Cppcheck .cfg format

Cppcheck team

Contents

Chapter 1

Introduction

This is a reference for the .cfg file format that Cppcheck uses.

Chapter 2

Memory and resource leaks

Cppcheck has configurable checking for leaks, e.g. you can specify which functions allocate and free memory or resources and which functions do not affect the allocation at all.

2.1 `<alloc>` , `<realloc>` and `<dealloc>`

Here is an example program:

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

The code example above has a resource leak - CreatePen() is a WinAPI function that creates a pen. However, Cppcheck doesn't assume that return values from functions must be freed. There is no error message:

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

If you provide a configuration file then Cppcheck detects the bug:

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

Here is a minimal windows.cfg file:

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

Functions that reallocate memory can be configured using a `<realloc>` tag. The input argument which points to the memory that shall be reallocated can also be configured (the default is the first argument). As an example, here is a configuration file for the fopen, freopen and fclose functions from the c standard library:

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

The allocation and deallocation functions are organized in groups. Each group is defined in a <resource> or <memory> tag and is identified by its <dealloc> functions. This means, groups with overlapping <dealloc> tags are merged.

2.2 <leak-ignore> and <use>

Often the allocated pointer is passed to functions. Example:

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

If Cppcheck doesn't know what dostuff does, without configuration it will assume that dostuff takes care of the memory so there is no memory leak.

To specify that dostuff doesn't take care of the memory in any way, use <leak-ignore/> in the <function> tag (see next section):

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

If instead dostuff takes care of the memory then this can be configured with:

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

The <use> configuration has no logical purpose. You will get the same warnings without it. Use it to silence –check-library information messages.

Chapter 3

Function behavior

To specify the behaviour of functions and how they should be used, `<function>` tags can be used. Functions are identified by their name, specified in the name attribute and their number of arguments. The name is a comma-separated list of function names. For functions in namespaces or classes, just provide their fully qualified name. For example: `<function name="memcpy,std::memcpy">`. If you have template functions then provide their instantiated names `<function name="dostuff<int>">`.

3.1 Function arguments

The arguments a function takes can be specified by `<arg>` tags. Each of them takes the number of the argument (starting from 1) in the nr attribute, `nr="any"` for arbitrary arguments, or `nr="variadic"` for variadic arguments. Optional arguments can be specified by providing a default value: `default="value"`. The specifications for individual arguments override this setting.

You can specify if an argument is an input or output argument. For example `<arg nr="1" direction="in">`. The allowed directions are `in`, `out` and `inout`.

3.1.1 Not bool

Here is an example program with misplaced comparison:

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

Cppcheck assumes that it is fine to pass boolean values to functions:

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

If you provide a configuration file then Cppcheck detects the bug:

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

Here is the minimal 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 Uninitialized memory

Here is an example program:

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

The bug here is that buffer2 is uninitialized. The second argument for CopyMemory needs to be initialized. However, Cppcheck assumes that it is fine to pass uninitialized variables to functions:

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

If you provide a configuration file then Cppcheck detects the bug:

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

Below windows.cfg is shown:

Version 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>
```

Version 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>
```

Version 1: If `indirect` attribute is not used then the level of indirection is determined automatically. The `<not-null/>` tells Cppcheck that the pointer must be initialized. The `<not-uninit/>` tells Cppcheck to check 1 extra level. This configuration means that both the pointer and the data must be initialized.

Version 2: The `indirect` attribute can be set to explicitly control the level of indirection used in checking. Setting `indirect` to 0 means no uninitialized memory is allowed. Setting it to 1 allows a pointer to uninitialized memory. Setting it to 2 allows a pointer to pointer to uninitialized memory.

3.1.3 Null pointers

Cppcheck assumes it's ok to pass NULL pointers to functions. Here is an example program:

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

The MSDN documentation is not clear if that is ok or not. But let's assume it's bad. Cppcheck assumes that it's ok to pass NULL to functions so no error is reported:

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

If you provide a configuration file then Cppcheck detects the bug:

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

Note that this implies `<not-uninit>` as far as values are concerned. Uninitialized memory might still be passed to the function.

Here is a minimal windows.cfg file:

```
<?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 Format string

You can define that a function takes a format string. Example:

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

No error is reported for that:

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

A configuration file can be created that says that the string is a format string. For instance:

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

Now Cppcheck will report an error:

```
$ 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.
```

The type attribute can be either:

printf - format string follows the printf rules

scanf - format string follows the scanf rules

3.1.5 Value range

The valid values can be defined. Imagine:

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

No error is reported for that:

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

A configuration file can be created that says that 1024 is out of bounds. For instance:

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

Now Cppcheck will report an error:

```
$ 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'.
```

Some example expressions you can use in the valid element:

0,3,5 => only values 0, 3 and 5 are valid -10:20 => all values between -10 and 20 are valid :0 => all values that are less or equal to 0 are valid 0: => all values that are greater or equal to 0 are valid 0,2:32 => the value 0 and all values between 2 and 32 are valid -1.5:5.6 => all values between -1.5 and 5.6 are valid

3.1.6 `<minsize>`

Some function arguments take a buffer. With minsize you can configure the min size of the buffer (in bytes, not elements). Imagine:

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

No error is reported for that:

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

A configuration file can for instance be created that says that the size of the buffer in argument 1 must be larger than the strlen of argument 2. For instance:

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

Now Cppcheck will report this error:

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

There are different types of minsizes:

strlen buffer size must be larger than other arguments string length. Example: see strcpy configuration in std.cfg

argvalue buffer size must be larger than value in other argument. Example: see memset configuration in std.cfg

sizeof buffer size must be larger than other argument buffer size. Example: see memcpy configuration in posix.cfg

mul buffer size must be larger than multiplication result when multiplying values given in two other arguments. Typically one argument defines the element size and another element defines the number of elements. Example: see fread configuration in std.cfg

strz With this you can say that an argument must be a zero-terminated string.

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

3.1.7 `<noreturn>`

Cppcheck doesn't assume that functions always return. Here is an example code:

```
void test(int x)
{
    int data, buffer[1024];
    if (x == 1)
        data = 123;
    else
        ZeroMemory(buffer, sizeof(buffer));
    buffer[0] = data;  // <- error: data is uninitialized if x is not 1
}
```

In theory, if ZeroMemory terminates the program then there is no bug. Cppcheck therefore reports no error:

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

However if you use `--check-library` and `--enable=information` you'll get this:

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

If a proper windows.cfg is provided, the bug is detected:

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

Here is a minimal windows.cfg file:

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

3.1.8 `<use-retval>`

As long as nothing else is specified, cppcheck assumes that ignoring the return value of a function is ok:

```
bool test(const char* a, const char* b)
{
```

```
    strcmp(a, b);  // <- bug: The call of strcmp does not have
       side-effects, but the return value is ignored.
    return true;
}
```

In case strcmp has side effects, such as assigning the result to one of the parameters passed to it, nothing bad would happen:

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

If a proper lib.cfg is provided, the bug is detected:

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

Here is a minimal lib.cfg file:

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

3.1.9 `<pure>` and `<const>`

These correspond to the GCC function attributes <pure> and <const>.

A pure function has no effects except to return a value, and its return value depends only on the parameters and global variables.

A const function has no effects except to return a value, and its return value depends only on the parameters.

Here is an example code:

```
void f(int x)
{
    if (calculate(x) == 213) {
    } else if (calculate(x) == 213) {
        // unreachable code
    }
}
```

If calculate() is a const function then the result of calculate(x) will be the same in both conditions, since the same parameter value is used.

Cppcheck normally assumes that the result might be different, and reports no warning for the code:

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

If a proper const.cfg is provided, the unreachable code is detected:

```
$ 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.
```

Here is a minimal const.cfg file:

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

3.1.10 Example configuration for strcpy()

The proper configuration for the standard strcpy() function would be:

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

The <leak-ignore/> tells Cppcheck to ignore this function call in the leaks checking. Passing allocated memory to this function won't mean it will be deallocated.

The <noreturn> tells Cppcheck if this function returns or not.

The first argument that the function takes is a pointer. It must not be a null pointer, therefore <not-null> is used.

The second argument the function takes is a pointer. It must not be null. And it must point at initialized data. Using <not-null> and <not-uninit> is correct. Moreover it must point at a zero-terminated string so <strz> is also used.

Chapter 4

`<type-checks>` ; check or suppress

The `<type-checks>`configuration tells Cppcheck to show or suppress warnings for a certain type.

Example:

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

In the `unusedvar` checking the `foo` type will be checked. Warnings for `bar` type variables will be suppressed.

Chapter 5

`<define>`

Libraries can be used to define preprocessor macros as well. For example:

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

Each occurrence of “NULL_VALUE” in the code would then be replaced by “0” at preprocessor stage.

Chapter 6

<podtype>

Use this for integer/float/bool/pointer types. Not for structs/unions.

Lots of code relies on typedefs providing platform independent types. “podtype”-tags can be used to provide necessary information to cppcheck to support them. Without further information, cppcheck does not understand the type “uint16_t” in the following example:

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

No message about variable ‘a’ being unused is printed:

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

If uint16_t is defined in a library as follows, the result improves:

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

The size of the type is specified in bytes. Possible values for the “sign” attribute are “s” (signed) and “u” (unsigned). Both attributes are optional. Using this library, cppcheck prints:

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

Chapter 7

<container>

A lot of C++ libraries, among those the STL itself, provide containers with very similar functionality. Libraries can be used to tell cppcheck about their behaviour. Each container needs a unique ID. It can optionally have a startPattern, which must be a valid Token::Match pattern and an endPattern that is compared to the linked token of the first token with such a link. The optional attribute “inherits” takes an ID from a previously defined container.

The `hasInitializerListConstructor` attribute can be set when the container has a constructor taking an initializer list.

Inside the `<container>` tag, functions can be defined inside of the tags `<size>`, `<access>` and `<other>` (on your choice). Each of them can specify an action like “resize” and/or the result it yields, for example “end-iterator”.

The following example provides a definition for std::vector, based on the definition of “stdContainer” (not shown):

```
<?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>
```

The tag `<type>` can be added as well to provide more information about the type of container. Here is some of the attributes that can be set:

- `string='std-like'` can be set for containers that match `std::string` interfaces.
- `associative='std-like'` can be set for containers that match C++’s `AssociativeContainer` interfaces.

Chapter 8

`<smart-pointer>`

Specify that a class is a smart pointer by using `<smart-pointer class-name"std::shared_ptr"/>`.