PUCRS | ESCOLA POLITÉCNICA

98G08-04

CONFIABILIDADE E SEGURANÇA DE SOFTWARE

Ferramentas de Análise de Código
(Estáticas e Dinâmicas)

Prof. Iaçanã Ianiski Weber

MARISTA

Baseado no material gentilmente cedido pelo Prof. ***Dr. Rafael Garibotti***

GCOV – Cobertura de Código

- ✓ **Gcov** é uma ferramenta de análise de cobertura de código fonte e de criação de perfil por instruções passo-a-passo.
  - **Gcov** gera contagens exatas do número de vezes que cada instrução em um programa é executada e anota o código-fonte para adicionar instrumentação.
  - **Gcov** vem como um utilitário padrão com o pacote GNU Compiler Collection (*GCC*).
  - Tutorial disponível em: https://www.tutorialspoint.com/unix_commands/gcov.htm

Analisem o tutorial disponível e verifiquem o que os seguintes comandos fazem no projeto base:

- **make cov**
- **./cov**
- **gcov -b cov-identifier.gcda**

CPPCHECK

- ✓ **Cppcheck** é uma *ferramenta de análise estática* de código para aplicações escritas em C/C++
- ✓ O objetivo principal da ferramenta é encontrar erros que normalmente não são identificados pelo compilador, incluindo:
  - Dereferenciamento de um ponteiro não inicializado
  - Acesso fora dos limites de um vetor
  - Utilização de variáveis não inicializadas
  - Vazamento de recursos, isto é, arquivos abertos mas não fechados, memória alocada mas não desalocada, etc.
  - Avisos sobre código não utilizado ou duplicado

EXEMPLO DE USO DO CPPCHECK

1. Você escreve uma aplicação em C (por exemplo, *file1.c*):

```
int main()
{
    char a[10];
    a[10] = 0;
    return 0;
}
```

2. Você executa a ferramenta sobre o código em C, sem precisar compilar a aplicação:

```
cppcheck file1.c
```

3. Então, a ferramenta gera a seguinte saída:

```
Checking file1.c...
[file1.c:4]: (error) Array 'a[10]' index 10 out of bounds
```

Obs.: Está disponível no Material de Apoio um Tutorial sobre o **cppcheck**.

VALGRIND

- ✓ **Valgrind** é um *conjunto de ferramentas (Tool Suite) para análise dinâmica* de código.
- ✓ Exemplos de ferramentas da **valgrind**:
  - *Memcheck*
  - *Cachegrind*
  - *Callgrind*
  - *Massif*
  - *Helgrind*
- ✓ *Memcheck* é a ferramenta mais utilizada, e serve, principalmente, para resolver dois problemas em seus programas:
  - Vazamento de memória
  - Acesso a posições inválidas de memória (o que pode levar a *segmentation fault*)

Link: https://valgrind.org/

FERRAMENTAS ENCONTRADAS NO VALGRIND

- ✓ ***Memcheck*** detecta problemas de gerenciamento de memória e é voltado principalmente para programas C e C++. Parâmetro: *--leak-check=full*.
- ✓ ***Cachegrind*** é um *cache profiler.* Ele executa uma simulação detalhada das caches I1, D1 e L2 em sua CPU e, portanto, pode localizar com precisão as fontes de falhas de cache em seu código. Parâmetro: *--tool=cachegrind*.
- ✓ ***Callgrind*** é uma extensão do *Cachegrind*, e fornece, por exemplo, a quantidade de vezes que uma função é chamada. Dessa forma, conseguimos analisar o gargalo de nosso código e conseguimos otimiza-lo. Parâmetro: *--tool=callgrind*.
- ✓ ***Massif*** é um *heap profiler*. Ele produz um gráfico que mostra o uso de heap ao longo do tempo, incluindo informações sobre quais partes do programa são responsáveis pela maioria das alocações de memória. Parâmetro: *--tool=massif*.
- ✓ ***Helgrind*** é um *thread debugger* utilizado para encontrar *data race conditions* em programas *multithread*. Parâmetro: *--tool=helgrind*.

EXEMPLO DE USO DO VALGRIND

1. Considere o seguinte programa que lê 10 números e depois imprime-os na ordem inversa.

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

int main(int argc, char const *argv[]) {
  int i, *vetor = malloc(10 * sizeof(int));
  for (i = 0; i < 10; i++)
    scanf("%d", &vetor[i]);
  for (i = 9; i >= 0; i--)
    printf("%d\n", vetor[i]);
  return 0;
}
```

2. Esse programa tem um vazamento de memória: o vetor alocado não é desalocado antes do programa terminar. No terminal, execute:

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

3. Temos algumas mensagens padrões no começo e depois o Valgrind avisa que houveram bytes perdidos na seções HEAP SUMMARY e LEAK SUMMARY.

```
==1918== Memcheck, a memory error detector
==1918== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==1918== Using Valgrind-3.12.0.SVN and LibVEX; rerun with -h for copyright info
==1918== Command: ./programa
==1918==

...

==1918==
==1918== HEAP SUMMARY:
==1918==     in use at exit: 40 bytes in 1 blocks
==1918==   total heap usage: 3 allocs, 2 frees, 5,160 bytes allocated
==1918==
==1918== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==1918==    at 0x4C2BBAF: malloc (vg_replace_malloc.c:299)
==1918==    by 0x108758: main (programa.c:5)
==1918==
==1918== LEAK SUMMARY:
==1918==    definitely lost: 40 bytes in 1 blocks
==1918==    indirectly lost: 0 bytes in 0 blocks
==1918==      possibly lost: 0 bytes in 0 blocks
==1918==    still reachable: 0 bytes in 0 blocks
==1918==         suppressed: 0 bytes in 0 blocks
==1918==
==1918== For counts of detected and suppressed errors, rerun with: -v
==1918== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

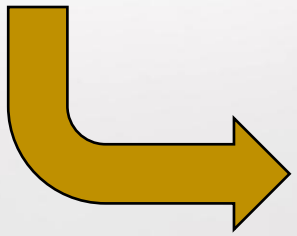
MEMORY / ADDRESS SANITIZER

- ✓ **Memory Sanitizer** é um detector de leituras de memória não inicializadas em aplicações escritas em C/C++
  - Ele consiste em um módulo de instrumentação do compilador e uma biblioteca de tempo de execução
  - Mais informações em: https://clang.llvm.org/docs/MemorySanitizer.html
- ✓ **Address Sanitizer** é uma ferramenta de programação de código aberto do Google que detecta *bugs* de corrupção de memória, como por exemplo, estouro de buffer ou acessos a um ponteiro desalocado (*use-after-free*)
  - Ela é baseada na instrumentação do compilador e é mapeada diretamente na *shadow memory* (uma técnica usada para rastrear e armazenar informações na memória do computador usada por um programa durante sua execução)
  - Mais informações em: https://en.wikipedia.org/wiki/AddressSanitizer

EXEMPLO DE USO DO ADDRESS SANITIZER

1. Considere o seguinte programa e compile-o utilizando a flag: **-fsanitize=address**

```
1 // To compile: g++ -O -g -fsanitize=address heap-use-after-free.cc
2 int main(int argc, char **argv) {
3   int *array = new int[100];
4   delete [] array;
5   return array[argc];  // BOOM
6 }
```



```
$ ./a.out
==5587==ERROR: AddressSanitizer: heap-use-after-free on address 0x61400000fe44 at pc 0x47b55f bp 0x7ffc36b28200 sp 0x7ffc36b281f8
READ of size 4 at 0x61400000fe44 thread T0
    #0 0x47b55e in main /home/test/example_UseAfterFree.cc:7
    #1 0x7f15cfe71b14 in __libc_start_main (/lib64/libc.so.6+0x21b14)
    #2 0x47b44c in _start (/root/a.out+0x47b44c)

0x61400000fe44 is located 4 bytes inside of 400-byte region [0x61400000fe40,0x61400000ffd0)
freed by thread T0 here:
    #0 0x465da9 in operator delete[](void*) (/root/a.out+0x465da9)
    #1 0x47b529 in main /home/test/example_UseAfterFree.cc:6

previously allocated by thread T0 here:
    #0 0x465aa9 in operator new[](unsigned long) (/root/a.out+0x465aa9)
    #1 0x47b51e in main /home/test/example_UseAfterFree.cc:5

SUMMARY: AddressSanitizer: heap-use-after-free /home/test/example_UseAfterFree.cc:7 main
Shadow bytes around the buggy address:
  0x0c287fff9f70: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c287fff9f80: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c287fff9f90: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c287fff9fa0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c287fff9fb0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
=>0x0c287fff9fc0: fa fa fa fa fa fa fa fa[fd]fd fd fd fd fd fd fd
  0x0c287fff9fd0: fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd
  0x0c287fff9fe0: fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd
  0x0c287fff9ff0: fd fd fd fd fd fd fd fd fd fd fa fa fa fa fa fa
  0x0c287fffa000: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c287fffa010: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:     fa
  Heap right redzone:    fb
  Freed heap region:     fd
  Stack left redzone:    f1
  Stack mid redzone:     f2
  Stack right redzone:   f3
  Stack partial redzone: f4
  Stack after return:    f5
  Stack use after scope: f8
  Global redzone:        f9
  Global init order:     f6
  Poisoned by user:      f7
  ASan internal:         fe
==5587==ABORTING
```

2. A saída do programa dando uma explicação mais detalhada da falha encontrada é mostrada ao lado

INFORMAÇÕES PARA INSTALAÇÃO NO LINUX

- ✓ **cppcheck**
  - **sudo apt-get install cppcheck**
- ✓ **Valgrind**
  - **sudo apt-get install valgrind**
- ✓ **Memory / Address Sanitizer**
  - **gcc/clang** “**-fsanitize=address**” vem junto com o compilador

EXEMPLOS

EXEMPLO Nº 1

Considere o seguinte programa:

```c
void calc(void) {
  int buf[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
  int result;
  int i;

  for (i = 0; i <= 10; i++) {
    result += buf[i];
  }
}

int Static[5];

int func(void) {
  int Stack[5];
  Static[5] = 0;
  Stack [5] = 0;
  return 0;
}

int main(void) {
  calc();
  func();
  return 0;
}
```

1. Compilando com **gcc**:

```
iacanaw@DESKTOP-0IJM1VP:~/workspace/tcs/aula6/Codigos_de_Aula$ gcc test.c
iacanaw@DESKTOP-0IJM1VP:~/workspace/tcs/aula6/Codigos_de_Aula$
```

Nenhuma mensagem de ***warning*** ou ***error***!

EXEMPLO Nº 1

Considere o seguinte programa:

```c
void calc(void) {
  int buf[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
  int result;
  int i;

  for (i = 0; i <= 10; i++) {
    result += buf[i];
  }
}

int Static[5];

int func(void) {
  int Stack[5];
  Static[5] = 0;
  Stack [5] = 0;
  return 0;
}

int main(void) {
  calc();
  func();
  return 0;
}
```

Note que o **cppcheck** não precisa nem compilar o código para realizar a análise, por isso é considerado um analisador estático!

Neste exemplo, temos indicativos do que melhorar usando uma **ferramenta de análise estática**!

2. Analisando com o **cppcheck**:

```
iacanaw@DESKTOP-0IJM1VP:~/workspace/tcs/aula6/Codigos_de_Aula$ cppcheck --enable=all test.c
Checking test.c ...
test.c:7:18: error: Array 'buf[10]' accessed at index 10, which is out of bounds. [arrayIndexOutOfBounds]
    result += buf[i];
                  ^
test.c:6:17: note: Assuming that condition 'i<=10' is not redundant
  for (i = 0; i <= 10; i++) {
                ^
test.c:7:18: note: Array index out of bounds
    result += buf[i];
                  ^
test.c:15:9: error: Array 'Static[5]' accessed at index 5, which is out of bounds. [arrayIndexOutOfBounds]
  Static[5] = 0;
        ^
test.c:16:9: error: Array 'Stack[5]' accessed at index 5, which is out of bounds. [arrayIndexOutOfBounds]
  Stack [5] = 0;
        ^
test.c:2:7: style: Variable 'buf' can be declared with const [constVariable]
  int buf[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
      ^
test.c:7:5: error: Uninitialized variable: result [uninitvar]
    result += buf[i];
    ^
test.c:7:12: style: Variable 'result' is assigned a value that is never used. [unreadVariable]
    result += buf[i];
           ^
test.c:16:13: style: Variable 'Stack[5]' is assigned a value that is never used. [unreadVariable]
  Stack [5] = 0;
            ^
```

EXEMPLO Nº 1

Considere o seguinte programa:

```
1  void calc(void) {
2    int buf[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
3    int result;
4    int i;
5
6    for (i = 0; i <= 10; i++) {
7      result += buf[i];
8    }
9  }
10
11 int Static[5];
12
13 int func(void) {
14   int Stack[5];
15   Static[5] = 0;
16   Stack [5] = 0;
17   return 0;
18 }
19
20 int main(void) {
21   calc();
22   func();
23   return 0;
24 }
```

Note que a **ferramenta de análise dinâmica, (valgrind)** não encontrou erros nesse exemplo.

3. Compilando com **gcc**, habilitando todas as mensagens de aviso do compilador:

```
iacanaw@DESKTOP-0IJM1VP:~/workspace/tcs/aula6/Codigos_de_Aula$ gcc -g -Wall -Wfatal-errors test.c -o test
test.c: In function ‘func’:
test.c:14:7: warning: variable ‘Stack’ set but not used [-Wunused-but-set-variable]
   14 |   int Stack[5];
      |       ^~~~~
```

4. Analisando com o **valgrind**:

```
iacanaw@DESKTOP-0IJM1VP:~/workspace/tcs/aula6/Codigos_de_Aula$ valgrind --leak-check=full
 --show-leak-kinds=all ./test
==1096== Memcheck, a memory error detector
==1096== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1096== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==1096== Command: ./test
==1096==
==1096==
==1096== HEAP SUMMARY:
==1096==     in use at exit: 0 bytes in 0 blocks
==1096==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==1096==
==1096== All heap blocks were freed -- no leaks are possible
==1096==
==1096== For lists of detected and suppressed errors, rerun with: -s
==1096== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

EXEMPLO Nº 1

Considere o seguinte programa:

```c
void calc(void) {
  int buf[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
  int result;
  int i;

  for (i = 0; i <= 10; i++) {
    result += buf[i];
  }
}

int Static[5];

int func(void) {
  int Stack[5];
  Static[5] = 0;
  Stack [5] = 0;
  return 0;
}

int main(void) {
  calc();
  func();
  return 0;
}
```

5. Analisando com o **Address Sanitizer**:

1) compilar usando a flag **`-fsanitize=address`**

```
iacanaw@DESKTOP-0IJM1VP:~/workspace/tcs/aula6/Codigos_de_Aula$ gcc -g -Wall -Wfatal-errors -fsanitize=address test.c -o test
```

EXEMPLO Nº 1

5. Analisando com o **Address Sanitizer**:

2) Executar

Relatório completo do **Address Sanitizer**

```
iacanaw@DESKTOP-0IJM1VP:~/workspace/tcs/aula6/Codigos_de_Aula$ ./test
=================================================================
==1526==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7ffcc67c5428 at pc 0x55
ecec9305aa bp 0x7ffcc67c53b0 sp 0x7ffcc67c53a0
READ of size 4 at 0x7ffcc67c5428 thread T0
    #0 0x55ecec9305a9 in calc /home/iacanaw/workspace/tcs/aula6/Codigos_de_Aula/test.c:7
    #1 0x55ecec9307c9 in main /home/iacanaw/workspace/tcs/aula6/Codigos_de_Aula/test.c:21
  #2 0x7f6f59948d8f in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58
#3 0x7f6f59948e3f in __libc_start_main_impl ../csu/libc-start.c:392
    #4 0x55ecec930164 in _start (/home/iacanaw/workspace/tcs/aula6/Codigos_de_Aula/test+0x1
164)

Address 0x7ffcc67c5428 is located in stack of thread T0 at offset 88 in frame
    #0 0x55ecec930238 in calc /home/iacanaw/workspace/tcs/aula6/Codigos_de_Aula/test.c:1

  This frame has 1 object(s):
    [48, 88) 'buf' (line 2) <== Memory access at offset 88 overflows this variable
HINT: this may be a false positive if your program uses some custom stack unwind mechanism,
 swapcontext or vfork
      (longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: stack-buffer-overflow /home/iacanaw/workspace/tcs/aula6/Codigos_
de_Aula/test.c:7 in calc
Shadow bytes around the buggy address:
  0x100018cf0a30: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100018cf0a40: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100018cf0a50: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100018cf0a60: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100018cf0a70: 00 00 00 00 00 00 00 00 00 00 f1 f1 f1 f1 f1 f1
=>0x100018cf0a80: 00 00 00 00 00[f3]f3 f3 f3 f3 00 00 00 00 00 00
  0x100018cf0a90: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100018cf0aa0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100018cf0ab0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100018cf0ac0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100018cf0ad0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
  Shadow gap:              cc
==1526==ABORTING
```

Note que a **ferramenta de análise dinâmica, (Address Sanitizer)** encontrou uma falha na linha 7 durante a execução do programa!

EXEMPLO Nº 1

Considere o seguinte programa:

```c
void calc(void) {
  int buf[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
  int result;
  int i;

  for (i = 0; i <= 10; i++) {
    result += buf[i];
  }
}

int Static[5];

int func(void) {
  int Stack[5];
  Static[5] = 0;
  Stack [5] = 0;
  return 0;
}

int main(void) {
  calc();
  func();
  return 0;
}
```

Sequência de Comandos:

GCC

- `gcc test.c`

Cppcheck

- `cppcheck --enable=all --suppress=missingIncludeSystem test.c`

Valgrind

- `gcc -g -Wall -Wfatal-errors test.c -o test`
- `valgrind --leak-check=full --show-leak-kinds=all ./test`

Address Sanitizer

- `gcc -g -Wall -Wfatal-errors -fsanitize=address test.c -o test`
- `./test`

EXEMPLO Nº 2

Considere o seguinte programa:

```
int main(int argc, char **argv) {
  int *array = new int[100];
  delete [] array;
  return array[argc]; // Buggy
}
```

1. Compilando com **g++**:

```
iacanaw@DESKTOP-0IJM1VP:~/workspace/tcs/aula6/Codigos_de_Aula$ g++ buggy.cpp -o buggy
iacanaw@DESKTOP-0IJM1VP:~/workspace/tcs/aula6/Codigos_de_Aula$
```

Nenhum erro encontrado.

2. Analisando com o **cppcheck**:

```
iacanaw@DESKTOP-0IJM1VP:~/workspace/tcs/aula6/Codigos_de_Aula$ cppcheck --enable=all buggy.cpp
Checking buggy.cpp ...
buggy.cpp:3:13: error: Memory is allocated but not initialized: array [uninitdata]
  delete [] array;
            ^
```

EXEMPLO Nº 2

```
int main(int argc, char **argv) {
  int *array = new int[100];
  delete [] array;
  return array[argc]; // Buggy
}
```

3. Analisando com o **valgrind**:

Encontrou o erro de leitura inválida na linha 4.

```
iacanaw@DESKTOP-0IJM1VP:~/workspace/tcs/aula6/Codigos_de_Aula$ g++ -g -Wall -Wfatal-errors buggy.cpp -o buggy
iacanaw@DESKTOP-0IJM1VP:~/workspace/tcs/aula6/Codigos_de_Aula$ valgrind --leak-check=full --show-leak-kinds=all ./buggy
==3608== Memcheck, a memory error detector
==3608== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3608== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==3608== Command: ./buggy
==3608==
==3608== Invalid read of size 4
==3608==    at 0x1091B1: main (buggy.cpp:4)
==3608==  Address 0x4dd4c84 is 4 bytes inside a block of size 400 free'd
==3608==    at 0x484CA8F: operator delete[](void*) (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==3608==    by 0x10919C: main (buggy.cpp:3)
==3608==  Block was alloc'd at
==3608==    at 0x484A2F3: operator new[](unsigned long) (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==3608==    by 0x109185: main (buggy.cpp:2)
==3608==
==3608==
==3608== HEAP SUMMARY:
==3608==     in use at exit: 0 bytes in 0 blocks
==3608==   total heap usage: 2 allocs, 2 frees, 73,104 bytes allocated
==3608==
==3608== All heap blocks were freed -- no leaks are possible
==3608==
==3608== For lists of detected and suppressed errors, rerun with: -s
```

EXEMPLO Nº 2

Considere o seguinte programa:

```
int main(int argc, char **argv) {
  int *array = new int[100];
  delete [] array;
  return array[argc]; // Buggy
}
```

4. Analisando com o **Address Sanitizer**:

Encontrou o erro de leitura inválida na linha 4.

```
iacanaw@DESKTOP-0IJM1VP:~/workspace/tcs/aula6/Codigos_de_Aula$ g++ -g -Wall -Wfatal-errors -fsanitize=address buggy.cpp -o buggy
iacanaw@DESKTOP-0IJM1VP:~/workspace/tcs/aula6/Codigos_de_Aula$ ./buggy
=================================================================
==3929==ERROR: AddressSanitizer: heap-use-after-free on address 0x614000000044 at pc 0x563a759c7249 bp 0x7ffc8f2b3c70 sp 0x7ffc8f2b3c60
READ of size 4 at 0x614000000044 thread T0
    #0 0x563a759c7248 in main /home/iacanaw/workspace/tcs/aula6/Codigos_de_Aula/buggy.cpp:4
    #1 0x7f37fe9b2d8f in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58
    #2 0x7f37fe9b2e3f in __libc_start_main_impl ../csu/libc-start.c:392
    #3 0x563a759c7104 in _start (/home/iacanaw/workspace/tcs/aula6/Codigos_de_Aula/buggy+0x1104)

0x614000000044 is located 4 bytes inside of 400-byte region [0x614000000040,0x6140000001d0)
freed by thread T0 here:
    #0 0x7f37fef98e37 in operator delete[](void*) ../../../../src/libsanitizer/asan/asan_new_delete.cpp:163
    #1 0x563a759c71fc in main /home/iacanaw/workspace/tcs/aula6/Codigos_de_Aula/buggy.cpp:3
    #2 0x7f37fe9b2d8f in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58

previously allocated by thread T0 here:
    #0 0x7f37fef98337 in operator new[](unsigned long) ../../../../src/libsanitizer/asan/asan_new_delete.cpp:102
    #1 0x563a759c71e5 in main /home/iacanaw/workspace/tcs/aula6/Codigos_de_Aula/buggy.cpp:2
    #2 0x7f37fe9b2d8f in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58

SUMMARY: AddressSanitizer: heap-use-after-free /home/iacanaw/workspace/tcs/aula6/Codigos_de_Aula/buggy.cpp:4 in main
Shadow bytes around the buggy address:
```

EXEMPLO Nº 3

Considere o seguinte programa:

```
int main(int argc, char **argv) {
  int *array = new int[100];
  array[0] = 0;
  int res = array[argc + 100]; // BOOM
  delete [] array;
  return res;
}
```

1. Compilando com **g++**:

```
iacanaw@DESKTOP-0IJM1VP:~/workspace/tcs/aula6/Codigos_de_Aula$ g++ buggy2.cpp -o buggy2
iacanaw@DESKTOP-0IJM1VP:~/workspace/tcs/aula6/Codigos_de_Aula$
```

Nenhum erro encontrado.

2. Analisando com o **cppcheck**:

```
iacanaw@DESKTOP-0IJM1VP:~/workspace/tcs/aula6/Codigos_de_Aula$ cppcheck --enable=all buggy2.cpp
Checking buggy2.cpp ...
```

Nenhum erro encontrado.

EXEMPLO Nº 3

```
int main(int argc, char **argv) {
  int *array = new int[100];
  array[0] = 0;
  int res = array[argc + 100]; // BOOM
  delete [] array;
  return res;
}
```

3. Analisando com o **valgrind**:

```
iacanaw@DESKTOP-0IJM1VP:~/workspace/tcs/aula6/Codigos_de_Aula$ g++ -g -Wall -Wfatal-errors buggy2.cpp
 -o buggy2
iacanaw@DESKTOP-0IJM1VP:~/workspace/tcs/aula6/Codigos_de_Aula$ valgrind --leak-check=full --show-leak
-kinds=all ./buggy2
==4664== Memcheck, a memory error detector
==4664== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4664== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==4664== Command: ./buggy2
==4664==
==4664== Invalid read of size 4
==4664==    at 0x1091AC: main (buggy2.cpp:4)
==4664==  Address 0x4dd4e14 is 4 bytes after a block of size 400 alloc'd
==4664==    at 0x484A2F3: operator new[](unsigned long) (in /usr/libexec/valgrind/vgpreload_memcheck-
amd64-linux.so)
==4664==    by 0x109185: main (buggy2.cpp:2)
==4664==
==4664==
==4664== HEAP SUMMARY:
==4664==     in use at exit: 0 bytes in 0 blocks
==4664==   total heap usage: 2 allocs, 2 frees, 73,104 bytes allocated
==4664==
==4664== All heap blocks were freed -- no leaks are possible
==4664==
==4664== For lists of detected and suppressed errors, rerun with: -s
==4664== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

EXEMPLO Nº 3

```
int main(int argc, char **argv) {
  int *array = new int[100];
  array[0] = 0;
  int res = array[argc + 100]; // BOOM
  delete [] array;
  return res;
}
```

4. Analisando com o **Address Sanitizer**:

```
iacanaw@DESKTOP-0IJM1VP:~/workspace/tcs/aula6/Codigos_de_Aula$ g++ -g -Wall -Wfatal-errors -fsanitize=address buggy2.cpp -o buggy2
iacanaw@DESKTOP-0IJM1VP:~/workspace/tcs/aula6/Codigos_de_Aula$ ./buggy2
=================================================================
==5044==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x6140000001d4 at pc 0x5607ee46d29b bp 0x7ffd1cdc7300 sp 0x7ffd1cdc72f0
READ of size 4 at 0x6140000001d4 thread T0
    #0 0x5607ee46d29a in main /home/iacanaw/workspace/tcs/aula6/Codigos_de_Aula/buggy2.cpp:4
    #1 0x7f4cd46b5d8f in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58
    #2 0x7f4cd46b5e3f in __libc_start_main_impl ../csu/libc-start.c:392
    #3 0x5607ee46d124 in _start (/home/iacanaw/workspace/tcs/aula6/Codigos_de_Aula/buggy2+0x1124)

0x6140000001d4 is located 4 bytes to the right of 400-byte region [0x614000000040,0x6140000001d0)
allocated by thread T0 here:
    #0 0x7f4cd4c9b337 in operator new[](unsigned long) ../../../../src/libsanitizer/asan/asan_new_delete.cpp:102
    #1 0x5607ee46d205 in main /home/iacanaw/workspace/tcs/aula6/Codigos_de_Aula/buggy2.cpp:2
    #2 0x7f4cd46b5d8f in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58
```

- ✓ **GitHub** é uma plataforma de hospedagem de código-fonte e arquivos com controle de versão usando o **git**. Ele permite que programadores, utilitários ou qualquer usuário cadastrado na plataforma contribuam em projetos privados e/ou Open Source de qualquer lugar do mundo

1. Criem um repositório para vocês baseado no projeto base do GitHub disponível em:
   - https://github.com/iacanaw/identifier

2. Façam um **clone** desse repositório e analisem seus arquivos.

```
git clone https://github.com/iacanaw/identifier
```

INTEGRAÇÃO CONTÍNUA

- ✓ Ferramentas de integração contínua (*Continuous Integration*) são usadas para construir e testar projetos de software hospedados no *GitHub* e *Bitbucket*
  - Integração Contínua é uma metodologia em que os desenvolvedores envolvidos em um projeto integram seus trabalhos continuamente. Cada integração é consolidada por uma ferramenta chamada de automatização de tarefas, que inclusive pode executar diversos testes para identificar erros de digitação de códigos, incompatibilidades entre comandos dados por desenvolvedores diferentes, etc.
  - Sobre o Github Actions: https://docs.github.com/pt/actions/about-github-actions/understanding-github-actions

- O arquivo .yml do projeto criado no GitHub precisa estar adaptado ao Makefile de vocês

.github > workflows > main.yml

```yaml
name: C/C++ CI

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v4
    - name: make
      run: "make"
```

Makefile

```
GCCFLAGS = -g -Wall -Wfatal-errors
ALL = identifier
GCC = gcc

all: $(ALL)

identifier: identifier.c
    $(GCC) $(GCCFLAGS) -o $@ $@.c

clean:
    rm -fr $(ALL) *.o cov* *.dSYM *.gcda *.gcno *.gcov

```

TRABALHO 1

Integração com o Projeto Base:

- Façam o **clone** do projeto do Unity disponível no link abaixo, e então façam a integração com o ambiente de projeto criado neste material, este será a base para o *Trabalho sobre Teste de Software*

  ✓ https://github.com/iacanaw/Unity

- Deve-se adicionar ao repositório um dos três problemas clássicos que foi desenvolvido por vocês (vide slide 03 - Teste de Software).

- Vocês devem fazer a integração das ferramentas **cppcheck**, **valgrind** e **sanitizer** no ambiente de projeto criado no item anterior, a fim de terem um ambiente completo.

Criação dos Testes:

- Além da integração com as ferramentas, vocês devem criar um conjunto de testes com base nos:
  1. **Caixa Preta -** Testes Funcionais (Metodologia Sistemática) que junta as classes de equivalência com a análise do valor limite.
  2. **Caixa Branca -** Testes Estruturais baseado no grafo de controle de fluxo, onde vocês devem realizar testes que cubram todos os nodos e todas as arestas.

Relatório de Verificação:

- Ao final vocês devem entregar um relatório contendo o *report* da verificação proposta por vocês.
  1. *Indique as classes de equivalência*
  2. *Indique os valores limites testados*
  3. *Indique o grafo de controle de fluxo*

- Nesse relatório, você também terá a oportunidade de relatar o funcionamento da integração contínua, demonstrando uma situação onde erros são encontrados e reportados pela ferramenta.

Entrega do Trabalho 1:

- **Via Moodle**
- **Deadline:** 23/04/2025 – 17:30
- Na aula do dia 23/04/2025 uma breve apresentação deverá ser realizada – demonstração da *continuous integration*
- **Não serão aceitas entregas atrasadas**