Debugging memory corruptions in C++

Mateusz Nowak

5.11.2019

EmbeddedMeetup

Who am I

Software Engineer @ habana

Located in Gdańsk, Poland

Works on: Enabling AI libraries on HW

Interested in: C++, team work organization, management, composing and recording music

Table of contents

Introduction

Introduction

The problem: Data corruptions

What is data corruption

Unintended change of data

May be caused by hardware failure

May be caused by programming mistake

Buffer overflows

Reading or writing outside of memory buffer range

Can happen on stack or heap

Usually related to array-like structures

Use after free

Access to object after:

- × underlying memory was free'd
- × it was deleted
- × it was moved from

Stack

	...	previous stack frame
stack frame	ebp+16	3rd function argument
	ebp+12	2nd argument
	ebp+8	1st argument
	ebp+4	return address
	ebp	old ebp value
	ebp-4	1st local variable
	...	
	ebp-n*4 = esp	current stack pointer

Stack smashing

```
void a()
{
  int32_t t[1];
  t[2] = 0xbad;
}
```

Stack smashing

```
void a()
{
    int32_t t[1];
    t[2] = 0xbad;
}
```

...	previous frame	
ebp+4	return address	0x12344321
ebp	old ebp value	0xffffbeef
ebp-4	t[0]	0

...	previous frame	
ebp+4	return address	0xbad
ebp	old ebp value	0xffffbeef
ebp-4	t[0]	0

Introduction

(Some) Dark corners of C++

C++ is complex

r/cpp · Posted by u/eyun89 11 days ago

Just started learning C++ coming from Python and just realized that that Python has been my programming mama, doing my laundry and spoon-feeding me all this time.

Pointer

Can point to either to:

- nothing (nullptr)
- anything valid
- invalid object
- innacessible memory on other device

Reference

Can point to:

- Valid object
- Invalid object

Behave like value if assigned to

SSO + C strings

SSO + C strings

```
// Store pointer in internal structure
// Return handle
extern "C" unsigned c_api(char const *name);
// Use the name somewhere
extern "C" void c_api2(unsigned handle);
```

SSO + C strings

```
context call(std::string&& prefix, std::string&& name) {
  std::string s_v{prefix + "_" + name};
  auto handle{c_api(s_v.c_str())};
  context c{std::move(s_v), handle};
  return c;
}
```

SSO + C strings

```
struct context {
  context(std::string&& name, unsigned handle)
    : name_{name}, handle_{handle} {}
  context(context const &) = delete;
  context& operator=(context const &) = delete;
  context(context&&) = default;
  context& operator=(context&&) = default;
  ~context() = default;
  std::string const name_;
  unsigned const handle_;
};
```

SSO + C strings

```
int main() {
  auto c{call("p", "n")};
  c_api2(c.handle_);
}
```

SSO + C strings

```
context call(std::string&& prefix, std::string&& name) {
  std::string s_v{prefix + "_" + name};
  auto handle{c_api(s_v.c_str())};
  context c{std::move(s_v), handle};
  return c;
}
```

String null termination

String null termination

C style string ✓ / ✗

std::string ✓

std::string_view ✓ / ✗

Implicit constructors and assign operators

Compiler implicitly does

You do	default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
any constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
destructor	defaulted	user declared	defaulted	defaulted	not declared	not declared
copy constructor	not declared	defaulted	user declared	defaulted	not declared	not declared
copy assignment	defaulted	defaulted	defaulted	user declared	not declared	not declared
move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

Based on Howard Hinnant Accu 2014 slides `https://accu.org/content/conf2014/Howard_Hinnant_Accu_2014.pdf`.

Overriding new and delete

The library provides default implementation for the global allocation and deallocation functions

Any definition of global allocation / deallocation function in C++ program replaces default one

Allocation and deallocation functions may also be declared and defined for any class

Flavours of allocation and deallocation functions change between standard versions

Flavours of operator new

replaceable allocation functions
`[[nodiscard]]` (since C++20)

`void* operator new  ( std::size_t count );`	(1)	
`void* operator new[]( std::size_t count );`	(2)	
`void* operator new  ( std::size_t count, std::align_val_t al);`	(3)	(since C++17)
`void* operator new[]( std::size_t count, std::align_val_t al);`	(4)	(since C++17)

replaceable non-throwing allocation functions
`[[nodiscard]]` (since C++20)

`void* operator new  ( std::size_t count, const std::nothrow_t& tag);`	(5)	
`void* operator new[]( std::size_t count, const std::nothrow_t& tag);`	(6)	
`void* operator new  ( std::size_t count, std::align_val_t al, const std::nothrow_t&);`	(7)	(since C++17)
`void* operator new[]( std::size_t count, std::align_val_t al, const std::nothrow_t&);`	(8)	(since C++17)

non-allocating placement allocation functions
`[[nodiscard]]` (since C++20)

`void* operator new  ( std::size_t count, void* ptr );`	(9)	
`void* operator new[]( std::size_t count, void* ptr );`	(10)	

user-defined placement allocation functions

`void* operator new  ( std::size_t count, user-defined-args... );`	(11)	
`void* operator new[]( std::size_t count, user-defined-args... );`	(12)	
`void* operator new  ( std::size_t count, std::align_val_t al, user-defined-args... );`	(13)	(since C++17)
`void* operator new[]( std::size_t count, std::align_val_t al, user-defined-args... );`	(14)	(since C++17)

class-specific allocation functions

`void* T::operator new  ( std::size_t count );`	(15)	
`void* T::operator new[]( std::size_t count );`	(16)	
`void* T::operator new  ( std::size_t count, std::align_val_t al );`	(17)	(since C++17)
`void* T::operator new[]( std::size_t count, std::align_val_t al );`	(18)	(since C++17)

class-specific placement allocation functions

`void* T::operator new  ( std::size_t count, user-defined-args... );`	(19)	
`void* T::operator new[]( std::size_t count, user-defined-args... );`	(20)	
`void* T::operator new  ( std::size_t count, std::align_val_t al, user-defined-args... );`	(21)	(since C++17)
`void* T::operator new[]( std::size_t count, std::align_val_t al, user-defined-args... );`	(22)	(since C++17)

Flavours of operator delete

replaceable usual deallocation functions		
`void operator delete ( void* ptr ) throw();`	(1)	(until C++11)
`void operator delete ( void* ptr ) noexcept;`		(since C++11)
`void operator delete[]( void* ptr ) throw();`	(2)	(until C++11)
`void operator delete[]( void* ptr ) noexcept;`		(since C++11)
`void operator delete ( void* ptr, std::align_val_t al ) noexcept;`	(3)	(since C++17)
`void operator delete[]( void* ptr, std::align_val_t al ) noexcept;`	(4)	(since C++17)
`void operator delete ( void* ptr, std::size_t sz ) noexcept;`	(5)	(since C++14)
`void operator delete[]( void* ptr, std::size_t sz ) noexcept;`	(6)	(since C++14)
`void operator delete ( void* ptr, std::size_t sz, std::align_val_t al ) noexcept;`	(7)	(since C++17)
`void operator delete[]( void* ptr, std::size_t sz, std::align_val_t al ) noexcept;`	(8)	(since C++17)
replaceable placement deallocation functions		
`void operator delete ( void* ptr, const std::nothrow_t& tag ) throw();`	(9)	(until C++11)
`void operator delete ( void* ptr, const std::nothrow_t& tag ) noexcept;`		(since C++11)
`void operator delete[]( void* ptr, const std::nothrow_t& tag ) throw();`	(10)	(until C++11)
`void operator delete[]( void* ptr, const std::nothrow_t& tag ) noexcept;`		(since C++11)
`void operator delete ( void* ptr, std::align_val_t al, const std::nothrow_t& tag ) noexcept;`	(11)	(since C++17)
`void operator delete[]( void* ptr, std::align_val_t al, const std::nothrow_t& tag ) noexcept;`	(12)	(since C++17)
non-allocating placement deallocation functions		
`void operator delete ( void* ptr, void* place ) throw();`	(13)	(until C++11)
`void operator delete ( void* ptr, void* place ) noexcept;`		(since C++11)
`void operator delete[]( void* ptr, void* place ) throw();`	(14)	(until C++11)
`void operator delete[]( void* ptr, void* place ) noexcept;`		(since C++11)
user-defined placement deallocation functions		
`void operator delete ( void* ptr, args... );`	(15)	
`void operator delete[]( void* ptr, args... );`	(16)	
class-specific usual deallocation functions		
`void T::operator delete ( void* ptr );`	(17)	
`void T::operator delete[]( void* ptr );`	(18)	
`void T::operator delete ( void* ptr, std::align_val_t al );`	(19)	(since C++17)
`void T::operator delete[]( void* ptr, std::align_val_t al );`	(20)	(since C++17)
`void T::operator delete ( void* ptr, std::size_t sz );`	(21)	
`void T::operator delete[]( void* ptr, std::size_t sz );`	(22)	
`void T::operator delete ( void* ptr, std::size_t sz, std::align_val_t al );`	(23)	(since C++17)
`void T::operator delete[]( void* ptr, std::size_t sz, std::align_val_t al );`	(24)	(since C++17)
class-specific placement deallocation functions		
`void T::operator delete ( void* ptr, args... );`	(25)	
`void T::operator delete[]( void* ptr, args... );`	(26)	
class-specific destroying deallocation functions		
`void T::operator delete(T* ptr, std::destroying_delete_t);`	(27)	(since C++20)
`void T::operator delete(T* ptr, std::destroying_delete_t, std::align_val_t al);`	(28)	(since C++20)
`void T::operator delete(T* ptr, std::destroying_delete_t, std::size_t sz);`	(29)	(since C++20)
`void T::operator delete(T* ptr, std::destroying_delete_t, std::size_t sz, std::align_val_t al);`	(30)	(since C++20)

Introduction

Knowing your hardware

Memory accesses and alignment

- Data can be read and write from memory in different size chunks
  - Depending on architecture / memory type
  - Memory accessess are always word size aligned
    - Shorter accesses are translated to longer ones under the hood
      - E.g. *first / last byte enable* in PCI TLBs
  - Currently word size is almost never less then 4 bytes, and always power of 2

Cache

- Most memory accesses are not directly accessing memory, but go through cache
- Caches may be handled automatically or manually
  - Coherent cache - no need to care
  - Incoherent cache - need to manually flush / invalidate to ensure data is valid
- Need to be aware how used architecture is handling caches

A tale of an impossible bug: big.LITTLE and caching

Rodrigo Kumpera and Bernhard Urban September 12, 2016

Solutions

Solutions

What C++ gives for preventing data corruptions

Defensive checks

Check if not null

Non-zero size / length check

Bounds checking

Secure memory access functions

Available either in Visual Studio or C11 compatible environment

memcpy_s

memset_s

strcpy_s

strncpy_s

memmove_s

getenv_s

qsort_s

std:: memory manipulation algorithms

std::copy

std::copy_n

std::fill

Smart pointers

Explicitly showing ownership and lifetime

Automatically deleted when all owners go out of scope

std::unique_ptr - single owner, lives as long as owner scope

std::shared_ptr - multiple owners, lives as long as all owners scope

std::weak_ptr - non owning, reference to std::shared_ptr with validity check

Range checked access to containers

Accessing containers with operator[] does not check bounds

- In particular, for std::array or std::vector memory outside container can be accessed

C++11 added .at() function that throws if index is out of range

Still little troublesome if -fno-exceptions is set

Solutions

What compiler and tools give

Static analyzers

Analyzing source code to find potential bugs

Some open source examples:

- Clang analyzer
- Cppcheck
- BLAST

Some proprietary examples:

- Visual Studio
- PVS-Studio
- Coverity
- Klocwork

Importance of static analysis

GGribkov July 15, 2019 at 04:30 PM

Errors that static code analysis does not find because it is not used

`https://habr.com/en/company/pvs-studio/blog/460121/`

Useful warnings to eliminate some leaks

-Wall -Wextra -Wpedantic

-Wuninitialized / -Wmaybe-uninitialized

-Wnon-virtual-dtor

-Weffc++

Any other warnings:

`https://gcc.gnu.org/onlinedocs/gcc-9.1.0/gcc/Warning-Options.html`

`https://gcc.gnu.org/onlinedocs/gcc-9.1.0/gcc/C_002b_002b-Dialect-Options.html`

`https://clang.llvm.org/docs/DiagnosticsReference.html`

Clang's consumed annotations

Class can marked with consumable attribute, and be in: unconsumed, consumed, or unknown state.

The compiler can generate warning (with -Wconsumed flag) if object is in unwanted state.

Feature in development and may change as in Clang's documentation.

Clang's consumed annotations (2)

```
extern void *OpenResource();
extern void CloseResource(void *);
extern void ExecuteResource(void *, int);
```

Clang's consumed annotations (3) - Example class

```
struct resource {
    resource() : handle_{OpenResource()}{}
    resource(resource const&) = delete;
    resource& operator=(resource const&) = delete;
    resource(resource&& other) : handle_{other.handle_} {
        other.invalidate();
    }
    resource& operator=(resource&& other) {
        this->handle_ = other.handle_;
        other.invalidate();
        return *this;
    }
    ~resource() { if (handle_) CloseResource(handle_); }
    void execute(int v) { ExecuteResource(handle_, v); }
private:
    void invalidate() { handle_ = nullptr; }
    void *handle_;
};
```

Clang's consumed annotations (4.1) - Using annotations

```
struct [[clang::consumable(unconsumed)]] resource {
```

Clang's consumed annotations (4.2) - Using annotations

```
[[clang::return_typestate(unconsumed)]]
resource() : handle_{OpenResource()}{}
```

Clang's consumed annotations (4.3) - Using annotations

```
[[clang::callable_when(unconsumed)]]
void execute(int v) { ExecuteResource(handle_, v); }
```

Clang's consumed annotations (4.4) - Using annotations

```
[[clang::set_typestate(consumed)]]
void invalidate() { handle_ = nullptr; }
```

Clang's consumed annotations (4.full) - Using annotations

```
struct [[clang::consumable(unconsumed)]] resource {
    [[clang::return_typestate(unconsumed)]]
    resource() : handle_{OpenResource()}{}
    resource(resource const&) = delete;
    resource& operator=(resource const&) = delete;
    resource(resource&& other) : handle_{other.handle_} {
        other.invalidate();
    }
    resource& operator=(resource&& other) {
        if (&other == this) return *this;
        this->handle_ = other.handle_;
        other.invalidate();
        return *this;
    }
    ~resource() { if (handle_) CloseResource(handle_); }
    [[clang::callable_when(unconsumed)]]
    void execute(int v) { ExecuteResource(handle_, v); }
private:
    [[clang::set_typestate(consumed)]]
    void invalidate() { handle_ = nullptr; }
    void *handle_;
};
```

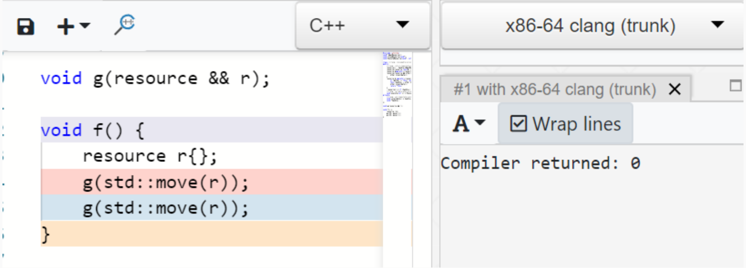
Clang's consumed annotations (5) - Compile with -Wconsumed

```
resource r{};
g(std::move(r));
r.execute(2);
```

warning: invalid invocation of method 'execute' on object 'r' while it is in the 'consumed' state [-Wconsumed]
r.execute(2);

Clang's consumed annotations (6) - not always work

With clang-trunk as of 14.07.2019



Sanitizers

Fast runtime memory issues detector

Can be built into binary with clang and gcc

Different types of sanitizers available:

1. AddressSanitizer - to detect user after free and buffer overlow
2. ThreadSanitizer - to detect data races
3. MemorySanitizer - to detect uninitialized memory usage
4. UndefinedBehaviorSanitizer - to detect some UB
5. Stack protector - to detect stack smashing

Compiling with sanitizer support

- Binary has to be compiled with -O0 to prevent false positives resulting from optimizations
- Binary has to be compiled with -g to have symbols
- Use one of flags:
  - *-fsanitize=[...]* to enable [...] sanitizer
    - More in docs:
    - `https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html`
    - (*)This is not necessary to execute with sanitizers enabled
  - *-fstack-protector[-(all/strong/explicit)]*

Running with sanitizer support

If compiled with flag, simply run executable.

If compiled without flag use *LD_PRELOAD=libasan.so*

```cpp
#include <cstdlib>
int main()
{
  malloc(7);
}
```

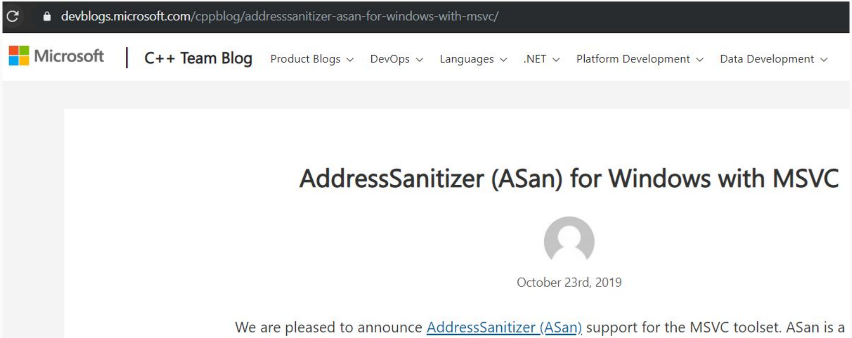
```
LD_PRELOAD=/usr/lib/x86_64-linux-gnu/libasan.so.4 ./testasan

=================================================================
==12309==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 7 byte(s) in 1 object(s) allocated from:
    #0 0x7f73fe91cb50 in __interceptor_malloc (/usr/lib/x86_64-linux-gnu/l
ibasan.so.4+0xdeb50)
    #1 0x55be95bc4657 in main /home/mnowak/test/testasan.cpp:4
    #2 0x7f73fe46eb96 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.
6+0x21b96)

SUMMARY: AddressSanitizer: 7 byte(s) leaked in 1 allocation(s).
```

So cool? MSVC also has it!



Runtime analyzers

Valgrind

Electric fence / DUMA

Microsoft Application Verifier / Static Driver Verifier

Solutions

Manual corruption detection techniques

Memory tagging

Reserve additional space before and after memory buffer

Fill it with unique value (canary) on allocation

Check if canary was not overwritten before deallocation

Allocation tracking

1. On every allocation record address and size of buffer.
2. On every deallocation delete recorded entry.
3. On library unload / process exit check if anything left on the list.

Memory pools

All allocations and deallocations go through custom functions instead of default new / delete / malloc / free

Can be made using:

- Overriding default new / delete operators
- Placement new and custom deleters
- Custom allocator
- std::pmr

Example - allocation tracking with pmr (1)

```
class pmr_memory_tracker : public std::pmr::memory_resource
{
public:
  explicit pmr_memory_tracker(std::pmr::memory_resource *const upstream) : upstream_(upstream) {}
  pmr_memory_tracker(pmr_memory_tracker const &) = delete;
  pmr_memory_tracker &operator=(pmr_memory_tracker const &) = delete;
  ~pmr_memory_tracker() {
    for (auto &[ptr, p] : allocation_map_)
    {
      std::cerr << "Unallocated " << ptr << " size " << p.first << " alignment " << p.second << std::endl;
    }
  }
  void *do_allocate(std::size_t bytes, std::size_t alignment) override {
    void *ptr = upstream_->allocate(bytes, alignment);
    allocation_map_[(uint64_t)ptr] = {bytes, alignment};
    return ptr;
  }
  void do_deallocate(void *ptr, std::size_t bytes, std::size_t alignment) override {
    allocation_map_.erase((uint64_t)ptr);
    upstream_->deallocate(ptr, bytes, alignment);
  }
  bool do_is_equal(memory_resource const &other) const noexcept override { return true; }
private:
  std::pmr::memory_resource *const upstream_;
  std::unordered_map<uint64_t, std::pair<size_t, size_t>> allocation_map_;
};
```

Example - allocation tracking with pmr (2)

```
int main()
{
  struct t
  {
  };
  pmr_memory_tracker tracker(std::pmr::get_default_resource());
  auto ptr = new (tracker.allocate(sizeof(t))) t;
  ptr->~t();
  //tracker.deallocate(ptr, sizeof(t));
}
```

```
Program stderr
Unallocated 23985776 size 1 alignment 16
```

Hand crafted stack smashing detector

Put canary value on stack and verify it after function call

Need to have optimizations and inlining disabled

Example works for both standalone functions and class members, need adjustments for std::function and lambdas

Hand crafted stack smashing detector (1)

```
void a()
{
  volatile int32_t c[100000];
  int const i = 12;
  std::cout << std::hex << c[100000 + i] << "\n";
  c[100000 + i] = 1;
}

struct asd
{
  int q(int &, int &&) { return 0; }
  static std::string p(int v) { return std::to_string(v); }
};

int main()
{
  func_caller(a)();

  asd a1;
  int s = 2;
  func_caller_o(a1, q)(s, std::move(s));
  func_caller(asd::p)(10);
}
```

Hand crafted stack smashing detector (2)

```
#define func_caller_o(i, f) stack_smashing_detector<decltype(i)>{i}(&decltype(i)::f)
#define func_caller(f) stack_smashing_detector<void>()(f)
```

Hand crafted stack smashing detector (3)

```
template <class W>
struct stack_smashing_detector
{
  template <typename Wi = W,
            typename = typename std::enable_if_t<std::is_void_v<Wi>>>
  stack_smashing_detector() : object_{nullptr} {}
  template <typename Wi = W,
            typename = typename std::enable_if_t <!std::is_void_v<Wi>>,
            typename = Wi>
  stack_smashing_detector(Wi &w) : object_{&w} {}

...

private:
  W *object_;
};
```

Hand crafted stack smashing detector (4)

```
template <typename R, typename... Args>
caller<R, Args...> operator()(
    typename function_t<W>::template type<R, Args...> func)
{
  return caller{*this, func};
}
```

Hand crafted stack smashing detector (5)

```
template <typename W>
struct function_t
{
  template <typename R, typename... Args>
  using type = R (W::*)(Args...);
};

template <>
struct function_t<void>
{
  template <typename R, typename... Args>
  using type = R (*)(Args...);
};
```

Hand crafted stack smashing detector (6)

```
template <typename R, typename... Args>
struct caller
{
  using function_concrete_t =
      typename function_t<W>::template type<R, Args...>;

private:
  function_concrete_t func_;
  stack_smashing_detector &parent_;

public:
  caller(stack_smashing_detector &parent, function_concrete_t func)
      : parent_{parent}, func_{func} {}
```

Hand crafted stack smashing detector (7)

```
    R __attribute__((noinline, optimize("O0"))) operator()(Args... args)
    {
      uint32_t __CANARY__[4]{0xDEADBEEF, 0xDEADBEEF, 0xDEADBEEF, 0xDEADBEEF};
#define CANARY_CHECK                                                    \
  if (__CANARY__[0] != 0xDEADBEEF || __CANARY__[1] != 0xDEADBEEF || \
      __CANARY__[2] != 0xDEADBEEF || __CANARY__[3] != 0xDEADBEEF)   \
  {                                                                 \
    std::cout << "Error" << std::endl;                              \
    std::abort();                                                   \
  }
```

Hand crafted stack smashing detector (8)

```
if constexpr (std::is_void_v<R>)
{
  if constexpr (std::is_same_v<void, W>)
  {
    func_(std::forward<Args>(args)...);
  }
  else
  {
    (parent_.object_->*func_)(std::forward<Args>(args)...);
  }
  CANARY_CHECK;
}
```

Hand crafted stack smashing detector (9)

```
      else
      {
        if constexpr (std::is_same_v<void, W>)
        {
          auto &&retval{func_(std::forward<Args>(args)...)};
          CANARY_CHECK;
          return retval;
        }
        else
        {
          auto &&retval{(parent_.object_->*func_)(std::forward<Args>(args)...)};
          CANARY_CHECK;
          return retval;
        }
      }
    }
  };
```

Have second pair of eyes look at your code

May see something you have missed

May just resolved similar issue

Will have unbiased, out of the box view

Thanks for your attention!

Mateusz Nowak

https://www.linkedin.com/in/mateusz-nowak-qq/

https://github.com/noqqaqq

Feedback is appreciated ☺