C++

Spring 2021

compscicenter.ru

Philipp Grabovoy

t.me/phil-grab

Lecture XII

Tools

static analyzers

error example

```
void f(size_t n) {
    char* s = static_cast<char*>(malloc(n));
    if (!s) { return; }

    for (size_t i = 0; i != n; ++i) { s[i] = 'q'; }

    delete[] s;
}
```

- warning: Memory allocated by malloc() should be deallocated by free(), not 'delete[]'

nullptr deref example

```
void test(int *p) {
    if (p)
        return;

    int x = p[0];
}
```

- `warning: Array access (from variable 'p') results in a null pointer dereference`
- посмотрим разные уровни оптимизаций `-O*` на godbolt.org

analyzer vs compiler

- компиляторы — балансируют между временем сборки и качеством поиска проблем
- анализаторы — тратят больше времени ⇒ лучше полнота/точность
- граница между ворнингами компилятора и сообщениями статического анализатора часто размыта

warnings

```c
int main() {
    printf("%p", 42);
}
```

- VS2017 (статический анализатор):
  - `warning C6066: Non-pointer passed as Param(2) when pointer is required in call to 'printf' Actual type: 'int'`
- VS2019 (компилятор):
  - `warning C4477: 'printf' : format string '%p' requires an argument`

precision/recall

- анализ программ с абсолютной точностью невозможен
- возможны ошибки ⇒ есть полнота и точность их нахождения
  - + ограничение на время исполнения

GCC PR18501 example, pt. 1

```
void something();
bool finish;

void f(void) {
    bool first;

    for (; !finish; ) {
        if (!first)
            something();
        first = false;
    }
}
```

- godbolt, где можно покрутить -○*

GCC PR18501 example, pt. 2

```
void something();

void consume(int value);

void g(bool flag) {
    int value;

    if (flag)
        value = 42;

    something();

    if (flag)
        consume(value);
}
```

- godbolt, где можно покрутить -○*

GCC PR18501

- bug report
- flag
- clang + `-Wconditional-uninitialized` — оба случая warn
- g++ — оба пропущены

interprocedural analysis example

```
#include <cstdlib>

void g(void* p) {
    free(p);
}

void f() {
    void* q = malloc(42);
    g(q);
    free(q);
}
```

- warning: Attempt to free released memory

annotations

- в каких случаях это ошибка?

```
void foo(int* val) {
    printf("%d", *val);
}
```

SAL example

```
void foo(_Out_ int* val) {
    printf("%d", *val);
}
```

- warning C6001: Using uninitialized memory '*val'
- godbolt

clang thread safety example

```
#include <mutex>

struct mytype {
    void foo();

private:
    std::mutex m;
    int a __attribute__((guarded_by(m)));
};

void mytype::foo() {
    a = 42;
}
```

- clang, `-Wthread-safety-analysis` ⇒ `warning: writing variable 'a' requires holding mutex 'm' exclusively`, godbolt

points

- инструменты могут выдавать ошибки
- сложность проверок варьируется (строчка исходника vs межмодульный анализ)
  - код можно проверять на разных этапах "компиляции"
- анализатору надо хорошо понимать язык программирования

dynamic analyzers

example

```
void f(int* p) {
    p[10] = 42;
}

int main() {
    int* p = new int[10];
    f(p);
    delete[] p;
}
```

- ⇒ `heap-buffer-overflow`

sanitizers

- Address sanitizer
  - обращения к неаллоцированой памяти
- Memory sanitizer
  - использование неинициализированной памяти

- Thread sanitizer
  - data races
- Undefined behavior sanitizer
  - множество различных проверок (например: переполнение при арифметике над знаковыми числами)

techs

- подмена библиотечных функций
- → инструментация кода компилятором (sanitizers)
  - для трекинга работы с данными — shadow-регионы (на 8 байт памяти программы — 1 байт мета-информации)
- → JIT-изменение скомпилированного кода (valgrind)

address sanitizer

- трекает аллоцированность памяти
- в shadow-регионах биты — флажки о доступности соотв. байтов
- на каждое обращение к памяти внутри программы — проверка на доступность (через эти биты)

memory sanitizer

- трекает инициализированность памяти
  - читать неинициализированную память разрешено
  - запрещено делать условные переходы на основании таких значений
- часто нужна инструментированность *всей* программы:
  - все зависимости пересобрать с `-fsanitize=memory`
  - иначе — возможны ложные срабатывания

thread sanitizer

- в shadow-состоянии — несколько последних обращений к переменной
  - если очередное обращение unordered с каким-то из предыдущих — нашли data race
- также требуется инструментированность всего кода
  - `-fsanitize=thread`

undefined behavior sanitizer

- "дешевые" проверки: на многие случаи UB есть флажки процессора
- overflow check example

valgrind memcheck

- совмещает проверки asan и msan
- не требует перекомпиляции программы
  - JIT-ит самостоятельно
  - теряет часть информации — про локальные стеки

valgrind modes

- memcheck — ошибки с памятью
- cachegrid/callgrind — профилировщик, на модельном процессоре
- massif — профилировщик использования памяти (heap)
- ...

debug modes

- `_GLIBCXX_DEBUG` — включает отладочный режим в `libstdc++`
  - проверки на правильное использование контейнеров и т.д.
- в своем коде тоже полезно писать ассерты для отладочного режима
  - проверки согласованности состояния класса и т.п.

more points

- динамические проверки сработают, если *код выполняется*
  - иметь тесты полезно
  - хорошо, если покрытие кода большое
  - fuzzy-testing, ...

profilers

tools overview

- Intel VTune Profiler
- Linux perf
  - + CPU flame graph
- valgrind callgrind
  - + `KCacheGrind`

processors' events

- perfmon-events
  - в `perf` можно сэмплировать по ним и другим

building for profiling/debugging

- -O2 или аналог
- -g или -g1
- возможно -fno-omit-frame-pointer

debuggers (+ reverse)

helpful features

- attach to process
  - `gdb -p 1234`
- open coredump
  - `gdb ./a.out core`
- watchpoints
  - `watch`
- breakpoint conditions
  - `break foo if x == 42`

reverse debuggers

- возможность исполнять программу назад
  - UndoDB
  - rr
  - Time Travel Debugging

techs for reverse debugging

- записывать блок действий для перемотки
  - В `gdb`: `(gdb) record`, `(gdb) step`, ...
- делать снапшоты + запоминать недетерминированные результаты
  - В `rr`: `$ rr ./a.out`, `$ rr replay` → `(gdb) run`, ...