[EXEMPLO]

```
void main() {
    int i;
    for (i=0; i<=LENGTH; i++) {
        vec[i] = nondet_int();
    }
    int r = maxarray(vec, LENGTH);
}
```

Verificação de acessos ilegais ao array

- cbmc lab.c **--bounds-check** --pointer-check

[OUTPUT]

[main.array_bounds.2] line 18 array 'vec' upper bound in vec[(signed long int)i]: FAILURE

Para corrigirmos este FAILURE, devemos modificar a condição do loop para: **i<LENGTH**. Deste modo, obtemos:

[main.array_bounds.2] line 18 array 'vec' upper bound in vec[(signed long int)i]: SUCCESS

```
int maxarray(int u[], int size) {
	int i = 1, a, max = 0;
	while (i < size) {
		if (u[i] > u[max]) { max = i; }
		i++;
	}
	__CPROVER_assume(i<=size);
	return max;
}

void main() {
	int i;
	for (i=0; i<=LENGTH; i++) {
		vec[i] = nondet_int();
	}
	int r = maxarray(vec, LENGTH);
	__CPROVER_assume(r >= 0 && r < LENGTH, "Index Invalid");
}
```

- cbmc lab.c **--function maxarray --pointer-check --unwind 10**

[OUTPUT]

Unwinding loop maxarray.0 iteration 656 file lab.c line 9 function maxarray thread 0

Quanto ao número de iterações consideradas, ele pode afetar o resultado da verificação. Um número maior de iterações pode levar a uma análise mais completa do código, mas também pode aumentar o tempo de verificação. Por outro lado, um número menor de iterações pode resultar em uma análise incompleta e, portanto, pode não capturar todas as propriedades desejadas. É importante encontrar um equilíbrio que permita uma verificação eficiente e abrangente. Experimente ajustar o número de iterações ou fornecer outras restrições para melhorar a análise e obter resultados mais satisfatórios.

Verificação funcional: intervalo do índice calculado por maxarray

Acrescente um assert na função maxarray que lhe permita comprovar que: "o índice devolvido está contido entre 0 e size-1".

```
int maxarray(int u[], int size) {
	int i = 1, a, max = 0;
	while (i < size) {
		if (u[i] > u[max]) { max = i; }
		i++;
	}
	__CPROVER_assume(max >= 0 && max < size-1, "Invalid index");
	return max;
}
```

[PARA TESTAR ESSE ASSERT]

- cbmc lab.c **--bounds-check**

[OUTPUT]

[main.array_bounds.1] line 24 array `vec' lower bound in vec[(signed long int)i]: <ins>SUCCESS</ins> [main.array_bounds.2] line 24 array` vec' upper bound in vec[(signed long int)i]: SUCCESS [main.assertion.1] line 28 Invalid index: SUCCESS

[PRÉ-CONDIÇÕES PARA O MAXARRAY()]

```
CPROVER_assume(u != NULL); // Check that the array is not null
CPROVER_assume(size >= 0 && size <= LENGTH); // N is the maximum valid size for the array u
```

- cbmc lab.c **–function maxarray –bounds-check –pointer-check –unwind 10 –trace –stop-on-fail**

Verificação funcional: máximo

```
void main() {
	int i;
	for (i=0; i<=LENGTH; i++) {
		vec[i] = nondet_int();
	}
	int r = maxarray(vec, LENGTH);
	for (i=0; i<LENGTH; i++) {
		assert (vec[i] <= vec[r]);
	}
}
```

O que é o mesmo que, em vez desse ciclo final, colocar:

```
i = nondet_int();
__CPROVER_assume (0 <= i && i < LENGTH);
assert (vec[i] <= vec[r]);
```

Equivalente a:

```
∀i. 0 <= i < LENGTH → vec[i] <= vec[r]
```

Teste de invariantes

Investigue, usando o CBMC, quais das seguintes fórmulas não são invariantes do ciclo da função maxarray:

```
Formula 1: (∀k. 0 ≤ k ≤ i → vec[k] ≤ vec[max])
Formula 2: (∀k. 0 ≤ k < i → vec[k] < vec[max])
Formula 3: (∀k. 0 ≤ k < i → vec[k] ≤ vec[max])
Formula 4: (∀k. 0 ≤ k ≤ i → vec[k] < vec[max])
```

```
void main() {
    int i;
    for (i=0; i<LENGTH; i++) {
        vec[i] = i;
    }
    int r = maxarray(vec, LENGTH);

    // Formula 1: (∀k. 0 ≤ k ≤ i → vec[k] ≤ vec[max])
    for (i = 0; i <= r; i++) {
        assert(vec[i] <= vec[r]);
    }

    // Formula 2: (∀k. 0 ≤ k < i → vec[k] < vec[max])
    for (i = 0; i < r; i++) {
        assert(vec[i] < vec[r]);
    }

    // Formula 3: (∀k. 0 ≤ k < i → vec[k] ≤ vec[max])
    for (i = 0; i < r; i++) {
        assert(vec[i] <= vec[r]);
    }

    // Formula 4: (∀k. 0 ≤ k ≤ i → vec[k] < vec[max])
    for (i = 0; i <= r; i++) {
        assert(vec[i] < vec[r]);
    }
}
```

[OUTPUT]

[main.assertion.1] line 39 assertion vec[(signed long int)i] <= vec[(signed long int)r]: SUCCESS
[main.assertion.2] line 44 assertion vec[(signed long int)i] < vec[(signed long int)r]: SUCCESS
[main.assertion.3] line 49 assertion vec[(signed long int)i] <= vec[(signed long int)r]: SUCCESS
[main.assertion.4] line 54 assertion vec[(signed long int)i] < vec[(signed long int)r]: FAILURE

Como tal, a fórmula 4 não é um invariante de ciclo da função maxarray.

Verificação de overflow

```
int sum (int u[], int size) {
    int i, s=0;
    for (i=0; i<size; i++)
        s += u[i];
    return s;
}

void main() {
    int i;
    for (i = 0; i < LENGTH; i++) {
        // Add constraints to ensure no overflow occurs
        __CPROVER_assume(vec[i] >= INT_MIN / LENGTH && vec[i] <= INT_MAX / LENGTH);
    }

    int result = sum(vec, LENGTH);
    printf("Sum: %d\n", result);
}
```

- cbmc lab.c **–signed-overflow-check**

[OUTPUT]

[__sputc.overflow.1] line 271 arithmetic overflow on signed - in _p->_w - 1: SUCCESS\

lab.c function sum
[sum.overflow.2] line 10 arithmetic overflow on signed + in i + 1: SUCCESS
[sum.overflow.1] line 11 arithmetic overflow on signed + in s + u[(signed long int)i]: SUCCESS

lab.c function maxarray

[maxarray.overflow.1] line 22 arithmetic overflow on signed + in i + 1: SUCCESS

lab.c function main
[main.overflow.2] line 38 arithmetic overflow on signed + in i + 1: SUCCESS
[main.overflow.1] line 40 arithmetic overflow on signed - in -2147483647 - 1: SUCCESS

Exercício Final

```
#include <stdio.h>
#define MAX 10

int vec[MAX];

int fun (int n)
{
    int i, x=0;

    for (i=0; i < MAX; i++)
        if (vec[i]>n)
            x = x + vec[i]/(n-i);
        else x = x + vec[i]*(n-i);

    return x;
    }

    int cpdiff (int A[], int na, int B[], int nb, int x)
    {
    int i, j=0;

    for (i=0; i<na; i++)
        if (A[i]!=x) {
            B[j] = A[i];
            j++;
        }
    return j;
    }

    void main()
    {
    int j, t, y;
    int a[MAX], b[MAX];

    for (j=0; j<MAX; j++) {
        a[j] = j;
        vec[j] = j*30;
    }
    a[4] = 2;
    a[5] = 1;

    y = fun(55);
    t = cpdiff(a,MAX,b,MAX,1);
    }
```

- 1. Para verificar a segurança das funções fun, cpdiff e main, relativamente a acessos ilegais a posições do array, invoque:

```
cbmc exercicioCBMC.c -function fun -bounds-check -pointer-check
cbmc exercicioCBMC.c -function cpdiff -bounds-check -pointer-check
cbmc exercicioCBMC.c -bounds-check
```

O que pode concluir? Por que razão segunda invocação não pára, e as outras duas sim?

[RESPOSTA]

A razão pela qual a segunda invocação não para (não termina) é provavelmente devido a um problema de escalabilidade. A função cpdiff envolve operações complexas de manipulação de arrays e pode gerar um grande número de caminhos de execução possíveis. O CBMC, nesse caso, pode levar muito tempo para analisar todas as possibilidades e, eventualmente, exceder o tempo limite. No entanto, é importante observar que, mesmo que o CBMC não tenha concluído a

verificação, não significa necessariamente que não há violações de segurança. Pode ser útil considerar estratégias de otimização ou limitar o escopo da verificação para torná-la mais escalável.

- 2. Repita com a invocação com

```
cbmc exercicioCBMC.c -function cpdiff -bounds-check -pointer-check -unwind 10
```

Por que razão falha agora a verificação? Será que podemos evitar esta falha? O que podemos concluir?

[RESPOSTA]

A verificação falha neste caso porque o limite de desenrolamento de 10 não é suficiente para alcançar todas as possíveis execuções do loop. O CBMC não é capaz de analisar todos os caminhos de execução relevantes e, portanto, não pode fornecer uma resposta definitiva sobre a segurança da função.

Para evitar essa falha, poderíamos aumentar o limite de desenrolamento (-unwind) para um valor maior, permitindo ao CBMC analisar um número maior de iterações do loop. No entanto, isso pode não ser uma solução prática, pois o tempo e os recursos necessários para analisar um número muito grande de iterações podem se tornar proibitivos.

- 4. Insira código no final da função main de modo a fazer uma atribuição não determinista ao vector vec e ao argumento de invocação da função fun, e teste esta invocação quanto a eventuais problemas de “safety”.

Indique as invocações que fez ao CBMC e as conclusões a que chegou.

Indique os contra-exemplos (i.e., situações onde há falha) que o CBMC encontrou, caso haja.

```
int nondet_int();

int main() {
  int j, t, y;
  int a[MAX], b[MAX];

  for (j = 0; j < MAX; j++) {
    a[j] = j;
    vec[j] = j * 30;
  }
  a[4] = 2;
  a[5] = 1;

  // Atribuição não determinística
  int n = nondet_int();
  __CPROVER_assume(n >= INT_MIN && n <= INT_MAX);
  vec[0] = nondet_int();

  y = fun(n);
  t = cpdiff(a, MAX, b, MAX, 1);

  return 0;
}
```

Verificação de estouro de inteiros (overflow):

```
cbmc exercicioCBMC.c -overflow-check
```

Verificação abrangente de todos os problemas de "safety":

```
cbmc exercicioCBMC.c -bounds-check -pointer-check -overflow-check
```