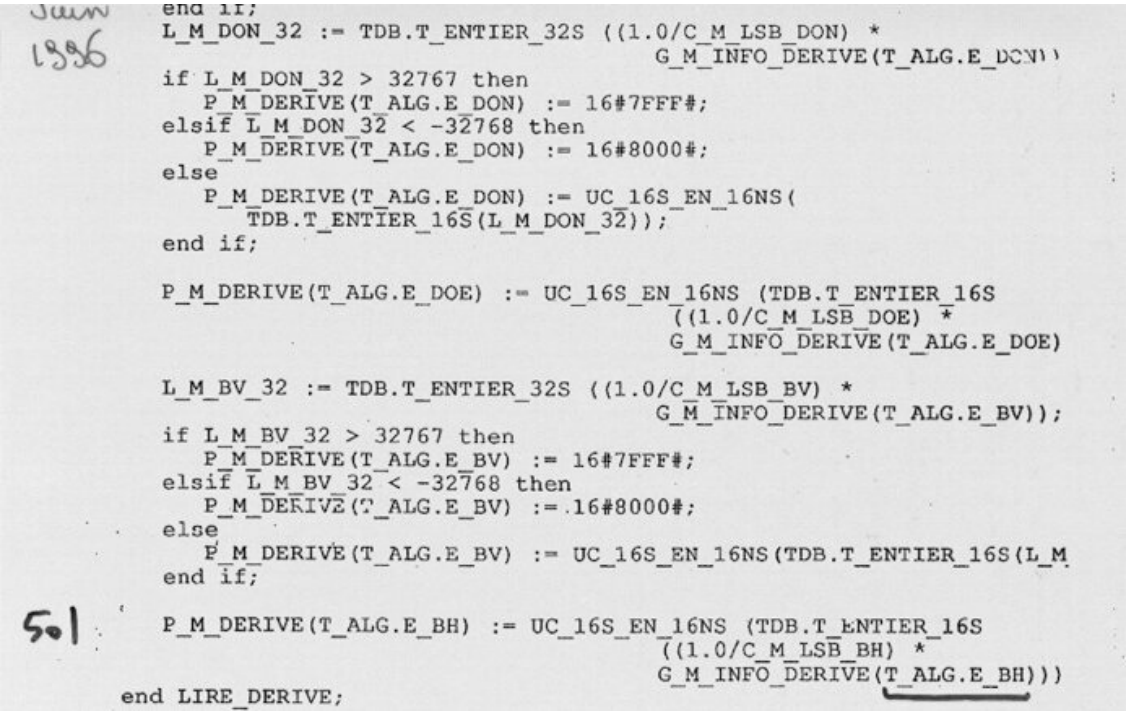
Algoritmo 1 – Código vulnerável ao *Stack Overflow*

```
#include <stdio.h>

typedef struct pessoa {
    char sobrenome[10];
    char nome[10];
} pessoa;

int main() {
    pessoa x;

    printf("Insira o nome: ");
    scanf("%s", x.nome);

    printf("Insira sobrenome: ");
    scanf("%s", x.sobrenome);

    printf("\nNome armazenado: %s", x.nome);
    printf("\nSobrenme armazenado: %s\n\n", x.sobrenome);

    return 0;
}
```

2.5.2 *Integer Overflow*

Integer Overflow (IOF) é uma condição em que as operações aritméticas (tais como: soma, subtração, multiplicação e divisão) ultrapassam a capacidade dos tipos das variáveis ao receber o resultado da operação, ocasionando estouro de memória.

Exemplificando na linguagem C, o tipo *int* compreende o intervalo válido de: -2147483648 até 2147483647, se a variável receber algum valor fora do intervalo, ocorrerá IOF. Outro cenário comum é realizar *casting* dos tipos quando há promoção explícita do dado (valor grande no tipo que não é capaz de comportar), por exemplo, o *casting* de *float* para *int*, o qual responsável pela queda do foguete Ariane 5 (JAZEQUEL; MEYER, 1997) (LéVY, 2010).



```
end if;
L_M_DON_32 := TDB.T_ENTIER_32S ((1.0/C_M_LSB_DON) *
                                G_M_INFO_DERIVE(T_ALG.E_DON));
if L_M_DON_32 > 32767 then
   P_M_DERIVE(T_ALG.E_DON) := 16#7FFF#;
elsif L_M_DON_32 < -32768 then
   P_M_DERIVE(T_ALG.E_DON) := 16#8000#;
else
   P_M_DERIVE(T_ALG.E_DON) := UC_16S_EN_16NS(
      TDB.T_ENTIER_16S(L_M_DON_32));
end if;

P_M_DERIVE(T_ALG.E_DOE) := UC_16S_EN_16NS (TDB.T_ENTIER_16S
                                ((1.0/C_M_LSB_DOE) *
                                G_M_INFO_DERIVE(T_ALG.E_DOE)

L_M_BV_32 := TDB.T_ENTIER_32S ((1.0/C_M_LSB_BV) *
                                G_M_INFO_DERIVE(T_ALG.E_BV));
if L_M_BV_32 > 32767 then
   P_M_DERIVE(T_ALG.E_BV) := 16#7FFF#;
elsif L_M_BV_32 < -32768 then
   P_M_DERIVE(T_ALG.E_BV) := 16#8000#;
else
   P_M_DERIVE(T_ALG.E_BV) := UC_16S_EN_16NS(TDB.T_ENTIER_16S(L_M
end if;

P_M_DERIVE(T_ALG.E_BH) := UC_16S_EN_16NS (TDB.T_ENTIER_16S
                                ((1.0/C_M_LSB_BH) *
                                G_M_INFO_DERIVE(T_ALG.E_BH)))
end LIRE_DERIVE;
```

Figura 2 – Trecho de código ADA responsável pela queda Ariane 5

Fonte: Lévy (2010).

Nota: Em destaque no canto inferior direito a variável responsável por medir a velocidade horizontal do foguete, entretanto sofreu *casting* e o valor da velocidade estourou a capacidade de armazenamento.

2.5.3 *Heap Overflow*

Heap Overflow ou *Heap Overun* normalmente é considerado como um *bug* que corrompe a área de memória ao sobrescrever as áreas adjacentes. Pode ocorrer na utilização do malloc(), e diferentemente do *Stack Overflow*, é preciso atender vários requisitos para que um *bug* de corrupção de *heap* seja explorável pelo atacante (JIA et al., 2017), pois os endereços alocados na região de *heap* são imprevisíveis.

Algoritmo 2 – Código vulnerável ao *Heap Overflow*

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

int main() {
    long diferenca = 0;
    int tamanho = 8;
    char *BUFF_1, *BUFF_2;

    BUFF_1 = (char *)malloc(tamanho);
    BUFF_2 = (char *)malloc(tamanho);

    diferenca = (long)BUFF_2 - (long)BUFF_1;

    printf("Buffer 1: %p\n", BUFF_1);
    printf("Buffer 2: %p\n\n", BUFF_2);
    printf("Diferenca: %ld\n\n", diferenca);

    memset(BUFF_2, 'A', tamanho - 1);
    printf("Buffer 2 antes Heap Overflow: %s\n", BUFF_2);

    memset(BUFF_1, 'B', diferenca + 4);
    printf("Buffer 2 depois Heap Overflow: %s\n", BUFF_2);

    free(BUFF_1);
    free(BUFF_2);

    return 0;
}
```

2.5.4 *Double Free*

Double Free é responsável por causar comportamento indefinido como falhas de execução ou alteração do fluxo de execução quando o free() é acionada duas vezes no mesmo endereço de memória ou por não apontar para o valor NULL após usar o free(). Através desta circunstância, o invasor pode sobrescrever os espaços para obter privilégio de permissão; o Algoritmo 3 ilustra exemplo de código vulnerável ao *Double Free*.

Algoritmo 3 – Código vulnerável ao *Double Free*

```
#include <stdlib.h>

int main() {
    int *x = malloc(10);

    free(x);
    free(x);    //Double Free

    return 0;
}
```

2.6 Técnicas de Mitigação contra o BOF

Existem diferentes técnicas que atenuam os riscos do BOF que podem ser utilizadas em conjunto para aprimorar a confiabilidade da aplicação.

2.6.1 Compilador

Os compiladores são empregados para mapear o código fonte para um objeto semanticamente correspondente. Este processo acontece em duas etapas: *front-end* do compilador e *back-end* do compilador.

Front-end reparte o código em componentes, impondo estruturas da linguagem de programação, criando código intermediário futuramente utilizado pela síntese. Neste processo é possível identificar as más formações e inconsistências na sintaxe do código fonte, em sequência poderão ser exibidas mensagens para esclarecer o erro ou até mesmo propor uma ação corretiva. *Back-end* é responsável por criar o código objeto a partir da representação intermediária gerada pela análise.

Os compiladores podem ter uma fase adicional de otimização, pois a tradução direta das linguagens de alto nível para linguagem de máquina tem custo elevado (SETHI; ULLMAN; LAM, 2008). É fundamental otimizar o código para reduzir as deficiências, eliminando instruções redundantes, substituindo uma instrução por outra mais eficiente e segura sem alterar o resultado. A execução da otimização pode ser feita no código local (aperfeiçoamento dentro de um bloco básico do código) ou otimização global (melhorando o funcionamento entre os blocos de códigos). Há também as abordagens dinâmica e estática, que podem ser executadas pela otimização dos compiladores ou ferramentas, para detectar as ameaças do BOF.

A análise dinâmica identifica as vulnerabilidades no programa testado, entretanto, a abordagem possui a penalidade de desempenho (LIU; CURTSINGER; BERGER, 2016). A análise estática realiza apuração do programa sem executar o mesmo, permitindo encontrar vulnerabilidades em circunstâncias muito específicas (KINDERMANN, 2008); a Seção 2.6.7 irá aprofundar o tema.

2.6.2 *Stack Shield*

Stack Shield tem objetivo de salvar uma cópia do endereço de retorno das funções em um local seguro (DU, 2017). No começo da função, o compilador realiza um procedimento de cópia de endereço de retorno em uma região conhecida como *Shadow Stack*, um local que não sofre BOF. E antes da função acionar o retorno, haverá a chamada de outro procedimento de comparação de endereço de retorno com o endereço salvo na etapa descrita anteriormente, e a partir disso, o *software* irá determinar a presença de BOF.

2.6.3 *Stack Guard*

Stack Guard cria um valor aleatório que é adicionado no bloco do código, conhecido como "canário", entre o endereço de retorno e o *buffer*, e salva o valor em um local seguro fora da pilha. Antes da função chamar o endereço de retorno, o "canário" é verificado para detectar a presença do BOF (DU, 2017). Esta técnica incrementa a segurança com modesto custo de desempenho (COWAN et al., 1998). A adição de *Stack Guard* no código fonte é descomplicada; o Algoritmo 4 apresenta exemplo de implementação.

Algoritmo 4 – Implementação de *Stack Guard*

```
int segredo;                    //Variavel global com valor aleatorio

void teste(char *entrada) {
    int guarda = segredo;       //Canario criado com o valor segredo
    char buffer[10];

    strcpy(buffer, entrada);

    if (guarda == segredo)      //Confere se houve sobrescrita
        return;                 //Sucesso
    else
        exit(1);                //Falha
}
```

Fonte: Adaptado Du (2017).

2.6.4 Bibliotecas Confiáveis

O programador deve estar ciente dos riscos de BOF na linguagem C, portanto, deve considerar o uso de bibliotecas confiáveis disponíveis, tais como: glib.h[1], strsafe.h[2], para amenizar os riscos.

glib.h: primeiramente, a glib.h fazia parte do projeto GTK+, formalmente conhecida como GIMP Toolkit, uma *Application Programming Interface* (API) para desenvolvimento de interface gráfica dirigida pela Gnome. Posteriormente, foi detectada a necessidade de separar as bibliotecas não referentes com desenvolvimento gráfico, então criou-se em paralelo o glib.h.

A glib.h é um conjunto de bibliotecas que disponibiliza componentes convenientes para desenvolver aplicações em linguagem C nos sistemas Unix-like. A vantagem do glib.h é poder usufruir funções equivalentes da linguagem C com implementações confiáveis, que podem conter o BOF. No Quadro 2, são exibidas tais funções.

[1] <https://developer.gnome.org/glib/stable/glib-String-Utility-Functions.html>
[2] <https://msdn.microsoft.com/pt-br/library/windows/desktop/ms647466(v=vs.85).aspx>

Quadro 2 – Funções confiáveis em glib.h

Função Confiável	**Substitui**
g_strlcat()	strcat()/strncat()
g_strlcpy()	strcpy()/strncpy()
g_snprintf()	sprintf()/snprintf()

Fonte: Gnome Developer[1].

strsafe.h: é uma biblioteca desenvolvida pela Microsoft, provém soluções exclusivas para o sistema operacional Windows, disponibilizado através do *kit* de desenvolvimento de *software* desde a versão do Windows XP SP2. As funções são exibidas no Quadro 3.

A strsafe.h monitora o tamanho do *buffer* de destino sempre fornecida na função, garantindo que a função não grave após o final do *buffer* e terminação nula. Inclui também funções para contagem de caracteres ("cch") ou contagem de *bytes* ("cb").

Quadro 3 – Funções confiáveis em strsafe.h

Função Confiável	**Substitui**
StringCchCat()/StringCbCat()	strcat()
StringCchCatN()/StringCbCatN()	strncat()
StringCchCopy()/StringCbCopy()	strcpy()
StringCchCopyN()/StringCbCopyN()	strncpy()

Fonte: Microsoft[2].

2.6.5 *Address Space Layout Randomization*

Address Space Layout Randomization (ASLR) é um processo de segurança que previne a exploração das vulnerabilidades ocasionadas pelo BOF, sendo considerada uma das maiores contramedidas da vulnerabilidade (DU, 2017). Esta técnica impede que o atacante encontre de forma certeira os endereços alocados na memória. Basicamente ASLR faz a alocação aleatória dos espaços de endereço das principais áreas de dados, tais como o executável, *heap*, bibliotecas e as posições das pilhas. A implementação ASLR existe em diversos sistemas e arquiteturas, entre elas: Android, DragonFly BSD, iOS, Linux, Windows, NETBSD, OpenBSD, macOS, Solaris.

2.6.6 Linguagem Imune ao *Buffer Overflow*

A linguagem C possui inúmeras falhas de segurança ocasionadas pela ausência de gerenciamento de memória, por não verificar o fluxo e limites dos dados, delegando a responsabilidade de verificação de limite ao desenvolvedor para impedir BOF; se ocorrer desatenção na implementação do *software*, o sistema estará comprometido.

É interessante considerar o desenvolvimento da aplicação empregando linguagens imunes ao BOF, que dispõem de controle do limite dinâmico, presente no Java ou redi-

mensionamento automático do *buffer* como ocorre em Perl. Caso seja indispensável o uso do C, é necessário realizar auditorias de código para mitigar a ameaça.

2.6.7 Ferramenta de Depuração

Os programadores podem incluir em seus projetos ferramentas de depuração para detectar: vazamentos de memória, violação e falha na alocação e desalocação (*segmentation fault*); ao constatar as irregularidades, medidas preventivas e corretivas serão adotadas para aprimorar a qualidade do código. Essas ferramentas são conhecidas como instrumentação binária que podem ser: *Dynamic Binary Instrumentation* (DBI) ou *Static Binary Instrumentation* (SBI).

SBI analisa o código fonte e gera um executável modificado. Esta abordagem tem a vantagem de produzir normalmente executáveis mais eficientes ao comparar com DBI, pois, as ações de SBI são feitas antes da execução e introduz somente o código de instrumentação, enquanto a DBI sofre sobrecarga de análise, desmontagem e outras deficiências em tempo de execução (LAURENZANO et al., 2010). Contudo, SBI tem dificuldades em aplicar instrumentação em bibliotecas compartilhadas precisando processá-las de forma individual para contornar o problema. Outra desvantagem, o código gerado é persistente, de modo não permitir a exclusão ou modificação do código em tempo de execução, diferentemente de DBI que oferece estes recursos.

SBI geralmente são aplicados em projetos com prioridade na eficiência, como os *Data Centers* e *High Performance Computing* que precisam lidar com paralelismo e múltiplos processadores (LAURENZANO et al., 2010). As seguintes ferramentas são exemplos de análise estática: Clang[3], Cppcheck[4] e IBM Security AppScan[5].

DBI realiza análise de código em tempo de execução (*run-time*) e anexa no código original (NETHERCOTE; SEWARD, 2007). Este método é conveniente para os desenvolvedores, por evitar a recompilação e suprimir a necessidade de ter em mãos o código fonte. Existem diversas soluções de Dynamic Binary Instrumentations (DBIs), tais como: Pin[6], DynamoRIO[7] e Valgrind[8].

Valgrind é uma DBI com licença da *GNU General Public License* (GPL). Essencialmente funciona como se fosse uma máquina virtual que cria ambiente de "*sandbox*"para executar o conjunto de ferramentas no programa analisado e também insere instruções próprias para fazer depuração e *profiling* avançada Zhang et al. (2014). O Quadro 4

[3] <https://clang.llvm.org/>
[4] <http://cppcheck.sourceforge.net/>
[5] <https://www.ibm.com/security/application-security/appscan>
[6] <https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool>
[7] <http://www.dynamorio.org/>
[8] <http://valgrind.org/docs/manual/manual.html>

descreve os componentes do *framework*, sendo o *Memcheck* o mais importante, por detectar erros de memórias das linguagens C e C++.

Quadro 4 – Principais ferramentas de Valgrind

Ferramenta	**Descrição**
Memcheck	Monitora a alocação dinâmica de memória e gera relatório de vazamentos de memória.
Helgrind	Conhecido como *Thread Error Detector*, faz detecção de *dead locks*, violação da POSIX e *data race* (acesso de memória sem sincronização ou bloqueio válido).
Cachegrind	Simula como o aplicativo interage com o *cache* do computador e fornece relatório sobre eventuais falhas de *cache*.
Nulgrind	Alternativa simplificada do Valgrind, não executa instrumentação ou análise; direcionado para desenvolvedores que querem realizar testes de regressão.
Massif	Módulo que analisa o gerenciamento de memória *heap* do programa.

Fonte: Valgrind User Manual[8].

Valgrind é capaz de identificar inúmeras anomalias de memória, entretanto, não consegue monitorar os limites de pilhas e *arrays* estáticos, logo, as falhas relacionadas ao *stack overflow* exemplificada no Algoritmo 1 passam despercebidas.

A Figura 3 é um relatório gerado a partir do comando: `valgrind --log-file=valgrind-out.txt ./nomeDoPrograma`. O programa examinado é exemplo do *Double Free* referenciado no Algoritmo 3; o relatório diagnosticou uma chamada inválida da função free() de modo premeditado.

```
==2347== Memcheck, a memory error detector
==2347== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2347== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==2347== Command: ./DoubleFree
==2347== Parent PID: 2331
==2347==
==2347== Invalid free() / delete / delete[] / realloc()
==2347==    at 0x4848B8C: free (vg_replace_malloc.c:530)
==2347==    by 0x10497: main (in /home/pi/Desktop/TCC/DoubleFree)
==2347==  Address 0x49cd028 is 0 bytes inside a block of size 10 free'd
==2347==    at 0x4848B8C: free (vg_replace_malloc.c:530)
==2347==    by 0x1048F: main (in /home/pi/Desktop/TCC/DoubleFree)
==2347==  Block was alloc'd at
==2347==    at 0x4847568: malloc (vg_replace_malloc.c:299)
==2347==    by 0x1047F: main (in /home/pi/Desktop/TCC/DoubleFree)
==2347==
==2347==
==2347== HEAP SUMMARY:
==2347==     in use at exit: 0 bytes in 0 blocks
==2347==   total heap usage: 1 allocs, 2 frees, 10 bytes allocated
==2347==
==2347== All heap blocks were freed -- no leaks are possible
==2347==
==2347== For counts of detected and suppressed errors, rerun with: -v
==2347== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 6 from 3)
```

Figura 3 – Relatório do Algoritmo *Double Free* com Valgrind

Fonte: relatório gerado pelo Valgrind.

3 Solução Aplicada

As soluções escolhidas envolvem as correções do uso de funções que apresentam o risco de ocasionar BOF. Foram selecionadas funções que manuseiam as *strings* apontando casos em que não ocorre a verificação de limite de dados com alternativas confiáveis para a entrada de dados. A categoria do BOF que manipula as *strings* é *Stack Overflow* e a proposta para correção é utilizar a implementação nativa da linguagem C e também empregar as soluções disponíveis pela biblioteca glib.h. A anomalia de memória *Double Free* não possui uma alternativa imune contra a ameça, entretanto, é possível de mitigar o BOF acrescentando uma etapa.

3.1 Funções não-Confiáveis

As funções strcat(), strcpy() e sprintf() realizam manipulação de *strings*, segundo o Quadro 1, apresentam alto risco de ocasionar BOF, pois os limites de memórias não podem ser conferidos. Ademais não há implementação em nível de código capaz de corrigir o estouro do *buffer*. O desenvolvedor deve estar ciente do risco e evitar o uso destas chamadas, usando versões homólogas, capazes de verificar os limites de memória, como o strncat(), strncpy() e snprintf() ou utilizar outras implementações de bibliotecas confiáveis.

3.1.1 strcat()

```
char *strcat (char *destino, const char *origem)
```

Para concatenar *strings*, strcat() acrescenta uma cópia da *string* **origem** na *string* **destino**, o caractere de terminação nula da *string* **destino** é sobrescrita pelo primeiro caractere da *string* **origem**, e depois é adicionado caractere nulo no final da *string* resultante da concatenação, a *string* **origem** se mantém inalterada na execução. Não é possível limitar o tamanho da entrada da *string* **origem** na *string* **destino**, e poderá acontecer BOF.

3.1.2 strcpy()

```
char *strcpy (char *destino, const char *origem )
```

strcpy() copia a *string* **origem** na *string* **destino** (inclusive o caractere de terminação nula da *string* **origem**, pois a cópia somente encerra ao encontrá-lo). Para evitar o BOF o tamanho da *string* **destino** precisa ser maior do que a *string* **origem** com caractere de terminação nula.

3.1.3 sprintf()

```
int sprintf (char *destino, const char *format, ...)
```

A *string* é gravada no formato de printf() pela sprintf(), o resultado é armazenado na *string* **destino**, a terminação nula é automaticamente anexado no final, a chamada retorna valor inteiro informando a quantidade de caracteres armazenados na *string* **destino**. Não é possível delimitar o conteúdo da *string* resultante, e eventualmente ocorrerá BOF, para evitá-lo, o tamanho da *string* **destino** precisa ser grande o suficiente para conter a *string* resultante.

3.2 Funções Confiáveis

As soluções strncat(), strncpy() e snprintf() são mais confiáveis do que seus equivalentes mencionados na Seção 3.1, pois são capazes de limitar a entrada de dados e evitar o BOF, entretanto, continuam sendo vulneráveis de acordo com o Quadro 1, apresentando risco baixo. Usar estas funções por si só não garante a confiabilidade, pois é preciso inserir valor válido e solucionar a ausência do terminador nulo em certas ocasiões.

3.2.1 strncat()

```
char *strncat (char *destino, const char *origem, size_t tamanho)
```

strncat() é análoga ao strcat(), com adição de limite de caracteres a ser concatenado, representado pelo parâmetro **tamanho**. O valor válido de **tamanho** é obtida com sizeof() (obtém o tamanho máximo) e depois subtrair o com strlen() (que medirá o espaço preenchido) e por último subtrair uma unidade para armazenar o terminador nulo. É recomendável inicializar a *string* **destino** com terminador nulo, para evitar lixo de memória (ou garantir que o conteúdo já armazenado tenha o terminador nulo); o Algoritmo 5 ilustra a implementação.

Algoritmo 5 – Implementação confiável do strncat()

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

int main(){
    char destino[6] = "\0", origem[6]  = "TESTE"
    int tamanho     =  sizeof(destino) - 1;
    strncat(destino, origem, tamanho - strlen(destino));
    printf("%s", destino);
    return 0;
}
```

3.2.2 strncpy()

```
char *strncpy (char *destino, const char *origem, size_t tamanho)
```

strncpy() substitui strcpy(), acrescentando limite de caracteres a ser copiado representado pelo parâmetro **tamanho**. strncpy() copia os primeiros caracteres da *string* **origem** até encontrar o primeiro terminador nulo, o restante da *string* **destino** será preenchido com terminador nulo. Caso a *string* **origem** for maior do que a *string* **destino**, o terminador nulo não será inserido no final. Para obter o valor válido do parâmetro **tamanho** é recomendado utilizar sizeof() sobre a *string* **destino** e subtrair uma unidade para gravar o terminador nulo no final, o Algoritmo 6 ilustrará a implementação.

Algoritmo 6 – Implementação confiável do strncpy()

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

int main(){
    char destino[6] = "\0", origem[6]  = "TESTE";
    int tamanho     = sizeof(destino) - 1;
    strncpy(destino, origem, tamanho);
    destino[tamanho] = '\0';
    printf("%s", destino);
    return 0;
}
```

3.2.3 snprintf()

```
int snprintf (char *destino, size_t tamanho, const char *format, ...)
```

snprintf() equivale ao sprintf(), com limite de caracteres a ser armazenada pelo parâmetro **tamanho**. É sugerido obter o valor do parâmetro **tamanho** usando sizeof() sobre a *string* **destino** e subtrair uma unidade para alocar o terminador nulo na última posição, que é adicionado automaticamente; o Algoritmo 7 demonstrará a implementação para mitigar o risco de BOF.

Algoritmo 7 – Implementação confiável do snprintf()

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

int main(){
    char destino[11], origem[6]  = "TESTE";
    int tamanho     = sizeof(destino) - 1;
    snprintf(destino, tamanho, "%s%s", origem, origem);
    printf("%s", destino);
    return 0;
}
```

3.3 Biblioteca glib.h

A biblioteca glib.h disponibiliza alternativas confiáveis de strcat(), strcpy() e sprintf() que respectivamente são substituídas por g_strlcat(), g_strlcpy() e g_snprintf(). Gnome renomeou os tipos primitivos da linguagem C para evidenciar a presença da biblioteca glib.h, adicionando um "g" no início das palavras reservadas, por exemplo, o tipo int se torna gint, o tipo char se torna gchar, o processo se repete aos demais.

3.3.1 g_strlcat()

```
gsize g_strlcat (gchar *destino, const gchar *origem, gsize tamanho)
```

g_strlcat() é análogo ao strncat(), anexa o terminador nulo no final da *string* **destino**, e o parâmetro **tamanho**, diferentemente de strncat(), é o tamanho total do destino, e não o espaço restante; o Algoritmo 8 exemplificará a implementação.

Algoritmo 8 – Implementação confiável do g_strlcat()

```
    #include <stdio.h>
    #include <string.h>
    #include <glib.h>
    #include <glib/gprintf.h>

    int main(){
        char destino[6] = "\0", origem[6]  = "TESTE";
        int tamanho     = sizeof(destino);
        g_strlcat(destino, origem, tamanho);
        return 0;
    }
```

3.3.2 g_strlcpy()

```
gsize g_strlcpy (gchar *destino, const gchar *origem, gsize tamanho)
```

A opção confiável de strcpy() apresentado pela biblioteca glib.h é g_strlcpy(), a cópia da *string* **origem** na *string* **destino** tem o terminador nulo acrescentado, o parâmetro **tamanho**, que recebe sizeof() com uma unidade decrementada diretamente para comportar o terminador nulo, ao contrário de strncpy(), não preenche a memória livre com o terminador nulo; o Algoritmo 9 representará a implementação.

Algoritmo 9 – Implementação confiável do g_strlcpy()

```
#include <stdio.h>
#include <string.h>
#include <glib.h>
#include <glib/gprintf.h>

int main(){
    char destino[6] = "\0", origem[6]  = "TESTE"
    int tamanho     = sizeof(destino);
    g_strlcpy(destino, origem, tamanho);
    return 0;
}
```

3.3.3 g_snprintf()

```
gint g_snprintf (gchar *destino, gulong tamanho, gchar const *format, ...)
```

g_snprintf() é a alternativa confiável do sprintf(), a saída produzida pela função não ultrapassará a quantidade de caracteres representado pelo parâmetro **tamanho** que recebe sizeof() com uma unidade decrementa para alocar o terminador nulo. O Algoritmo 10 demonstrará a recomendação.

Algoritmo 10 – Implementação confiável do g_snprintf()

```
#include <stdio.h>
#include <string.h>
#include <glib.h>
#include <glib/gprintf.h>

int main(){
    char destino[11] = "\0", origem[6]   = "TESTE"
    int tamanho      = sizeof(destino);
    g_snprintf(destino, tamanho, %s%s, origem, origem);
    return 0;
}
```

3.4 *Double Free*

Na Seção 2.5.4 foi mencionada a vulnerabilidade *Double Free*, que resulta em comportamento indefinido ao usar free() em um espaço liberado ou utilizar o valor após seu uso sem atribuir o terminador nulo. A vulnerabilidade é neutralizada ao adicionar uma etapa de verificação do ponteiro, ou, após uso de free(), fazer com que o ponteiro receba valor NULL imediatamente, conforme mostrado no Algoritmo 11.

Algoritmo 11 – Implementação confiável do *Double Free*

```
#include <stdlib.h>

int main(){
    int *x = malloc(8);
    free(x);
    x = NULL;   //Aterramento do ponteiro
    return 0;
}
```

4 Metodologia

As abordagens de prevenção e correção que existem não corrigem os problemas produzidos pelo BOF no nível código, pois a verificação de limite é um problema indecidível por não ser possível prever em tempo de compilação os endereços das memórias reservadas. A iniciativa mais eficiente e instrutiva para correção do problema é realizar implementações confiáveis direto no código (SHAW; DOGGETT; HAFIZ, 2014) (ZHIVICH, 2005).

A execução dos códigos para analisar as soluções contra o BOF decorreu no equipamento Raspberry Pi considerado como dispositivo de IoT que será descrita na próxima seção.

4.1 Equipamento Utilizado

O equipamento utilizado para ilustrar os dispositivos de IoT foi Raspberry Pi, sendo um *hardware* desenvolvido pela Raspberry Pi Foundation[1], que é uma organização de caridade britânica. A versão específica é o Raspberry Pi 3 modelo B+, disponibilizado em 2018; o Quadro 5 descreve as especificação do dispositivo[2].

Quadro 5 – Especificação do Raspberry Pi 3 modelo B+

Especificação	Descrição
SoC	Broadcom BCM2837B0 quad-core A53 (ARMv8) 64-bit @ 1.4GHz
GPU	Broadcom Videocore-IV
Memória RAM	1GB LPDDR2 *Synchronous Dynamic Random Access Memory* (SDRAM)
Placa de Rede	Gigabit Ethernet (canal *Universal Serial Bus* (USB)), 2.4GHz e 5GHz 802.11b/g/n/ac Wi-Fi
Bluetooth	Bluetooth 4.2, *Bluetooth Low Energy* (BLE)
Armazenamento	Micro-SD
GPIO	40 pinos *General Purpose Input/Output* (GPIO)
Conectores	HDMI, tomada de áudio analógico de 3,5 mm, 4 x USB 2.0, Ethernet, interface serial de câmera (CSI), interface serial de vídeo (DSI)
Dimensões	82mm x 56mm x 19.5mm, 50g

Fonte: Site oficial Raspberry Pi[2].

O dispositivo tem objetivo de estimular o ensino da Ciência da Computação e conceder soluções para atender as necessidade de quem está desenvolvendo através da acessibilidade de aquisição, com valor estabelecido de $35. O *hardware* possibilita criar

[1] <https://www.raspberrypi.org/about/>
[2] <https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/>

uma variedade de projetos, podendo ser empregado como: central de multimídia, *desktop*, servidor de impressão sem fio, estação de rádio FM ou até mesmo controlador de *drones* (LI; LING, 2015). Este componente é capaz de executar diversas tarefas e podendo substituir alguns computadores potentes que normalmente são subutilizados (PAIVA; MOREIRA, 2014). A plataforma IoT permite desenvolver aplicações direcionadas para redes de sensores sem fio, viabilizando soluções inovadoras através da sua flexibilidade e conectividade. Raspberry Pi também tem o potencial para fins acadêmicos, pois, possui suporte para várias linguagens de programação por exemplo C e Python; estas linguagens possuem abstrações ideais para aprendizagem.

O sistema operacional é instalado no dispositivo através de cartão Micro-SD, que recebe designação de classe conforme a capacidade de leitura e escrita. O cartão utilizado é da classe 10 (velocidade leitura e escrita de 10MB/s), pois haverá uma sensível perda de performance ao utilizar cartões com velocidade de leitura e escrita menor[3]. Foi utilizado o sistema operacional Raspbian na arquitetura ARM[4], sendo o sistema oficial da Raspberry Pi[5] baseado na distribuição Debian Linux, mas o dispositivo alcançou uma certa popularidade e recebeu suporte de sistemas de terceiros, tais como: Ubuntu Mate, Snappy Ubuntu Core, Windows IoT Core, *Open Source Media Center* (OSMC), Pinet e RISC OS. A Figura 4 e 5 exibe o dispositivo utilizado para executar a análise de sobrecarga da Seção 4.3 e o estudo de caso apresentado nesta monografia.

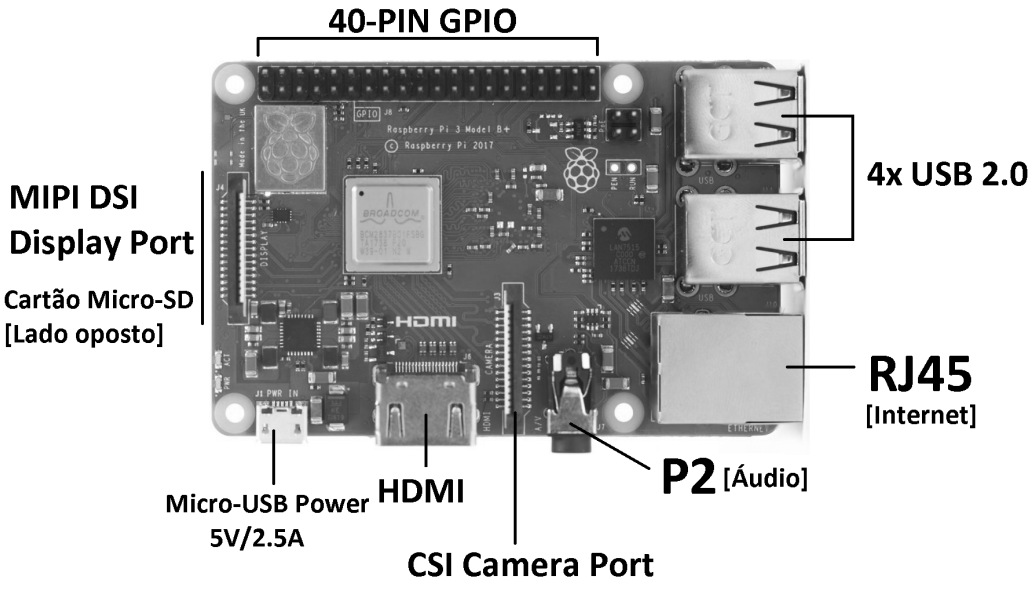


Figura 4 – Conectores do Raspberry Pi 3 Modelo B+

Fonte: Adaptado do site oficial Raspberry Pi[6].

[3] <https://www.raspberrypi.org/documentation/installation/sd-cards.md>
[4] <https://www.raspberrypi.org/downloads/raspbian/>
[5] <https://www.raspberrypi.org/downloads/>

Figura 5 – Raspberry Pi dentro de uma case de acrílico

Fonte: Disponível na Amazon[7].

4.2 Compilação

Os códigos testados (presente no repositório GitHub) foram compilados pelo *GNU Compiler Collection* (GCC) versão: 6.3.0 usando o comando: `gcc main.c 'pkg-config -cflags -libs glib-2.0' -o main -lm`, onde `"pkg- config -cflags -libs glib-2.0"` representa a biblioteca glib.h e '-lm' significa a biblioteca math.h. O *software* foi executado no modo administrador, e antes de cada iteração o *buffer* e o *cache* foram limpos através do comando: `"free && sync && echo 3 > /proc/sys/vm/drop_caches && free"`.

4.3 Análise de Sobrecarga

Neste trabalho foram realizados testes obtendo o tempo de execução de um programa em três situações: (a) função vulnerável ao BOF; (b) função com implementação nativa da

[6] <https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/>
[7] <https://www.amazon.fr/TRIXES-Transparente-Acrylique-Ventilateur-Refroidissement/dp/B01BKIQJD2>

linguagem C com confiabilidade no código; (c) função confiável usando biblioteca confiável glib.h.

As funções selecionadas descritas no Quadro 6, compõem uma pequena amostra, visto que existem outras funções para analisar. As funções: strcat(), strcpy() e sprintf() funcionarão como grupo controle, e respectivamente as funções: strncat(), strncpy() e snprintf() serão o grupo teste de implementação confiável nativa da linguagem C e também as funções: g_strlcat(), g_strlcpy() e g_snprintf() serão outro grupo teste usando implementações confiáveis disponibilizadas pela biblioteca glib.h. A função free() se difere das demais, pois não há uma opção segura que substitui a mesma, entretanto, é possível acrescentar um procedimento que garantirá a segurança e comparar uma eventual diferença de desempenho.

Quadro 6 – Funções Avaliadas

Função não-Confiável	**Função Confiável**	**glib.h**
free()	-	-
strcat()	strncat()	g_strlcat()
strcpy()	strncpy()	g_strlcpy()
sprintf()	snprintf()	g_snprintf()

O *software* que foi utilizado para analisar sobrecarga está disponível no repositório GitHub[8]. A configuração do *software* avaliado usou os parâmetros padrões, o primeiro argumento **BUFFER_SIZE** (tamanho do *buffer*) definido como 10 e o segundo argumento **LOOP** (quantidade de vezes que a função é acionado) recebeu o valor de 100 milhões. O tempo é medido em segundos pela função clock(), que é marcado antes da entrada do laço de repetição, e finalizado ao sair do laço; o fluxo de execução é representado pela Figura 6.

[8] <https://github.com/VitorDeSiqueiraCotta/analise-buffer-overflow-c>

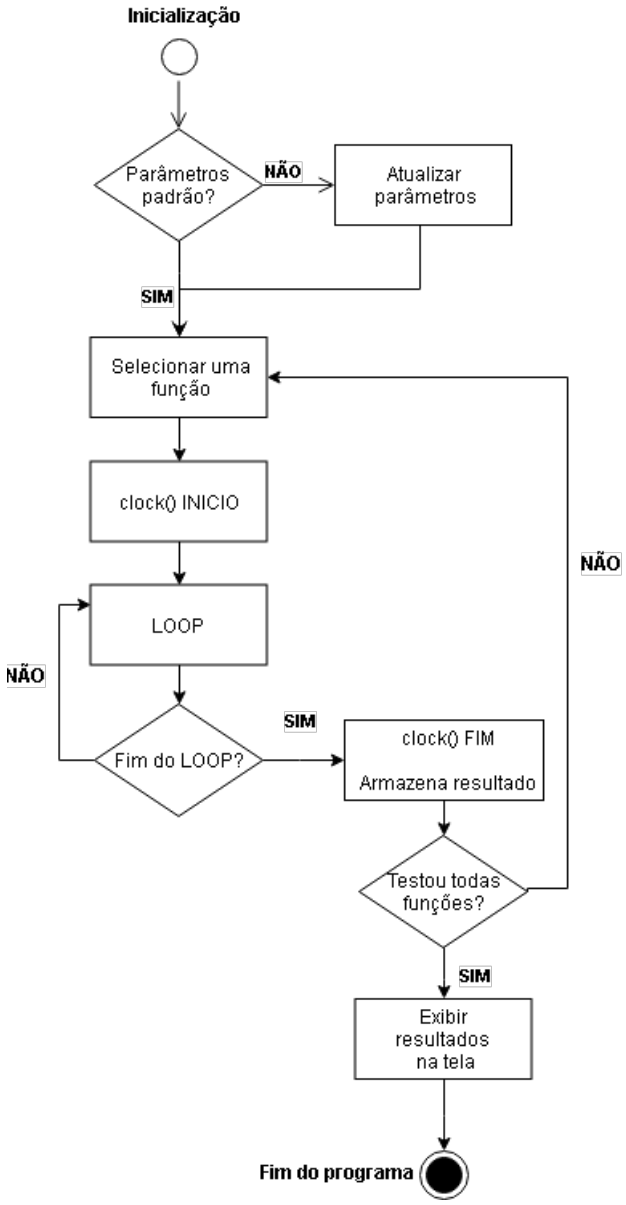


Figura 6 – Fluxograma da Análise de Sobrecarga

Para analisar o desempenho foi considerado a execução de 4 repetições (presente na Tabela 1) para coletar dados e construir a tabela de resultado composta por 4 parâmetros: **Duração Média** (corresponde a média dos tempos de execução), **Desvio Padrão** (indica o grau de oscilação entre as amostras), **Amplitude** (diferença entre o maior e menor tempo das amostras) e **Sobrecarga** que compara os resultados do grupo controle (funções não-confiáveis) com o grupo teste (funções confiáveis). A Figura 7 é exemplo de iteração executado no Raspberry Pi.

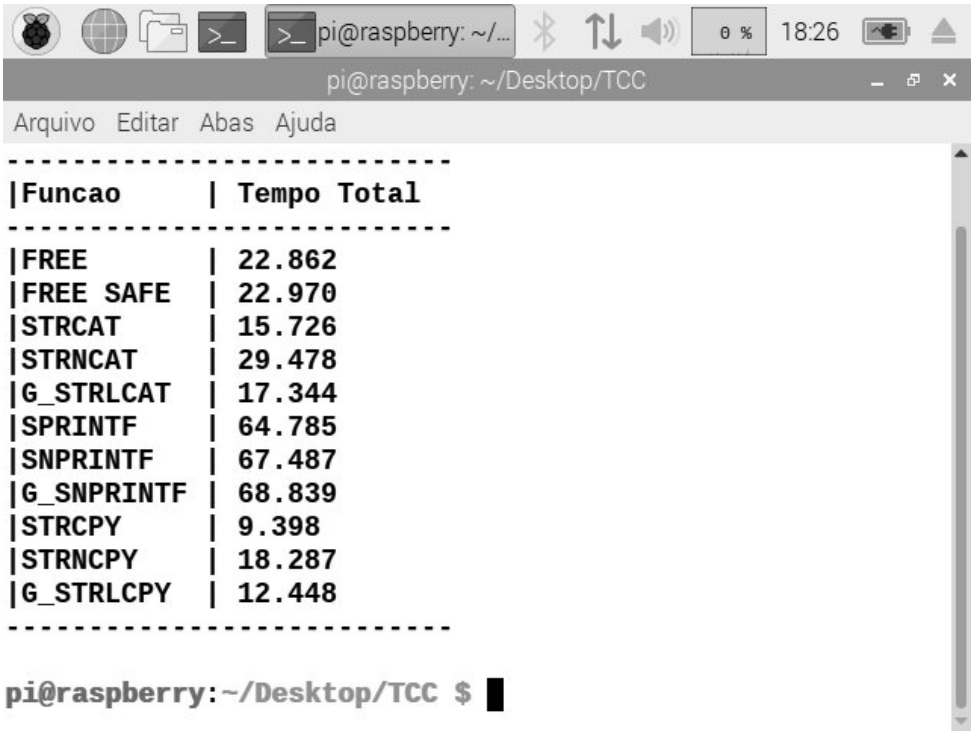


Figura 7 – Exemplo de Iteração no Raspberry Pi

5 Resultados

Os resultados foram elaborados a partir de duas seções, a Seção 5.1 examinará o desempenho das implementações confiáveis de código com as não-confiáveis, enquanto a Seção 5.2 ilustrará uma aplicação dos dispositivos de IoT vulnerável ao BOF.

5.1 Análise de Desempenho

O desvio padrão e amplitude encontrados validam os resultado obtidos no dispositivo Raspberry Pi. A performance entre o desenvolvimento confiável e não-confiável constata a diferença entre as soluções. A implementação confiável da biblioteca glib.h apresentou em média sobrecarga de 18%, enquanto a alternativa confiável da linguagem C apresentou em média 62% de sobrecarga. De forma sucinta, a implementação não-confiável executou a tarefa mais rápida como esperado, pois quaisquer soluções de segurança acrescentam etapas que consumirão recursos. A glib.h apresentou menor impacto de performance sobre a solução nativa em C. A etapa adicional de segurança para *Double Free* apresentou o impacto inferior a 1% em comparação com a versão não-confiável, sendo o único exemplo que não afeta a performance. A Tabela 1 exibe os números obtidos em cada iteração.

Tabela 1 – Resultado das iterações

Função	Iteração 1	Iteração 2	Iteração 3	Iteração 4
strcat	15,726s	15,619s	15,724s	15,723s
strncat	29,478s	29,267s	29,478s	29,478s
g_strlcat	17,344s	17,829s	17,944s	18,364s
strcpy	9,398s	9,331s	9,616s	9,775s
strncpy	18,287s	18,157s	18,536s	18,697s
g_strlcpy	12,448s	12,762s	12,785s	13,131s
sprintf	64,785s	64,271s	64,864s	65,328s
snprintf	67,487s	66,984s	69,057s	69,724s
g_snprintf	68,839s	68,465s	70,012s	72,278s
Free	22,862s	22,694s	22,863s	22,862s
Free Seguro	22,970s	22,809s	23,215s	22,961s

5.1.1 Concatenar *String*

Funcionalidades encarregadas por concatenar *strings*, apresentou sobrecarga nas duas soluções confiáveis, mas o pior cenário foi para strncat(), com 87,45% contra 13,84%

para g_strlcat(). Fica evidente que a alternativa confiável disponibilizada pela glib.h apresentou vantagem com margem de 84,17% sobre strncat(), pois g_strlcat() concatena apenas o tamanho do *buffer* e não o limite do *buffer*.

Tabela 2 – Resultado para concatenar *string*

Função	Duração Média	Desvio Padrão	Amplitude	Diferença %
strcat	15,698s	0,046	0,107s	-
strncat	29,425s	0,091	0,211s	87,45%
g_strlcat	17,870s	0,363	1,020s	13,84%

5.1.2 Copiar *String*

Os componentes que copiam *strings*, tem perda de desempenho de 34,14% para g_strlcpy() e 93,28% para strncpy(); a variação de 63,40% das soluções é explicada na Seção 3.3.2, pois strncpy() preenche o terminador nulo nas áreas de memória não utilizada, enquanto o g_strlcpy() insere apenas o terminador nulo no final da *string* copiado. Conforme a Tabela 3, para desenvolvimento confiável e eficiente é apontado g_strlcpy() como melhor alternativa.

Tabela 3 – Resultado para copiar *string*

Função	Duração Média	Desvio Padrão	Amplitude	Diferença %
strcpy	9,530s	0,176	0,444s	-
strncpy	18,419s	0,210	0,540s	93,28%
g_strlcpy	12,782s	0,242	0,683s	34,12%

5.1.3 Armazenar *String* no Formato Printf

As opções responsáveis por armazenar *string* no formato printf apontou sobrecarga inferior à 10% nas amostras confiáveis, evidentemente a implementação snprintf() foi mais eficiente do que g_snprintf(), com 31,21% de vantagem; conforme a Tabela 4.

Tabela 4 – Resultado para armazenar *string* no formato printf

Função	Duração Média	Desvio Padrão	Amplitude	Diferença %
sprintf	64,812s	0,375	1,057s	-
snprintf	68,313s	1,117	2,740s	5,40%
g_snprintf	69,899s	1,488	3,813s	7,85%

5.1.4 *Double Free*

O resultado da implementação confiável free(), indicada na Tabela 5, obteve sobrecarga inferior a 1%. A leve diferença é devido a etapa adicional para instanciar o ponteiro com valor NULL, imunizando-a contra a vulnerabilidade do *Double Free*, discutida na Seção 2.5.4. Com base no resultado obtido, é válido considerar a alternativa confiável sem sofrer penalidade de performance.

Tabela 5 – Resultado para *Double Free*

Função	Duração Média	Desvio Padrão	Amplitude	Diferença %
Free	22,820s	0,073	0,169s	-
Free Seguro	22,989s	0,145	0,406s	0,74%

5.2 Hub IoT do Windows Azure

Hub IoT do Windows Azure é uma iniciativa desenvolvida pela Microsoft que conecta, monitora e controla vários dispositivos de IoT por meio do serviço da nuvem[1]. A nuvem assume o papel de *hub* central que gerencia a comunicação dos dispositivos que compõem a rede IoT. Com a solução Azure será possível estabelecer conexão confiável e segura dos dispositivos numa rede resiliente a falhas. Hub IoT possui telemetria e relatórios de rastreamento de eventos, essenciais para gerenciar equipamentos industriais e monitorar instrumentos valiosos na área da saúde; a Figura 8 descreve as plataformas suportadas pelo Hub IoT[2].

Figura 8 – Plataformas suportadas pelo *hub* IoT

Fonte: Microsoft[2].

O projeto Azure IoT C SDK está disponível no GitHub[3] e tem o objetivo de facilitar o trabalho do desenvolvedor que quer conectar os dispositivos de IoT utilizando

[1] <https://docs.microsoft.com/pt-br/azure/iot-hub/about-iot-hub>
[2] <https://docs.microsoft.com/pt-br/azure/iot-hub/>
[3] <https://github.com/Azure/azure-iot-sdk-c/tree/0ca9b92d4ff7f7a44fe1687e81f919bd7dcaa944>

a linguagem C. A licença do repositório é a MIT[4] garantindo a liberdade de: modificar, distribuir, vender e sublicenciar. O código foi desenvolvido seguindo as orientações da ANSI C revisão C99; com esta revisão será possível maximizar a portabilidade do código para diversas extensões de compiladores existentes ampliando a compatibilidade da plataforma.

5.2.1 Função getMbedParameter() vulnerável

Neste trabalho foi realizado uma análise do código fonte do Hub IoT e identificou uma vulnerabilidade de BOF no arquivo **iothub_account.c**[5] na linha 227, dentro da função **getMbedParameter()**, descrito no Algoritmo 12. A vulnerabilidade em específico é do tipo *Stack Overflow* originada pela função scanf(), que de acordo com Viega e McGraw (2001) é uma função de alto risco de ocasionar o BOF. A função scanf() não obriga o desenvolvedor limitar o *buffer* de entrada, e, isto poderá ocasionar falhas de execução ou até mesmo um *exploit*.

Algoritmo 12 – Função getMbedParameter() vulnerável

```
223     static const char* getMbedParameter(const char* name)
224     {
225         static char value[MBED_PARAM_MAX_LENGTH];
226         (void)printf("%s?\r\n", name);
227         (void)scanf("%s", &value);
228         (void)printf("Received '%s'\r\n", value);
229 
230         return value;
231     }
```

Fonte: Trecho do código extraído no repositório **azure-iot-sdk-c**[5].

Com objetivo de ilustrar a vulnerabilidade do BOF, o dispositivo Raspberry Pi executou somente o módulo **iothub_account.c** e em específico a função **getMbedParameter()**. A constante definida pelo **MBED_PARAM_MAX_LENGTH** recebe o valor 256, ou seja, se o *buffer* de entrada ultrapassar este valor haverá o BOF. Para testar a integridade da aplicação foi inserido 256 caracteres para alcançar o limite e posteriormente foi adicionado o seguimento "BUFFER_OVERFLOW" para denotar a sobrecarga.

A Figura 9 ilustra o resultado da execução que constata a existência do BOF no código quando o conteúdo "BUFFER_OVERFLOW" é armazenada na variável **value**. Esta anomalia é capaz de: sobrescrever dados, alterar fluxo de execução, ocasionar falhas de execução, injeção de código malicioso e sobrescrever o endereço de retorno para acionar código malicioso (DU, 2017).

[4] <https://mit-license.org/>
[5] <https://github.com/Azure/azure-iot-sdk-c/blob/0ca9b92d4ff7f7a44fe1687e81f919bd7dcaa944/testtools/iothub_test/src/iothub_account.c>

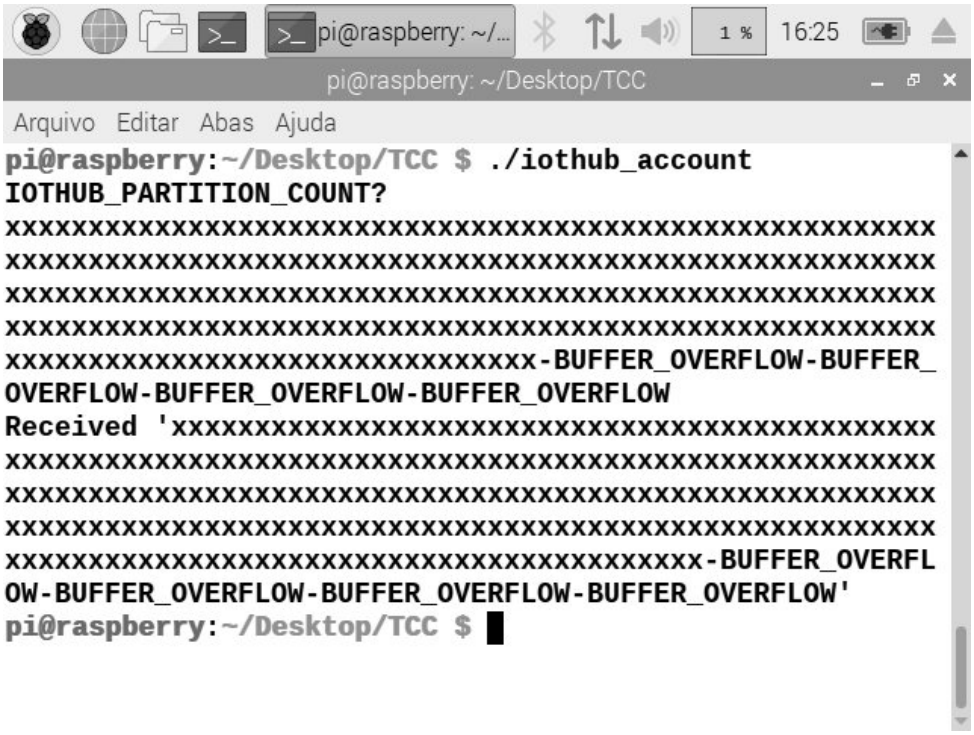


Figura 9 – Execução da função getMbedParameter() vulnerável no Raspberry Pi

5.2.2 Função getMbedParameter() corrigido

A solução do problema está apresentada no Algoritmo 13, em que a função scanf() foi substituída pela fgets(). Apesar do fgets() continuar sendo uma função vulnerável, este apresenta risco menor de ocasionar o BOF (VIEGA; MCGRAW, 2001), pois diferentemente do scanf(), fgets() torna obrigatório delimitação do *buffer* de entrada, mitigando assim a ameaça. Executando o mesmo teste de integridade e atribuindo a constante **MBED_PARAM_MAX_LENGTH** como limite do *buffer*, é possível observar na Figura 10 que apenas os 256 primeiros caracteres foram armazenados, enquanto o restante que seria o BOF foi descartado.

O caso de estudo foi um exemplo prático de como o BOF poderia afetar uma aplicação, mas ao mesmo tempo pode ser corrigida facilmente. Entretanto, a maior dificuldade da vulnerabilidade não é aplicar a correção, e sim identificar a sua existência, pois nem toda iteração resulta em falha e algumas ferramentas de detecção do BOF (por exemplo Valgrind) pode não encontrá-la.

Algoritmo 13 – Função getMbedParameter() corrigido

```
223     static const char* getMbedParameter(const char* name)
224     {
225         static char value[MBED_PARAM_MAX_LENGTH];
226         (void)printf("%s?\r\n", name);
227         (void)fgets(value, MBED_PARAM_MAX_LENGTH, stdin);
228         (void)printf("Received '%s'\r\n", value);
229 
230         return value;
231     }
```

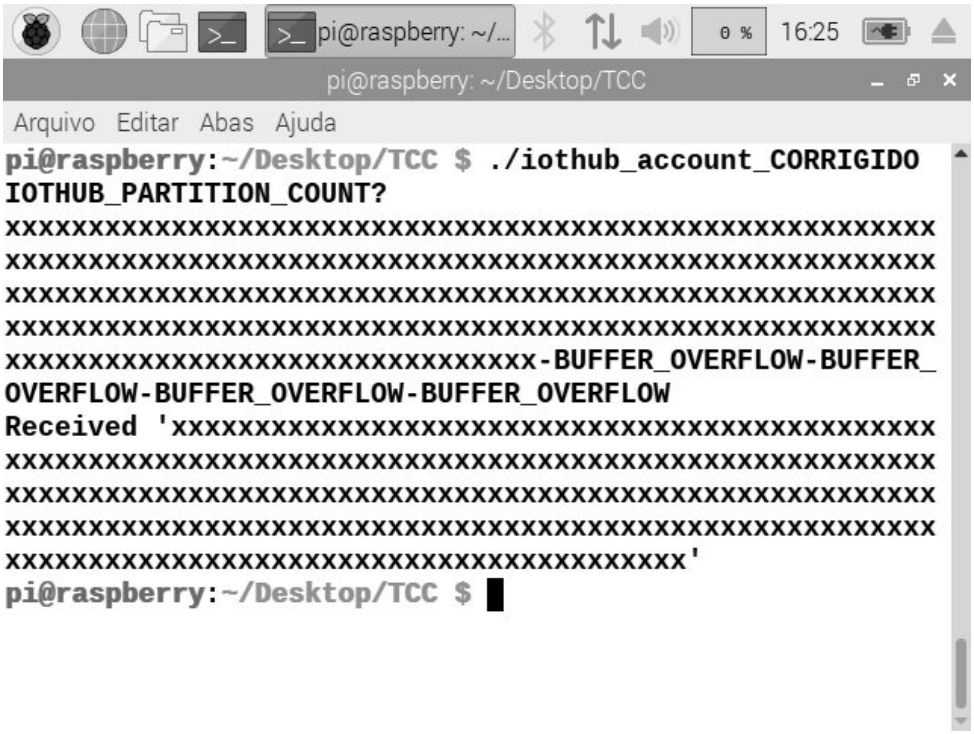


Figura 10 – Execução da função getMbedParameter() corrigido no Raspberry Pi

6 Considerações Finais

As linguagens de programação de alto nível são preparadas para prevenir situações de erro do *Buffer Overflow* (BOF), por exemplo, Java é capaz de realizar verificação de limite de memória tornando-o imune contra a vulnerabilidade. O uso destas linguagens facilitam o desenvolvimento de códigos confiáveis através da sua abstração, contudo, há um custo de processamento que exigirá *hardwares* potentes como *desktops* e *notebooks*. Os dispositivos de IoT tem limitação de recursos e ao utilizar linguagens de alto nível haverá implicações de desempenho. Por este motivo a linguagem C é empregada, pois apesar de exigir mais do programador para implementar códigos confiáveis, C possibilita desenvolvimento de alta performance, graças a sua proximidade com o *hardware*.

A prevenção de erros elementares do BOF não é complexa, porém o desenvolvedor precisa estar atento com as falhas de implementação, pois existem funções comumente da linguagem C suscetíveis à vulnerabilidade. O trabalho analisou o *software* Hub IoT do Windows Azure sendo uma aplicação dos dispositivos IoT que pode ser inviabilizado pelo BOF por uma função que não verifica o tamanho do *buffer* de entrada. Também foi avaliado soluções de código que corrigem o problema, entretanto, foi notado uma sobrecarga que prejudica o desempenho. Em alguns projetos de IoT, a performance é vista como elemento crítico e com isso confiabilidade será tratada como elemento secundário. O BOF continua sendo uma ameaça computacional difícil de detectar e com grande potencial de ser explorada e resultar em falhas de execução. As propostas de confiabilidade dos dispositivos de IoT não se diferem dos computadores tradicionais.

6.1 Recomendação de trabalho futuro

O trabalho estudou implementação confiável ao BOF no nível de código, contudo, há outras alternativas capazes de mitigar o problema. Segue como recomendação de trabalho futuro o estudo de performance das linguagens imunes ao BOF e o uso de outras bibliotecas da linguagem C imune ao BOF.

Referências

ANDERSON, J. P. *Computer Security Technology Planning Study. Volume 2.* [S.l.], 1972. Citado na página 18.

COWAN, C. et al. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: SAN ANTONIO, TX. *USENIX Security Symposium.* [S.l.], 1998. v. 98, p. 63–78. Citado 3 vezes nas páginas 18, 21 e 26.

DECKARD, J. *Buffer overflow attacks: detect, exploit, prevent.* [S.l.]: Elsevier, 2005. Citado na página 20.

DU, W. *Computer Security: A Hands-on Approach.* CreateSpace Independent Publishing Platform, 2017. ISBN 154836794X. Disponível em: <https://www.amazon.com/Computer-Security-Hands-Approach-Wenliang/dp/154836794X?SubscriptionId=AKIAIOBINVZYXZQZ2U3A&tag=chimbori05-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=154836794X>. Citado 4 vezes nas páginas 25, 26, 27 e 46.

GUPTA, S. Buffer overflow attack. *IOSR Journal of Computer Engineering*, v. 1, n. 1, p. 10–23, 2012. Citado 2 vezes nas páginas 19 e 20.

JAZEQUEL, J.-M.; MEYER, B. Design by contract: The lessons of ariane. *Computer*, IEEE, v. 30, n. 1, p. 129–130, 1997. Citado 2 vezes nas páginas 18 e 22.

JIA, X. et al. Towards efficient heap overflow discovery. In: *26th USENIX Security Symposium.* [S.l.: s.n.], 2017. Citado na página 23.

KINDERMANN, R. Static detection of buffer overflows in executables. 2008. Citado na página 25.

LAURENZANO, M. A. et al. Pebil: Efficient static binary instrumentation for linux. In: IEEE. *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS 2010).* 2010. p. 175–183. Disponível em: <https://www.computer.org/csdl/proceedings/ispass/2010/6024/00/05452024.pdf>. Citado na página 28.

LI, C. J.; LING, H. Synthetic aperture radar imaging using a small consumer drone. In: IEEE. *Antennas and Propagation & USNC/URSI National Radio Science Meeting, 2015 IEEE International Symposium on.* 2015. p. 685–686. Disponível em: <https://pdfs.semanticscholar.org/b82a/c0b90548c51e09a7a71c1ccc5264eeaf1a21.pdf>. Citado na página 38.

LIU, T.; CURTSINGER, C.; BERGER, E. D. Doubletake: fast and precise error detection via evidence-based dynamic analysis. In: IEEE. *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on.* [S.l.], 2016. p. 911–922. Citado na página 25.

LéVY, J.-J. Un petit bogue, un grand boum! *Centre de Recherche Commun*, INRIA Microsoft Research, 2010. Disponível em: <http://moscova.inria.fr/~levy/talks/10enslongo/enslongo.pdf>. Citado 2 vezes nas páginas 22 e 23.

NAKAMURA, G.; MARINO, K.; MURASE, I. 2-4 buffer-overflow detection in c program by static detection. *Journal of the National Institute of Information and Communications Technology Vol*, v. 52, n. 1/2, 2005. Disponível em: <https://pdfs.semanticscholar.org/61d9/b190aa3029aafc6a18338a4bcf9607270796.pdf>. Citado na página 18.

NETHERCOTE, N.; SEWARD, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In: ACM. *ACM Sigplan notices*. [S.l.], 2007. v. 42, n. 6, p. 89–100. Citado na página 28.

ORMAN, H. The morris worm: A fifteen-year perspective. *IEEE Security & Privacy*, IEEE, v. 99, n. 5, p. 35–43, 2003. Citado na página 18.

PAIVA, O. A.; MOREIRA, R. d. O. Raspberry pi: a 35-dollar device for viewing dicom images. *Radiologia brasileira*, SciELO Brasil, v. 47, n. 2, p. 99–100, 2014. Citado na página 38.

RITCHIE, D. M. The development of the c language. *ACM Sigplan Notices*, v. 28, n. 3, p. 201–208, 1993. Citado na página 17.

SCHILDT, H.; COMPLETO, C. *Total, 3ª edição*. [S.l.]: Editora Makron Books, 827p, 1997. Citado na página 19.

SETHI, R.; ULLMAN, J. D.; LAM monica S. *Compiladores: princípios, técnicas e ferramentas*. [S.l.]: Pearson Addison Wesley, 2008. Citado na página 25.

SHAW, A.; DOGGETT, D.; HAFIZ, M. Automatically fixing c buffer overflows using program transformations. In: IEEE. *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. [S.l.], 2014. p. 124–135. Citado 2 vezes nas páginas 15 e 37.

SIMPSON, A. K.; ROESNER, F.; KOHNO, T. Securing vulnerable home iot devices with an in-hub security manager. In: IEEE. *Pervasive Computing and Communications Workshops (PerCom Workshops), 2017 IEEE International Conference on*. [S.l.], 2017. p. 551–556. Citado 2 vezes nas páginas 15 e 17.

TEIXEIRA, F. A. et al. Siot–defendendo a internet das coisas contra exploits. *Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC)*, 2014. Citado na página 18.

VIEGA, J.; MCGRAW, G. R. *Building secure software: how to avoid security problems the right way*. [S.l.]: Pearson Education, 2001. Citado 3 vezes nas páginas 21, 46 e 47.

WAHER, P. *Learning internet of things*. [S.l.]: Packt Publishing Ltd, 2015. Citado na página 17.

ZHANG, M. et al. A platform for secure static binary instrumentation. *ACM SIGPLAN Notices*, ACM, v. 49, n. 7, p. 129–140, 2014. Disponível em: <http://seclab.cs.stonybrook.edu/seclab/pubs/vee14.pdf>. Citado na página 28.

ZHIVICH, M. A. *Detecting buffer overflows using testcase synthesis and code instrumentation*. Tese (Doutorado) — Massachusetts Institute of Technology, 2005. Citado 3 vezes nas páginas 15, 19 e 37.

Anexos

TERMO DE RESPONSABILIDADE

Eu, **Vitor de Siqueira Cotta** declaro que o texto do trabalho de conclusão de curso intitulado "***Análise de implementação de código contra o Buffer Overflow e o impacto em dispositivos de IoT***" é de minha inteira responsabilidade e que não há utilização de texto, material fotográfico, código fonte de programa ou qualquer outro material pertencente a terceiros sem as devidas referências ou consentimento dos respectivos autores.

João Monlevade, 20 de dezembro de 2018

Vitor de Siqueira Cotta

UNIVERSIDADE FEDERAL DE OURO PRETO

INSTITUTO DE CIÊNCIAS EXATAS E APLICADAS

COLEGIADO DO CURSO DE SISTEMAS DE INFORMAÇÃO

ANEXO IX – DECLARAÇÃO DE CONFORMIDADE

Certifico que o aluno Vitor de Siqueira Cotta, autor do trabalho de conclusão de curso intitulado "*Análise de implementação de código contra o Buffer Overflow e o impacto em dispositivos de IoT*" efetuou as correções sugeridas pela banca examinadora e que estou de acordo com a versão final do trabalho.

Houve alteração no título do trabalho devido à solicitação da banca.

João Monlevade, 27 de janeiro de 2019.

Marlon Paolo Lima

Prof. Dr. Marlon Paolo Lima

Professor Orientador / DECSI – UFOP

Rua Trinta e seis, 115 – Bairro Loanda – CEP 35931-008 – João Monlevade – MG – Brasil
http://www.icea.ufop.br – cosi@ufop.edu.br – **(31) 3852-8709**