

Яндекс

Яндекс

Мойте руки перед едой,  
или Санитайзеры  
в тестировании

Андрей Сатарин

# О чем я сегодня расскажу

- › Что такое санитайзеры?
- › Address Sanitizer
- › Устройство Address Sanitizer
- › Memory Sanitizer
- › Thread Sanitizer

**Не буду** рассказывать какие нужны ключи и как интегрировать в вашу сборку

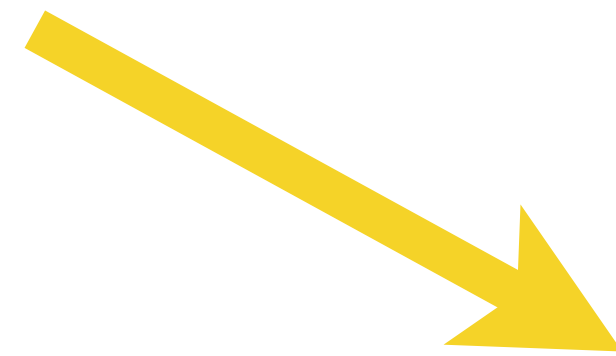
Что такое санитайзеры?



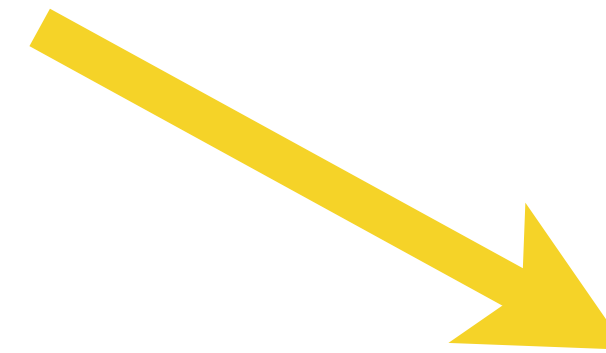
# Что такое санитайзеры?

- › Инструменты для динамического **поиска дефектов** кода на C++
- › Динамический — значит **работают на запущенном коде**, например в тестах
- › Требуют **специальной компиляции** кода программы (работает в GCC, Clang)
- › Поддерживаются на Linux x86\_64


Компиляция тестов  
с санитайзерами



Запуск тестов



**Новые дефекты**  
найденные санитайзерами



Зачем нам нужны  
специальные  
инструменты для C++?

# Ошибки работы с памятью

- › Переполнение буфера (buffer overflow)
- › Использование после освобождения (use after free)
- › Использование не инициализированного значения (uninitialized value)



# CVE: buffer overflow

CVE — Common Vulnerabilities and Exposures

<http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=buffer+overflow>

Search Results:

There are **8322** CVE entries that match your search.

# WannaCrypt

This is based on the Eternalblue tool stolen from the NSA, and was developed by infosec biz RiskSense. It reveals that the SMB server bug is the result of a **buffer overflow** in Microsoft's code. [WC]



Payment will be raised on

5/16/2017 00:47:55

Time Left

02:23:57:37

Your files will be lost on

5/20/2017 00:47:55

Time Left

06:23:57:37

What happened to  
Your important files are e  
Many of your documents  
accessible because they h  
recover your files, but do  
our decryption service.

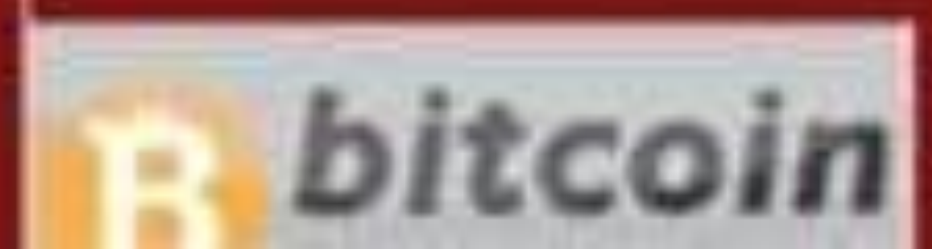
## Can I Recover My

Sure. We guarantee that y  
not so enough time.  
You can decrypt some of  
But if you want to decryp  
You only have 3 days to s  
Also, if you don't pay in 7  
We will have free events.

## How Do I Pay?

Payment is accepted in B  
Please check the current  
click <How to buy bitcoi  
And send the correct amo  
After your payment, click

About bitcoin



# CVE: use after free

CVE — Common Vulnerabilities and Exposures

<http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=use+after+free>

Search Results:

There are **1006** CVE entries that match your search.

# CVE: uninitialized memory

CVE — Common Vulnerabilities and Exposures

<http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=uninitialized+memory>

Search Results

There are **202** CVE entries that match your search.



- Do you know the first thing about **bug finding**?
- Stick 'em with the pointy end.
- That's the essence of it.

# Address Sanitizer



# Buffer overflow: C++ vs Java

C++

```
int* a =  
new int[10];  
a[10] = 1;  
// nothing
```

Java

```
int[] a =  
new int[10];  
a[10] = 1;  
// ArrayIndexOutOfBoundsException
```



# Пример 1



```
int sum(int* array, int lo, int hi) {  
    int res = 0;  
    for (int i = lo; i <= hi; i++) {  
        res += array[i];  
    }  
    return res;  
}  
int main(int argc, char **argv) {  
    int *array = new int[10] {0, ..., 9};  
    int res = sum(array, argc, 10);  
    delete [] array;  
    return res;  
}
```

# Address Sanitizer: heap-buffer-overflow

```
AddressSanitizer: heap-buffer-overflow on address ...  
READ of size 4 at ... thread T0
```

```
#0 ... in sum(int*, int, int) /heap_buffer_overflow.cpp:6:16  
#1 ... in main /heap_buffer_overflow.cpp:14
```

... is located 0 bytes to the right of 40-byte region [...,...) allocated by thread T0 here:

```
#0 ...  
#1 ... in main /heap_buffer_overflow.cpp:13:18
```

```
SUMMARY: AddressSanitizer: heap-buffer-overflow  
/heap_buffer_overflow.cpp:6:16 in sum(int*, int, int)
```

```
int sum(int* array, int lo, int hi) {  
    int res = 0;  
    for (int i = lo; i <= hi; i++) {  
        res += array[i];  
    }  
    return res;  
}  
int main(int argc, char **argv) {  
    int *array = new int[10] {0, ..., 9};  
    int res = sum(array, argc, 10);  
    delete [] array;  
    return res;  
}
```

allocated here



# Пример 2

```
int fib(int n) {  
    int *arr = new int[n + 2] {0};  
    arr[0] = 1;  
    arr[1] = 1;  
    for (int i=2; i < n; i++) {  
        arr[i] = arr[i - 1] + arr[i - 2];  
    }  
    int *x = &arr[n - 1];  
    delete [] arr;  
    return *x;  
}
```

# Address Sanitizer: heap-use-after-free

```
AddressSanitizer: heap-use-after-free on address ...  
READ of size 4 at ... thread T0
```

```
#0 ... in fib(int) /heap_use_after_free.cpp:12:12
```

```
...
```

```
... is located 0 bytes inside of 12-byte region [...,...) freed by thread  
T0 here:
```

```
#0 ...
```

```
#1 ... in fib(int) /heap_use_after_free.cpp:11:5
```

```
previously allocated by thread T0 here:
```

```
#0 ...
```

```
#1 ... in fib(int) /heap_use_after_free.cpp:4:18
```

```
SUMMARY: AddressSanitizer: heap-use-after-free  
/heap_use_after_free.cpp:12:12 in fib(int)
```

```

int fib(int n) {
    int *arr = new int[n + 2] {0};
    arr[0] = 1;
    arr[1] = 1;
    for (int i=2; i < n; i++) {
        arr[i] = arr[i - 1] + arr[i - 2];
    }
    int *x = &arr[n - 1];
    delete [] arr;      freed by thread T0 here
    return *x;        heap-use-after-free
}

```

# Устройство Address Sanitizer





# Как это все работает?

- › **Инструментация** при компиляции — добавляем проверки на каждое чтение/запись
- › **Runtime** библиотека для проверки доступов
- › Специальная **«теневая» область памяти** для отслеживания состояния памяти (shadow memory)

# Инструментация при КОМПИЛЯЦИИ

```
*address = ...;
```

```
if (IsPoisoned(address)) {  
    ReportError(...);  
}  
*address = ...;
```

# Runtime библиотека

- › Подменяет malloc/free
- › malloc создает **«красные зоны»** при аллокации
- › free **«отравляет»** (poisons) освобожденные регионы памяти и держит их в карантине

# Теневая память (Shadow memory)

- › На 8 байт памяти создается 1 байт «теневой» памяти
- › Содержит **метаданные** о памяти вашего приложения
- › **«Отравление»** (poisoning) блока основной памяти — специальная метка в теневой памяти, соответствующей этому блоку основной памяти

# Теневая память (Shadow memory)

Shadow byte legend (one shadow byte represents 8 application bytes):

<b>Addressable:</b>	00
<b>Partially addressable:</b>	01 02 03 04 05 06 07
<b>Heap left redzone:</b>	fa
<b>Heap right redzone:</b>	fb
<b>Freed heap region:</b>	fd

...



# Пример 3

# Детектирование use-after-free

```
int main(int argc, char **argv) {  
    int *array = new int[10] {0};  
    int *x = &array[argc];  
    delete [] array;  
    return *x;  
}
```

```
int *array = new int[10]{0};
```

Shadow memory:

0x9bd0:	fa	fa	fa	fa	fa	fa	fa	fa	fa	fa
0x9be0:	fa	fa	fa	fa	fa	fa	fa	fa	fa	fa
0x9bf0:	fa	fa	fa	fa	00	00	00	00	00	fa
0x9c00:	fa	fa	fa	fa	ta	ta	ta	ta	ta	fa
0x9c20:	fa	fa	fa	fa	fa	fa	fa	fa	fa	fa

fa – Heap left redzone



```
int *x = &array[argc];
```

Shadow memory:

0x9bd0:	fa	fa	fa	fa	fa	fa	fa	fa	fa	fa
0x9be0:	fa	fa	fa	fa	fa	fa	fa	fa	fa	fa
0x9bf0:	fa	fa	fa	fa	00	00	00	00	00	fa
0x9c00:	fa	fa	fa	fa	fa	fa	fa	fa	fa	fa
0x9c20:	fa	fa	fa	fa	fa	fa	fa	fa	fa	fa

00 – Addressable

**delete** [] array;

Shadow memory:

0x9bd0:	fa	fa	fa	fa	fa	fa	fa	fa	fa	fa
0x9be0:	fa	fa	fa	fa	fa	fa	fa	fa	fa	fa
0x9bf0:	fa	fa	fa	fa	fd	fd	fd	fd	fd	fa
0x9c00:	fa	fa	fa	fa	ta	ta	ta	ta	ta	fa
0x9c20:	fa	fa	fa	fa	fa	fa	fa	fa	fa	fa

fd – Freed heap region

**return \*x;**

Shadow memory:

0x9bd0:	fa	fa	fa	fa	fa	fa	fa	fa	fa	fa
0x9be0:	fa	fa	fa	fa	fa	fa	fa	fa	fa	fa
0x9bf0:	fa	fa	fa	fa	[fd]	fd	fd	fd	fd	fa
0x9c00:	fa	fa	fa	fa	fa	fa	fa	fa	fa	fa
0x9c20:	fa	fa	fa	fa	fa	fa	fa	fa	fa	fa

fd – Freed heap region

# ИТОГОВЫЙ ОТЧЕТ

AddressSanitizer: heap-use-after-free on address ...

READ of size 4 at ... thread T0

#0 ... in main /heap\_use\_after\_free.cpp:8:12

... is located 4 bytes inside of 40-byte region ... here:

#0 ...

#1 ... in main /heap\_use\_after\_free.cpp:7:5

previously allocated by thread T0 here:

#0 ...

#1 ... in main /heap\_use\_after\_free.cpp:5:18

SUMMARY: AddressSanitizer: heap-use-after-free  
/heap\_use\_after\_free.cpp:8:12 in main

# Детектирование use-after-free

```
int main(int argc, char **argv) {  
    int *array = new int[10] {0};  
    int *x = &array[argc];  
    delete [] array; inside of 40-byte region  
    return *x; READ of size 4  
}
```

# Address Sanitizer: итоги

- › Практически **нет false positive** ошибок
- › **Высокая точность**, если код сделал «что-то плохое» — это будет обнаружено
- › Проще всего для первоначального внедрения в проекте
- › В наших тестах нет замедления, обычно ~2x

# Memory Sanitizer



# Не инициализированная память: Java vs C++

C++

```
int* a =  
new int[10];  
a[0] // == ???
```

Java

```
int[] a =  
new int[10];  
a[0] // == 0
```





# Пример 4

```
int main(int argc, char** argv) {  
    int* a = new int[10] {0};  
    int* b = new int[10];  
    memcpy(b, a, 10);  
    int res = b[argc + 5];  
    delete [] a;  
    delete [] b;  
    return res;  
}
```

# Memory Sanitizer: Use-of-uninitialized-value

```
MemorySanitizer: use-of-uninitialized-value  
#0 ... in main /uninitialized-memory.cpp:17:5  
...
```

```
SUMMARY: MemorySanitizer: use-of-uninitialized-value  
/uninitialized-memory.cpp:17:5 in main
```

```
int main(int argc, char** argv) {  
    int* a = new int[10] {0};  
    int* b = new int[10];  
    memcpy(b, a, 10);  
    int res = b[argc + 5];  
    delete [] a;  
    delete [] b;  
    return res;  
}
```

← Проблемная память

use-of-uninitialized-value

# Memory Sanitizer: origin tracking

```
MemorySanitizer: use-of-uninitialized-value  
  #0 ... in main /uninitialized-memory.cpp:17:5  
  ...
```

```
Uninitialized value was created by a heap allocation  
  #0 ...  
  #1 ... in main /uninitialized-memory.cpp:12:14
```

```
SUMMARY: MemorySanitizer: use-of-uninitialized-value  
/uninitialized-memory.cpp:17:5 in main
```

```
int main(int argc, char** argv) {  
    int* a = new int[10] {0};  
    int* b = new int[10];           Heap allocation  
    memcpy(b, a, 10);  
    int res = b[argc + 5];  
    delete [] a;  
    delete [] b;  
    return res;           use-of-uninitialized-value  
}
```

# Memory Sanitizer: тонкая настройка

```
__msan_unpoison(a, size);
```


```
__msan_poison(a, size);
```

```
__msan_check_mem_is_initialized(  
    a, size  
);
```

# Memory Sanitizer: итоги

- › Memory Sanitizer находит только одну проблему — это **не делает** его менее полезным
- › Не инициализированная память может приводить к произвольным последствиям в коде
- › В сложных случаях может помочь origins tracking и тонкая настройка





Важны ли дефекты  
найденные  
санитайзерами?

# Важны ли дефекты от санитайзеров?

Было:

- › несколько дефектов найденных Address/Memory Sanitizer
- › **странные баги**, которые было непонятно как чинить

=> приняли решение починить все дефекты найденные санитайзерами

# Важны ли дефекты от санитайзеров?

Стало:

- › все проблемы, найденные Address/Memory Sanitizer были исправлены
- › странные **баги пропали**, хотя напрямую их никто не чинил

# Thread Sanitizer



# Состояние гонки (race condition/data race)

Состояние гонки является классическим **гейзенбагом**.

Состояние гонки возникает тогда, когда несколько потоков многопоточного приложения пытаются одновременно получить доступ к данным, причем **хотя бы один поток выполняет запись [DR]**.



# CVE: race condition

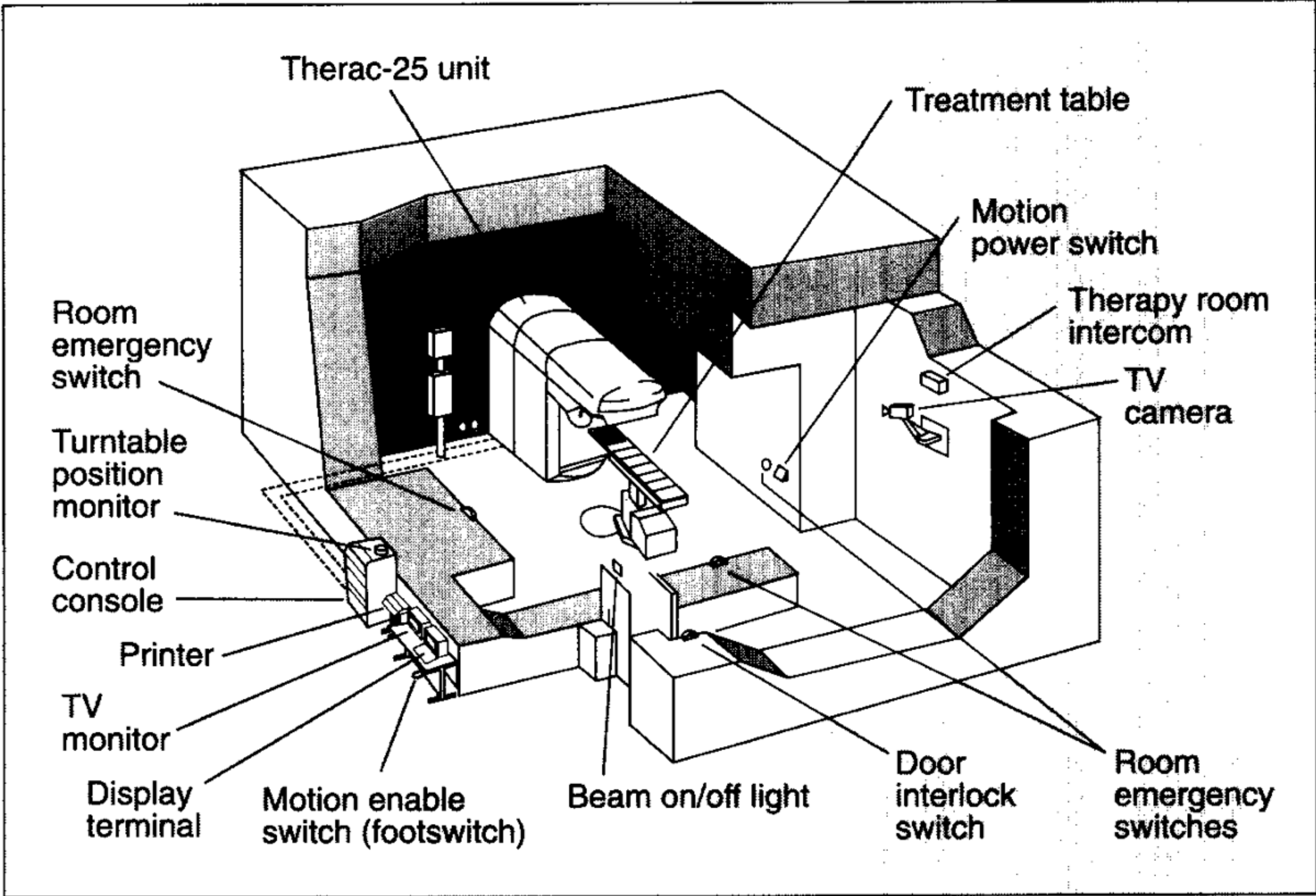
CVE — Common Vulnerabilities and Exposures

<http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=race+condition>

Search Results

There are **580** CVE entries that match your search.

# Therac-25





# Пример 5



```
void Init() {  
    if (!inited) {  
        mutex.lock();  
        if (!inited) {  
            Global = 1;  
        }  
        inited = true;  
        mutex.unlock();  
    }  
}
```

# Thread Sanitizer: data race

```
ThreadSanitizer: data race
```

```
Write of size 1 at ... by main thread (mutexes: write M7):
```

```
#0 Init() /dcl.cpp:17:12
```

```
Previous read of size 1 at ... by thread T1:
```

```
#0 Init() /dcl.cpp:11:8
```

```
Location is global 'inited' of size 1 at ...
```

```
Mutex M7 (...) created at:
```

```
...
```

```
SUMMARY: ThreadSanitizer: data race /dcl.cpp:17:12 in Init()
```

```
void Init() {  
    if (!inited) {  
        mutex.lock();  
        if (!inited) {  
            Global = 1;  
        }  
        inited = true;  
        mutex.unlock();  
    }  
}
```

Previous read of size 1

Write of size 1

# Thread Sanitizer: итоги

- › Дефекты от Thread Sanitizer чинят с наибольшей неохотой — лучше внедрять его последним из всех
- › **Не 100% точен** — тесты с ним «мигают»
- › На нашем коде из всех троих **больше всего замедляет тесты**
- › Требуется **много (~5x) памяти** by design

# Частые проблемы и решения





Санитайзеры тормозят  
и ломают тесты

# Тесты тормозят — что делать?

```
constexpr TDuration TIMEOUT
    = NSan::PlainOrUnderSanitizer(
        TDuration::Seconds(120),
        TDuration::Seconds(240)
    );
```

# Тесты тормозят — что делать?

```
inline constexpr static T PlainOrUnderSanitizer(  
    T plain, T sanitized  
) noexcept {  
#if defined(_tsan_enabled_)  
    || defined(_msan_enabled_)  
    || defined(_asan_enabled_)  
        return sanitized;  
#else  
        return plain;  
#endif  
}
```



| Это прекрасно, но мы  
не пишем на C++

# Санитайзеры на других платформах

- › go race — это Thread Sanitizer работающий в Go [[GO1](#)][[GO2](#)]
- › go -msan — Memory Sanitizer для Go [[GO3](#)]  
«Such interoperation is useful mainly for testing a program containing suspect C or C++ code»
- › В компилятор языка Rust с февраля 2017 включена поддержка Address, Leak, Memory, Thread санитайзеров [[RST](#)]
- › Для Java есть инструмент поиска дедлоков (deadlock) от компании Devexperts [[DL](#)]

# Наша статистика

Address Sanitizer	59 дефектов
Memory Sanitizer	26 дефектов
Thread Sanitizer	52 дефекта



# Выводы

- › Санитайзеры можно использовать не зная C++
- › Внедрение лучше начинать с Address Sanitizer
- › Если вы еще не используете санитайзеры, в вашем коде **100% есть дефекты**, которые они найдут
- › Разные санитайзеры находят разные дефекты — надо использовать их все
- › Использование санитайзеров в тестировании — **простой и дешевый** способ решения сложной проблемы

All bugs must die

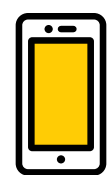


Андрей Сатарин

Ведущий инженер по автоматизации тестирования



[asatarin@yandex-team.ru](mailto:asatarin@yandex-team.ru)



<https://twitter.com/asatarin>

# Ссылки

- › «AddressSanitizer: A Fast Address Sanity Checker»
- › «MemorySanitizer: fast detector of uninitialized memory use in C++»
- › «ThreadSanitizer: data race detection in practice»
- › <https://github.com/google/sanitizers>



# Ссылки

- › [AddressSanitizer, или как сделать программы на C/C++ надежнее и безопаснее](#)
- › [""go test -race" Under the Hood" by Kavya Joshi](#)
- › [Konstantin Serebryany](#)

# Credits

- › <https://www.flickr.com/photos/16210667@N02/8681651088/>
- › Therac-25 <http://www.cs.umd.edu/class/spring2003/cmssc838p/Misc/therac.pdf>
- › <https://www.flickr.com/photos/43326207@N00/4810894062/>