Segurança de Memória e Valgrind

Segurança de Memória

- Segurança contra ataques
  - Menos relevante em ED
- Segurança contra bugs
  - Ativamente avaliado em ED
- A linguagem C não é segura em memória por padrão

Erros mais comuns

- Use-after-free: Acessar um ponteiro depois de desalocar
  - Pode funcionar, pode não funcionar
- Ponteiros não inicializados
  - Pode conter lixo de memória, ou NULL
  - Depende do compilador
- Vazamento de memória
  - Memória alocada perdida

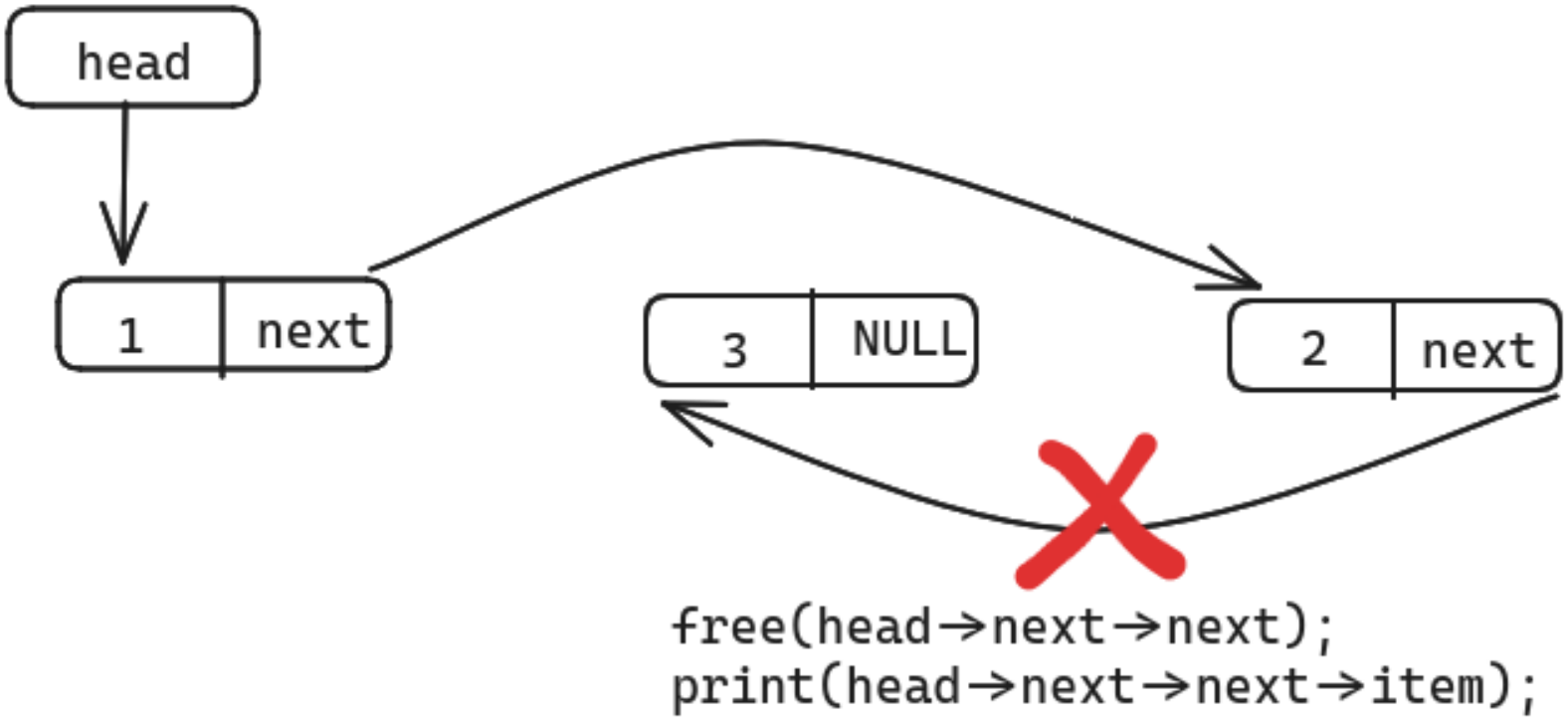
Use-after-free

- Quando um ponteiro é desalocado, mas ainda é usado

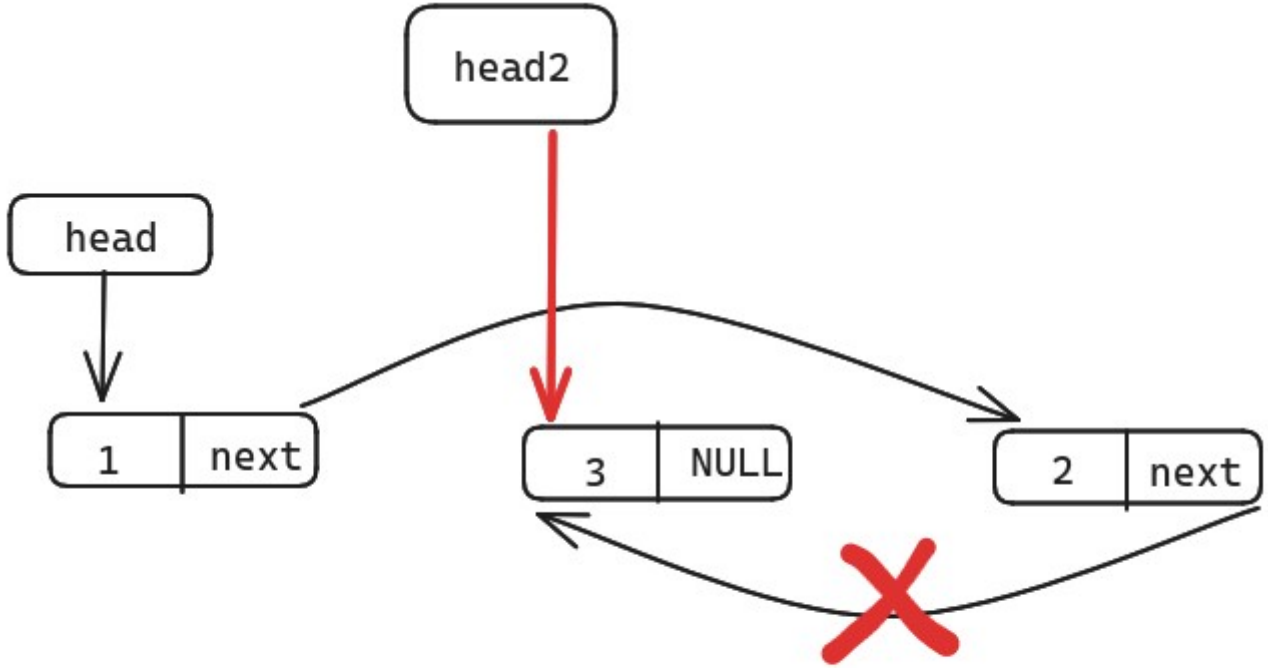
```
int *ponteiro;
ponteiro = malloc(sizeof(int));
*ponteiro = 1;
free(ponteiro);
*ponteiro = 2; /* W: pointer 'ponteiro' used after 'free'
```

- Detectado pelo compilador em casos simples (-Wall)
- Gera comportamento “indefinido”

Perigo 1 do Use-After-Free (em ED)



Perigo 2 do Use-After-Free (em ED)



Ponteiros não-inicializados

- O conteúdo inicial de qualquer variável em C é desconhecido
  - Alguns compiladores colocam NULL, mas não confie!
- Erro comum com alocações de estruturas com muitos ponteiros
- Provavelmente vai acessar lixo de memória
  - Endereço inválido: Segmentation Fault
  - Endereço válido: Valor também será lixo de memória

Exemplo do que (não) fazer

```
 9 Celula *nova_celula(int valor){
10      Celula *nova = malloc(sizeof(Celula));
11      nova->valor = valor;
12      // celula->esq = NULL
13      // celula->dir = NULL
14      return nova;
15 }
```

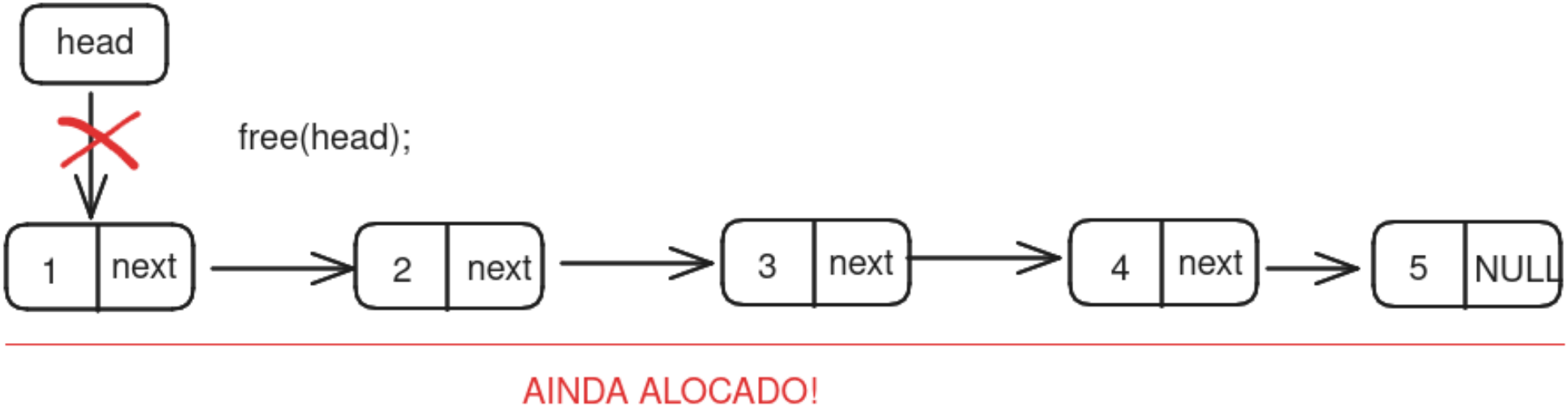
Vazamento de Memória

- Memória é alocada, mas não é desalocada

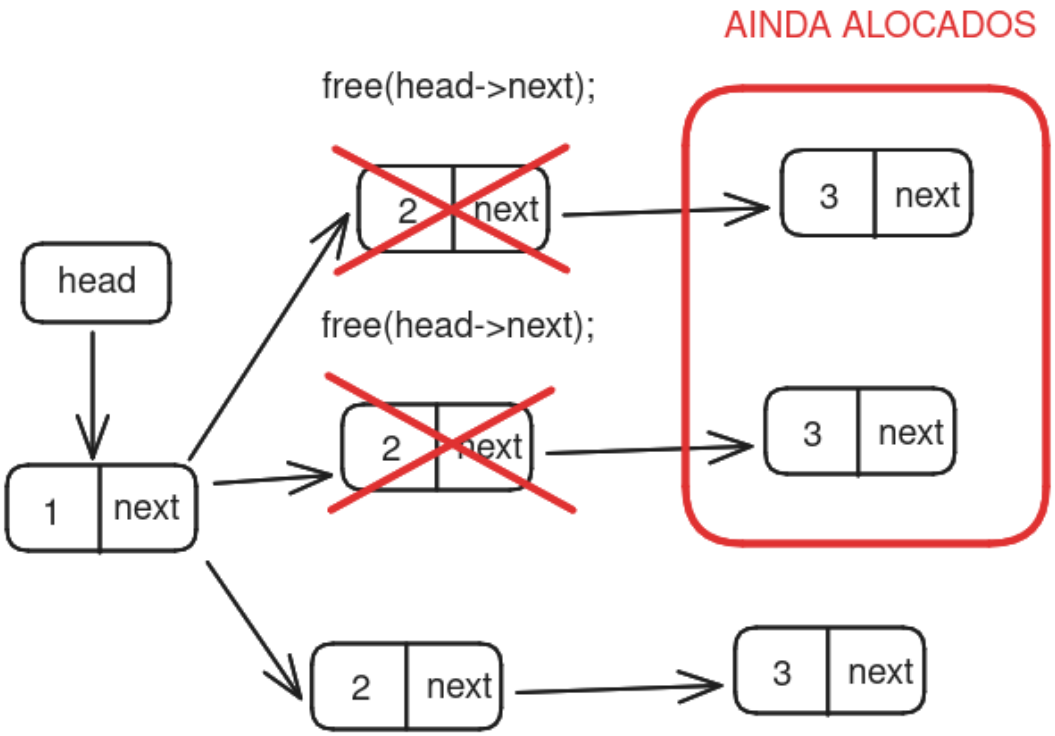
- Memória pode ainda estar acessível por um ponteiro
- Ou pode ter tido seu endereço perdido

- Em processos intensivos, pode desperdiçar memória até crashar
- Geralmente não gera “erros”

Exemplo Ilustrativo



Exemplo provável



Valgrind Memcheck

- Ferramenta para encontrar falhas de segurança de memória
- Automaticamente detecta os erros listados
- Gera um log (difícil de ler) no stderr
- Não precisa de acesso ao seu código

Como usar

```
[castro@kwarismi Valgrind]$ ./a.out
2
[castro@kwarismi Valgrind]$ valgrind ./a.out_
```

```
HEAP SUMMARY:
    in use at exit: 21,445 bytes in 10 blocks
  total heap usage: 44 allocs, 34 frees, 59,312 bytes allocated
```

Vazamentos

```
LEAK SUMMARY:
   definitely lost: 4 bytes in 1 blocks
   indirectly lost: 0 bytes in 0 blocks
     possibly lost: 0 bytes in 0 blocks
   still reachable: 0 bytes in 0 blocks
        suppressed: 0 bytes in 0 blocks
```

- Caso não haja vazamentos, terá apenas a essa mensagem

```
All heap blocks were freed -- no leaks are possible
```

Leak-check=full

- Flag do Valgrind Memcheck para memory leak
- Lista em que função cada “bloco” perdido foi alocado
- Util se você estiver usando mais de uma estrutura de dados

```
[castro@kwarismi Valgrind]$ valgrind --leak-check=full ./main.out_
```

```
==25862== 4 bytes in 1 blocks are definitely lost in loss record 1 of 1
==25862==    at 0x4841828: malloc (vg_replace_malloc.c:442)
==25862==    by 0x10914A: main (in /home/castro/Projects/ED/Valgrind/main.out)
```

Ponteiros Não-Inicializados

- Valgrind Memcheck automaticamente detecta isso
- Lista em qual função isso acontece

```
Use of uninitialised value of size 8
   at 0x109145: main (in /home/castro/Projects/ED/Valgrind/main.out)
```

- As flags -Wall e -Werror também tentam detectar casos simples

Use-After-Free

- Ele também detecta leituras e escritas invalidas
  - Isso inclui “use-after-free”, mas também outras leituras/escritas ilegais
  - Muito útil quando o código “só funciona na sua máquina”

```
Invalid read of size 4
   at 0x109189: main (in /home/castro/Projects/ED/Valgrind/main.out)
 Address 0x4a6c040 is 0 bytes inside a block of size 4 free'd
   at 0x484488F: free (vg_replace_malloc.c:985)
   by 0x109184: main (in /home/castro/Projects/ED/Valgrind/main.out)
 Block was alloc'd at
   at 0x4841828: malloc (vg_replace_malloc.c:442)
   by 0x10916A: main (in /home/castro/Projects/ED/Valgrind/main.out)
```

Demonstração

Exercício Bash