C++ now

ALSan (Attachable Leak Sanitizer)

Bojun Seo

2024

About me

I am a LG Electronics software engineer committed to memory leak problems

Github: https://github.com/Bojun-Seo

LinkedIn: https://www.linkedin.com/in/bojun-seo-5361b31a4

Email: bojun.seo@lge.com

Contents

- Motivation
  - Memory leak matters in C++?
  - Memory leak detection tools and limitations
    1. Test again
    2. Report on process exit
    3. Heisenbug
- Solution
  - Background: eBPF, uprobe
  - ALSan(Attachable Leak Sanitizer)
  - Demo

Contents

- Motivation
  - **Memory leak matters in C++?**
  - Memory leak detection tools and limitations
    1. Test again
    2. Report on process exit
    3. Heisenbug
- Solution
  - Background: eBPF, uprobe
  - ALSan(Attachable Leak Sanitizer)
  - Demo

Which of these do you find frustrating about C++ dev?	*Major %*
Managing libraries my application depends on	45 %
Build times	43 %
Setting up a CI pipeline from scratch	30 %
Managing CMake projects	30 %
Concurrency safety: Races, deadlocks, performance bottlenecks	27 %
Setting up a dev env from scratch	26 %
Parallelism support	23 %
Managing Makefiles	20 %
Memory safety: Bounds safety issues	20 %
Memory safety: Use-after-delete/free	20 %
Debugging issues in my code	18 %
Managing MSBuild projects	16 %
Unicode, internationalization, and localization	16 %
Security issues: disclosure, vulnerabilities, exploits	12 %
Type safety: Using an object as the wrong type	12 %
Memory safety: Memory leaks	12 %
Moving existing code to the latest language standard	9 %

Source: C++ Now 2024 opening keynote, C++ painkillers, Anastasia Kazakova

Which of these do you find frustrating about C++ dev?	*Major %*
Managing libraries my application depends on	45 %
Build times	43 %
Setting up a CI pipeline from scratch	30 %
Managing CMake projects	30 %
Concurrency safety: Races, deadlocks, performance bottlenecks	27 %
Setting up a dev env from scratch	26 %
Parallelism support	23 %
Managing Makefiles	20 %
Memory safety: Bounds safety issues	20 %
Memory safety: Use-after-delete/free	20 %
Debugging issues in my code	18 %
Managing MSBuild projects	16 %
Unicode, internationalization, and localization	16 %
Security issues: disclosure, vulnerabilities, exploits	12 %
Type safety: Using an object as the wrong type	12 %
Memory safety: Memory leaks	12 %
Moving existing code to the latest language standard	9 %

Source: C++ Now 2024 opening keynote, C++ painkillers, Anastasia Kazakova

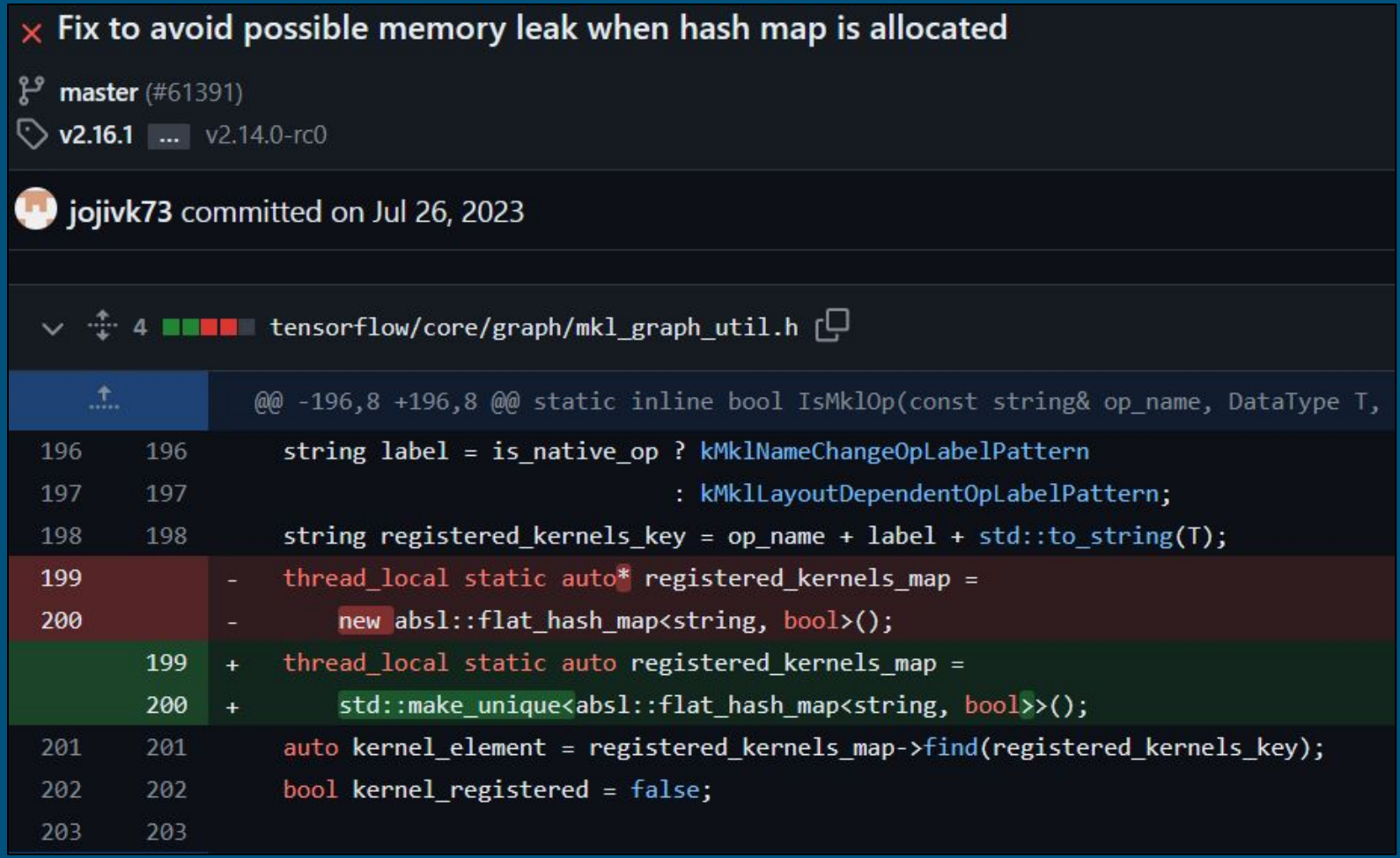
Memory leak matters in C++?

- Using smart pointer correctly → no unreachable memory leak

Memory leak matters in C++?

- Using smart pointer correctly → no unreachable memory leak

- But, sometimes
  - Use raw pointer
  - Use smart pointer incorrectly
  - Use C library incorrectly



source: https://github.com/tensorflow/tensorflow/pull/61391

Memory leak matters in C++?

- Using smart pointer correctly → no unreachable memory leak

- But, sometimes
  - Use raw pointer
  - Use smart pointer incorrectly
  - Use C library incorrectly

- Inevitable issue even for expert C++ developers

Contents

- Motivation
  - Memory leak matters in C++?
  - **Memory leak detection tools and limitations**
    1. Test again
    2. Report on process exit
    3. Heisenbug
- Solution
  - Background: eBPF, uprobe
  - ALSan(Attachable Leak Sanitizer)
  - Demo

Memory leak detection tools

- Valgrind
  - Runs target program on its own VM
  - Detect memory issues including leak but very slow(5 - 100 times)[1]
- AddressSanitizer - ASan
  - Native run(need recompile)
  - Detect memory issues including leak but slow(about 2 times)[2]

[1]: https://valgrind.org/info/about.html

[2]: https://github.com/google/sanitizers/wiki/AddressSanitizerPerformanceNumbers

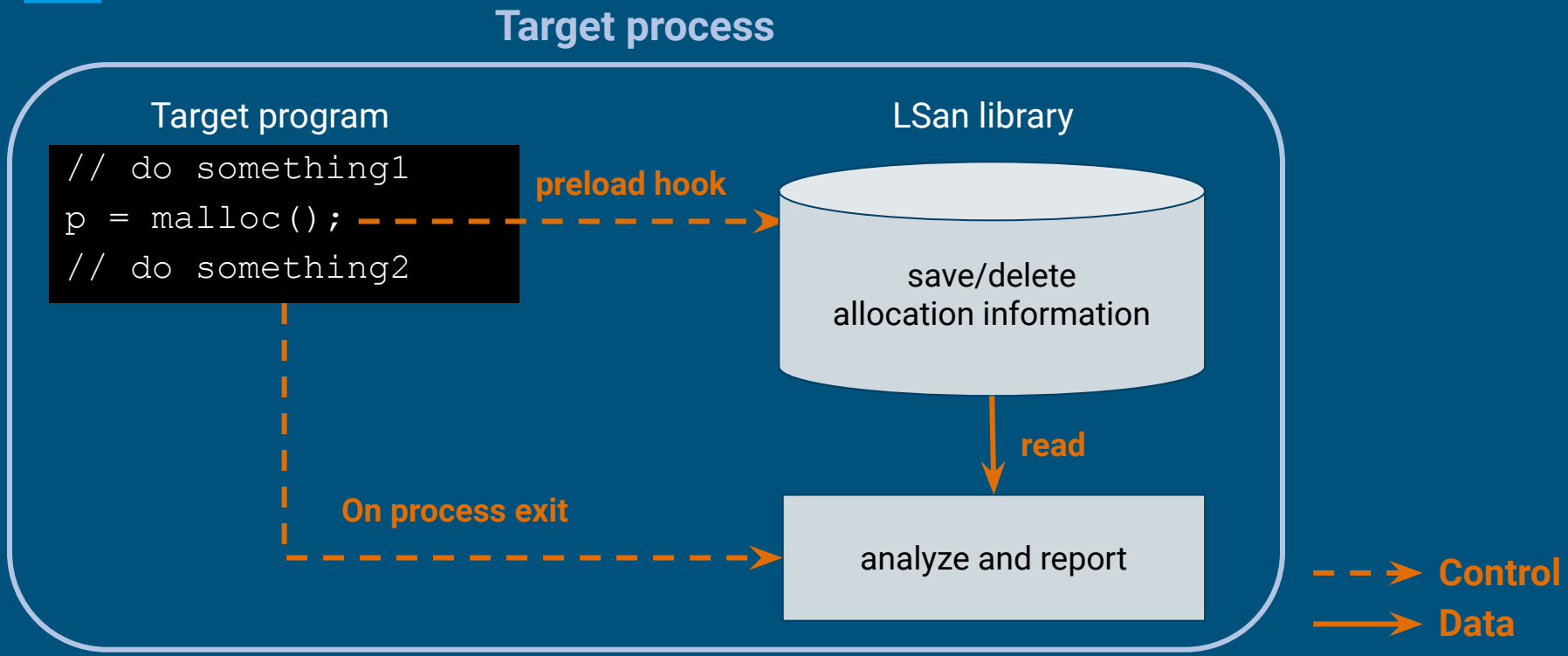
Memory leak detection tools

- Valgrind
  - Runs target program on its own VM
  - Detect memory issues including leak but very slow(5 - 100 times)[1]
- AddressSanitizer - ASan
  - Native run(need recompile)
  - Detect memory issues including leak but slow(about 2 times)[2]
- **LeakSanitizer - LSan**
  - **Native run(library preload)**
  - **Detect memory leak only and not slow**

[1]: https://valgrind.org/info/about.html

[2]: https://github.com/google/sanitizers/wiki/AddressSanitizerPerformanceNumbers

LeakSanitizer overview



How LeakSanitizer works

LSan library is preloaded on target process starting

1. Hooks memory allocator functions
2. Save/delete allocation information to analyze and report
3. Analyze and report unreachable memory

Contents

- Motivation
  - Memory leak matters in C++?
  - Memory leak detection tools and limitations
    1. **Test again**
    2. Report on process exit
    3. Heisenbug
- Solution
  - Background: eBPF, uprobe
  - ALSan(Attachable Leak Sanitizer)
  - Demo

1. Need to test again

LSan library is preloaded on target process starting

1. Need to test again

LSan library is preloaded on target process starting

User need to test again!

1. Need to test again

LSan library is preloaded on target process starting

User need to test again!

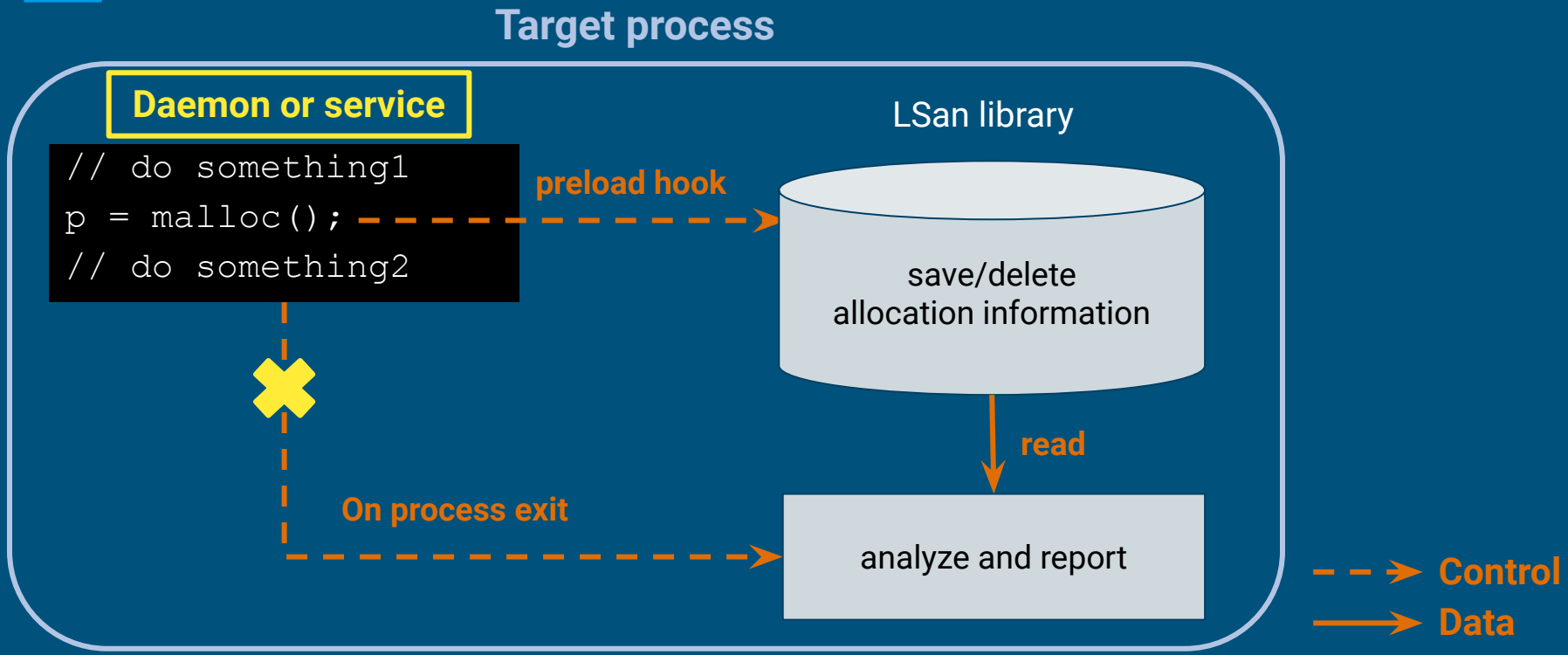
Problem of retest

- Hard to know reproduction scenario
- Take a long time to perform reproduction scenario

Contents

- Motivation
  - Memory leak matters in C++?
  - Memory leak detection tools and limitations
    1. Test again
    2. **Report on process exit**
    3. Heisenbug
- Solution
  - Background: eBPF, uprobe
  - ALSan(Attachable Leak Sanitizer)
  - Demo

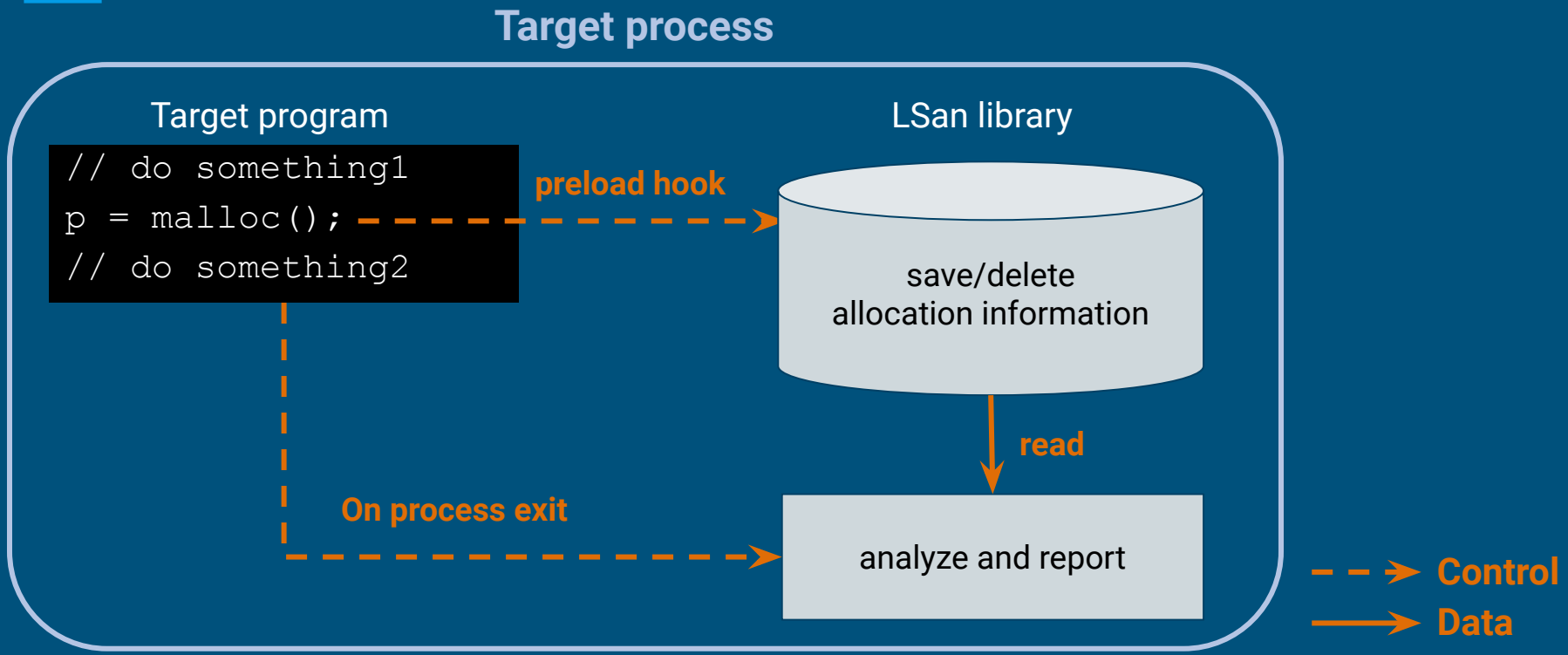
2. Report on process exit



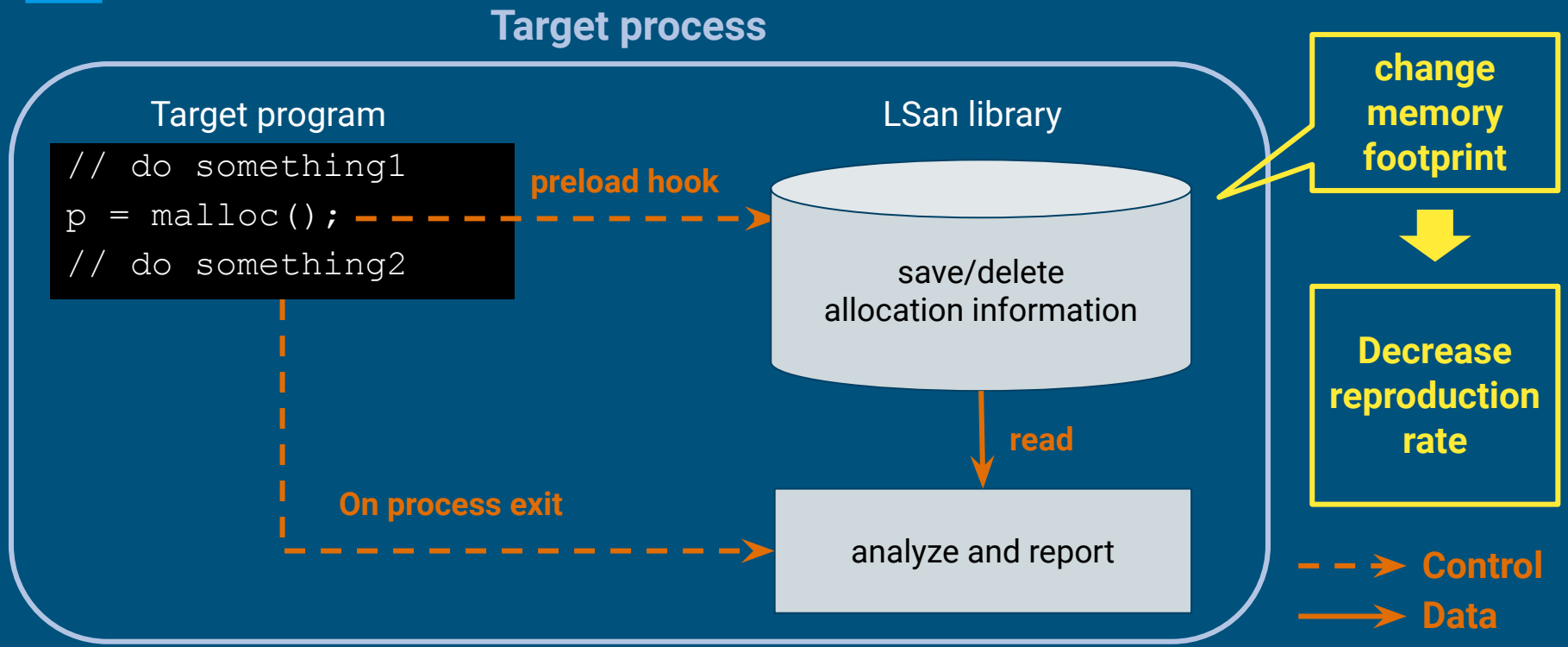
Contents

- Motivation
  - Memory leak matters in C++?
  - Memory leak detection tools and limitations
    1. Test again
    2. Report on process exit
    3. **Heisenbug**
- Solution
  - Background: eBPF, uprobe
  - ALSan(Attachable Leak Sanitizer)
  - Demo

3. Heisenbug


Target process
Target program
// do something1
p = malloc();
// do something2
preload hook
LSan library
save/delete
allocation information
read
On process exit
analyze and report
Control
Data

3. Heisenbug


Target process
Target program
// do something1
p = malloc();
// do something2
preload hook
LSan library
save/delete
allocation information
read
On process exit
analyze and report
change
memory
footprint
Decrease
reproduction
rate
Control
Data

Contents

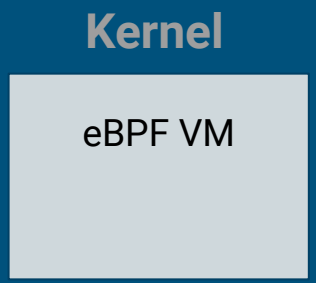
- Motivation
  - Memory leak matters in C++?
  - Memory leak detection tools and limitations
    1. Test again
    2. Report on process exit
    3. Heisenbug
- **Solution**
  - Background: eBPF, uprobe
  - ALSan(Attachable Leak Sanitizer)
  - Demo

Contents

- Motivation
  - Memory leak matters in C++?
  - Memory leak detection tools and limitations
    1. Test again
    2. Report on process exit
    3. Heisenbug
- Solution
  - **Background: eBPF, uprobe**
  - ALSan(Attachable Leak Sanitizer)
  - Demo

eBPF(extended Berkeley Packet Filter)

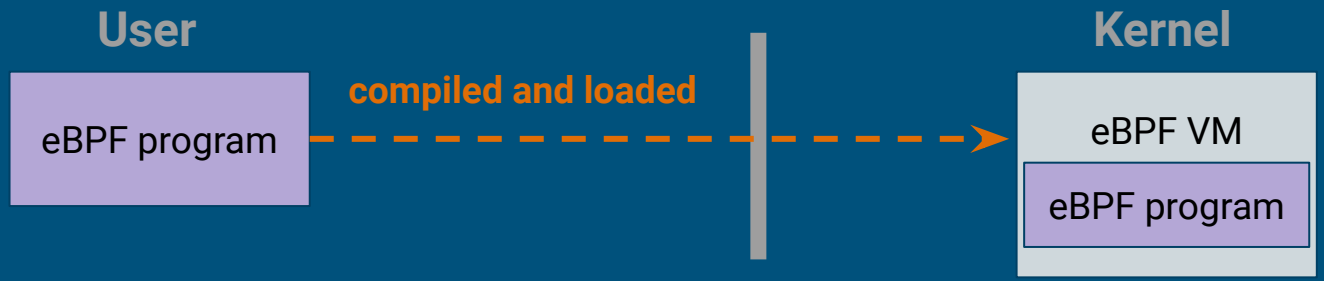
- Virtual machine running inside Linux Kernel



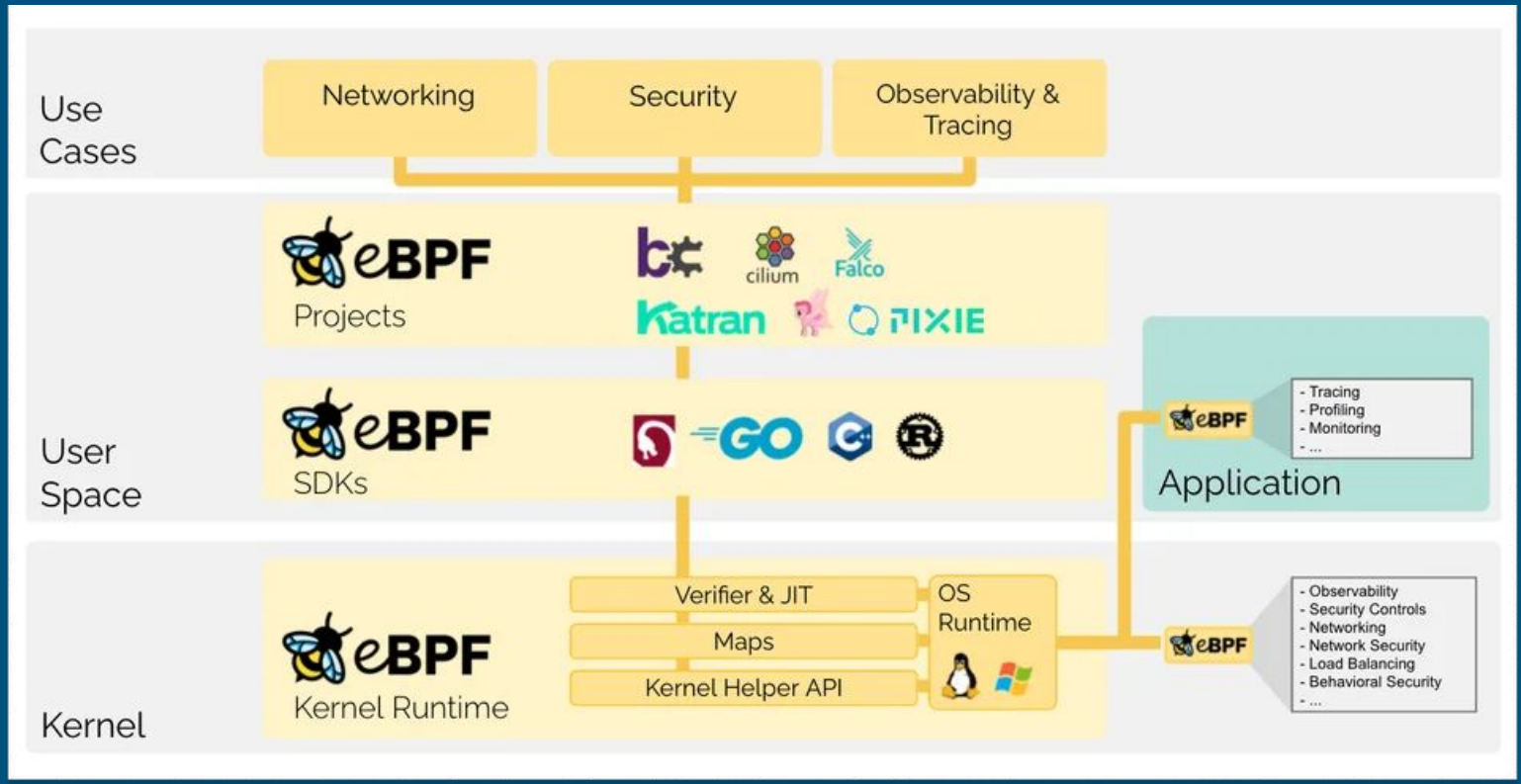
eBPF(extended Berkeley Packet Filter)

- Virtual machine running inside Linux Kernel

- Allow user defined code can be run inside Linux kernel

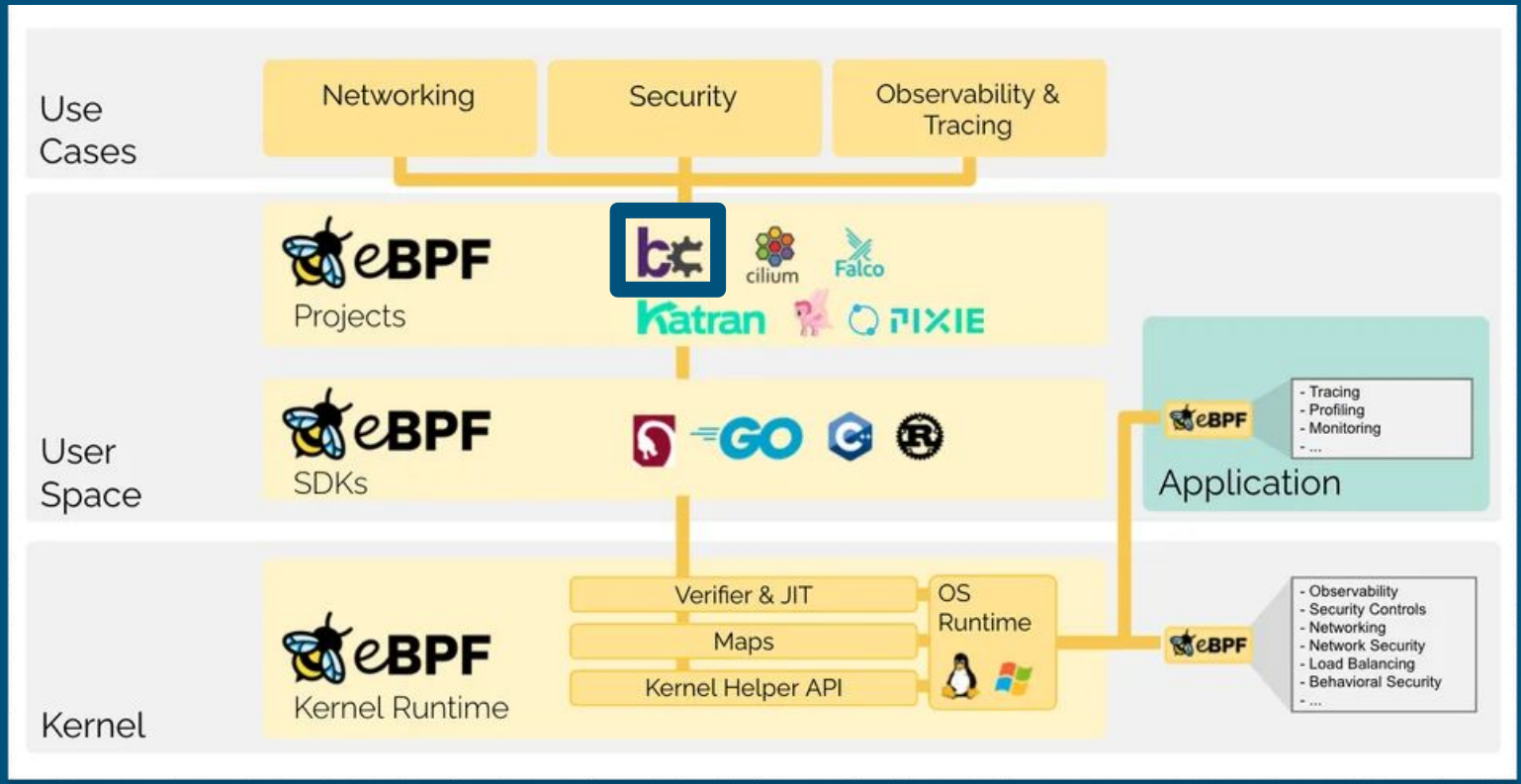


eBPF use cases



Source: https://ebpf.io/static/e293240ecccb9d506587571007c36739/f2674/overview.png

eBPF use cases



Source: https://ebpf.io/static/e293240ecccb9d506587571007c36739/f2674/overview.png

Uprobe

- Provide dynamic tracing for user-level applications

Uprobe

- Provide dynamic tracing for user-level applications

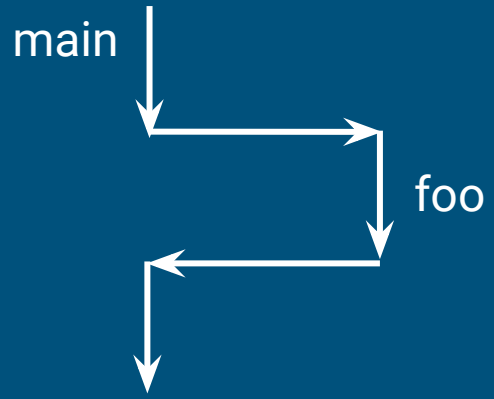
Able to hook user function

Uprobe

- Provide dynamic tracing for user-level applications

Able to hook user function

```
int main() {
  foo();
}
```

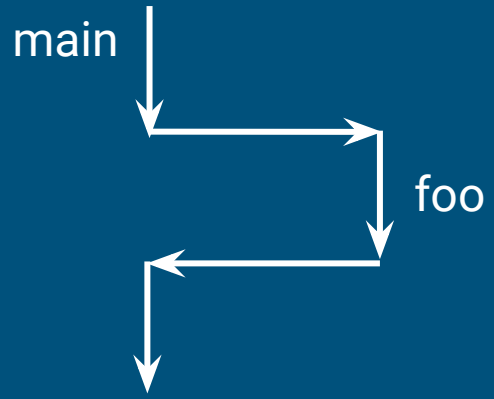


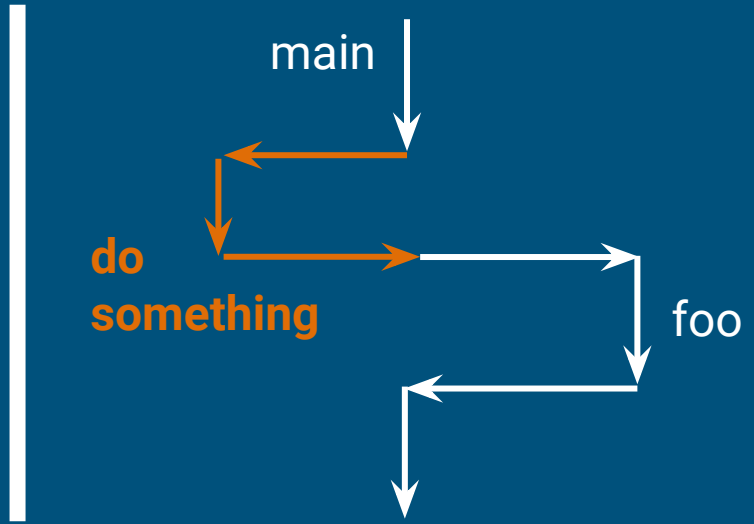
Uprobe

- Provide dynamic tracing for user-level applications

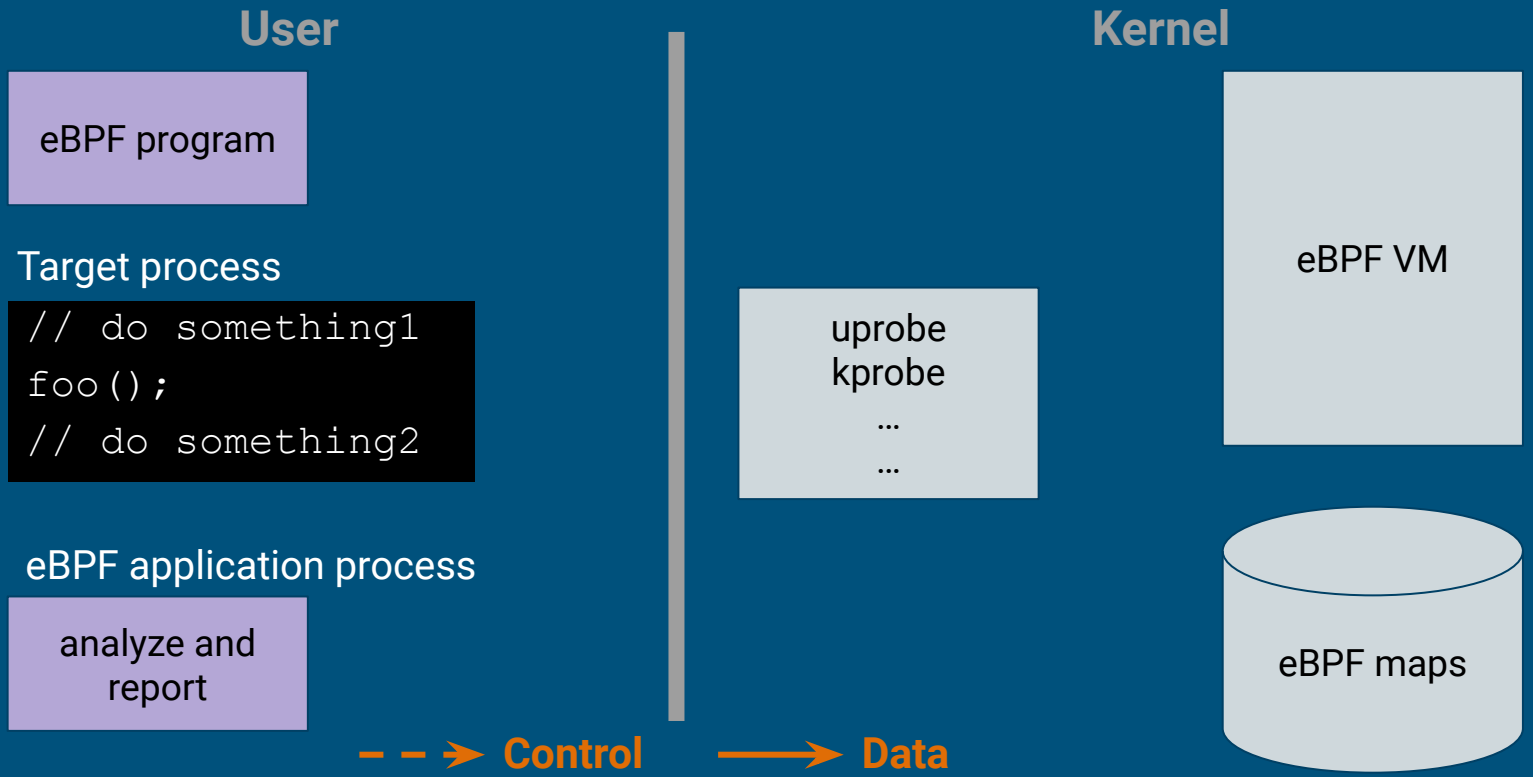
Able to hook user function

```
int main() {
  foo();
}
```

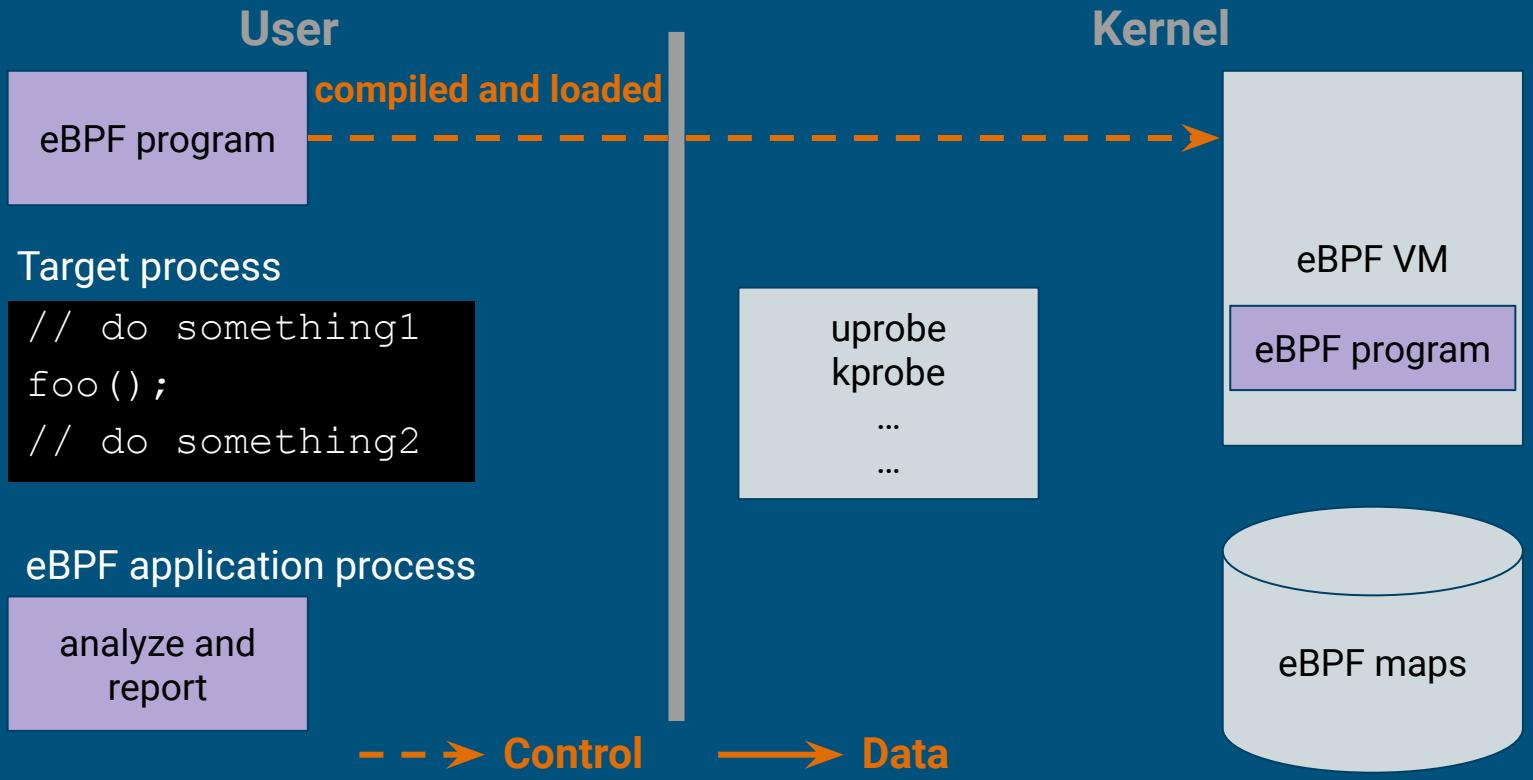




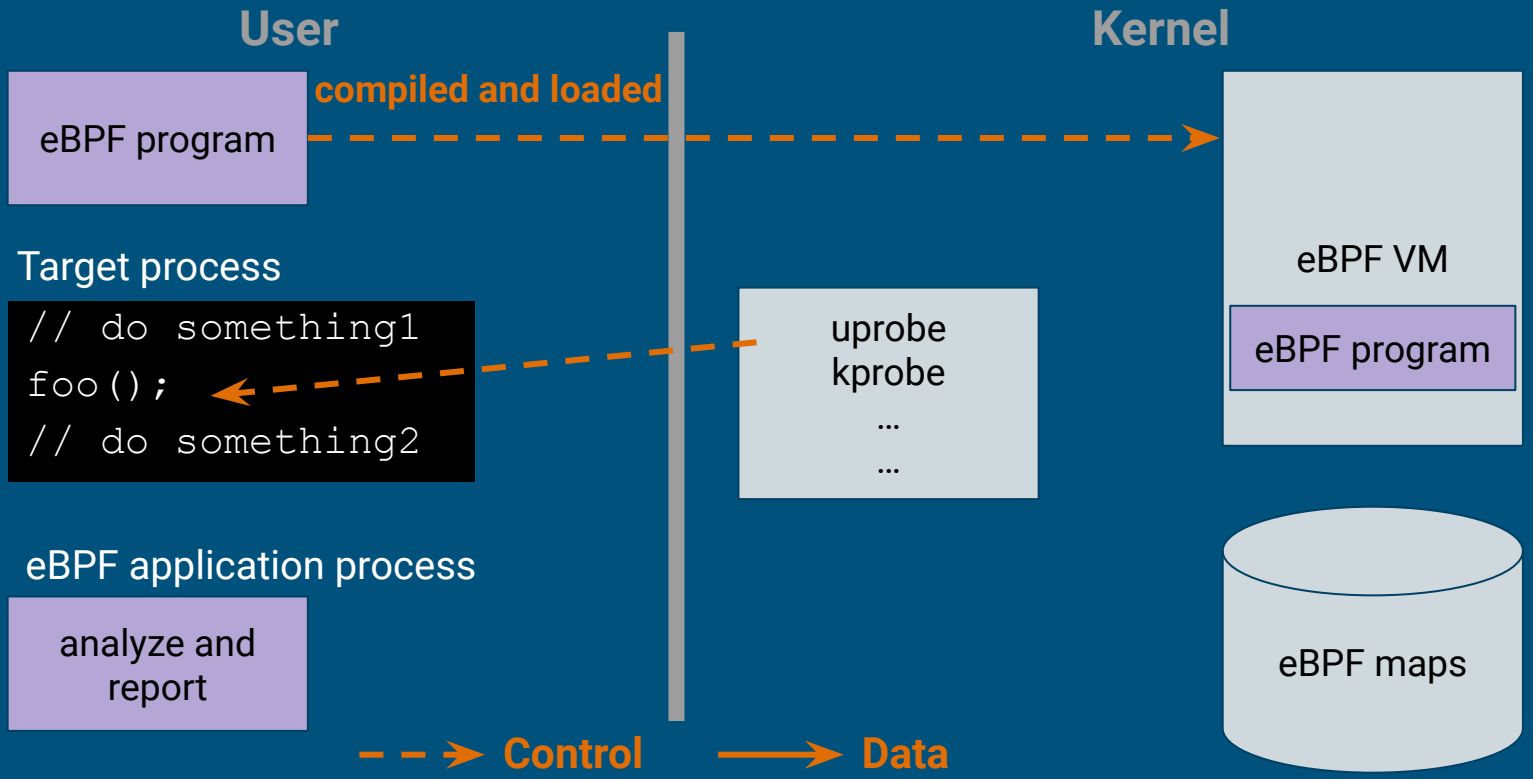
eBPF application overview



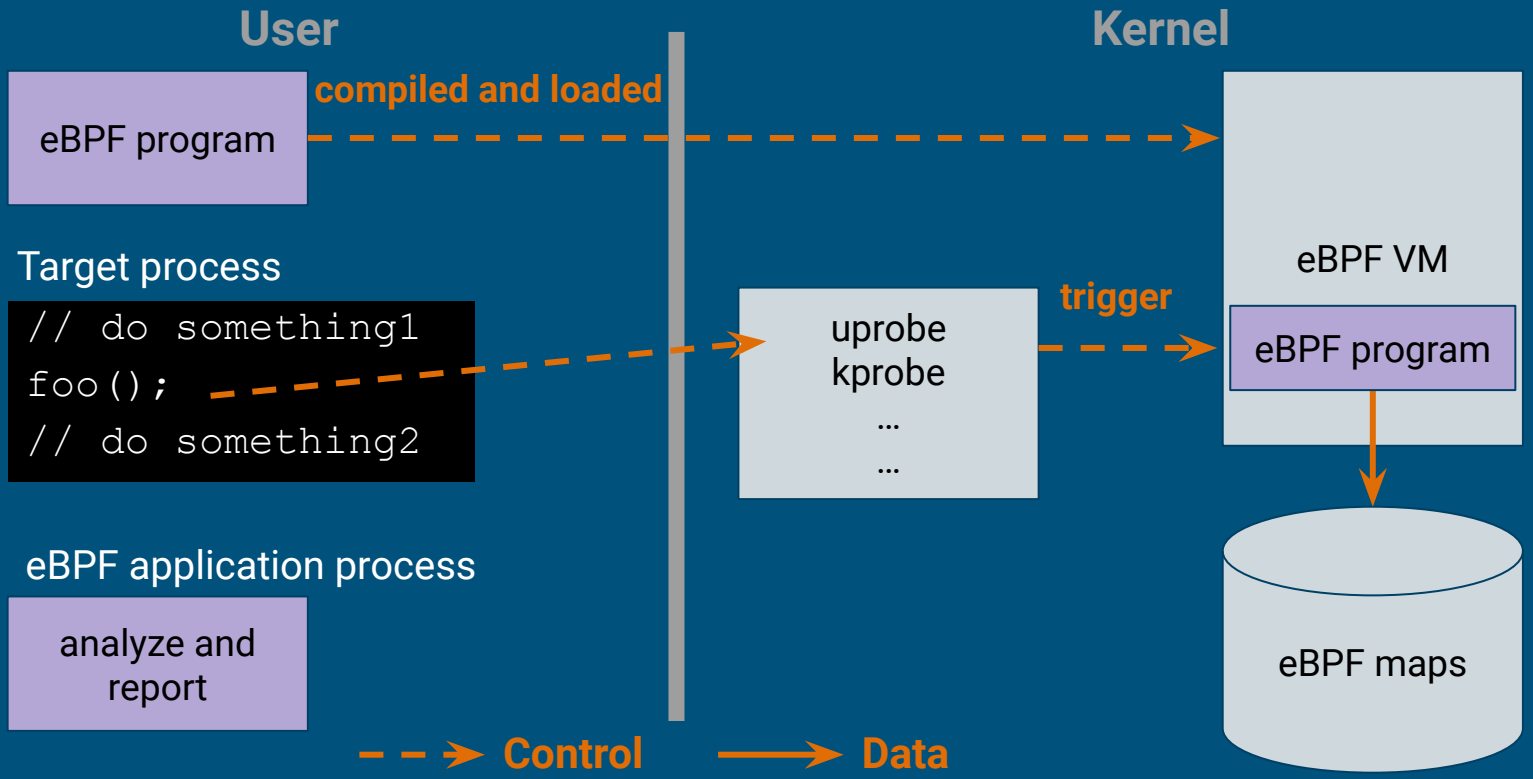
eBPF application overview



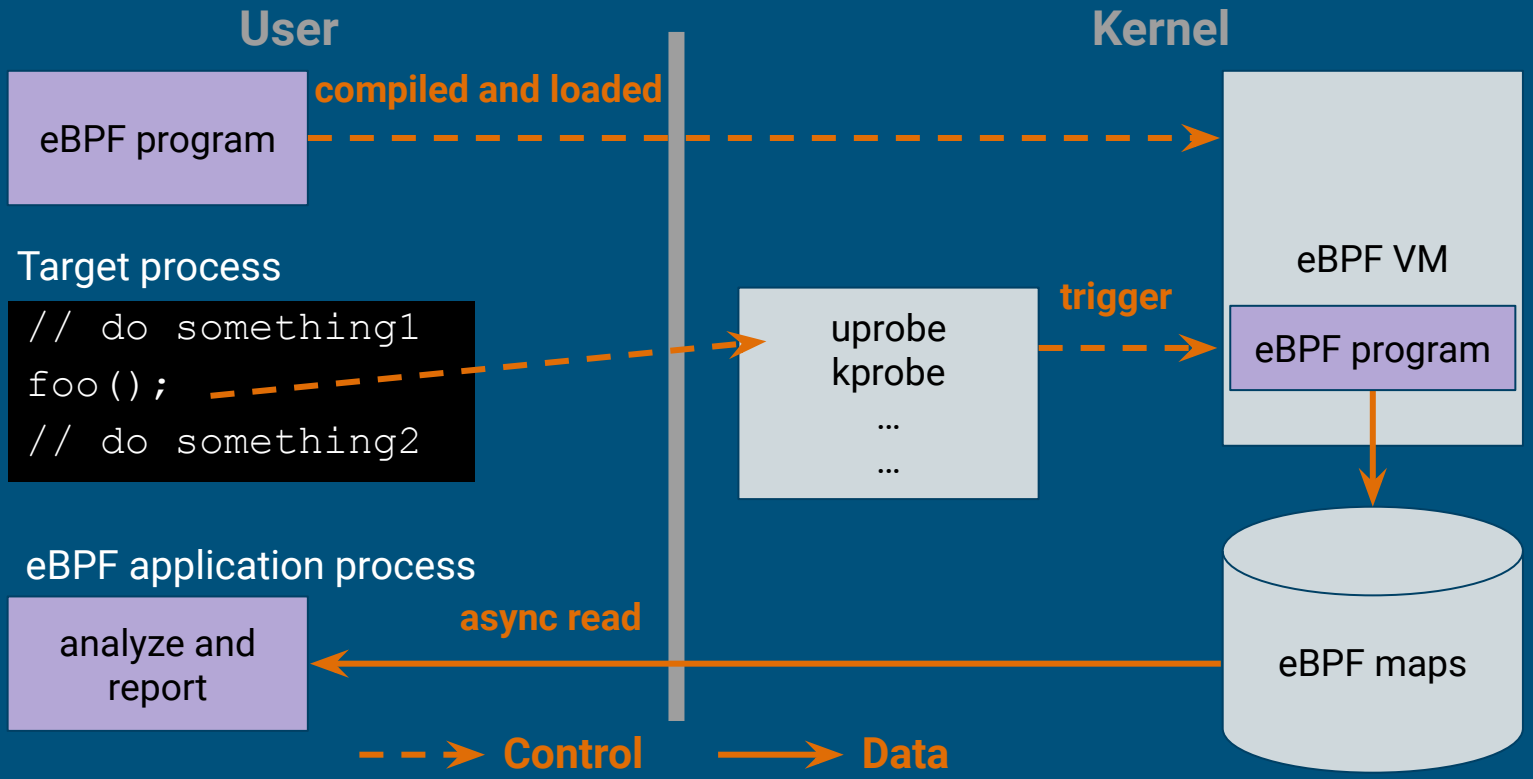
eBPF application overview



eBPF application overview



eBPF application overview



Prerequisite

1. Root privilege

2. Kernel configs
   - CONFIG_UPROBES, CONFIG_UPROBE_EVENTS
   - CONFIG_BPF, CONFIG_BPF_xxx, etc

   Enabled by default on major Linux distributions

Contents

- Motivation
  - Memory leak matters in C++?
  - Memory leak detection tools and limitations
    1. Test again
    2. Report on process exit
    3. Heisenbug
- Solution
  - Background: eBPF, uprobe
  - **ALSan(Attachable Leak Sanitizer)**
  - Demo

What is Attachable LSan?

eBPF application

What is Attachable LSan?

eBPF application

1. Hooks memory allocator functions
2. Save/delete allocation information to analyze and report
3. Analyze and report unreachable memory

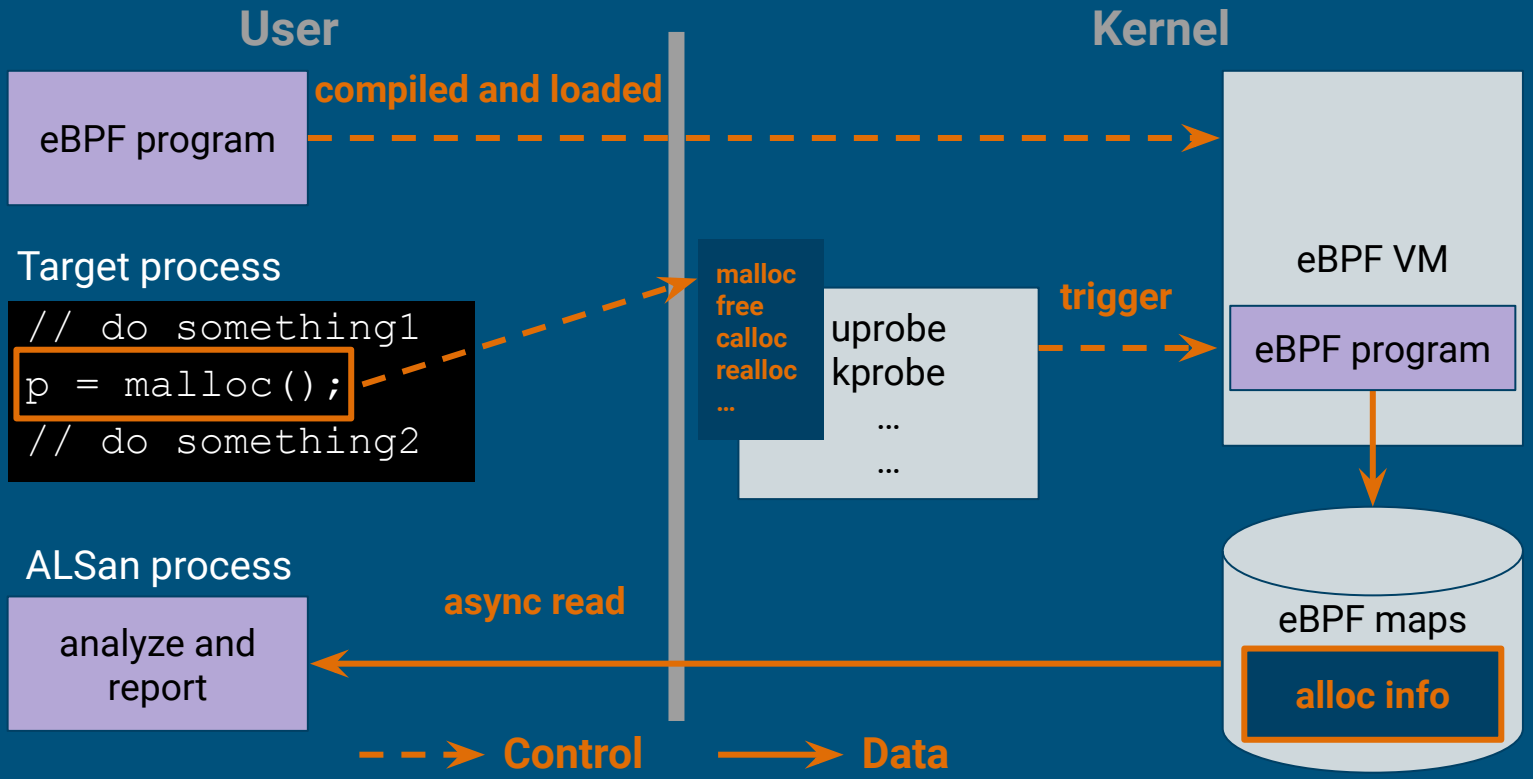
What is Attachable LSan?

eBPF application

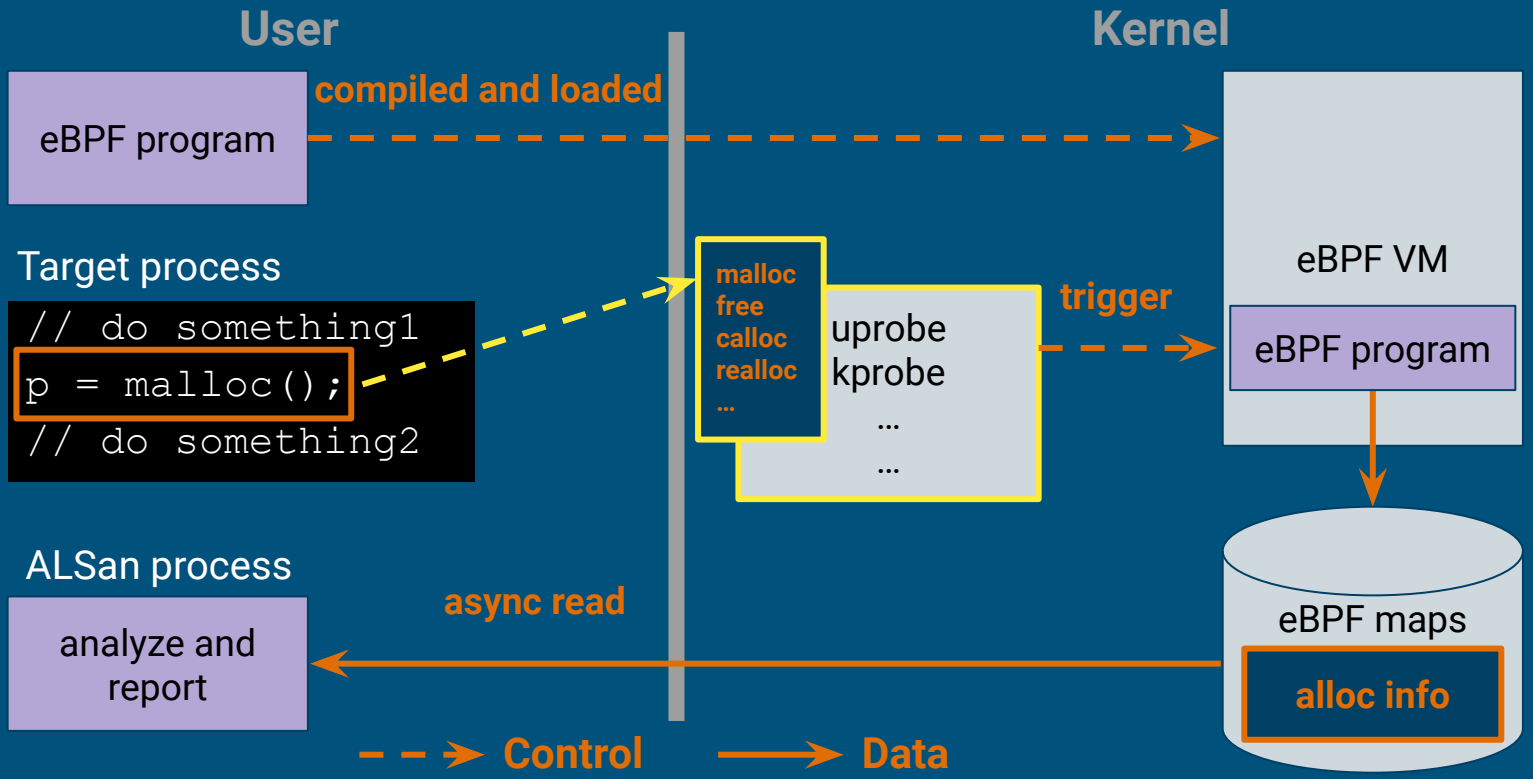
Same as LSan

1. Hooks memory allocator functions
2. Save/delete allocation information to analyze and report
3. Analyze and report unreachable memory

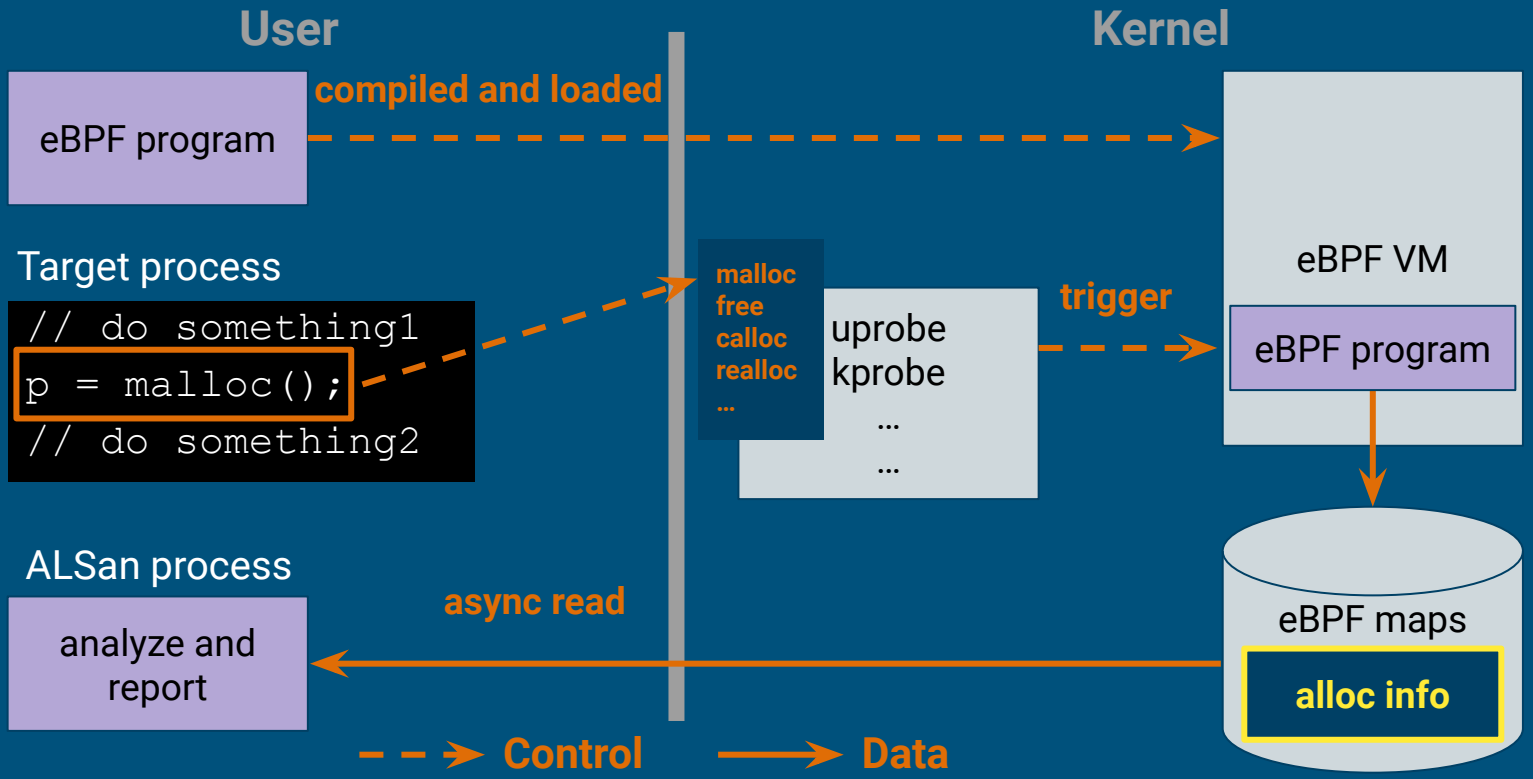
ALSan overview



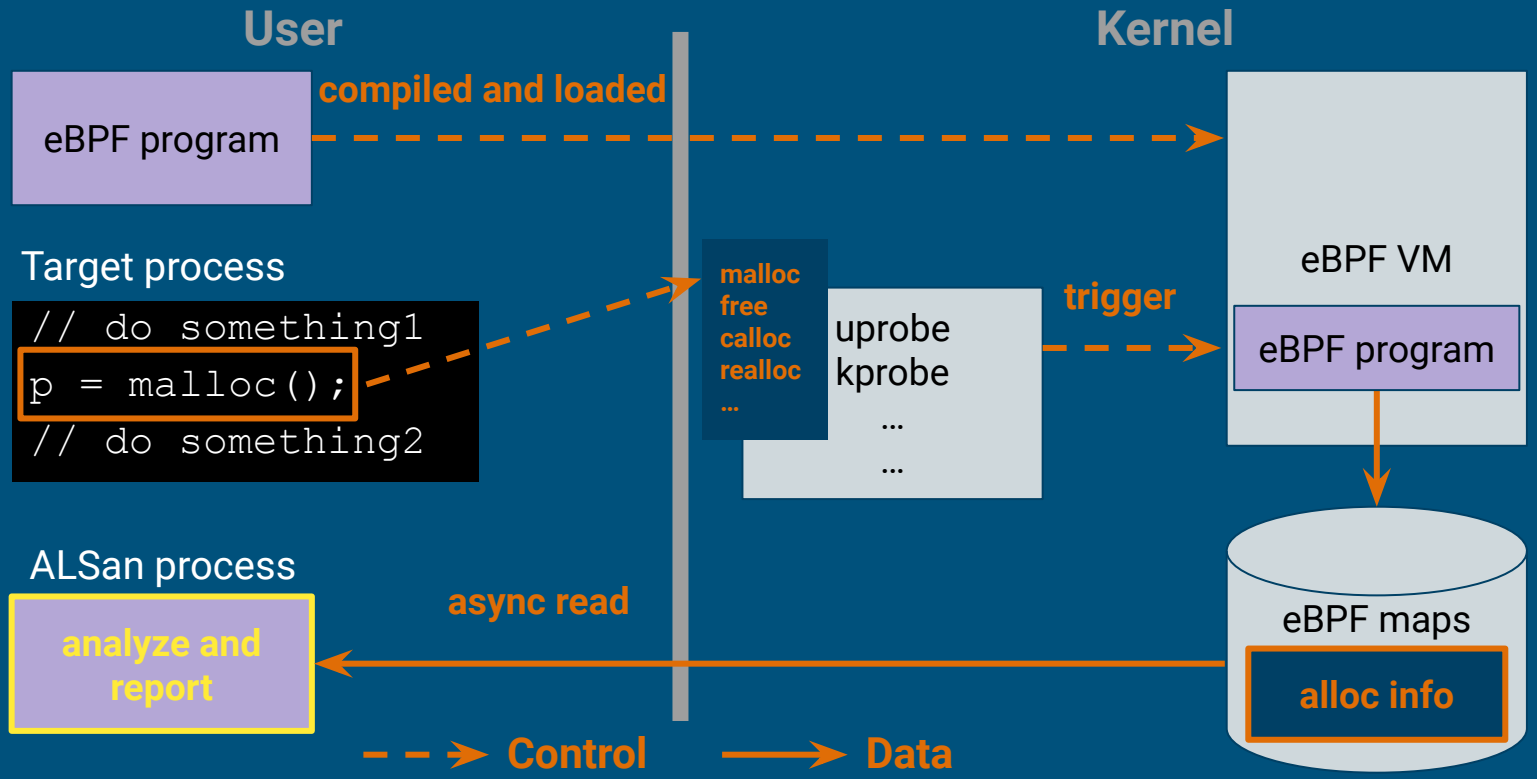
Hook functions on running process

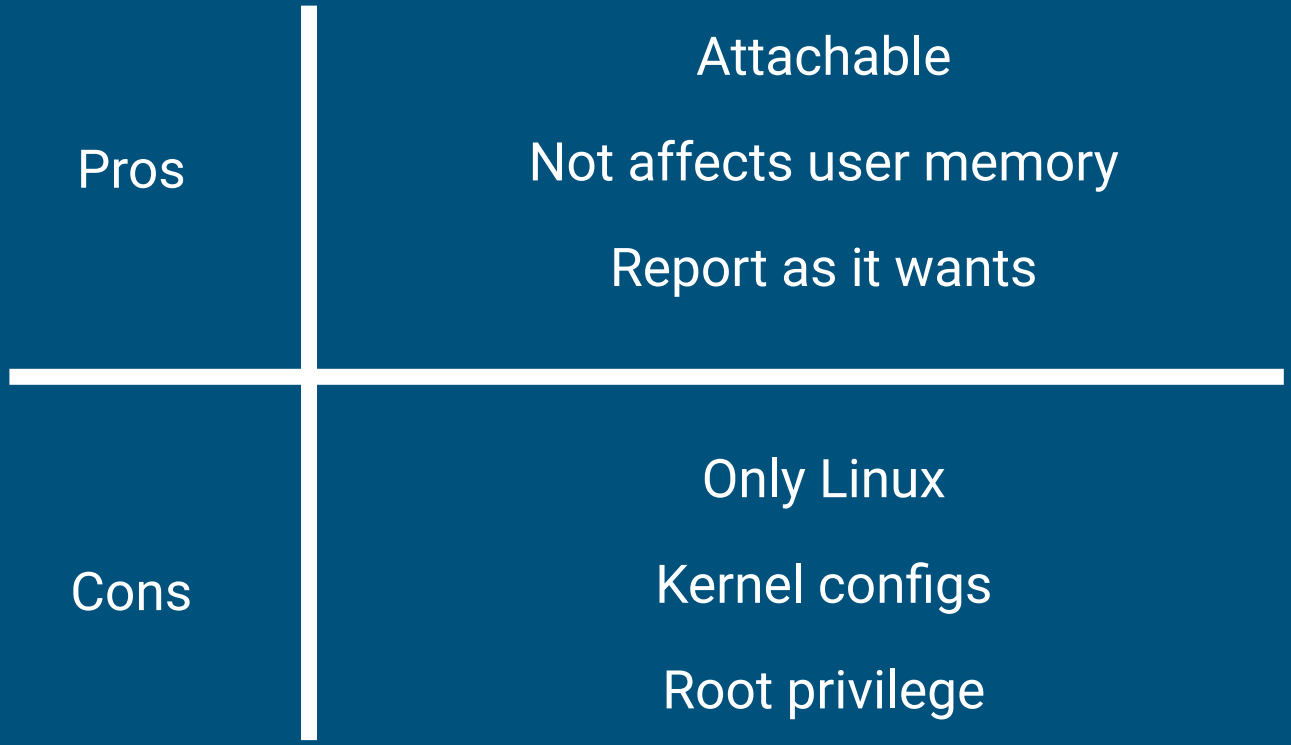


Allocation info saved in kernel



Analyze and report as it wants




Pros
Attachable
Not affects user memory
Report as it wants
Cons
Only Linux
Kernel configs
Root privilege

Contents

- Motivation
  - Memory leak matters in C++?
  - Memory leak detection tools and limitations
    1. Test again
    2. Report on process exit
    3. Heisenbug
- Solution
  - Background: eBPF, uprobe
  - ALSan(Attachable Leak Sanitizer)
  - **Demo**

Demo

- Examples
  1. Simple case
  2. Long test case
- Actual open source leaks
  1. https://github.com/nlohmann/json/issues/2865
  2. https://github.com/nlohmann/json/issues/3881

Simple example

Simple example - original leak sanitizer

```
$ cat test_new.cpp
int* baz() { return new int; }
int* bar() { return baz(); }
int* foo() { return bar(); }
int main() { auto p = foo(); }
```

Simple example - original leak sanitizer

```
$ cat test_new.cpp
int* baz() { return new int; }
int* bar() { return baz(); }
int* foo() { return bar(); }
int main() { auto p = foo(); }
$ g++ -fsanitize=leak test_new.cpp
```

Simple example - original leak sanitizer

```
$ cat test_new.cpp
int* baz() { return new int; }
int* bar() { return baz(); }
int* foo() { return bar(); }
int main() { auto p = foo(); }
$ g++ -fsanitize=leak test_new.cpp
$ ./a.out
=================================================================
==22642==ERROR: LeakSanitizer: detected memory leaks
Direct leak of 4 byte(s) in 1 object(s) allocated from:
    #0 0x709c4ca12a92 in operator new(unsigned long) lsan_interceptors.cpp:248
    #1 0x645bc216615a in baz() (/home/bojun/a.out+0x115a)
    #2 0x645bc2166169 in bar() (/home/bojun/a.out+0x1169)
    #3 0x645bc2166178 in foo() (/home/bojun/a.out+0x1178)
    #4 0x645bc216618b in main (/home/bojun/a.out+0x118b)
    #5 0x709c4c229d8f in __libc_start_call_main libc_start_call_main.h:58
SUMMARY: LeakSanitizer: 4 byte(s) leaked in 1 allocation(s).
```

Simple example - attachable leak sanitizer

```
$ cat test_new_loop.cpp
#include <chrono>
#include <thread>
int* baz() { return new int; }
int* bar() { return baz(); }
int* foo() { return bar(); }
int main() {
  while (true) {
    std::this_thread::sleep_for(std::chrono::seconds(5));
    auto p = foo();
  }
}
```

Simple example - attachable leak sanitizer

```
$ cat test_new_loop.cpp
#include <chrono>
#include <thread>
int* baz() { return new int; }
int* bar() { return baz(); }
int* foo() { return bar(); }
int main() {
  while (true) {
    std::this_thread::sleep_for(std::chrono::seconds(5));
    auto p = foo();
  }
}
$ g++ test_new_loop.cpp
```

Simple example - attachable leak sanitizer

```
$ cat test_new_loop.cpp
#include <chrono>
#include <thread>
int* baz() { return new int; }
int* bar() { return baz(); }
int* foo() { return bar(); }
int main() {
  while (true) {
    std::this_thread::sleep_for(std::chrono::seconds(5));
    auto p = foo();
  }
}
$ g++ test_new_loop.cpp
$ ./a.out &
[1] 22685
```

Simple example - attachable leak sanitizer

```
$ sudo ./alsan -p 22685
[2024-04-21 12:47:14] Print leaks:
4 bytes direct leak found in 1 allocations from stack id(26903)
        #1 0x0070a8758ae98c _Znwm+0x1c (libstdc++.so.6.0.30+0xae98c)
        #2 0x005dfdc5c921ca _Z3barv+0xd (/home/bojun/a.out+0x11ca)
        #3 0x005dfdc5c921d9 _Z3foov+0xd (/home/bojun/a.out+0x11d9)
        #4 0x005dfdc5c92221 main+0x46 (/home/bojun/a.out+0x1221)
        #5 0x0070a875429d90 [unknown] (libc.so.6+0x29d90)
```

Simple example - attachable leak sanitizer

```
$ sudo ./alsan -p 22685
[2024-04-21 12:47:14] Print leaks:
4 bytes direct leak found in 1 allocations from stack id(26903)
        #1 0x0070a8758ae98c _Znwm+0x1c (libstdc++.so.6.0.30+0xae98c)
        #2 0x005dfdc5c921ca _Z3barv+0xd (/home/bojun/a.out+0x11ca)
        #3 0x005dfdc5c921d9 _Z3foov+0xd (/home/bojun/a.out+0x11d9)
        #4 0x005dfdc5c92221 main+0x46 (/home/bojun/a.out+0x1221)
        #5 0x0070a875429d90 [unknown] (libc.so.6+0x29d90)

[2024-04-21 12:47:24] Print leaks:
12 bytes direct leak found in 3 allocations from stack id(26903)
        #1 0x0070a8758ae98c _Znwm+0x1c (libstdc++.so.6.0.30+0xae98c)
        #2 0x005dfdc5c921ca _Z3barv+0xd (/home/bojun/a.out+0x11ca)
        #3 0x005dfdc5c921d9 _Z3foov+0xd (/home/bojun/a.out+0x11d9)
        #4 0x005dfdc5c92221 main+0x46 (/home/bojun/a.out+0x1221)
        #5 0x0070a875429d90 [unknown] (libc.so.6+0x29d90)
```

The symbol of the function that calls “new” disappeared

https://github.com/iovisor/bcc/issues/4958

One solution has been suggested on LinuxCon 2024

https://sched.co/1aBwB

Simple example - attachable leak sanitizer

```
$ sudo ./alsan -p 22685
[2024-04-21 14:51:42] Print leaks:
4 bytes direct leak found in 1 allocations from stack id(26903)
        #1 0x0070a8758ae98c _Znwm+0x1c (libstdc++.so.6.0.30+0xae98c)
        #2 0x005dfdc5c921bb _Z3bazv+0xd (/home/bojun/a.out+0x11bb)
        #3 0x005dfdc5c921ca _Z3barv+0xd (/home/bojun/a.out+0x11ca)
        #4 0x005dfdc5c921d9 _Z3foov+0xd (/home/bojun/a.out+0x11d9)
        #5 0x005dfdc5c92221 main+0x46 (/home/bojun/a.out+0x1221)
        #6 0x0070a875429d90 [unknown] (libc.so.6+0x29d90)
```

Long test case example

Long test case example

```
int idx = 0;
std::string log[N];
int* data[M];

void send(const std::string& msg) {
  // save msg for logging
  log[idx++] = msg;

  // do something
}
```

Long test case example

```
constexpr int N = 1000 * 1000;
int idx = 0;
std::string log[N];
int* data[M];

void send(const std::string& msg) {
  // save msg for logging
  log[idx++] = msg;

  // do something
}
```

Buffer overflow will be happened after ‘send’ function is called more than ‘N’ times

Long test case example

```
#include <chrono>
#include <thread>
constexpr int N = 5000; constexpr int M = 100000;
int idx = 0; int logs[N]; int* data[M];
void send(int msg) { logs[idx++] = msg; }
int main(int argc, char* argv[]) {
  for (int i = 0; i < N; ++i) {
    send(1024);
    std::this_thread::sleep_for(std::chrono::milliseconds( 5));
  }
  for (int i = 0; i < M; ++i) {
    data[i] = new int;
    send(2024);
    std::this_thread::sleep_for(std::chrono::seconds( 5));
  }
  return 0;
}
```

Long test case example

```
$ g++ long_test.cpp
$ ./a.out &
[1] 66285
```

Long test case example

```
$ g++ long_test.cpp
$ ./a.out &
[1] 66285
$ sudo ./alsan -p 66285
[2024-05-01 15:47:32] Print leaks:

[2024-05-01 15:47:43] Print leaks:
4 bytes direct leak found in 1 allocations from stack id(37558)
        #1 0x007c064c0ae98c _Znwm+0x1c (libstdc++.so.6.0.30+0xae98c)
        #2 0x007c064bc29d90 [unknown] (libc.so.6+0x29d90)

[2024-05-01 15:47:54] Print leaks:
8 bytes direct leak found in 2 allocations from stack id(37558)
        #1 0x007c064c0ae98c _Znwm+0x1c (libstdc++.so.6.0.30+0xae98c)
        #2 0x007c064bc29d90 [unknown] (libc.so.6+0x29d90)
```

Actual open source leak

Json

https://github.com/nlohmann/json

Case 1. ASAN detects memory leaks

https://github.com/nlohmann/json/issues/2865

Original leak sanitizer - Actual case 1

```
$ ./a.out
=================================================================
==23043==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 32 byte(s) in 1 object(s) allocated from:
    #0 0x7c4041812a92 in operator new(unsigned long) ../../../../src/libsanitizer/lsa
    #1 0x587d6d759f0f in __gnu_cxx::new_allocator<std::__cxx11::basic_string<char, st
    #2 0x587d6d7584d1 in std::allocator_traits<std::allocator<std::__cxx11::basic_str
    #3 0x587d6d7576b2 in std::__cxx11::basic_string<char, std::char_traits<char>, std
    #4 0x587d6d755e47 in nlohmann::basic_json<std::map, std::vector, std::__cxx11::ba
    #5 0x587d6d754ce4 in nlohmann::basic_json<std::map, std::vector, std::__cxx11::ba
    #6 0x587d6d75e2f2 in std::pair<std::__cxx11::basic_string<char, std::char_traits<
    #7 0x587d6d75e364 in void __gnu_cxx::new_allocator<std::_Rb_tree_node<std::pair<s
    #8 0x587d6d75e25c in void std::allocator_traits<std::allocator<std::_Rb_tree_node
    #9 0x587d6d75e1ab in void std::_Rb_tree<std::__cxx11::basic_string<char, std::cha
    #10 0x587d6d75e0b1 in std::_Rb_tree_node<std::pair<std::__cxx11::basic_string<cha
    #11 0x587d6d75df62 in std::_Rb_tree_node<std::pair<std::__cxx11::basic_string<cha
```

Original leak sanitizer - Actual case 1

```
#12 0x587d6d75dd0f in std::_Rb_tree_node<std::pair<std::__cxx11::basic_string<cha
#13 0x587d6d75d89d in std::_Rb_tree_node<std::pair<std::__cxx11::basic_string<cha
#14 0x587d6d75d1e0 in std::_Rb_tree_node<std::pair<std::__cxx11::basic_string<cha
#15 0x587d6d75c796 in std::_Rb_tree<std::__cxx11::basic_string<char, std::char_tr
#16 0x587d6d75b99c in std::_Rb_tree<std::__cxx11::basic_string<char, std::char_tr
#17 0x587d6d75a9fe in std::map<std::__cxx11::basic_string<char, std::char_traits<
#18 0x587d6d75aa4c in void __gnu_cxx::new_allocator<std::map<std::__cxx11::basic_
#19 0x587d6d758fe5 in void std::allocator_traits<std::allocator<std::map<std::__c
#20 0x587d6d757410 in std::map<std::__cxx11::basic_string<char, std::char_traits<
#21 0x587d6d755df3 in nlohmann::basic_json<std::map, std::vector, std::__cxx11::b
#22 0x587d6d754c94 in nlohmann::basic_json<std::map, std::vector, std::__cxx11::b
#23 0x587d6d753afe in main (/home/bojun/json/include/a.out+0x3afe)
#24 0x7c4041029d8f in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main
```

Attachable leak sanitizer - Actual case 1

```
$ sudo ./alsan -p 23064
[2024-04-21 14:02:57] Print leaks:
32 bytes direct leak found in 1 allocations from stack id(31826)
        #1 0x0071b13faae98c _Znwm+0x1c (/usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.30
        #2 0x0060ec8ec909a4 _ZNSt16allocator_traitsISaINSt7__cxx1112basic_stringIcSt1
        #3 0x0060ec8ec8fb85 _ZN8nlohmann10basic_jsonISt3mapSt6vectorNSt7__cxx1112basi
        #4 0x0060ec8ec8e1e6 _ZN8nlohmann10basic_jsonISt3mapSt6vectorNSt7__cxx1112basi
        #5 0x0060ec8ec8ce2b _ZN8nlohmann10basic_jsonISt3mapSt6vectorNSt7__cxx1112basi
        #6 0x0060ec8ec967c5 _ZNSt4pairIKNSt7__cxx1112basic_stringIcSt11char_traitsIcE
        #7 0x0060ec8ec96837 _ZN9__gnu_cxx13new_allocatorISt13_Rb_tree_nodeISt4pairIKN
        #8 0x0060ec8ec9672f _ZNSt16allocator_traitsISaISt13_Rb_tree_nodeISt4pairIKNSt
        #9 0x0060ec8ec9667e _ZNSt8_Rb_treeINSt7__cxx1112basic_stringIcSt11char_traits
        #10 0x0060ec8ec96584 _ZNSt8_Rb_treeINSt7__cxx1112basic_stringIcSt11char_trait
        #11 0x0060ec8ec96435 _ZNKSt8_Rb_treeINSt7__cxx1112basic_stringIcSt11char_trai
        #12 0x0060ec8ec961e2 _ZNSt8_Rb_treeINSt7__cxx1112basic_stringIcSt11char_trait
        #13 0x0060ec8ec95d70 _ZNSt8_Rb_treeINSt7__cxx1112basic_stringIcSt11char_trait
        #14 0x0060ec8ec956b3 _ZNSt8_Rb_treeINSt7__cxx1112basic_stringIcSt11char_trait
```

Attachable leak sanitizer - Actual case 1

```
#15 0x0060ec8ec94c69 _ZNSt8_Rb_treeINSt7__cxx1112basic_stringIcSt11char_trait
#16 0x0060ec8ec93e6f _ZNSt8_Rb_treeINSt7__cxx1112basic_stringIcSt11char_trait
#17 0x0060ec8ec92ed1 _ZNSt3mapINSt7__cxx1112basic_stringIcSt11char_traitsIcES
#18 0x0060ec8ec92f1f _ZN9__gnu_cxx13new_allocatorISt3mapINSt7__cxx1112basic_s
#19 0x0060ec8ec914b8 _ZNSt16allocator_traitsISaISt3mapINSt7__cxx1112basic_str
#20 0x0060ec8ec8f8e3 _ZN8nlohmann10basic_jsonISt3mapSt6vectorNSt7__cxx1112bas
#21 0x0060ec8ec8e192 _ZN8nlohmann10basic_jsonISt3mapSt6vectorNSt7__cxx1112bas
#22 0x0060ec8ec8cddb _ZN8nlohmann10basic_jsonISt3mapSt6vectorNSt7__cxx1112bas
#23 0x0060ec8ec8bb3f _Z3foov+0xa4 (/home/bojun/json/include/a.out+0x3b3f)
#24 0x0060ec8ec8bc52 main+0x46 (/home/bojun/json/include/a.out+0x3c52)
#25 0x0071b13f629d90 [unknown] (/usr/lib/x86_64-linux-gnu/libc.so.6+0x29d90)
```

Case 2.

Memory leak when exception is thrown in adl_serializer::to_json

https://github.com/nlohmann/json/issues/3881

Original leak sanitizer - Actual case 2

```
$ ./a.out
=================================================================
==7259==ERROR: LeakSanitizer: detected memory leaks
Indirect leak of 80 byte(s) in 1 object(s) allocated from:
    #0 0x79c1f2212a92 in operator new(unsigned long) ../../../../src/libsanitizer/lsa
    #1 0x5db58d730649 in __gnu_cxx::new_allocator<std::_Rb_tree_node<std::pair<std::_
    #2 0x5db58d730005 in std::allocator_traits<std::allocator<std::_Rb_tree_node<std:
    #3 0x5db58d72f3b6 in std::_Rb_tree<std::__cxx11::basic_string<char, std::char_tra
    #4 0x5db58d72e304 in std::_Rb_tree_node<std::pair<std::__cxx11::basic_string<char
    #5 0x5db58d72d141 in std::pair<std::_Rb_tree_iterator<std::pair<std::__cxx11::bas
    #6 0x5db58d72bfe7 in std::pair<std::_Rb_tree_iterator<std::pair<std::__cxx11::bas
    #7 0x5db58d72b22c in nlohmann::json_abi_v3_11_2::basic_json<std::map, std::vector
    #8 0x5db58d72a50b in nlohmann::json_abi_v3_11_2::basic_json<std::map, std::vector
    #9 0x5db58d7297a8 in nlohmann::json_abi_v3_11_2::adl_serializer<Foo, void>::to_js
    #10 0x5db58d72a6a6 in nlohmann::json_abi_v3_11_2::basic_json<std::map, std::vecto
    #11 0x5db58d729875 in main (/home/bojun/json/include/a.out+0x2875)
    #12 0x79c1f1a29d8f in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main
```

Original leak sanitizer - Actual case 2

```
Indirect leak of 48 byte(s) in 1 object(s) allocated from:
    #0 0x79c1f2212a92 in operator new(unsigned long) ../../../../src/libsanitizer/lsa
    #1 0x5db58d72e12a in __gnu_cxx::new_allocator<std::map<std::__cxx11::basic_string
    #2 0x5db58d72cf80 in std::allocator_traits<std::allocator<std::map<std::__cxx11::
    #3 0x5db58d72bec8 in std::map<std::__cxx11::basic_string<char, std::char_traits<c
    #4 0x5db58d72b1c9 in nlohmann::json_abi_v3_11_2::basic_json<std::map, std::vector
    #5 0x5db58d72a50b in nlohmann::json_abi_v3_11_2::basic_json<std::map, std::vector
    #6 0x5db58d7297a8 in nlohmann::json_abi_v3_11_2::adl_serializer<Foo, void>::to_js
    #7 0x5db58d72a6a6 in nlohmann::json_abi_v3_11_2::basic_json<std::map, std::vector
    #8 0x5db58d729875 in main (/home/bojun/json/include/a.out+0x2875)
    #9 0x79c1f1a29d8f in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.

SUMMARY: LeakSanitizer: 128 byte(s) leaked in 2 allocation(s).
```

Attachable leak sanitizer - Actual case 2

```
$ sudo ./alsan -p 22883
[2024-04-21 13:42:39] Print leaks:
80 bytes indirect leak found in 1 allocations from stack id(50619)
        #1 0x0076d4ad2ae98c _Znwm+0x1c (/usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.30
        #2 0x005c11b1f954d2 _ZNSt16allocator_traitsISaISt13_Rb_tree_nodeISt4pairIKNSt
        #3 0x005c11b1f94883 _ZNSt8_Rb_treeINSt7__cxx1112basic_stringIcSt11char_traits
        #4 0x005c11b1f937d1 _ZNSt8_Rb_treeINSt7__cxx1112basic_stringIcSt11char_traits
        #5 0x005c11b1f9260e _ZNSt8_Rb_treeINSt7__cxx1112basic_stringIcSt11char_traits
        #6 0x005c11b1f914b4 _ZNSt3mapINSt7__cxx1112basic_stringIcSt11char_traitsIcESa
        #7 0x005c11b1f905c5 _ZN8nlohmann16json_abi_v3_11_210basic_jsonISt3mapSt6vecto
        #8 0x005c11b1f8f64c _ZN8nlohmann16json_abi_v3_11_210basic_jsonISt3mapSt6vecto
        #9 0x005c11b1f8e7e9 _ZN8nlohmann16json_abi_v3_11_214adl_serializerI3FoovE7to_
        #10 0x005c11b1f8f7e7 _ZN8nlohmann16json_abi_v3_11_210basic_jsonISt3mapSt6vect
        #11 0x005c11b1f8e8b6 _Z3foov+0x3c (/home/bojun/json/include/a.out+0x28b6)
        #12 0x005c11b1f8e933 main+0x46 (/home/bojun/json/include/a.out+0x2933)
        #13 0x0076d4ace29d90 [unknown] (/usr/lib/x86_64-linux-gnu/libc.so.6+0x29d90)
```

Attachable leak sanitizer - Actual case 2

```
48 bytes indirect leak found in 1 allocations from stack id(40760)
        #1 0x0076d4ad2ae98c _Znwm+0x1c (/usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.30
        #2 0x005c11b1f9244d _ZNSt16allocator_traitsISaISt3mapINSt7__cxx1112basic_stri
        #3 0x005c11b1f91395 _ZN8nlohmann16json_abi_v3_11_210basic_jsonISt3mapSt6vecto
        #4 0x005c11b1f90562 _ZN8nlohmann16json_abi_v3_11_210basic_jsonISt3mapSt6vecto
        #5 0x005c11b1f8f64c _ZN8nlohmann16json_abi_v3_11_210basic_jsonISt3mapSt6vecto
        #6 0x005c11b1f8e7e9 _ZN8nlohmann16json_abi_v3_11_214adl_serializerI3FoovE7to_
        #7 0x005c11b1f8f7e7 _ZN8nlohmann16json_abi_v3_11_210basic_jsonISt3mapSt6vecto
        #8 0x005c11b1f8e8b6 _Z3foov+0x3c (/home/bojun/json/include/a.out+0x28b6)
        #9 0x005c11b1f8e933 main+0x46 (/home/bojun/json/include/a.out+0x2933)
        #10 0x0076d4ace29d90 [unknown] (/usr/lib/x86_64-linux-gnu/libc.so.6+0x29d90)
```

Appendix

Way to build ALSan

```
# Way to build and run attachable lsan on Ubuntu 22.04.3
# Need to install some packages
~$ sudo apt install git cmake libclang-dev libelf-dev llvm clang

~$ git clone https://github.com/Bojun-Seo/bcc.git -b lsan
~$ cd bcc/
~/bcc$ mkdir build
~/bcc$ cd build/
~/bcc/build$ cmake ..
~/bcc/build$ cd ../libbpf-tools/
~/bcc/libbpf-tools$ make lsan
~/bcc/libbpf-tools$
```

Thanks