Cppcheck в формате .cfg

Команда cppcheck

Содержание

Глава 1

Вступление

Руководство описывает референсы для файлов .cfg, которые используются в Cppcheck.

Глава 2

Утечки памяти и ресурсов.

Cppcheck имеет настраиваемую проверку на утечки, например, вы можете указать, какие функции выделяют и освобождают память или ресурсы, а какие функции вообще не влияют на выделение.

2.1 <alloc>, <realloc> и <dealloc>

Вот пример программы:

```
void    test()
{
      HPEN pen = CreatePen(PS_SOLID , 1, RGB(255,0,0));
}
```

В приведенном выше примере кода обнаружена утечка ресурсов: CreatePen () - это функция WinAPI, которая создает перо. Однако Cppcheck не предполагает, что значения, возвращаемые функциями, должны быть освобождены. Сообщение об ошибке отсутствует:

```
$ cppcheck pen1.c
Checking pen1.c…
```

Если вы предоставите файл конфигурации, Cppcheck обнаружит ошибку:

```
$ cppcheck --library=windows.cfg pen1.c
Checking pen1.c...
[pen1.c:3]: (error) Resource leak: pen
```

Вот минимальный файл windows.cfg:

```
<?xml   version=”1.0”?>
<def>
  <resource>
    <alloc>CreatePen</alloc>
    <dealloc>DeleteObject</dealloc>
  </resource>
</def>
```

Функции, которые перераспределяют память, можно настроить с помощью тега <realloc>. Входной аргумент, указывающий на память, которая должна быть перераспределена, также может быть настроен (по умолчанию это первый аргумент). В качестве примера приведем файл конфигурации для функций fopen, freopen и fclose из стандартной библиотеки c:

```
<?xml   version=”1.0”?>
<def>
  <resource>
    <alloc>fopen</alloc>
    <realloc  realloc-arg=”3”>freopen</realloc>
    <dealloc>fclose</dealloc>
  </resource>
</def>
```

Функции распределения и освобождения организованы в группы. Каждая группа определяется в тегах <resource> или <memory> и идентифицируется своими функциями <dealloc>. Это означает, что группы с перекрывающимися тегами <dealloc> объединяются.

2.2 <leak-ignore> u <use>

Часто назначенный указатель передается функциям. Пример:

```
void test()
{
       char *p = malloc(100);
       dostuff(p);
 }
```

Если Cppcheck не знает, что делает dostuff, без конфигурации он будет считать, что dostuff заботится о памяти, поэтому утечки памяти нет.

Чтобы указать, что dostuff никоим образом не заботится о памяти, используйте <leak-ignore/> в теге <function> (см. Следующий раздел):

```
<?xml version="1.0"?>
<def>
   <function name="dostuff">
      <leak-ignore/>
      <arg nr="1"/>
   </function >
</def>
```

Если вместо этого dostuff позаботится о памяти, это можно настроить с помощью:

```
<?xml version="1.0"?>
<def>
  <memory >
    <dealloc >free </dealloc>
    <use>dostuff </use>
  </memory >
</def>
```

Конфигурация <use> не имеет логического назначения. Без него вы получите такие же предупреждения. Используйте его, чтобы заглушить информационные сообщения –check-library.

Глава 3

Функциональное поведение.

Чтобы указать поведение функций и способы их использования, можно использовать теги <function>. Функции идентифицируются своим именем, указанным в атрибуте name и количеством аргументов. Имя представляет собой список имен функций, разделенных запятыми. Для функций в пространствах имен или классах просто укажите их полное имя. Например: <function name = "memcpy, std :: memcpy">. Если у вас есть функции-шаблоны, укажите их имена экземпляров <function name = "dostuff <int>">.

3.1 Аргументы функции

Аргументы, которые принимает функция, могут быть указаны тегами <arg>. Каждый из них принимает номер аргумента (начиная с 1) в атрибуте nr, nr = «любой» для произвольных аргументов или nr = «вариативный» для вариативных аргументов. Необязательные аргументы можно указать, указав значение по умолчанию: default = "value". Спецификации для отдельных аргументов переопределяют этот параметр.

Вы можете указать, является ли аргумент входным или выходным аргументом. Например, <arg nr="1" direction="in">. Разрешенные направления: внутрь, наружу и внутрь.

3.1.1 Сравнение в аргументах

Вот пример программы с неуместным сравнением:

```
void test()
{
    if (MemCmp(buffer1 , buffer2 , 1024 == 0)) {}
}
```

Cppcheck предполагает, что в функции можно передавать логические значения:

```
$ cppcheck notbool.c
Checking notbool.c...
```

Если вы предоставите файл конфигурации, Cppcheck обнаружит ошибку:

```
$ cppcheck --library=notbool.cfg notbool.c
Checking notbool.c...
[notbool.c:5]: (error) Invalid MemCmp() argument nr 3. A non-boolean value
is required.
```

Вот минимальный notbool.cfg:

```
<?xml version="1.0"?>
<def>
   <function name="MemCmp">
      <arg nr="1"/>
      <arg nr="2"/>
      <arg nr="3">
         <not-bool/>
      </arg>
  </function >
</def>
```

3.1.2 Неинициализированная память.

Вот пример программы:

```
void test()
{
    char buffer1[1024];
    char buffer2[1024];
    CopyMemory(buffer1 , buffer2 , 1024);
}
```

Ошибка здесь в том, что buffer2 не инициализирован. Второй аргумент для CopyMemory необходимо инициализировать. Однако Cppcheck предполагает, что в функции можно передавать неинициализированные переменные:

```
$ cppcheck uninit.c
Checking uninit.c...
```

Если вы предоставите файл конфигурации, Cppcheck обнаружит ошибку:

```
$ cppcheck --library=windows.cfg uninit.c
Checking uninit.c...
[uninit.c:5]: (error) Uninitialized variable: buffer2
```

Ниже показан файл windows.cfg:
Версия 1:

```
<?xml version="1.0"?>
<def>
   <function name="CopyMemory">
      <arg nr="1"/>
      <arg nr="2">
         <not-null/>
         <not-uninit/>
      </arg>
      <arg nr="3"/>
   </function >
</def>
```

Версия 2:

```
<?xml version="1.0"?>
<def>
   <function name="CopyMemory">
     <arg nr="1"/>
     <arg nr="2">
        <not-uninit indirect="2"/>
     </arg>
     <arg nr="3"/>
   </function >
</def>
```

Версия 1: Если косвенный атрибут не используется, уровень косвенного обращения определяется автоматически. <not-null /> сообщает Cppcheck, что указатель должен быть инициализирован. <not-uninit /> указывает Cppcheck проверить 1 дополнительный уровень. Эта конфигурация означает, что и указатель, и данные должны быть инициализированы.

Версия 2: косвенный атрибут может быть установлен для явного управления уровнем косвенности, используемой при проверке. Установка косвенного значения на 0 означает, что неинициализированная память не допускается. Установка его в 1 разрешает указатель на неинициализированную память. Установка его в 2 позволяет указывать указатель на неинициализированную память.

3.1.3 Нулевые указатели

Cppcheck предполагает, что можно передавать в функции указатели NULL. Вот пример программы:

```
void test()
{
    CopyMemory(NULL, NULL, 1024);
}
```

В документации MSDN неясно, нормально это или нет. Но допустим, это плохо. Cppcheck предполагает, что функциям можно передавать NULL, чтобы не сообщать об ошибках:

```
$ cppcheck null.c
Checking null.c...
```

Если вы предоставите файл конфигурации, Cppcheck обнаружит ошибку:

```
$ cppcheck --library=windows.cfg null.c
Checking null.c...
[null.c:3]: (error) Null pointer dereference
```

Обратите внимание, что это подразумевает <not-uninit> в отношении значений. Неинициализированная память все еще может быть передана функции.

Вот минимальный файл windows.cfg:

```
<?xml version="1.0"?>
<def>
    <function name="CopyMemory">
      <arg nr="1">
        <not-null/>
      </arg>
      <arg nr="2"/>
      <arg nr="3"/>
    </function >
</def>
```

3.1.4 Формат строки

Вы можете определить, что функция принимает строку формата. Пример:

```
void test()
{
    do_something("%i %i\n", 1024);
}
```

Для этого не сообщается об ошибке:

```
$ cppcheck formatstring.c
Checking formatstring.c...
```

Можно создать файл конфигурации, в котором указано, что строка является строкой формата. Например:

```
<?xml version="1.0"?>
<def>
     <function name="do_something">
       <formatstr type="printf"/>
       <arg nr="1">
         <formatstr/>
       </arg>
     </function >
</def>
```

Теперь Cppcheck сообщит об ошибке:

```
$ cppcheck --library=test.cfg formatstring.c
Checking formatstring.c...
[formatstring.c:3]: (error) do_something format string requires 2
parameters but only 1 is given
```

Атрибут type может быть одним из следующих:
printf - форматная строка соответствует правилам printf
scanf - строка формата соответствует правилам scanf

3.1.5 Диапазон значений

Допустимые значения могут быть определены. Задумано:

```
void test()
{
   do_something(1024);
}
```

Для этого не сообщается об ошибке:

```
$ cppcheck valuerange.c
Checking valuerange.c...
```

Можно создать файл конфигурации, в котором указано, что 1024 находится за пределами допустимого диапазона. Например:

```
<?xml version="1.0"?>
<def>
    <function name="do_something">
      <arg nr="1">
        <valid >0:1023</valid >
      </arg>
   </function >
</def>
```

Теперь Cppcheck сообщит об ошибке:

```
$ cppcheck --library=test.cfg range.c
Checking range.c...
[range.c:3]: (error) Invalid do_something() argument nr 1. The value is
1024 but the valid values are '0-1023'.
```

Некоторые примеры выражений, которые вы можете использовать в допустимом элементе:

0,3,5 => действительны только значения 0, 3 и 5 -10: 20 => действительны все значения от -10 до 20: 0 => все значения меньше или равны 0 действительны 0: => все значения, которые больше или равны 0, действительны 0,2: 32 => значение 0 и все значения от 2 до 32 действительны -1,5: 5.6 => все значения от -1,5 до 5,6 действительны

3.1.6 <minsize>

Некоторые аргументы функции принимают буфер. С помощью minsize вы можете настроить минимальный размер буфера (в байтах, а не в элементах). Задумано:

```
void test()
{
  char str[5];
  do_something(str ,"12345");
}
```

Для этого не сообщается об ошибке:

```
$ cppcheck minsize.c
Checking minsize.c...
```

Например, можно создать файл конфигурации, в котором указано, что размер буфера в аргументе 1 должен быть больше, чем strlen аргумента 2. Например:

```
<?xml version="1.0"?>
<def>
  <function name="do_something">
    <arg nr="1">
       <minsize type="strlen" arg="2"/>
    </arg>
    <arg nr="2"/>
  </function >
</def>
```

Теперь Cppcheck сообщит об этой ошибке:

```
$ cppcheck --library=1.cfg minsize.c
Checking minsize.c...
[minsize.c:4]: (error) Buffer is accessed out of bounds: str
```

Существуют разные типы миниатюрных размеров:

Размер буфера strlen должен быть больше, чем длина строки других аргументов.
Пример: см. Конфигурацию strcpy в std.cfg

Размер буфера argvalue должен быть больше, чем значение в другом аргументе. Пример: см. Конфигурацию memset в std.cfg

Размер буфера sizeof должен быть больше, чем размер буфера другого аргумента. Пример: см. Конфигурацию memcpy в posix.cfg

Размер буфера mul должен быть больше результата умножения при умножении значений, заданных в двух других аргументах. Обычно один аргумент определяет размер элемента, а другой элемент определяет количество элементов. Пример: см. Конфигурацию fread в std.cfg

strz Этим вы можете сказать, что аргумент должен быть строкой с нулевым символом в конце.

```
<?xml version="1.0"?>
<def>
  <function name="do_something">
    <arg nr="1">
      <strz/>
    </arg>
  </function >
</def>
```

3.1.7 <noreturn>

Cppcheck не предполагает, что функции всегда возвращаются. Вот пример кода:

```
void test(int x)
{
   int data, buffer[1024];
   if (x == 1)
      data = 123;
    else
       ZeroMemory(buffer , sizeof(buffer));
    buffer[0] = data; // <- ошибка: данные не инициализированы, если x не 1
}
```

Теоретически, если ZeroMemory завершает программу, ошибки нет. Таким образом, Cppcheck не сообщает об ошибке:

```
$ cppcheck noreturn.c
Checking noreturn.c...
```

Однако если вы используете --check-library и --enable=information, вы получите следующее:

```
$ cppcheck --check -library --enable=information noreturn.c
Checking noreturn.c...
[noreturn.c:7]: (information) --check -library: Function ZeroMemory()
should have <noreturn > configuration
```

Если указан правильный файл windows.cfg, ошибка обнаруживается:

```
$ cppcheck --library=windows.cfg noreturn.c
Checking noreturn.c...
[noreturn.c:8]: (error) Uninitialized variable: data
```

Вот минимальный файл windows.cfg:

```
<?xml version="1.0"?>
<def>
    <function name="ZeroMemory">
      <noreturn >false </noreturn >
      <arg nr="1"/>
      <arg nr="2"/>
   </function >
</def>
```

3.1.8 <use-retval>

Пока ничего не указано, cppcheck предполагает, что игнорирование возвращаемого значения функции допустимо:

```
bool test(const char* a, const char* b)
{
    strcmp(a, b); // ошибка: вызов strcmp не имеет побочных эффектов, но возвращаемое
значение игнорируется.
   return true;
}
```

Если strcmp имеет побочные эффекты, такие как присвоение результата одному из переданных ему параметров, ничего плохого не произойдет:

```
$ cppcheck useretval.c
Checking useretval.c...
```

Если предоставлен правильный lib.cfg, ошибка обнаруживается:

```
$ cppcheck --library=lib.cfg --enable=warning useretval.c
Checking useretval.c...
[useretval.c:3]: (warning) Return value of function strcmp() is not used.
```

Вот минимальный файл lib.cfg:

```
<?xml version="1.0"?>
<def>
    <function name="strcmp">
        <use-retval/>
        <arg nr="1"/>
        <arg nr="2"/>
    </function >
</def>
```

3.1.9 <pure> и <const>

Они соответствуют атрибутам функции GCC <pure> и <const>.

Чистая функция не имеет никаких эффектов, кроме возврата значения, а ее возвращаемое значение зависит только от параметров и глобальных переменных.

Константная функция не имеет никаких эффектов, кроме возврата значения, а ее возвращаемое значение зависит только от параметров.

Вот пример кода:

```
void f(int x)
{
    if (calculate(x) == 213) {
   } else if (calculate(x) == 213) {
    // недостижимый код
   }
}
```

Если calculate () является константной функцией, то результат метода calculate (x) будет одинаковым в обоих условиях, поскольку используется одно и то же значение параметра.

Cppcheck обычно предполагает, что результат может быть другим, и не выдает никаких предупреждений для кода:

```
$ cppcheck const.c
Checking const.c...
```

Если указан правильный const.cfg, обнаруживается недостижимый код:

```
$ cppcheck --enable=style --library=const const.c
Checking const.c... [const.c:7]: (style)
Expression is always false because 'else if'
condition matches previous condition at line 5.
```

Вот минимальный файл const.cfg:

```
<?xml version="1.0"?>
<def>
   <function name="calculate">
     <const/>
     <arg nr="1"/>
   </function >
</def>
```

3.1.10 Пример конфигурации для strcpy ()

Правильная конфигурация стандартной функции strcpy () будет следующей:

```
<function name="strcpy">
   <leak-ignore/>
   <noreturn >false </noreturn >
   <arg nr="1">
      <not-null/>
   </arg>
   <arg nr="2">
     <not-null/>
     <not-uninit/>
     <strz/>
   </arg>
</function >
```

<leak-ignore/> указывает Cppcheck игнорировать этот вызов функции при проверке утечек. Передача выделенной памяти в эту функцию не означает, что она будет освобождена.

<noreturn> сообщает Cppcheck, возвращает эта функция или нет.

Первый аргумент, который принимает функция, - это указатель. Это не должен быть нулевой указатель, поэтому используется <not-null>.

Второй аргумент, который принимает функция, - это указатель. Он не должен быть нулевым. И он должен указывать на инициализированные данные. Использование <not-null> и <not-uninit> является правильным. Более того, он должен указывать на строку с нулевым завершением, поэтому также используется <strz>.

Глава 4

<type-checks>; проверить или подавить

Конфигурация <type-checks> указывает Cppcheck показывать или подавлять предупреждения для определенного типа.
Пример:

```
<?xml version="1.0"?>
<def>
   <type-checks >
     <unusedvar >
        <check >foo </check >
        <suppress >bar </suppress >
     </unusedvar >
   </type-checks >
</def>
```

При проверке неиспользуемых переменных будет проверяться тип foo. Предупреждения для переменных типа бара будут подавлены.

Глава 5

<define>

Библиотеки также могут использоваться для определения макросов препроцессора. Например:

```
<?xml version="1.0"?>
<def>
   <define name="NULL_VALUE" value="0"/>
</def>
```

Тогда каждое появление «NULL_VALUE» в коде будет заменено на «0» на стадии препроцессора.

Глава 6

<podtype>

Используйте это для типов integer / float / bool / pointer. Не для структур / союзов.

Большая часть кода полагается на typedef, обеспечивающий независимые от платформы типы. Теги «Podtype» могут использоваться для предоставления необходимой информации в cppcheck для их поддержки. Без дополнительной информации cppcheck не распознает тип «uint16_t» в следующем примере:

```
void test() {
uint16_t a;
}
```

Сообщение о неиспользовании переменной 'a' не выводится:

```
$ cppcheck --enable=style unusedvar.cpp
Checking unusedvar.cpp...
```

Если uint16_t определен в библиотеке следующим образом, результат улучшится:

```
<?xml version="1.0"?>
<def>
    <podtype name="uint16_t" sign="u" size="2"/>
</def>
```

Размер типа указывается в байтах. Возможные значения для атрибута «знак»: «s» (со знаком) и «u» (без знака). Оба атрибута не обязательны. Используя эту библиотеку, cppcheck напечатает:

```
$ cppcheck --library=lib.cfg --enable=style unusedvar.cpp
Checking unusedvar.cpp...
[unusedvar.cpp:2]: (style) Unused variable: a
```

Глава 7

<container>

Многие библиотеки C ++, в том числе сам STL, предоставляют контейнеры с очень похожей функциональностью. Библиотеки можно использовать, чтобы сообщить cppcheck об их поведении. Каждому контейнеру нужен уникальный идентификатор. При желании он может иметь startPattern, который должен быть допустимым шаблоном Token :: Match, и endPattern, который сравнивается со связанным токеном первого токена с такой ссылкой. Необязательный атрибут «наследует» принимает идентификатор из ранее определенного контейнера.

Атрибут hasInitializerListConstructor может быть установлен, если в контейнере есть конструктор, принимающий список инициализаторов.

Внутри тега <container> функции могут быть определены внутри тегов <size>, <access> и <other> (по вашему выбору). Каждый из них может указывать действие, такое как «изменение размера» и / или результат, который он дает, например, «конечный итератор».

В следующем примере представлено определение std :: vector на основе определения «stdContainer» (не показано):

```
<?xml version="1.0"?>
<def>
    <container id="stdVector" startPattern="std :: vector &lt;"
          inherits="stdContainer">
      <size>
          <function name="push_back" action="push"/>
          <function name="pop_back" action="pop"/>
      </size>
      <access indexOperator="array -like">
          <function name="at" yields="at_index"/>
          <function name="front" yields="item"/>
          <function name="back" yields="item"/>
      </access >
   </container >
</def>
```

Также можно добавить тег <type>, чтобы предоставить дополнительную информацию о типе контейнера. Вот некоторые из атрибутов, которые можно установить:

- string = 'std-like' может быть установлен для контейнеров, которые соответствуют интерфейсам std :: string.
- associative = 'std-like' может быть установлена для контейнеров, которые соответствуют интерфейсам C ++ AssociativeContainer.

Глава 8

<smart-pointer>

Укажите, что класс является интеллектуальным указателем, используя <smart-pointer class-name"std::shared_ptr"/>