Топ 10 ошибок в C++ проектах за 2018 год

Источник: https://habr.com/ru/company/pvs-studio/blog/444570/

Предисловие

Существует класс программ для разработчика, который помогает значительно повысить качество кода. Это так называемые **статические анализаторы** кода.

Анализаторы кода похожи на компилятор тем, что получают на вход исходный код программы, но результат их работы - предупреждения о потенциальных ошибках, причем гораздо более подробные и конкретные, чем предупреждения компилятора.

Существуют и анализаторы с открытым кодом, и коммерческие, в том числе с ознакомительными версиями или даже бесплатные для открытых проектов.

Предисловие

Здесь описываются результат поиска ошибок в открытых проектах, написанных на языках C, C++, C# и Java.

Занимательные места были найдены с помощью статического анализатора кода PVS-Studio.

Он умеет обнаруживать ошибки и потенциальные уязвимости в коде, написанном на упомянутых выше языках.

Десятое место

Ошибка была обнаружена при проверке виртуального планетария Stellarium.

Приведенный фрагмент кода хоть и является небольшим, но таит в себе довольно хитрую ошибку:

```
Plane::Plane(Vec3f &v1, Vec3f &v2, Vec3f &v3)
 : distance(0.0f), sDistance(0.0f)
{
 Plane(v1, v2, v3, SPolygon::CCW);
}
```

Предупреждение PVS-Studio: V603 The object was created but it is not being used. If you wish to call constructor, 'this->Plane::Plane(....)' should be used. Plane.cpp 29

Автор кода хотел инициализировать часть полей объекта, используя еще один конструктор, вложенный в основной.

Вместо этого у него получилось создать временный объект, который будет уничтожен при покидании своей области видимости.

Таким образом, несколько полей объекта так и останутся неинициализированными.

Десятое место

Вместо вложенного вызова конструктора следовало использовать делегирующий конструктор, введенный в C++11:

```
Plane::Plane(Vec3f& v1, Vec3f& v2, Vec3f& v3)
 : Plane(v1, v2, v3, SPolygon::CCW)
{
 distance = 0.0f;
 sDistance = 0.0f;
}
```

Тогда все необходимые поля были бы корректно проинициализированы.

Девятое место

На использование макроса анализатором было выдано предупреждение:

```
PP(pp_match)
{
  ....
 MgBYTEPOS_set(mg, TARG, truebase, RXp_OFFS(prog)[0].end);
  ....
}
```

Предупреждение PVS-Studio: V502 Perhaps the '?:' operator works in a different way than it was expected. The '?:' operator has a lower priority than the '&&' operator. pp_hot.c 3036

При открытии определения макроса было обнаружено, что он содержит еще несколько вложенных макросов, некоторые из которых тоже имели вложенные макросы.

Разобраться в этом было так сложно, что пришлось использовать препроцессированный файл. Но и это не помогло.

Девятое место

На месте предыдущей строки кода было обнаружено вот это:

```
(((targ)->sv_flags & 0x00000400) && (!((targ)->sv_flags & 0x00200000) ||
S_sv_only_taint_gmagic(targ)) ? (mg)->mg_len = ((prog->offs)[0].end), (mg)->mg_flags |= 0x40 :
((mg)->mg_len = (((targ)->sv_flags & 0x20000000)  &&
!__builtin_expect(((((PL_curcop)->cop_hints + 0) & 0x00000008) ? (_Bool)1 :(_Bool)0),(0))) ?
(ssize_t)Perl_utf8_length( (U8 *)(truebase), (U8 *)(truebase)+((prog->offs)[0].end)) :
(ssize_t)((prog->offs)[0].end), (mg)->mg_flags &= ~0x40));
```

Найти такую ошибку глазами будет сложно. Авторы статьи пришли к выводу, что на самом деле никакой ошибки здесь нет. Но в любом случае, это довольно занимательный пример нечитабельного кода.

Говорят, что макросы — зло. Конечно, бывают моменты, когда они оказываются незаменимыми, но если есть возможность заменить макрос на функцию — следует обязательно это сделать.

Особенно чреваты вложенные макросы. Не только потому что в них сложно разобраться, но и потому что они могут давать непредсказуемый результат. Если автор макроса случайно допустит в таком макросе ошибку — найти её будет гораздо сложнее, чем в функции.

Восьмое место

Следующий пример был взят из цикла статей об анализе проекта Chromium. Крылась она в библиотеке WebRTC.

```
std::vector<SdpVideoFormat>
StereoDecoderFactory::GetSupportedFormats() const
{
  std::vector<SdpVideoFormat> formats = ....;
  for (const auto& format : formats) {
    if (cricket::CodecNamesEq(....)) {
      ....
      formats.push_back(stereo_format);
    }
  }
  return formats;
}
```

Предупреждение PVS-Studio: V789 CWE-672 Iterators for the 'formats' container, used in the range-based for loop, become invalid upon the call of the 'push_back' function. stereocodecfactory.cc 89

Ошибка заключается в том, что размер вектора *formats* изменяется внутри range-based-for цикла. Range-based циклы основаны на итераторах, поэтому изменение размера контейнера внутри таких циклов может привести к инвалидации этих итераторов.

Восьмое место

Данная ошибка сохранится, если переписать цикл с явным использованием итераторов. Поэтому для наглядности можно привести такой код:

```
for (auto format = begin(formats), __end = end(formats);   format != __end; ++format)
    {
      if (cricket::CodecNamesEq(....)) {
        ....
        formats.push_back(stereo_format);
      }
    }
```

Например, при использовании метода *push_back* может произойти реаллокация вектора — и тогда итераторы станут указывать на недопустимую область памяти.

Чтобы избежать подобных ошибок, следует придерживаться правила: ***никогда не изменяйте размер контейнера внутри цикла, условия которого привязаны к этому контейнеру.*** Это касается range-based-циклов и циклов, использующих итераторы.

Седьмое место

Первым примером из видеоигровой индустрии будет отрывок кода, обнаруженный нами в игровом движке Godot:

```
void AnimationNodeBlendSpace1D::add_blend_point(
  const Ref<AnimationRootNode> &p_node, float p_position, int p_at_index)
{
  ERR_FAIL_COND(blend_points_used >= MAX_BLEND_POINTS);
  ERR_FAIL_COND(p_node.is_null());
  ERR_FAIL_COND(p_at_index < -1 || p_at_index > blend_points_used);
  if (p_at_index == -1 || p_at_index == blend_points_used) {
    p_at_index = blend_points_used;
  } else {
    for (int i = blend_points_used - 1; i > p_at_index; i++) {
      blend_points[i] = blend_points[i - 1];
    }
  }
  ....
}
```

Предупреждение PVS-Studio: V621 CWE-835 Consider inspecting the 'for' operator. It's possible that the loop will be executed incorrectly or won't be executed at all. animation_blend_space_1d.cpp 113

Седьмое место

Рассмотрим подробнее условие цикла. Переменная-счетчик инициализируется значением *blend_points_used — 1*. При этом, исходя из двух предыдущих проверок (в *ERR_FAIL_COND* и в *if*), становится понятно, что на момент выполнения цикла *blend_points_used* будет всегда больше, чем *p_at_index*. Таким образом, либо условие цикла всегда будет истинно, либо цикл не будет выполняться совсем.

Если *blend_points_used — 1 == p_at_index*, то цикл не выполняется.

Во всех остальных случаях проверка *i > p_at_index* всегда будет истинной, так как счётчик *i* увеличивается на каждой итерации цикла.

Может показаться, что цикл будет выполняться вечно, но это не так.

Во-первых, возникнет целочисленное переполнение переменной *i*, что является неопределенным поведением. Следовательно, полагаться на это не стоит.

Если бы *i* имела тип *unsigned int*, то после достижения счетчиком максимально возможного значения оператор *i++* превратил бы его в *0*. Такое поведение определено стандартом и называется «Unsigned wrapping». Однако следует знать, что использование такого механизма тоже является не очень хорошей идеей.

Седьмое место

Это было во-первых, но ведь еще есть во-вторых! Дело в том, что до целочисленного переполнения даже не дойдет. Куда раньше произойдет выход за границу массива. Это значит, что произойдет попытка доступа к области памяти за пределами выделенного для массива блока. И это тоже является неопределенным поведением. Классический пример :)

Чтобы было легче избегать подобных ошибок, можно дать пару рекомендаций:

- пишите более простой и понятный код;
- проводите более тщательный Code Review и пишите больше тестов для свеженаписанного кода;
- используйте статические анализаторы.

Шестое место

Еще один пример из индустрии геймдева, а именно — из исходного кода AAA-движка Amazon Lumberyard.

```
void TranslateVariableNameByOperandType(....)
{  //  Igor: yet another Qualcomm's special case  //  GLSL compiler thinks that -2147483648 is
  //  an integer overflow which is not
  if (*((int*)(&psOperand->afImmediates[0])) == 2147483648)
   {   bformata(glsl, "-2147483647-1");  }
  else
  {    //  Igor: this is expected to fix     //  paranoid compiler checks such as Qualcomm's
    if (*((unsigned int*)(&psOperand->afImmediates[0])) >= 2147483648)
         {
   bformata(glsl, "%d", *((int*)(&psOperand->afImmediates[0])));
        }
    else
      {
   bformata(glsl, "%d", *((int*)(&psOperand->afImmediates[0])));
      }
  }
  bcatcstr(glsl, ")");  //  ....
}
```

Предупреждение PVS-Studio: V523 The 'then' statement is equivalent to the 'else' statement. toglsloperand.c 700

Шестое место

Amazon Lumberyard разрабатывается как кроссплатформенный движок. Поэтому разработчики стараются поддерживать как можно больше компиляторов. Программист Игорь столкнулся с компилятором Qualcomm, об этом говорят комментарии.

Неизвестно, смог ли Игорь выполнить свою задачу и справиться с «параноидальными» проверками компилятора, но он оставил после себя весьма странный код. Странный он тем, что как *then*-, так и *else*-ветви оператора *if* содержат абсолютно идентичный код. Скорее всего, такая ошибка допущена в результате неаккуратного Copy-Paste.

Совет: будьте внимательны при Copy-Paste!

Пятое место

При написании кода анализатор выдал предупреждение, которое программист посчитал ложным. Вот соответствующий фрагмент кода и предупреждение:

```
QWindowsCursor::CursorState QWindowsCursor::cursorState()
{
  enum { cursorShowing = 0x1, cursorSuppressed = 0x2 };
  CURSORINFO cursorInfo;
  cursorInfo.cbSize = sizeof(CURSORINFO);
  if (GetCursorInfo(&cursorInfo)) {
    if (cursorInfo.flags & CursorShowing)   // <= V616
  ....
}
```

Предупреждение PVS-Studio: V616 CWE-480 The 'CursorShowing' named constant with the value of 0 is used in the bitwise operation. qwindowscursor.cpp 669

То есть PVS-Studio ругался на место, в котором, очевидно, ошибки нет! Не может быть, чтобы константа *CursorShowing* равнялась *0*, ведь буквально парой строк выше она инициализирована значением *1*.

Пятое место

При подробном анализе выяснилось, что PVS-Studio вновь оказался внимательнее человека. Значение *0x1* присваивается именованной константе *cursorShowing*, а в побитовой операции «и» участвует именованная константа *CursorShowing*. Это совершенно разные константы, ведь первая начинается со строчной буквы, а вторая — с заглавной.

Код успешно компилируется, ведь класс *QWindowsCursor* действительно содержит константу с таким именем. Вот её определение:

```
class QWindowsCursor : public QPlatformCursor {
public: enum CursorState { CursorShowing, CursorHidden, CursorSuppressed }; .... }
```

Если именованной enum-константе не присвоить значение явно, оно будет инициализировано по умолчанию. Так как *CursorShowing* является первым элементом перечисления, ему будет присвоено значение *0*.

Чтобы не допускать подобные ошибки, не следует давать сущностям чересчур похожие имена. Стоит особенно внимательно следовать этому правилу, если эти сущности имеют одинаковый тип или могут быть неявно приведены друг ко другу. Ведь в таких случаях обнаружить ошибку на глаз будет практически невозможно, а некорректный код будет успешно компилироваться и жить припеваючи внутри

Четвертое место

Ошибка из проекта FreeSWITCH:

```
static const char *basic_gets(int *cnt)
{
  ....
  int c = getchar();
  if (c < 0) {
    if (fgets(command_buf, sizeof(command_buf) - 1, stdin)
          != command_buf) {
      break;
    }
    command_buf[strlen(command_buf)-1] = '\0'; /* remove endline */
    break;
  }
  ....
}
```

Предупреждение PVS-Studio: V1010 CWE-20 Unchecked tainted data is used in index: 'strlen(command_buf)'.

Анализатор предупреждает, что в выражении *strlen(command_buf) — 1* используются непроверенные данные.

Четвертое место

И действительно: если *command_buf* окажется пустой с точки зрения языка C строкой (содержащей единственный символ — '\0') то *strlen(command_buf)* вернет *0*. В таком случае произойдет обращение *command_buf[-1]*, что представляет собой неопределенное поведение.

Беда!

Самый сок этой ошибки даже не в том, **почему** она случается, а в том, **как** она случается.

Данная ошибка является одним из тех приятных примеров, которые можно «пощупать» самостоятельно, воспроизвести. Можно запустить FreeSwitch, произвести некоторые действия, которые приведут к выполнению приведенного выше участка кода, и передать программе пустую строку на вход.

Четвертое место

В итоге легким движением руки работающая программа превращается (да нет, не в элегантные шорты) в неработающую!

Помните, что входные данные могут быть какими угодно, и стоит всегда их проверять. Тогда и анализатор не будет ругаться, и программа будет надёжнее.


1
WINNER
2
3

Третье место

Участок кода из проекта NCBI Genome Workbench — набора инструментов для изучения и анализа генетических данных.

```
/**   * Crypt a given password using schema required for NTLMv1 authentication
  * @param passwd clear text domain password
  * @param challenge challenge data given by server
  * @param flags NTLM flags from server side
  * @param answer buffer where to store crypted password
  */
void tds_answer_challenge(....)
{//  ....
  if (ntlm_v == 1) { //    ....    /* with security is best be pedantic */
    memset(hash, 0, sizeof(hash));
    memset(passwd_buf, 0, sizeof(passwd_buf));  //    ...
  } else {  //    ....
  }
}
```

Предупреждения PVS-Studio:

V597 The compiler could delete the 'memset' function call, which is used to flush 'hash' buffer. The memset_s() function should be used to erase the private data. challenge.c 365

V597 The compiler could delete the 'memset' function call, which is used to flush 'passwd_buf' buffer. The memset_s() function should be used to erase the private data. challenge.c 366

Третье место

Дело в том, что современные оптимизирующие компиляторы умеют делать очень многое для того, чтобы собранная программа работала быстрее. В том числе, компиляторы умеют отслеживать, что буфер, переданный в memset, больше нигде не используется.

В таком случае они могут удалить «ненужный» вызов memset, и имеют на это полное право. Тогда буфер, который хранит важные данные, может остаться в памяти на радость злоумышленникам.

На фоне этого грамотейский комментарий «с безопасностью есть хорошо быть педантичным» выглядит еще забавнее. Судя по очень небольшому количеству предупреждений, выданных на этот проект, разработчики очень старались быть аккуратными и писать безопасный код. Однако, как мы видим, пропустить этот дефект безопасности очень просто. Согласно Common Weakness Enumeration дефект классифицируется как CWE-14: Compiler Removal of Code to Clear Buffers.

Чтобы очищение памяти было безопасным, следует использовать функцию memset_s(). Она не только является более безопасной, чем memset(), но еще и не может быть «проигнорирована» компилятором.

Второе место

Серебряного призера данного топа прислал один из клиентов автора статьи. Он был уверен, что анализатор выдает ложные предупреждения. Данных код просмотрели несколько человек. Ошибки действительно присутствовали в коде, но ни один из программистов не смог их обнаружить. Честно говоря, у автора данной статьи сделать этого тоже не получилось.

И это при том, что анализатор явно выдал предупреждения на ошибочные места!

Получится ли у вас найти такую пронырливую ошибку? Проверьте себя на зоркость и внимательность.

```
523  if (ch >= 0x0FF00)
524  {
525      if (!((ch >= 0x0FF10) && (ch <= 0x0FF19)) || ((ch >= 0x0FF21) && (ch <= 0x0FF3A)) || ((ch >= 0x0FF41) && (ch <= 0x0FF5A)))
526      {
527          if (j == 0)
528              continue;
529          ch = chx;
530      }
531  }
```

Предупреждение PVS-Studio:

V560 A part of conditional expression is always false: (ch >= 0x0FF21). decodew.cpp 525
V560 A part of conditional expression is always true: (ch <= 0x0FF3A). decodew.cpp 525
V560 A part of conditional expression is always false: (ch >= 0x0FF41). decodew.cpp 525
V560 A part of conditional expression is always true: (ch <= 0x0FF5A). decodew.cpp 525

Второе место

Кроется ошибка в том, что оператор логического отрицания (!) применяется не ко всему условию, а только к его первому подвыражению:

!((ch >= 0x0FF10) && (ch <= 0x0FF19))

Если это условие выполняется, то значение переменной ch лежит в отрезке [0x0FF10...0x0FF19]. Тем самым, четыре дальнейших сравнения уже не имеют смысла: они всегда будут либо истинны, либо ложны.

Чтобы избежать подобных ошибок, стоит придерживаться нескольких правил:

Во-первых, очень удобно и наглядно выравнивать код таблицей.

Во-вторых, не стоит перегружать выражения скобками. Например, данный код можно переписать так:

```
const bool isLetterOrDigit =    (ch >= 0x0FF10 && ch <= 0x0FF19)  // 0..9
                              || (ch >= 0x0FF21 && ch <= 0x0FF3A)  // A..Z
                              || (ch >= 0x0FF41 && ch <= 0x0FF5A); // a..z
if (!isLetterOrDigit)
```

Тогда, во-первых, скобок становится гораздо меньше, а во-вторых — увеличивается вероятность «поймать» случайно допущенную ошибку глазами.

Первое место

Ошибка из легендарного System Shock! Эта игра, вышедшая аж в 1994 году, стала прародителем и вдохновителем таких культовых игр, как Dead Space, BioShock и Deus Ex.

Приведу слова автора:

«Но сначала я должен кое в чем признаться. То, что я вам сейчас покажу, не содержит никакой ошибки. По большому счету, оно даже не является отрывком кода, но я просто не смог удержаться, чтобы не поделиться этим с вами!

Дело в том, что в процессе анализа исходного кода игры моя коллега Виктория обнаружила множество интересных комментариев. То тут, то там внезапно встречались шутливые и ироничные замечания, и даже стихи:

```
// I'll give you fish, I'll give you candy,
// I'll give you, everything I have in my hand

// that kid from the wrong side came over my house again,
// decapitated all my dolls
// and if you bore me, you lose your soul to me
// - "Gepetto", Belly, _Star_

//  And here, ladies and gentlemen,
// is a celebration of C and C++ and their untamed
passion...
//  ==================
TerrainData  terrain_info;
//  Now the actual stuff...
//  ======================
```

Вольный перевод:

```
// Я дам тебе рыбку, я дам тебе конфетку,
// Я дам тебе все, что есть у меня в руках

// этот мальчик с противоположной стороны
// снова пришел ко мне домой
// обезглавил всех моих кукол
// и если ты мне надоешь, ты потеряешь душу для
меня
// - "Gepetto", Belly, _Star_

//  И вот, дамы и господа,
// перед нами триумф C и C++ и их неукрощенной
страсти
//  ==================
TerrainData  terrain_info;
```

Первое место

```
// this is all outrageously horrible, as we dont know what
// we really need to deal with here

// And if you thought the hack for papers was bad,
// wait until you see the one for datas... - X

// Returns whether or not in the humble opinion of the
// sound system, the sample should be politely
obliterated
// out of existence

// it's a wonderful world, with a lot of strange men
// who are standing around, and they all wearing towels
```

```
// это всё невыразимо ужасающе, так как мы не знаем,
// с чем нам действительно нужно иметь здесь дело

// И если вы думали, что что хак для работы с
бумагами был плох
// это вы еще не видели тот, что используется для
работы с данными... - X

// Возвращает скромное мнение звуковой системы о
том,
// должен ли семпл быть любезно вычеркнут из
существования

// это чудный мир, с кучей странных мужчин
// что повсюду стоят, и они в полотенцах
```

Вот такие комментарии оставляли разработчики игры в начале далеких девяностых… Между прочим, Даг Чёрч — главный дизайнер System Shock — занимался еще и написанием кода. Кто знает, может быть какие-нибудь из этих комментариев написаны лично им? Надеюсь, про мужчин в полотенцах — это не его рук дело :)»

Заключение

Большая часть ошибок в работающих программах происходит из-за копирования (скопировали часть кода, но забыли исправить какой-то фрагмент).

Не стоит задавать сущностям похожие имена.

К использованию макросов нужно подходить с осторожностью.

Внедрение автоматической проверки исходного кода в процесс разработки выявляет такие ошибки, которые человек пропускает из-за «замыленности взгляда».

Хороший анализатор кода выявляет ошибки в логических выражениях.

Применять анализатор кода стоит не в конце проекта, а в процессе разработки, этим экономится время разработчика на отладку.

Избавление от предупреждений не даёт 100% гарантию качества, но значительно его повышает.

Использование анализатора кода не делает код читаемым, не исправляет ошибки, а только указывает места для рефакторинга (переработки).