Valgrind & Sanitizer

yuan

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
- $ valgrind --tool=<toolname> <program>
- 注意 : 使用 Valgrind 會讓程式執行速度比平常更慢。

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

Valgrind 實做方法

- 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

使用範例：

- Valgrind 是動態追蹤且會追蹤到 glibc，使用前要安裝對應的 glibc debug 套件
  - $ sudo apt install libc6-dbg
- 以下舉個例子：

```
➜  test vim test.c
➜  test gcc test.c
➜  test ./a.out
a
```

```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;
}
```

使用範例：

```
➜ test valgrind ./a.out
==27569== Memcheck, a memory error detector
==27569== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==27569== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==27569== Command: ./a.out
==27569==
==27569== Invalid write of size 1
==27569==    at 0x1086F8: main (in /home/yuan/test/a.out)
==27569==  Address 0x522f044 is 0 bytes after a block of size 4 alloc'd
==27569==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==27569==    by 0x1086EB: main (in /home/yuan/test/a.out)
==27569==
==27569== Invalid read of size 1
==27569==    at 0x108703: main (in /home/yuan/test/a.out)
==27569==  Address 0x522f044 is 0 bytes after a block of size 4 alloc'd
==27569==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==27569==    by 0x1086EB: main (in /home/yuan/test/a.out)
==27569==
a
==27569==
==27569== HEAP SUMMARY:
==27569==     in use at exit: 0 bytes in 0 blocks
==27569==   total heap usage: 2 allocs, 2 frees, 1,028 bytes allocated
==27569==
==27569== All heap blocks were freed -- no leaks are possible
==27569==
==27569== For counts of detected and suppressed errors, rerun with: -v
==27569== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

```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;
}
```

```
Invalid write of size 1
   at 0x1086F8: main (in /home/yuan/test/a.out)
 Address 0x522f044 is 0 bytes after a block of size 4 alloc'd
   at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-a
   by 0x1086EB: main (in /home/yuan/test/a.out)
```

```
Invalid read of size 1
   at 0x108703: main (in /home/yuan/test/a.out)
 Address 0x522f044 is 0 bytes after a block of size 4 alloc'd
   at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-a
   by 0x1086EB: main (in /home/yuan/test/a.out)
```

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。

原理

- 主要透過兩個方法：程式碼插樁（Instrumentation）和 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

例子

- 可以看到多了很多 library

```
➜  test gcc -o t1 test.c
➜  test ldd t1
        linux-vdso.so.1 (0x00007ffff7ffb000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ffff77e0000)
        /lib64/ld-linux-x86-64.so.2 (0x00007ffff7dd3000)
➜  test gcc -fsanitize=address -o t2 test.c
➜  test ldd t2
        linux-vdso.so.1 (0x00007ffff7ffb000)
        libasan.so.4 => /usr/lib/x86_64-linux-gnu/libasan.so.4 (0x00007ffff6c16000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ffff6825000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007ffff6621000)
        librt.so.1 => /lib/x86_64-linux-gnu/librt.so.1 (0x00007ffff6419000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007ffff61fa000)
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007ffff5e5c000)
        libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007ffff5c44000)
        /lib64/ld-linux-x86-64.so.2 (0x00007ffff7dd3000)
```

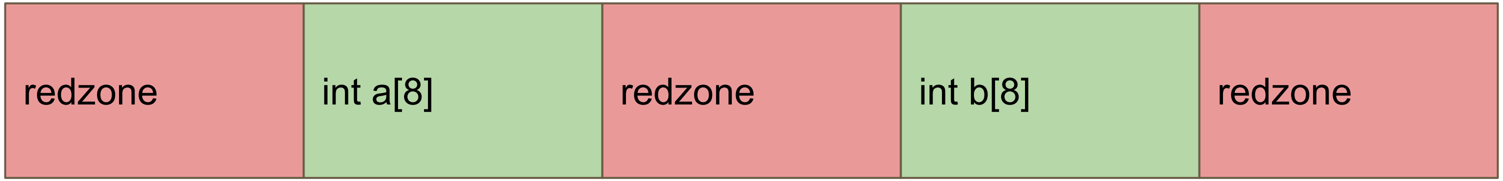
例子

```
char cVar1;
undefined8 *puVar2;
ulong uVar3;
long lVar4;
undefined8 *puVar5;
long in_FS_OFFSET;
undefined auVar6 [16];
undefined8 local_d8 [23];
long local_20;

puVar5 = local_d8;
if (__asan_option_detect_stack_use_after_return != 0) {
  puVar2 = (undefined8 *)__asan_stack_malloc_2(0xa0);
  if (puVar2 != (undefined8 *)0x0) {
    puVar5 = puVar2;
  }
}
*puVar5 = 0x41b58ab3;
puVar5[1] = 0x100c24;
puVar5[2] = 0x10097a;
uVar3 = (ulong)puVar5 >> 3;
auVar6 = CONCAT88(puVar5 + 0x18,uVar3);
*(undefined4 *)(uVar3 + 0x7fff8000) = 0xf1f1f1f1;
*(undefined4 *)(uVar3 + 0x7fff8008) = 0xf2f2f2f2;
*(undefined4 *)(uVar3 + 0x7fff8010) = 0xf3f3f3f3;
local_20 = *(long *)(in_FS_OFFSET + 0x28);
cVar1 = *(char *)(((long)puVar5 + 0x3fU >> 3) + 0x7fff8000);
if (cVar1 <= (char)((byte)((long)puVar5 + 0x3fU) & 7) && cVar1 != '\0' ||
    0x7f < *(byte *)(((ulong)(puVar5 + 4) >> 3) + 0x7fff8000)) {
  auVar6 = __asan_report_store_n(puVar5 + 4,0x20);
}
lVar4 = SUB168(auVar6 >> 0x40,0);
*(undefined8 *)(lVar4 + -0xa0) = 0;
*(undefined8 *)(lVar4 + -0x98) = 0;
*(undefined8 *)(lVar4 + -0x90) = 0;
*(undefined8 *)(lVar4 + -0x88) = 0;
*(undefined8 *)(lVar4 + -0x60) = 0;
*(undefined8 *)(lVar4 + -0x58) = 0;
```

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

int main()
{
    int a[8] = {0};
    int b[8] = {0};
    a[8] = 0xcafe;
    return 0;
}
```

redzone

- 在每個變數中間插一塊不可讀寫的區域，有做存取就噴錯



例子

```
=================================================================
==31359==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7fffffffdf00 at pc 0x555555554aff bp 0x7fffffffdeb0 sp 0x7fffffffdea0
WRITE of size 4 at 0x7fffffffdf00 thread T0
    #0 0x555555554afe in main /home/yuan/test/test.c:9
    #1 0x7ffff6a48bf6 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x21bf6)
    #2 0x555555554899 in _start (/home/yuan/test/t2+0x899)

Address 0x7fffffffdf00 is located in stack of thread T0 at offset 64 in frame
    #0 0x555555554989 in main /home/yuan/test/test.c:6

  This frame has 2 object(s):
    [32, 64) 'a' <== Memory access at offset 64 overflows this variable
    [96, 128) 'b'
HINT: this may be a false positive if your program uses some custom stack unwind mechanism or swapcontext
      (longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: stack-buffer-overflow /home/yuan/test/test.c:9 in main
Shadow bytes around the buggy address:
  0x10007fff7b90: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10007fff7ba0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10007fff7bb0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10007fff7bc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10007fff7bd0: 00 00 00 00 00 00 00 00 f1 f1 f1 f1 00 00 00 00
=>0x10007fff7be0:[f2]f2 f2 f2 00 00 00 00 f3 f3 f3 f3 00 00 00 00
  0x10007fff7bf0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10007fff7c00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10007fff7c10: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10007fff7c20: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10007fff7c30: 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
==31359==ABORTING
```

例子

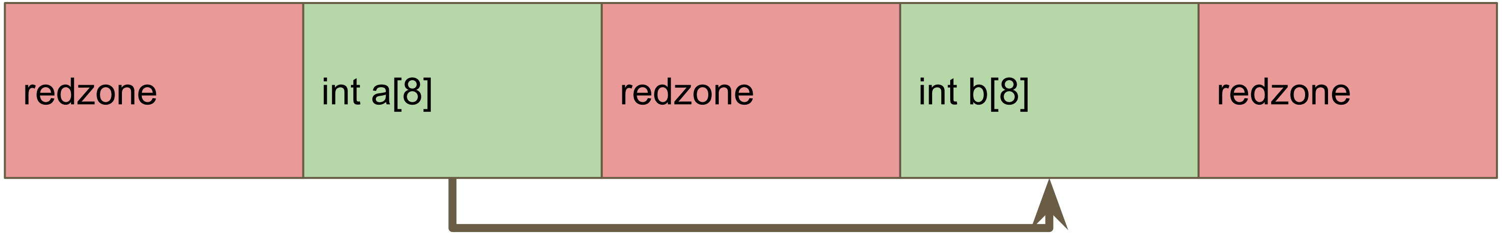
```
=================================================================
==31359==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7fffffffdf00 at pc 0x555555554aff bp 0x7fffffffdeb0
WRITE of size 4 at 0x7fffffffdf00 thread T0
    #0 0x555555554afe in main /home/yuan/test/test.c:9
    #1 0x7ffff6a48bf6 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x21bf6)
    #2 0x555555554899 in _start (/home/yuan/test/t2+0x899)

Address 0x7fffffffdf00 is located in stack of thread T0 at offset 64 in frame
    #0 0x555555554989 in main /home/yuan/test/test.c:6

  This frame has 2 object(s):
    [32, 64) 'a' <== Memory access at offset 64 overflows this variable
    [96, 128) 'b'
HINT: this may be a false positive if your program uses some custom stack unwind mechanism or swapcontext
      (longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: stack-buffer-overflow /home/yuan/test/test.c:9 in main
```

例子

```
Shadow bytes around the buggy address:
  0x10007fff7b90: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10007fff7ba0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10007fff7bb0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10007fff7bc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10007fff7bd0: 00 00 00 00 00 00 00 00 f1 f1 f1 f1 00 00 00 00
=>0x10007fff7be0:[f2]f2 f2 f2 00 00 00 00 f3 f3 f3 f3 00 00 00 00
  0x10007fff7bf0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10007fff7c00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10007fff7c10: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10007fff7c20: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10007fff7c30: 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
```

作業

- 1. 下面是常見的漏洞，請分別寫出有下面漏洞的簡單程式，並告訴我 Valgrind 和 ASan 兩個分別找不找的出來
    - Heap out-of-bounds read/write
    - Stack out-of-bounds read/write
    - Global out-of-bounds read/write
    - Use-after-free
    - Use-after-return
- 2. 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)，並告訴我 ASan 能否找的出來？



繳交格式

- 用 Markdown 格式寫，並匯出成 pdf 上傳
- 每個漏洞都需要附上程式碼和執行結果，並告訴我你用啥編譯器及版本
- example:
    - ### Heap out-of-bounds
    - ```
    - 有問題的程式碼
    - ```
    - ```
    - ASan report
    - ```
    - ```
    - valgrind report
    - ```
    - ASan 能/不能 , valgrind 能/不能

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
- sanitizers
  - 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