Problemas com Ponteiros

Camilla Hollmann, André Grégio

Neste capítulo, serão discutidos alguns problemas com ponteiros, focando no gerenciamento e acesso incorretos à memória (em geral inválida ou não-inicializada).

Ponteiro não-inicializado (*wild pointer*)

Um ponteiro não-inicializado aponta para uma região de memória qualquer, que pode estar inclusive fora do escopo de memória do seu programa.

```
// Declaração de ponteiro para inteiro.
int *ptri;
```

No exemplo, **`ptri`** é um ponteiro para inteiro não-inicializado, ou seja, apesar de poder armazenar um endereço, não sabemos o que ele armazena (pode ser NULL ou algum valor considerado lixo). A seguir, um exemplo de uso incorreto do ponteiro declarado acima:

```
// Atribuição INCORRETA.
*ptri = 99;
```

Por não ter sido inicializado, o ponteiro **`ptri`** ainda não armazena um endereço de memória válido. Dessa forma, o programa irá tentar armazenar o valor 99 em um endereço de memória qualquer para a qual **`ptri`** apontava, causando um comportamento indefinido ou um *crash*...

É importante lembrar que, se você inicializar o ponteiro com NULL, ele não pode ser desreferenciado, pois não há um endereço de memória válido no ponteiro. Veja exemplos:

```
// Declaração de ponteiro para inteiro.
int *ptri;

/* Atribuição de valor nulo para o ponteiro e
   inicialização INCORRETA do ponteiro. */
ptri = NULL;
*ptri = 99;
```

```
// Declaração de ponteiro para inteiro.
int *ptri;

// Inicialização CORRETA do ponteiro.
ptri = malloc(sizeof(int));
```

Ponteiro pendente (*dangling pointer*)

Um ponteiro pendente ou *dangling pointer* é um ponteiro que aponta para uma área da memória que foi excluída ou liberada.

```
// Declaração e inicialização de ponteiro para inteiro.
int *ptri = malloc(sizeof(int));

// Liberação do bloco alocado para ptri, tornando-o pendurado.
free(ptri);
```

No exemplo abaixo, o ponteiro pendente será apontado para NULL, assim deixando de ser *dangling pointer*. Pode-se chamar essa medida de “aterramento” do ponteiro.

```
// Declaração e inicialização de ponteiro para inteiro.
int *ptri = malloc(sizeof(int));

// Liberação do bloco alocado para ptri, tornando-o pendurado.
free(ptri);

// Ponteiro não mais aponta para bloco de memória inválido.
ptri = NULL;
```

Um dos problemas que podem ser causados por *dangling* pointers é o chamado *user-after-free*, no qual o valor contido em um endereço de memória pode ser lido após sua liberação.

```
// Declaração e inicialização do ponteiro para inteiro com 32
Bytes.
int *ptrc = malloc(sizeof(int)*8);
// Atribuição de valor para o vetor e impressão.
for(int i=0; i < N; i++) {
    ptri[i] = i;
    printf("%d ", ptri[i]);
}
// Liberação do bloco alocado para o vetor, tornando-o pendurado.
free(ptri);
printf("\nDepois do free:\n");
// Uso da variável após liberação, gerando comportamento
indefinido.
for(int i=0; i < N; i++)
    printf("%d ", *(ptri+i));
```

SAÍDA:
0 1 2 3 4 5 6 7
Depois do free:
1582040296 5 -242498083 -842059238 4 5 6 7

Ler a memória que foi liberada é comportamento indefinido e não há como saber o que será acessado. Além disso, tal prática também pode corromper a *heap*, permitindo a execução arbitrária de código.

Vazamento de memória (*memory leak*)

O *memory leak* ocorre quando se perde a referência para uma memória previamente alocada (perda de ponteiro) e não se libera essa memória depois do uso.

```
// Função que remove elemento de uma pilha dinâmica
void pop(pilha *P) {
    if(P->topo)
        P->topo = P->topo->proximo;

/* Ao ajustar o topo sem guardar o endereço do elemento a ser
 * removido, o elemento que ocupava o topo anteriormente continua
 * ocupando memória e não pode mais ser liberado… */
```

No exemplo a seguir, supõe-se o uso de ponteiros e alocação dinâmica dentro do escopo de uma função qualquer:

```
// Declaração e alocação do ponteiro para inteiro.
int *ptri = malloc(100 * sizeof(int))

// Retorno da função sem dar free, gerando um vazamento de memória.
if (ptri != condicao)
    return 0;
```

Para evitar que isso ocorra, é importante liberar a memória alocada após o uso da variável, utilizando a função *free ( )*:

```
// Declaração e alocação do ponteiro para inteiro.
int *ptri = malloc(sizeof(int))

// Libera a memória alocada após seu uso, evitando o vazamento.
if (ptri != condicao)
    free(ptri);
return 0;
```

O vazamento de memória pode gerar uma série de problemas, desde o mau funcionamento do programa por consumo excessivo de memória, até vulnerabilidades que podem ser exploradas por terceiros, como *denial of service* (negação de serviço), quebrando o programa em execução, até a leitura do intervalo de endereços de memória de um programa visando vazamento de informações. Segue um exemplo de programa com consumo excessivo de memória:

```
// Declaração do ponteiro para inteiro.
int *ptri;

/* Alocação repetitiva de memória sem haver a liberação, levando ao
esgotamento de memória e falha do programa. */
while (1)
    ptri = malloc(sizeof(int));
```

Identificando vazamentos com Valgrind

Uma boa prática para aqueles que desejam evitar problemas com alocação de memória em seus programas é o uso do software *Valgrind,* que detecta os erros decorrentes do uso incorreto da alocação dinâmica de memória.

Depois de feita a compilação, use o seguinte comando para executar seu programa exibindo o relatório de erros do Valgrind:

```
valgrind --leak-check=full ./PROGRAMA < entrada**
```

A Figura 1 mostra-se um exemplo de uso do Valgrind para detectar um vazamento de memória:

```
==4743==
==4743== HEAP SUMMARY:
==4743==     in use at exit: 32 bytes in 2 blocks
==4743==   total heap usage: 11 allocs, 9 frees, 1,184 bytes allocated
==4743==
==4743== 16 bytes in 1 blocks are definitely lost in loss record 1 of 2
==4743==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==4743==    by 0x109652: pilha_cria (libpilha.c:12)
==4743==    by 0x1091FE: teste_criar_pilha (testa_pilha.c:14)
==4743==    by 0x109439: main (testa_pilha.c:67)
==4743==
==4743== 16 bytes in 1 blocks are definitely lost in loss record 2 of 2
==4743==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==4743==    by 0x109652: pilha_cria (libpilha.c:12)
==4743==    by 0x1095B3: main (testa_pilha.c:102)
==4743==
==4743== LEAK SUMMARY:
==4743==    definitely lost: 32 bytes in 2 blocks
==4743==    indirectly lost: 0 bytes in 0 blocks
==4743==      possibly lost: 0 bytes in 0 blocks
==4743==    still reachable: 0 bytes in 0 blocks
==4743==         suppressed: 0 bytes in 0 blocks
==4743==
==4743== For lists of detected and suppressed errors, rerun with: -s
==4743== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

Figura 1: Saída do Valgrind com detecção de vazamento de memória

É interessante rodar o comando com a *flag* **-s**, que faz com que os erros detectados e suprimidos sejam exibidos de modo mais detalhado.

Note no início da Figura 1 (*HEAP SUMMARY*) que o Valgrind informa que, ao fim do programa, ainda havia 32 bytes em uso, isto é, alocações cuja memória não foi liberada (linha: ***in use at exit: 32 bytes in 2 blocks***). Na linha subsequente, é mostrado o uso total da *heap*, que ajuda a ver quantos *mallocs* não tiveram seus respectivos *frees* (a conta deve bater!): ***total heap usage: 11 allocs, 9 frees, 1184 bytes allocated***, indicando que foram feitos 11 mallocs e apenas 9 frees, e mais de 1Kbyte foi alocado durante a execução do programa monitorado pelo Valgrind.

A seguir, mostra-se cada um dos 2 blocos ***in use at exit*.** No primeiro bloco, onde é indicado que 16 bytes foram perdidos, note que o Valgrind exibe a função, o arquivo do código e a linha de código que geraram o alerta:

- O *main()* do arquivo *testa_pilha.c*, na **linha 67** (Figura 2), chama a função *teste_criar_pilha()*, a qual faz uma chamada de *pilha_cria()* na **linha 14** (Figura 3)do mesmo arquivo:

```
65      printf ("Teste 1: criar pilha e ver se esta vazia:\n");
66      printf ("Esperado: tem que dar pilha vazia com tamanho zero\n");
67      p= teste_criar_pilha ();
68      teste_pilha_vazia (p);
69      printf ("\n\n");
```

Figura 2: função *main()* do arquivo *testa_pilha.c*

```
10  pilha_t* teste_criar_pilha ()
11  {
12      pilha_t* p;
13
14      if (! (p = pilha_cria ()))
15          fprintf (stderr, "Falha na alocacao da pilha\n");
16
17      printf ("\tPilha criada com sucesso\n");
18      return p;
19  }
```

Figura 3: definição da função *teste_criar_pilha()* no início do arquivo *testa_pilha.c*

- A função *teste_criar_pilha()* por sua vez chama *pilha_cria()*, que faz um mallloc na **linha 12** do arquivo *libpilha.c* (Figura 4).

```
5   /*
6    * Cria e retorna uma nova pilha.
7    * Retorna NULL em caso de erro de alocação.
8   */
9   pilha_t *pilha_cria (){
10      pilha_t *pilha;
11
12      if(!(pilha = (malloc(sizeof(pilha_t)))))
13          return NULL;
14      pilha->topo = NULL;
15      pilha->tamanho = 0;
16      return pilha;
17  }
```

Figura 4: definição da função *pilha_cria()* no arquivo *libpilha.c*

- No segundo bloco do relatório mostrado na Figura 1, o Valgrind informa que mais 16 bytes foram perdidos em alocações não liberadas, desta vez devido a algo iniciado na **linha 102** da função *main()* do arquivo *testa_pilha.c* (Figura 5).

```
 99      printf ("Teste 6: destruir uma pilha com elementos:\n");
100      printf ("Esperado: nao pode ter leak (conferir com valdrind)\n");
101      printf ("          E nao pode ter segfault\n");
102      p = pilha_cria ();
103      if (push (p, 1) && push (p, 2) && push (p, 3))
104          pilha_destroi (&p);
105      else
106          printf ("Falha na alocação dos elementos!!!");
```

Figura 5: chamada da função *pilha_cria()* no *main()* do arquivo *testa_pilha.c*

O que ocorre é que após a criação do espaço de memória para a pilha "p" e seu uso, não é feita a liberação da memória após o tempo de vida esperado de "p".

Isto ocorre devido a um problema na função *pilha_destroi()*, que libera a memória alocada para cada *nodo_t*, mas não a da *pilha_t*... Para corrigir o erro, basta adicionar a chamada de *free* na **linha 27** da função responsável pela liberação, *pilha_destroi()*, como mostrado na Figura 6.

```
19  /* Desaloca toda memoria da pilha e faz pilha receber NULL. */
20  void pilha_destroi (pilha_t **pilha){
21      nodo_t *aux;
22      while ((*pilha)->topo != NULL){
23          aux = (*pilha)->topo;
24          (*pilha)->topo = aux->prox;
25          free(aux);
26      }
27      free(*pilha);
28      *pilha =  NULL;
29  }
```

Figura 6: Função *pilha_destroi()* corrigida no arquivo *libpilha.c*

Ao adicionar a chamada faltante ao *free*, os 32 bytes "vazados" da memória referentes às chamadas à função *pilha_cria()*, nas linhas 67 e 102 do arquivo *testa_pilha.c* desaparecem. Após a correção e reexecução do Valgrind, a saída esperada é que não se tenha nenhum byte/bloco em uso ao finalizar a execução do programa, como ilustrado na Figura 7.

```
==4838==
==4838== HEAP SUMMARY:
==4838==     in use at exit: 0 bytes in 0 blocks
==4838==   total heap usage: 11 allocs, 11 frees, 1,184 bytes allocated
==4838==
==4838== All heap blocks were freed -- no leaks are possible
==4838==
==4838== For lists of detected and suppressed errors, rerun with: -s
==4838== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figura 7: Saída do Valgrind sem erros!

A mensagem ***"All heap blocks were freed - - no leaks are possible"*** nos informa que não há vazamentos possíveis. Na última linha, é exibido um sumário de erros - no caso, 0.

O comando a seguir permite observar a origem de todos os valores não inicializados, porém ressalta-se que o custo de execução com o parâmetro *yes* no argumento *--track-origins* é alto.

```
valgrind --leak-check=full --track-origins=yes ./PROGRAMA
```

A *flag* **-*v*** significa *verbose* e permite que o relatório seja mais detalhado de forma compreensível para o usuário. É possível obter um relatório mais detalhado do que pode ser feito usando o *Valgrind* com o comando:

```
valgrind --help
```

Pontos importantes!

Em resumo:

- Um ponteiro não-inicializado pode armazenar um valor indeterminado, enquanto que um ponteiro nulo armazena o valor determinado “NULL“ (que não é o endereço válido para qualquer objeto do programa ao qual o ponteiro pertence).
- Um ponteiro pendente é o ponteiro que é liberado de forma incorreta (sem ser apontado para NULL), podendo causar seu uso posterior com comportamento indeterminado.
- O vazamento de memória ocorre quando um ponteiro não é liberado após sua vida útil, tornando aquela parte da memória inacessível e desperdiçando recursos do sistema.
- A ferramenta Valgrind ajuda a identificar vazamentos de memória em programas, contabilizando bytes perdidos e chamadas à *malloc* e free, e exibindo as funções envolvidas no vazamento (indicando a linha e o arquivo do código-fonte).

Referências:

- https://wiki.sei.cmu.edu/confluence/display/c/MEM31-C.+Free+dynamically+allocated+memory+when+no+longer+needed - Carnegie Mellon University Software Engineering Institute - SEI CERT C Coding Standard Wiki.
- https://www.geeksforgeeks.org/dangling-void-null-wild-pointers/
- https://cwe.mitre.org/data/definitions/401.html - Common Weakness Enumeration - CWE 401 - Missing Release of Memory after Effective Lifetime.
- https://encyclopedia.kaspersky.com/glossary/use-after-free/ - Encyclopedia by Kaspersky - Use-After-Free.
- https://owasp.org/www-community/vulnerabilities/Memory_leak - OWASP - Memory Leak.
- https://web.stanford.edu/class/archive/cs/cs107/cs107.1174/guide_valgrind.html - Nate Hardison & Julie Zelenski - Guide to Valgrind.