PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
CAMPUS EM POÇOS DE CALDAS

Claudio Correa
correa@pucpcaldas.br

Atividade complementar

Curso de Ciência da Computação
Lab. GR - Laboratório de Gerência de Redes

Nome: Alexandre Peluchi de Oliveira Junior

Poços de Caldas, 2020.

Transbordo de zona de memória
buffer overflow

O termo transbordo de zona de memória é mais conhecido pela designação de *buffer overflow*. Trata- se de um dos erros de programação que mais frequentemente está associada a falhas de segurança, como se pode constatar ao observar uma das mais consolidadas lista de falhas de segurança, a lista CVE (*Common Vulnerabilities and Exposures*).

A falha **CVE-2015-7547**, divulgada publicamente em fevereiro de 2016, afeta a GNU C Library (glibc) e, representa mais um exemplo de vulnerabilidade devido a um transbordo de memória. Assim, nos sistemas afetados, e em certas circunstâncias, o uso da **função getaddrinfo()** pode levar a um transbordo de memória que deixa o sistema vulnerável à execução remota de código.

O fato da **glibc** estar presente na maioria das distribuiçoes **Linux**, da função **getaddrinfo()** ser utilizada por um número significativo de aplicações e da falha existir desde 2008 tornaram a vulnerabilidade particularmente preocupante (O'Donell, 2016).

Um transbordo de zona de memória ocorre quando uma operação escreve fora da zona de memória que lhe foi previamente atribuída, corrompendo a informação armazenada na zona indevidamente acrescida. Por não possuir suporte nativo para verificação de fronteiras de zonas de memória (*bound checking*), a linguagem **C** pode originar programas que efetuem transbordos de zonas de memória. Situação similar ocorre, por exemplo, ao se utilizar a função **strcpy** para efetuar a cópia de uma *string* para um vetor de caracteres sem o espaço suficiente para armazenar todos os caracteres da *string*. Em particular, é relativamente comum o esquecimento de que uma *string* na linguagem **C** requer sempre o terminador \0, o que acresce em uma unidade o número de octetos necessários para o armazenamento da *string*.

Este tipo de erro (transbordo em uma unidade de uma zona de memória) é tão comum que é identificado como *off-by-one error* (Chris Anley, 2007). O programa apresentado na Listagem 1 exemplifica um transbordo de zona de memória.

Listing 1 – Transbordo de zona de memória em operação de cópia de *string*.

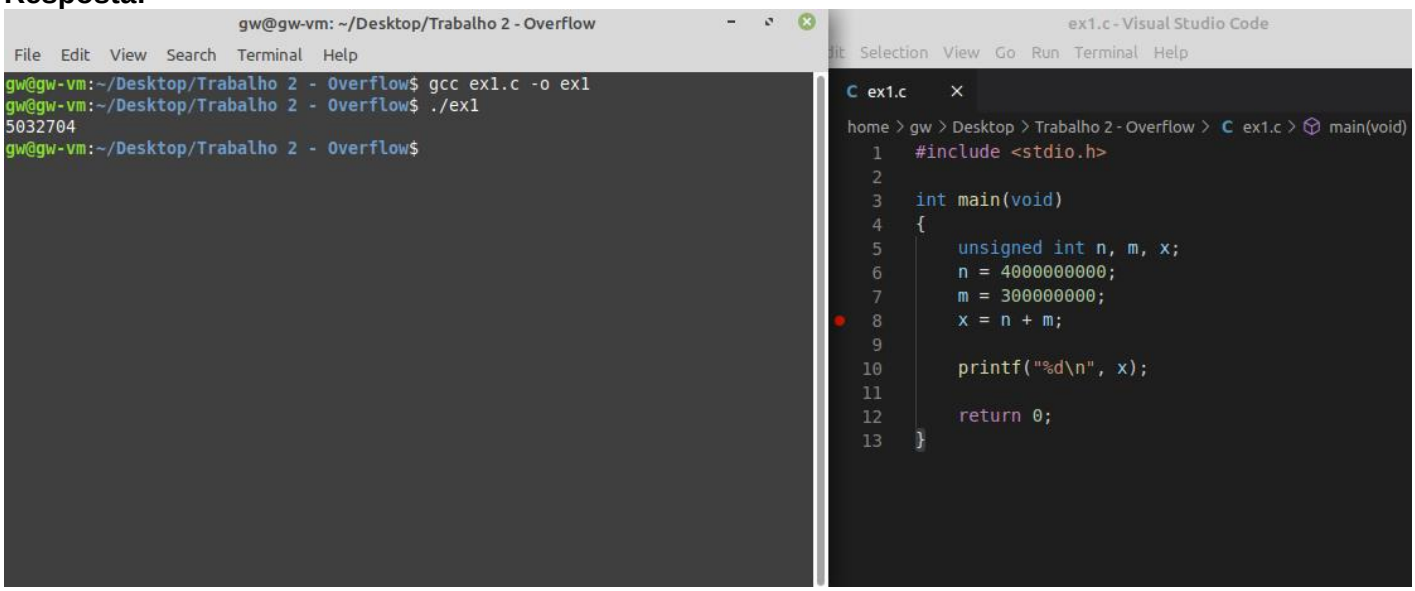
```c
#include <stdio.h>
#include <string.h>

int main(void) {
        char dst_str [8];
        char src_str [] = "ABCDEFGHI";
        printf(" [ INFO ]  Preparando    a copia \ n " );
        printf(" [ INFO ]'% s '  (% zu    octetos ) para "
                        " zona com %zu bytes \ n ",
                        src_str ,
                        strlen (src_str) + 1,
                        sizeof (dst_str));
        strcpy(dst_str,src_str);

        printf(" [ INFO ] Copia    efetuada \ n ");
        printf(" [INFO] string    origem = '% s '\ n " , src_str);
        return 0;
}
```

Exercício 1:
Apresente o resultado da execução do seguinte fragmento de código (Listagem 2) escrito em linguagem **C**. Justifique o valor obtido e implemente uma forma adequada para tratar essa situação.

Listing 2 – Transbordo de zona de memória com *unsigned int*

```c
    unsigned int n, m,x;
    n = 4000000000;
    m =300000000;
    x = n + m;
```

Resposta:



O valor obtido foi “5032704” e ocorreu overflow na linha 8, isto porque operações aritméticas entre números do tipo “*unsigned int*” são inseguras e sujeitas a overflow, o resultado pode não se encaixar no intervalo “0..UINT_MAX” (que é 4294967295U para um objeto do tipo unsigned int), isso só ocorre caso utilize números muito grandes, para números pequenos não é um problema.

Por isso a solução mais simples seria reduzir o limite de valores, ficando condizentes com o tipo da variável.

Exercício 2:

Suponha que seu computador represente cada inteiro em 2 bytes e que cada byte tem 8 bits. Qual o efeito do seguinte fragmento de código (Listagem 3)? Comente e descreva a forma de implementação adequada.

Listing 3 – Transbordo de zona de memória em laço

```
1   unsigned int i ;
2   for ( i = 0; i < 65536; i ++) x[i] = 0;
```

Resultado:

Palavra chave	Tipo	Tamanho	Intervalo
char	Caracter	1	-128 a 127
signed char	Caractere com sinal	1	-128 a 127
unsigned char	Caractere sem sinal	1	0 a 255
Int	Inteiro	2	-32.768 a 32.767
signed int	Inteiro com sinal	2	-32.768 a 32.767
unsigned int	Inteiro sem sinal	2	0 a 65.535
short int	Inteiro curto	2	-32.768 a 32 767
signed short int	Inteiro curto com sinal	2	-32.768 a 32.767
unsigned short int	Inteiro curto sem sinal	2	0 a 65.535
long int	Inteiro long	4	-2.147.483.648 a 2.147.483.647
signed long int	Inteiro longo com sinal	4	-2.147.483.648 a 2.147.483.647
unsigned long int	Inteiro longo sem sinal	4	0 a 4.294.967.295
float	Ponto flutuante com precisão simples	4	3.4 E-38 a 3.4E+38
double	Ponto flutuante com precisão simples	8	1.7 E-308 a 1.7E+308
long double	Ponto flutuante com precisão dupla longo	16	3.4E-4932 a 1.1E+4932

Tabela com todos os tipos de variáveis da linguagem C, seu tamanho em bytes e intervalo.

O que acontece é um overflow, pois o valor 65.536 irá precisar de 17 bits para ser representado. Por isso é necessário substituir o valor que a estrutura de repetição recebe, para o valor máximo representado por 16 bits (11111111 11111111), que seria 65.535 e não ocorrerá o overflow.

Exercício 3:
Comente cada uma das linhas de código apresentada na Listagem 1 e descreva o problema de transbordo observado.

Linha 1:

#include <stdio.h>

Esta biblioteca contém várias funções de ”entrada e saída”. Quase todos os programas C usam essa biblioteca.

Linha 2:

#include <string.h>

Esta biblioteca contém funções que manipulam strings.

Linha 5:

Declaração da variável string “*dst_str*” com capacidade de armazenamento de 7 caracteres mais o “\0” (8 octetos).

Linha 6:

Declaração da variável string “*src_str*” com o valor “ABCDEFGHI” que possui 9 caracteres mais o “\0” (10 octetos).

Linha 8-12:

Printf informando a string da variável “*src_str*” (fonte) que possui o valor “ABCDEFGHI” e o tamanho em bytes da variável “*dst_str”* que possui um espaço menor apenas 8 bytes.

Linha 13:

Copia a string da variável “*src_str”* para “*dst_str*”.

Linha 16:

Exibe o valor da variável “*src_str”.*

Saida após execução do código:

```
[INFO]A preparar cópia
[INFO]'ABCDEFGHI' (10 octetos) para zona com 8 bytes
[INFO]Cópia efetuada
[INFO]string origem='I'
```

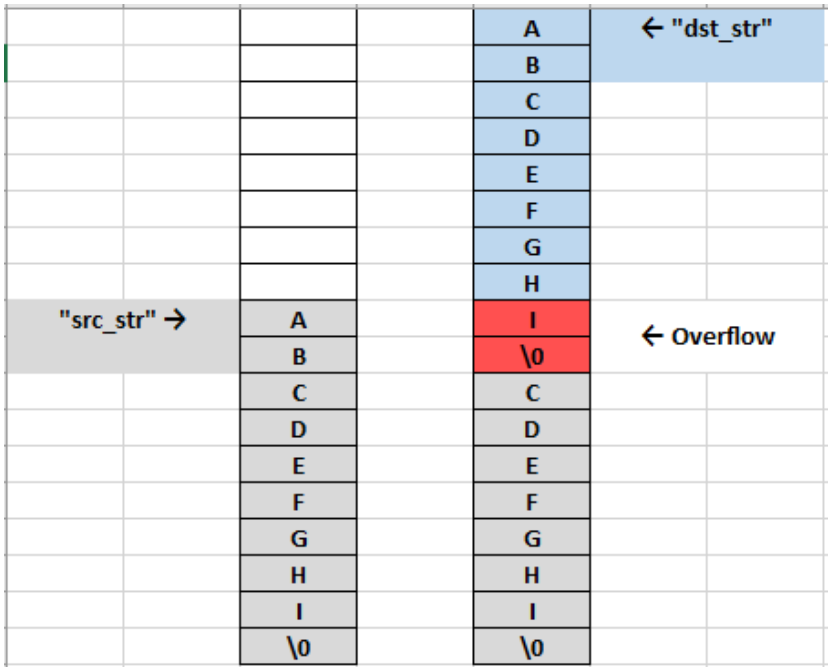
Descrição do problema:

O que acontece no exemplo, é a declaração de duas strings, uma possui capacidade de armazenamento de 8 caracteres (*dst_str)* e a outra possui uma string com 9 caracteres (*src_str)*.

A string “ABCDEFGHI” da variável “*src_str”* é copiada pela função “*strcpy*” para o endereço de memória da variável “*dst_str*” e é ai que acontece o problema. A área na memória reservada pela váriavel “*dst_str”* pode apenas armazenar 8 octetos e está recebendo 10, levando ao “stack overflow” dos dois últimos octetos na zona de memória de “*dst_str*”.

Ou seja os octetos “I” e “\0” serão copiados na memória, após o fim do espaço reservado por “*dst_str*” que corresponde ao início do endereço de memória reservado por “*src_str”,* por isso o resultado passa a ser de “ABCDEFGHI” para “I”, pois logo após o “I” encontra-se o valor “\0” significando o fim da string.

A solução ideal seria, adicionar um limite máximo nas variáveis string para que sejam iguais em tamanho.



Exemplo das posições reservadas na memória pelas duas variáveis e do stack overflow

Exercício 4:
Pesquise e apresente um código de exemplo em linguagem **C** relacionado com a falha **CVE-2015-7547** descrita no texto inicial.

Como descrito no enunciado, no inicio do ano de 2016 o Google anunciou que um engenheiro de segurança, tinha encontrado uma vulnerabilidade, sempre que um cliente SSH tentava se conectar a um host especifico ocorria o erro “segmentation fault” (falha de segmentação), que ocorre em um programa quando este tenta acessar (para leitura ou escrita) um endereço na memória RAM que está reservado para outro programa (ou o próprio sistema operativo) ou que não existe.

Esse engenheiro abriu um ticket para investigar o comportamento e, após uma investigação intensa, descobriram que o problema estava na glibc e não no SSH como eles esperavam.

Graças à observação atenta deste engenheiro, foram capazes de determinar que o problema poderia resultar na execução remota de código. Começaram imediatamente uma análise aprofundada do problema para determinar se ele poderia ser explorado e possíveis soluções. Viram isso como um desafio e, após algumas sessões de hacking intensas, foram capazes de criar um exploit funcional completo.

Quando o Google disponibilizou o código, faltava algumas informações a respeito do shellcode, mas alguns desenvolvedores tentaram completar e conseguirar simular o exploit.

Para acessar o repositório com o exploit funcional do Google acesse abaixo:
https://github.com/fjserna/CVE-2015-7547.

Segue abaixo o código (em C e Python):

Código em C

```
* Copyright 2016 Google Inc
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*
*/

#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <err.h>
#include <stdio.h>
#include <string.h>
int main(void)
{
        struct addrinfo hints, *res;
        int r;

        memset(&hints, 0, sizeof(hints));
        hints.ai_socktype = SOCK_STREAM;

        if ((r = getaddrinfo("foo.bar.google.com", "22",
            &hints, &res)) != 0)
                errx(1, "getaddrinfo: %s", gai_strerror(r));

        return 0;
}
```

Código em python

https://github.com/fjserna/CVE-2015-7547/blob/master/CVE-2015-7547-poc.py

Referências

https://www.exploit-db.com/exploits/40339
https://security.googleblog.com/2016/02/cve-2015-7547-glibc-getaddrinfo-stack.html

1 Bibliografia

. O'Donell, C. (02 de 2016). *CVE-2015-7547 - glibc getaddrinfo() stack-based buffer overflow*. Obtido de https://sourceware.org/ml/libc-alpha/2016-02/msg00416.html

. Chris Anley, J. H. (2007). *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*
(2 ed.). Wiley Publishing, Inc.

Bons estudos!

Escrito em LaTeX