Lab 6

Software Testing 2023

2023/04/20

#whoami

- Software Quality Lab @ EC547
- TA
  1. 蔡惠喬
     - hctsai.cs10@nycu.edu.tw
  2. 陳舜寧
     - xdev11.cs11@nycu.edu.tw

GitHub Repo

- <student_id>-ST-2023
- Add collaborators
  - XDEv11, chameleon10712, skhuang

Program Security Detect

Program Security Detect

- Valgrind
- ASAN

Valgrind

Valgrind

- An open source system memory debugger
- Simple and easy to use
    - does not require re-compilation and re-linking
- Used to validate many large Projects
- Language suport
    - C/C++, Python, Java, Javascript

Valgrind

- 在 user space 層級對程式進行動態分析的框架
- 有多種工具能追蹤和分析程式效能
- EX: 偵測記憶體錯誤
  - 未初始化的記憶體
  - 不當的記憶體配置
  - 記憶體越界存取
- 注意 : 使用 Valgrind 會讓程式執行速度比平常更慢

Valgrind 原理

- 透過 動態重新編譯（dynamic binary re-compilation）的方法把測試程式的 machine code 轉成 IR (VEX intermediate representation)

- 如果發生有興趣的事件執行 (例如：記憶體配置), 就會使用對應的工具對 IR 加入一些分析程式碼, 再轉成 machine code 存到 code cache 中

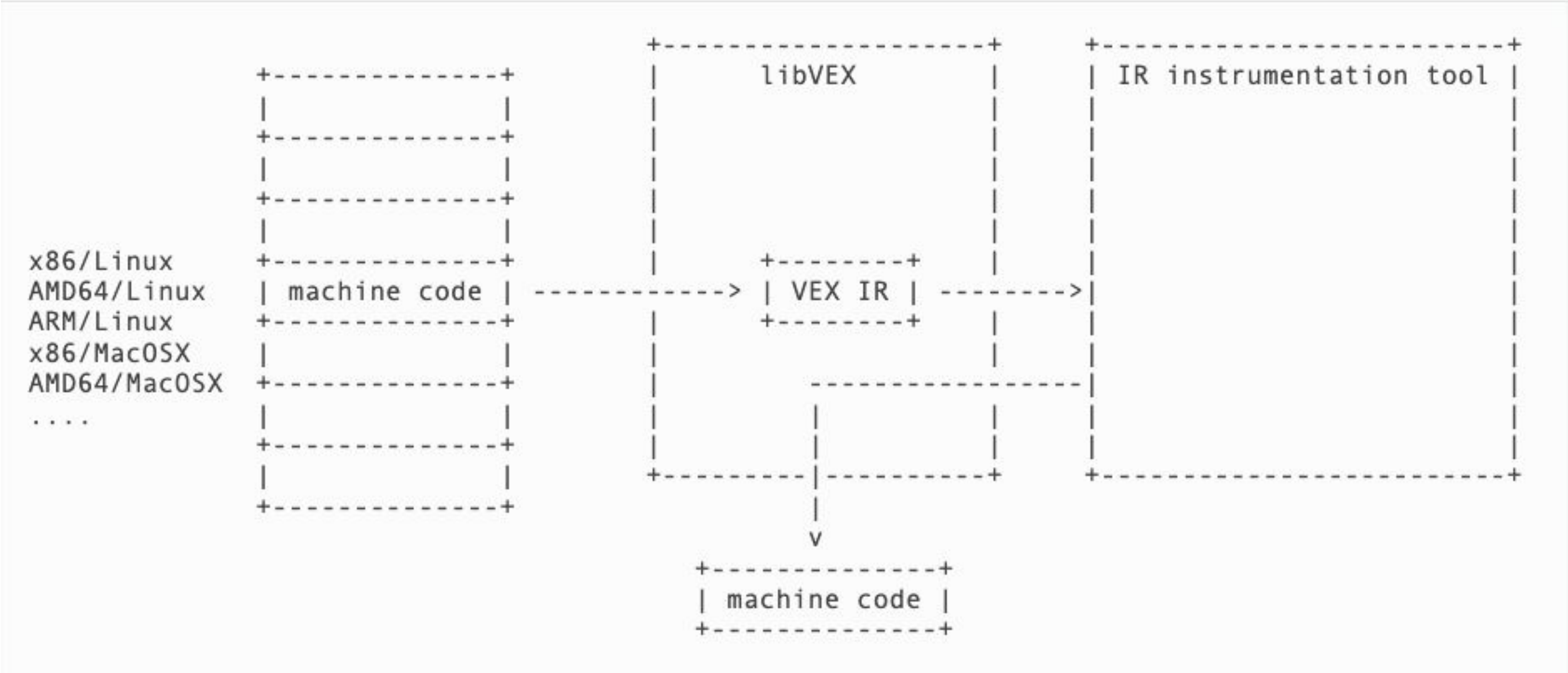
- 簡單的來說
    - Valgrind 執行的都是他們所加工過後的程式

動態分析 Dynamic Binary Instrumentation

- Valgrind 透過 shadow values 技術來實作

- 對所有的 register 和使用到的 memory 做 shadow（自行維護的副本）
  - shadow State
  - shadow registers
  - shadow memory
  - read / write

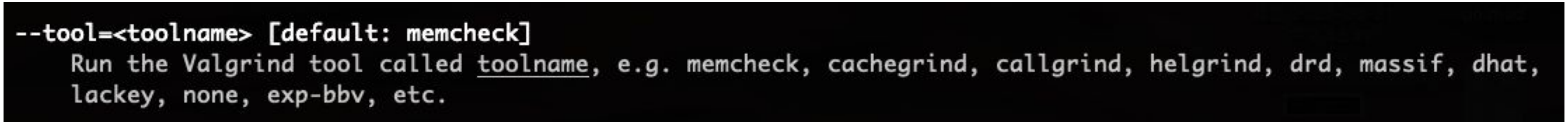
Dynamic Binary Instrumentation



```
                                     +--------------------+     +-------------------------+
              +--------------+       |      libVEX        |     | IR instrumentation tool |
              |              |       |                    |     |                         |
              +--------------+       |                    |     |                         |
              |              |       |                    |     |                         |
              +--------------+       |                    |     |                         |
              |              |       |                    |     |                         |
x86/Linux     +--------------+       |      +--------+    |     |                         |
AMD64/Linux   | machine code | -----------> | VEX IR | -------->|                         |
ARM/Linux     +--------------+       |      +--------+    |     |                         |
x86/MacOSX    |              |       |                    |     |                         |
AMD64/MacOSX  +--------------+       |         -----------------|                         |
....          |              |       |         |          |     |                         |
              +--------------+       |         |          |     |                         |
              |              |       +---------|----------+     +-------------------------+
              +--------------+                 |
                                               v
                                        +--------------+
                                        | machine code |
                                        +--------------+
```

Valgrind

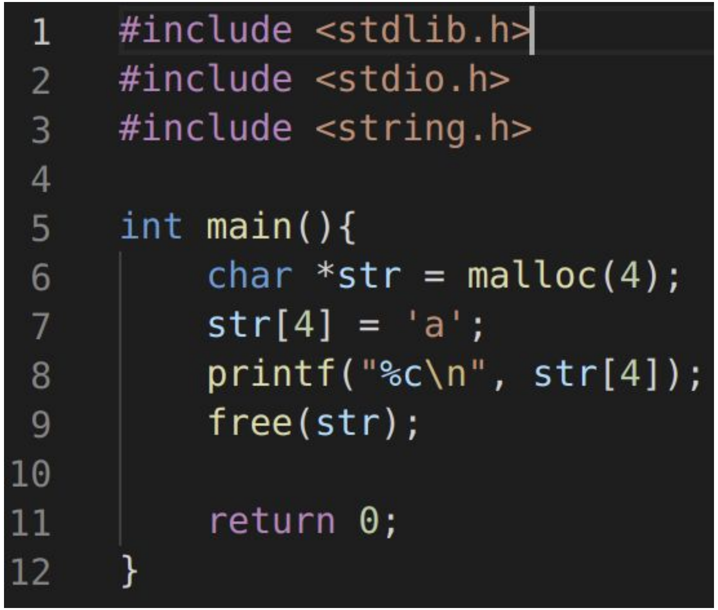
- $ valgrind --tool=<toolname> <program>



```
--tool=<toolname> [default: memcheck]
    Run the Valgrind tool called toolname, e.g. memcheck, cachegrind, callgrind, helgrind, drd, massif, dhat,
    lackey, none, exp-bbv, etc.
```

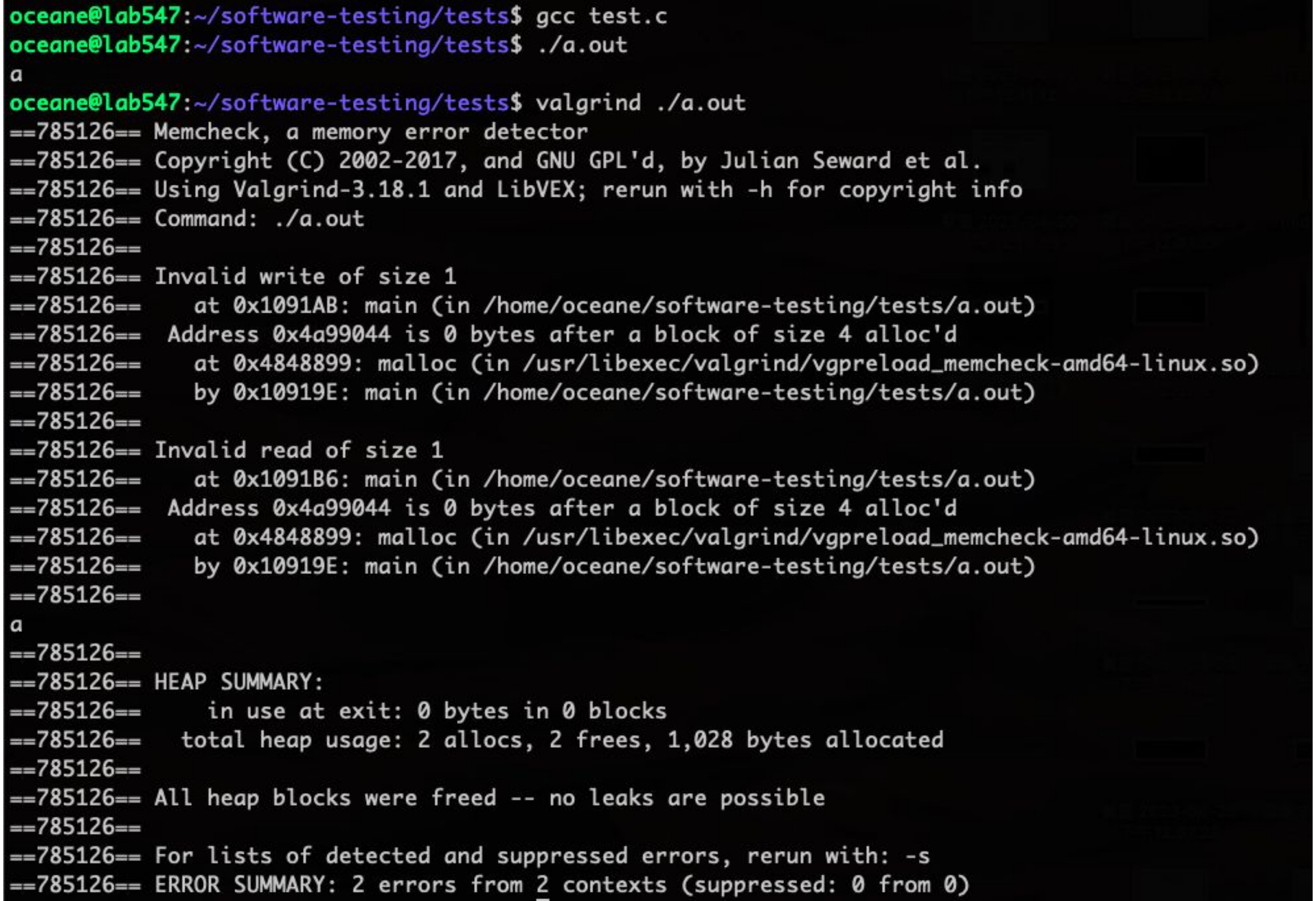
Valgrind

- sudo apt install valgrind

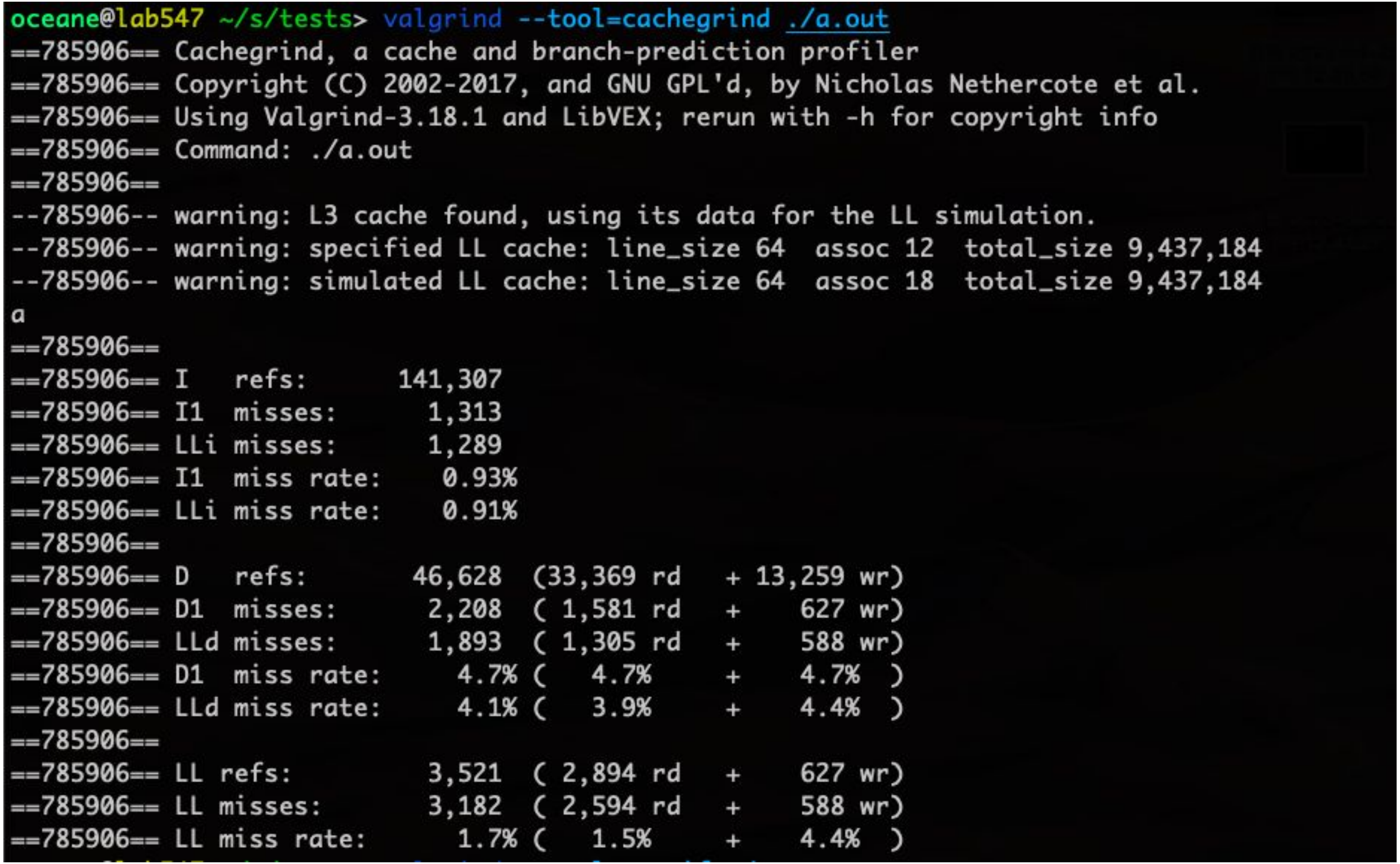


```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(){
    char *str = malloc(4);
    str[4] = 'a';
    printf("%c\n", str[4]);
    free(str);

    return 0;
}
```

test.c



```
oceane@lab547:~/software-testing/tests$ gcc test.c
oceane@lab547:~/software-testing/tests$ ./a.out
a
oceane@lab547:~/software-testing/tests$ valgrind ./a.out
==785126== Memcheck, a memory error detector
==785126== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==785126== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==785126== Command: ./a.out
==785126==
==785126== Invalid write of size 1
==785126==    at 0x1091AB: main (in /home/oceane/software-testing/tests/a.out)
==785126==  Address 0x4a99044 is 0 bytes after a block of size 4 alloc'd
==785126==    at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==785126==    by 0x10919E: main (in /home/oceane/software-testing/tests/a.out)
==785126==
==785126== Invalid read of size 1
==785126==    at 0x1091B6: main (in /home/oceane/software-testing/tests/a.out)
==785126==  Address 0x4a99044 is 0 bytes after a block of size 4 alloc'd
==785126==    at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==785126==    by 0x10919E: main (in /home/oceane/software-testing/tests/a.out)
==785126==
a
==785126==
==785126== HEAP SUMMARY:
==785126==     in use at exit: 0 bytes in 0 blocks
==785126==   total heap usage: 2 allocs, 2 frees, 1,028 bytes allocated
==785126==
==785126== All heap blocks were freed -- no leaks are possible
==785126==
==785126== For lists of detected and suppressed errors, rerun with: -s
==785126== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```



```
oceane@lab547 ~/s/tests> valgrind --tool=cachegrind ./a.out
==785906== Cachegrind, a cache and branch-prediction profiler
==785906== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==785906== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==785906== Command: ./a.out
==785906==
--785906-- warning: L3 cache found, using its data for the LL simulation.
--785906-- warning: specified LL cache: line_size 64  assoc 12  total_size 9,437,184
--785906-- warning: simulated LL cache: line_size 64  assoc 18  total_size 9,437,184
a
==785906==
==785906== I   refs:      141,307
==785906== I1  misses:      1,313
==785906== LLi misses:      1,289
==785906== I1  miss rate:    0.93%
==785906== LLi miss rate:    0.91%
==785906==
==785906== D   refs:       46,628  (33,369 rd   + 13,259 wr)
==785906== D1  misses:      2,208  ( 1,581 rd   +    627 wr)
==785906== LLd misses:      1,893  ( 1,305 rd   +    588 wr)
==785906== D1  miss rate:     4.7% (   4.7%     +    4.7%  )
==785906== LLd miss rate:     4.1% (   3.9%     +    4.4%  )
==785906==
==785906== LL refs:         3,521  ( 2,894 rd   +    627 wr)
==785906== LL misses:       3,182  ( 2,594 rd   +    588 wr)
==785906== LL miss rate:      1.7% (   1.5%     +    4.4%  )
```

Sanitizers

Sanitizers

- 以下是常見的 Sanitizers

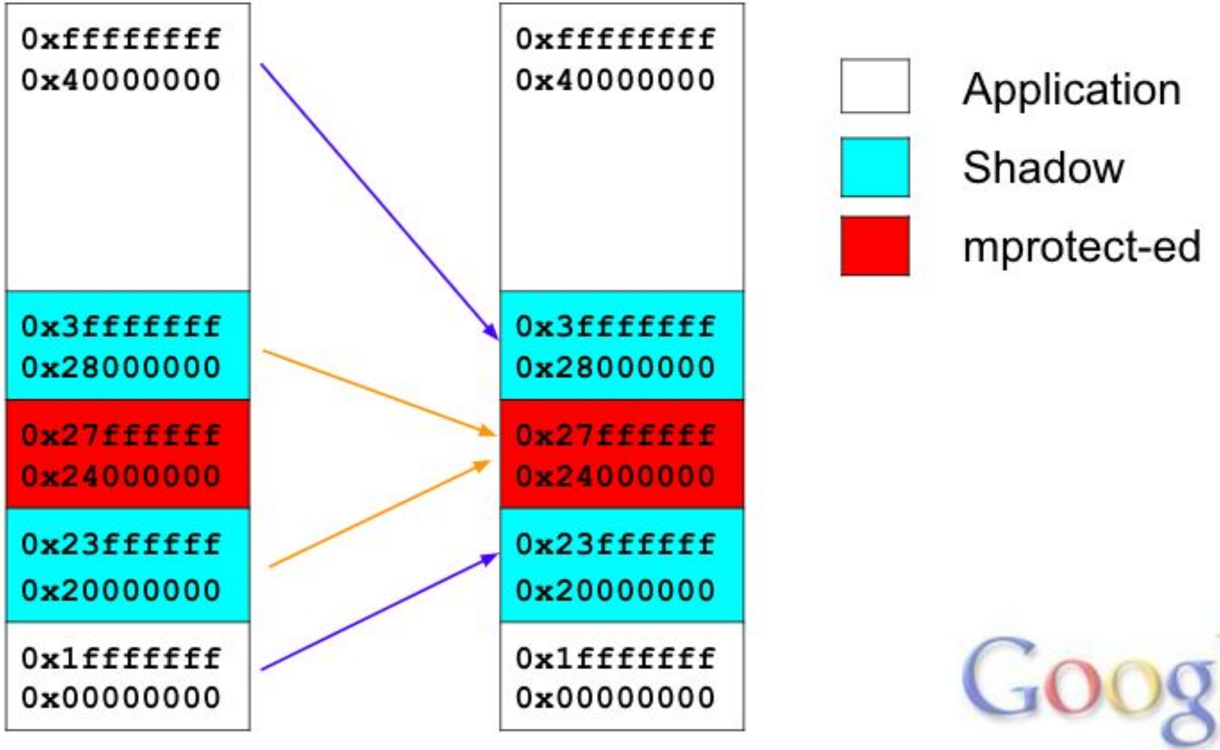
AddressSanitizer	檢查記憶體存取
LeakSanitizer	檢查 memory leak
ThreadSanitizer	檢查 deadlocks，race condition
MemorySanitizer	檢查未初始化的問題
UndefinedBehaviorSanitizer（UBsan）	

- 以下用 AddressSanitizer (ASan) 當例子

ASAN 原理

- 主要透過兩個方法
  - 程式碼插樁（Instrumentation)
  - 動態運行庫（Run-time library）
- 插樁
  - 在程式碼編譯時期對程式碼加料，來處理一些對記憶體的操作
- 動態運行庫：攔截一些特別的程式碼，並改由特定 library 處理
  - malloc
  - free
  - strcpy
  - ……
- 有用到 gcc 特有的東西會炸掉
- 使用： gcc -fsanitize=address

Mapping: `Shadow = (Addr>>3) + Offset`



0xffffffff
0x40000000

0x3fffffff
0x28000000

0x27ffffff
0x24000000

0x23ffffff
0x20000000

0x1fffffff
0x00000000

0xffffffff
0x40000000

0x3fffffff
0x28000000

0x27ffffff
0x24000000

0x23ffffff
0x20000000

0x1fffffff
0x00000000

☐ Application

🟦 Shadow

🟥 mprotect-ed

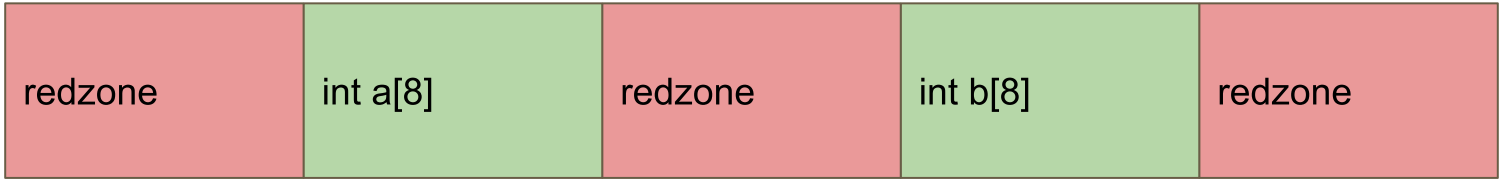
Instrumenting stack

```
void foo() {

  char a[328];






  <------------- CODE ------------->

}
```

Instrumenting stack

```
void foo() {
  char rz1[32];  // 32-byte aligned
  char a[328];
  char rz2[24];
  char rz3[32];
  int  *shadow = (&rz1 >> 3) + kOffset;
  shadow[0] = 0xffffffff;    // poison rz1

  shadow[11] = 0xffffff00;  // poison rz2
  shadow[12] = 0xffffffff;  // poison rz3
  <------------- CODE ------------->
  shadow[0] = shadow[11] = shadow[12] = 0;
}
```

Red Zone



redzone

int a[8]

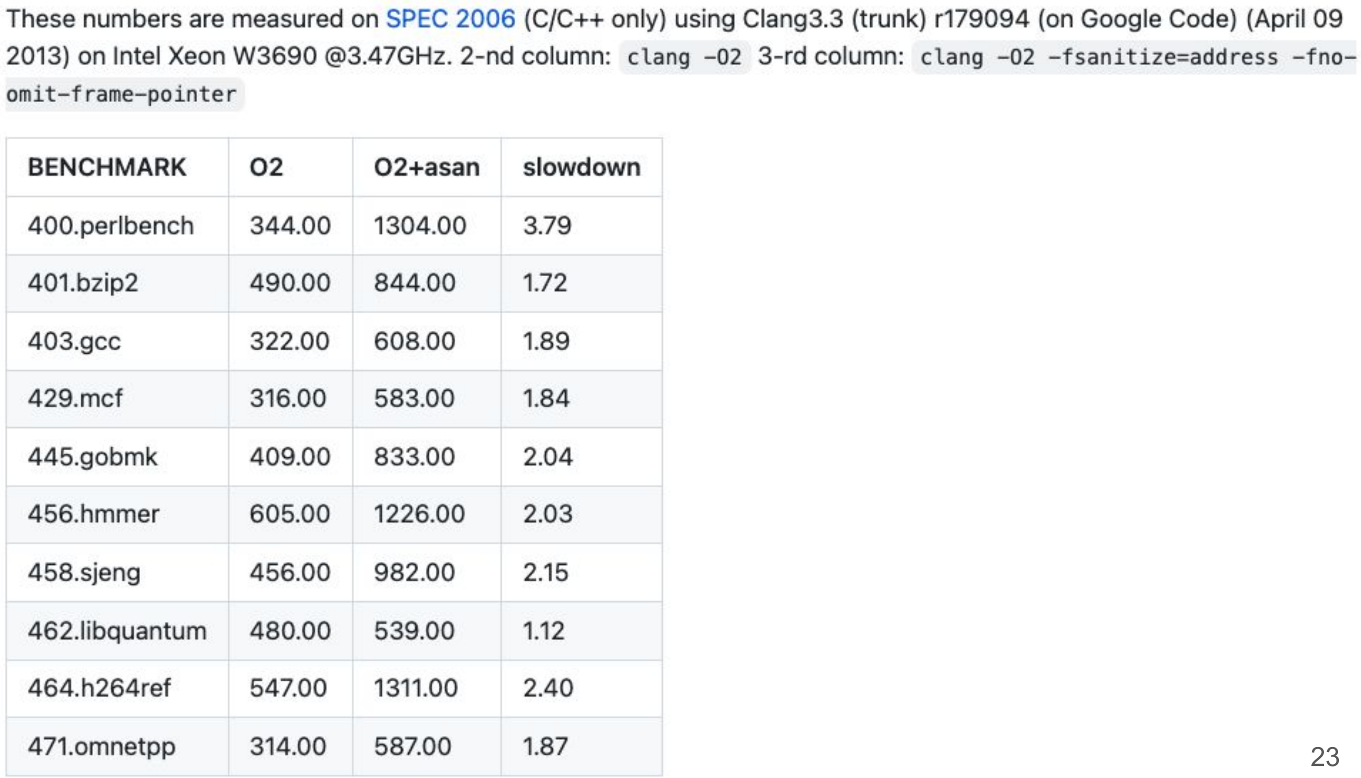
redzone

int b[8]

redzone

ASAN

- slowdown: ~2x

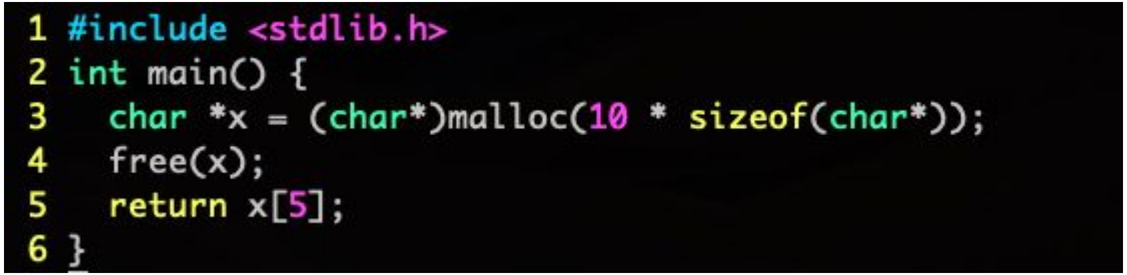


These numbers are measured on SPEC 2006 (C/C++ only) using Clang3.3 (trunk) r179094 (on Google Code) (April 09 2013) on Intel Xeon W3690 @3.47GHz. 2-nd column: `clang -O2` 3-rd column: `clang -O2 -fsanitize=address -fno-omit-frame-pointer`

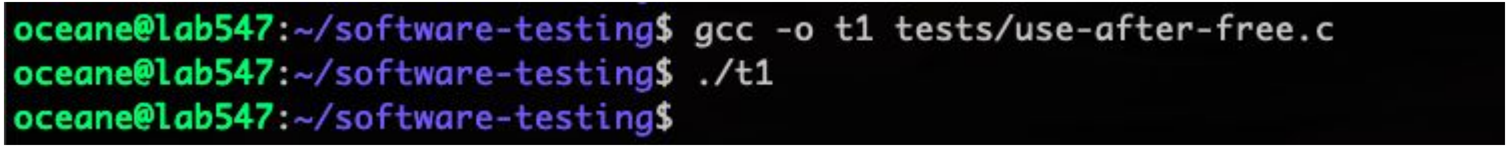
BENCHMARK	O2	O2+asan	slowdown
400.perlbench	344.00	1304.00	3.79
401.bzip2	490.00	844.00	1.72
403.gcc	322.00	608.00	1.89
429.mcf	316.00	583.00	1.84
445.gobmk	409.00	833.00	2.04
456.hmmer	605.00	1226.00	2.03
458.sjeng	456.00	982.00	2.15
462.libquantum	480.00	539.00	1.12
464.h264ref	547.00	1311.00	2.40
471.omnetpp	314.00	587.00	1.87

ASAN

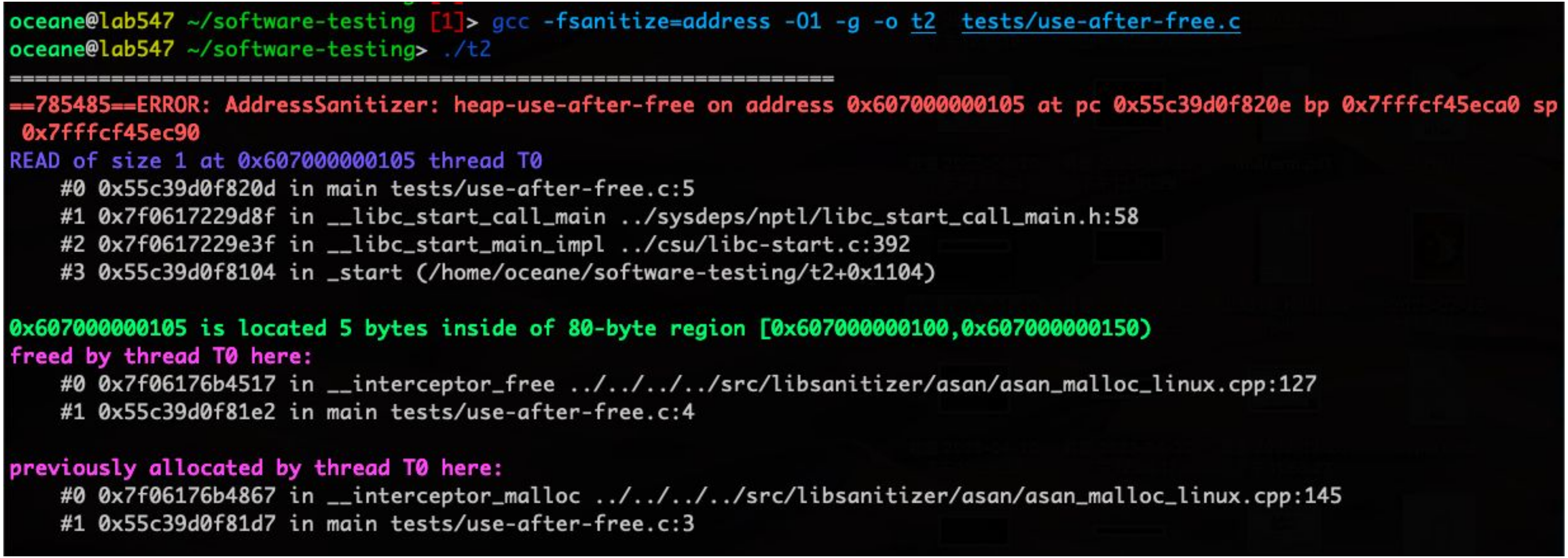
- use-after-free.c



```c
#include <stdlib.h>
int main() {
  char *x = (char*)malloc(10 * sizeof(char*));
  free(x);
  return x[5];
}
```



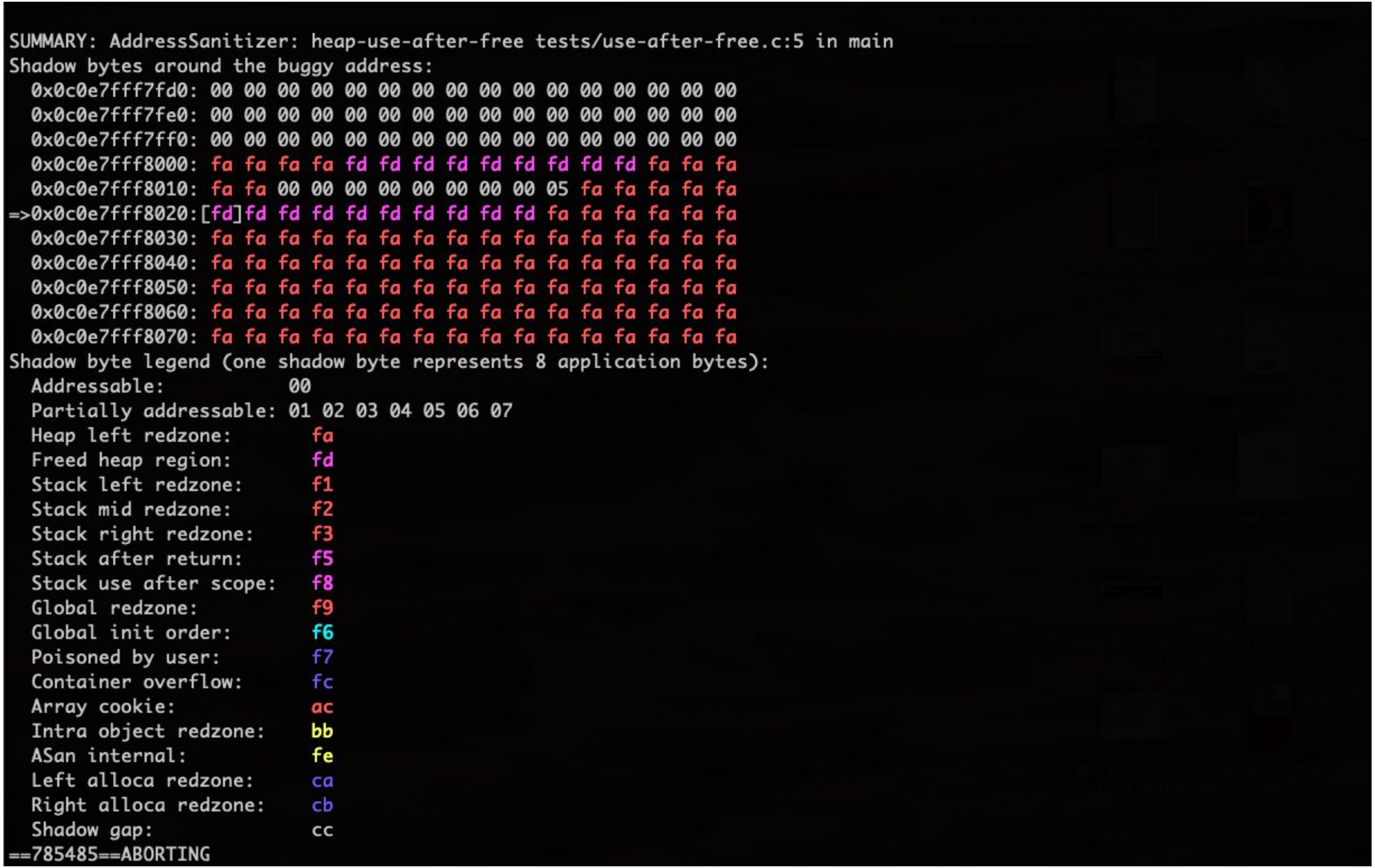
```
oceane@lab547:~/software-testing$ gcc -o t1 tests/use-after-free.c
oceane@lab547:~/software-testing$ ./t1
oceane@lab547:~/software-testing$
```



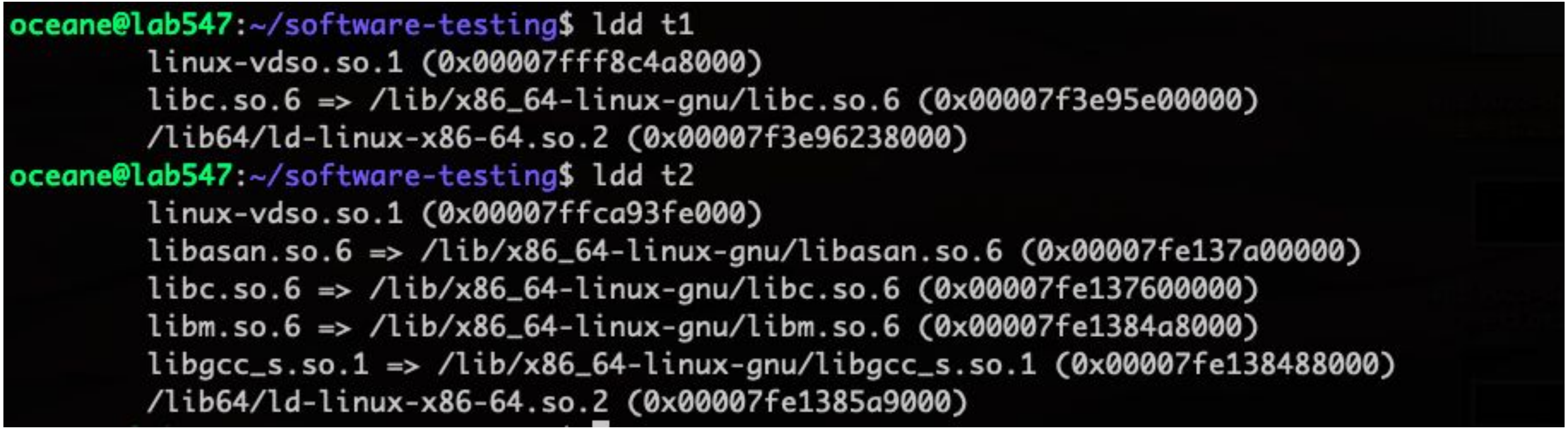
```
oceane@lab547 ~/software-testing [1]> gcc -fsanitize=address -O1 -g -o t2  tests/use-after-free.c
oceane@lab547 ~/software-testing> ./t2
=================================================================
==785485==ERROR: AddressSanitizer: heap-use-after-free on address 0x607000000105 at pc 0x55c39d0f820e bp 0x7fffcf45eca0 sp
 0x7fffcf45ec90
READ of size 1 at 0x607000000105 thread T0
    #0 0x55c39d0f820d in main tests/use-after-free.c:5
    #1 0x7f0617229d8f in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58
    #2 0x7f0617229e3f in __libc_start_main_impl ../csu/libc-start.c:392
    #3 0x55c39d0f8104 in _start (/home/oceane/software-testing/t2+0x1104)

0x607000000105 is located 5 bytes inside of 80-byte region [0x607000000100,0x607000000150)
freed by thread T0 here:
    #0 0x7f06176b4517 in __interceptor_free ../../../../src/libsanitizer/asan/asan_malloc_linux.cpp:127
    #1 0x55c39d0f81e2 in main tests/use-after-free.c:4

previously allocated by thread T0 here:
    #0 0x7f06176b4867 in __interceptor_malloc ../../../../src/libsanitizer/asan/asan_malloc_linux.cpp:145
    #1 0x55c39d0f81d7 in main tests/use-after-free.c:3
```



```
SUMMARY: AddressSanitizer: heap-use-after-free tests/use-after-free.c:5 in main
Shadow bytes around the buggy address:
  0x0c0e7fff7fd0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c0e7fff7fe0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c0e7fff7ff0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c0e7fff8000: fa fa fa fa fd fd fd fd fd fd fd fd fd fa fa fa
  0x0c0e7fff8010: fa fa 00 00 00 00 00 00 00 00 05 fa fa fa fa fa
=>0x0c0e7fff8020:[fd]fd fd fd fd fd fd fd fd fd fa fa fa fa fa fa
  0x0c0e7fff8030: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c0e7fff8040: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c0e7fff8050: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c0e7fff8060: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c0e7fff8070: 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
  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
==785485==ABORTING
```

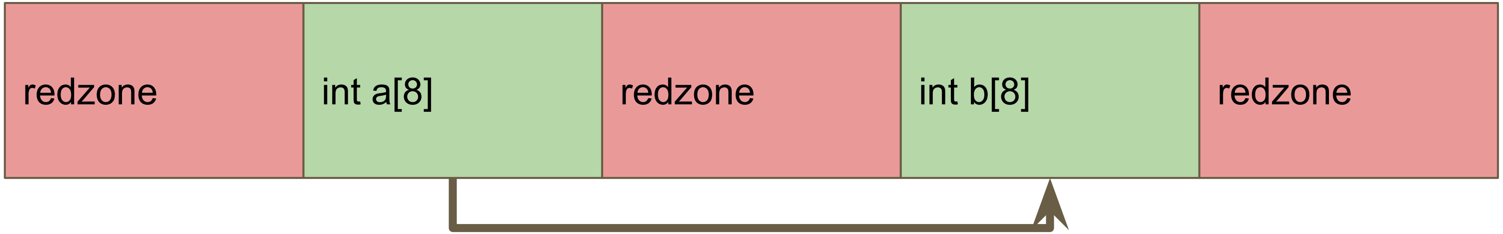


```
oceane@lab547:~/software-testing$ ldd t1
        linux-vdso.so.1 (0x00007fff8c4a8000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f3e95e00000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f3e96238000)
oceane@lab547:~/software-testing$ ldd t2
        linux-vdso.so.1 (0x00007ffca93fe000)
        libasan.so.6 => /lib/x86_64-linux-gnu/libasan.so.6 (0x00007fe137a00000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fe137600000)
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007fe1384a8000)
        libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007fe138488000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fe1385a9000)
```

Lab 6

Lab 6

- 下面是常見的記憶體操作問題，請分別寫出有下列記憶體操作問題的簡單程式，並說明 Valgrind 和 ASan 能否找的出來
  - Heap out-of-bounds read/write
  - Stack out-of-bounds read/write
  - Global out-of-bounds read/write
  - Use-after-free
  - Use-after-return
- 寫一個簡單程式 with ASan，Stack buffer overflow 剛好越過 redzone(並沒有對 redzone 做讀寫)，並說明 ASan 能否找的出來？



redzone

int a[8]

redzone

int b[8]

redzone

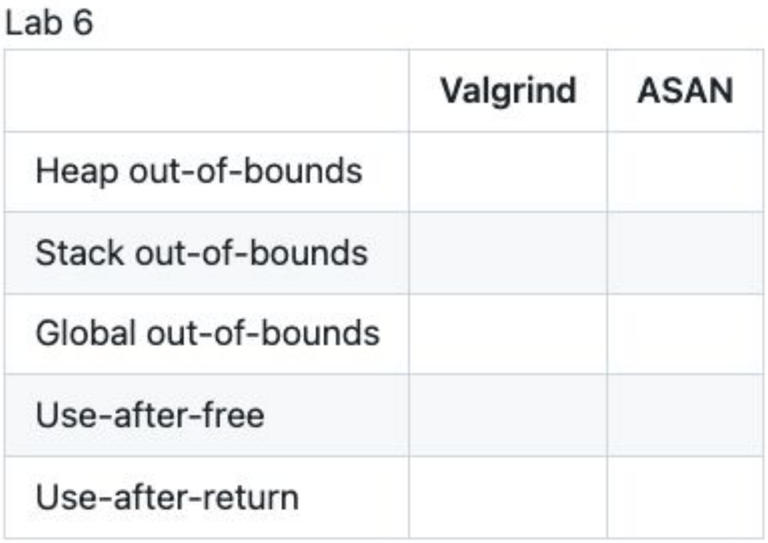
Lab 6

- 請使用 Markdown / RST 格式撰寫
- 每個問題都需要附上程式碼和執行結果，並說明你使用哪種編譯器及版本
- example:
  - ### Heap out-of-bounds

  -
    ```
  - 有問題的程式碼
  - ```
  - ```
  - ASan report
  - ```
  - ```
  - valgrind report
  - ```
  - ASan 能/不能 , valgrind 能/不能
    ```

Lab 6

- 請在 report 中填寫下列表格



Lab 6

	Valgrind	ASAN
Heap out-of-bounds		
Stack out-of-bounds		
Global out-of-bounds		
Use-after-free		
Use-after-return		

Lab 6

- 請將你的 report 命名為 README.md / README.rst 並放置在 <student_id>-ST-2023 的 Lab06 資料夾中
  - Lab06/README.md

Submission

Submission

- Please submit your Github repo <student_id>-ST-2023 (1) commit URL to E3
- Please submit your URL as link
- commit URL
  - refer to Lab 1 submission

Reference

- Valgrind
  - https://access.redhat.com/documentation/zh-tw/red_hat_enterprise_linux/6/html/performance_tuning_guide/s-memory-valgrind
  - https://valgrind.org/
  - https://valgrind.org/docs/valgrind2007.pdf
- Sanitizer
  - https://github.com/google/sanitizers
  - https://gcc.gnu.org/onlinedocs/gcc-4.9.2/gcc/Debugging-Options.html#index-fsanitize_003daddress-593
  - https://en.wikipedia.org/wiki/Stack_buffer_overflow