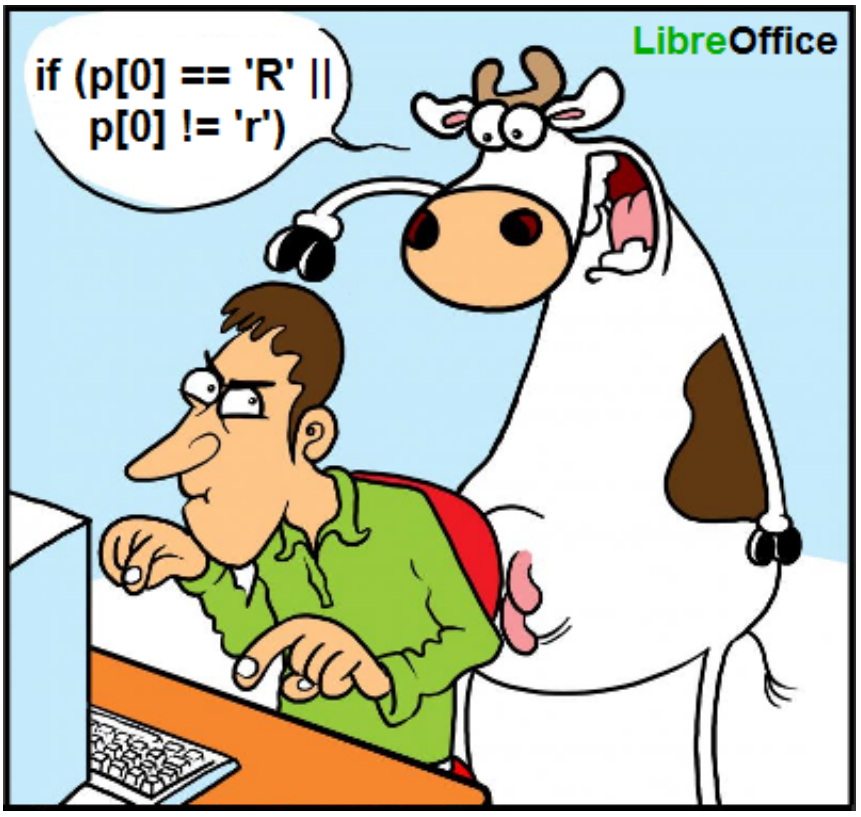
LibreOffice Project's Check

01.03.2015 Andrey Karpov

We invite you to read a new article about how we analyzed another well-known open-source project. This time it is the LibreOffice office suite that I have examined. The project is developed by more than 480 programmers. We have found that it is pretty high-quality and that it is regularly checked by the Coverity static analyzer. But, like in any other large project, we still managed to find previously undetected bugs and defects and in this article we are going to discuss them. Just for a change, this time we will be accompanied by cows instead of unicorns.



LibreOffice is a powerful office suite completely compatible with 32/64-bit systems. It has been translated into more than 30 languages and supports most of the popular operating systems, including GNU/Linux, Microsoft Windows, and Mac OS X.

LibreOffice is free and open-source. It includes code written in Java, Python, and C++. We analyzed the part written in C++ (and a small part in C, C++/CLI). Version: 4.5.0.0.alpha0+ (Git revision: 368367).

Analysis was done with the static code analyzer PVS-Studio.

Let's have a look at the errors found in this project and see what can be done about them. I'd like to notice right away that some of what I think to be bugs may actually be no bugs at all. As I'm not familiar with the code, I could have easily mixed up a real defect and correct code. However, since such code confused both me and the analyzer, it certainly means something isn't right there. This code smells and should be refactored to reduce the probability of its being misunderstood in the course of project development and maintenance.

Typos

No code can do without typos. Many of them are found and fixed at the testing stage of course, but some manage to make it through and keep living inside programs for many years. They are usually found in rarely used functions or have almost no effect on the program execution.

For example, I came across the following comparison function of which only one third is executed:

```
class SvgGradientEntry
{
  ....
```

```
  bool operator==(const SvgGradientEntry& rCompare) const
  {
    return (getOffset() == rCompare.getOffset()
            && getColor() == getColor()
            && getOpacity() == getOpacity());
  }
  ....
}
```

PVS-Studio's diagnostic message: V501 There are identical sub-expressions to the left and to the right of the '==' operator: getColor() == getColor() svggradientprimitive2d.hxx 61

I guess this bug doesn't do much harm. Perhaps this '==' operator is not even used at all. But since the analyzer has managed to find this bug, then it sure can find more serious things right after a new code has been written. That's why static analysis is most valuable when used regularly, not occasionally.

What could be done to avoid this error? I don't know. Perhaps if one trained oneself to be more careful and accurate when aligning blocks of homogeneous code, this error would be more visible. For example, we can write the function in the following way:

```
bool operator==(const SvgGradientEntry& rCompare) const
{
  return    getOffset()  == rCompare.getOffset()
         && getColor()   == getColor()
         && getOpacity() == getOpacity();
}
```

Now you can see it clear that "rCompare" is missing in the right column. But, to be honest, it doesn't make it that prominent. So it may fail too. To err is human. That's why a static analyzer can be very helpful.

And here is an example where a typo could have definitely been avoided. The programmer wrote a poor code to exchange values between two variables.


```
void TabBar::ImplGetColors(....)
{
  ....
  aTempColor = rFaceTextColor;
  rFaceTextColor = rSelectTextColor;
  rSelectTextColor = rFaceTextColor;
  ....
}
```

PVS-Studio's diagnostic message: V587 An odd sequence of assignments of this kind: A = B; B = A;. Check lines: 565, 566. tabbar.cxx 566

In the last line, 'aTempColor' should have been used instead of 'rFaceTextColor'.

The programmer who wrote this code for value exchange shouldn't have done it "manually". It would have been much easier and safer to use the standard function std::swap():

```
swap(rFaceTextColor, rSelectTextColor);
```

Let's go on. I'm not sure there's any protection against the next error. It's a classical typo:

```
void SAL_CALL Theme::disposing (void)
{
  ChangeListeners aListeners;
  maChangeListeners.swap(aListeners);

  const lang::EventObject aEvent (static_cast<XWeak*>(this));
```

```
  for (ChangeListeners::const_iterator
           iContainer(maChangeListeners.begin()),
           iContainerEnd(maChangeListeners.end());
       iContainerEnd!=iContainerEnd;
       ++iContainerEnd)
  {
    ....
  }
}
```

PVS-Studio's diagnostic message: V501 There are identical sub-expressions to the left and to the right of the '!=' operator: iContainerEnd != iContainerEnd theme.cxx 439

The loop won't be executed as the "iContainerEnd!=iContainerEnd" condition is always false. What failed the programmer were the similar names of iterators. The code should actually look as follows: "iContainer!=iContainerEnd". By the way, I suspect there's one more error here: The "iContainerEnd" iterator is incremented, which is strange.

Another bad loop:

```
static void lcl_FillSubRegionList(....)
{
  ....
  for( IDocumentMarkAccess::const_iterator_t
      ppMark = pMarkAccess->getBookmarksBegin();     <<<<----
      ppMark != pMarkAccess->getBookmarksBegin();    <<<<----
      ++ppMark)
  {
    const ::sw::mark::IMark* pBkmk = ppMark->get();
    if( pBkmk->IsExpanded() )
      rSubRegions.InsertEntry( pBkmk->GetName() );
  }
}
```

PVS-Studio's diagnostic message: V625 Consider inspecting the 'for' operator. Initial and final values of the iterator are the same. uiregionsw.cxx 120

The loop won't execute. In the condition, the 'ppMark' iterator should be compared to 'pMarkAccess->getBookmarksEnd()'. I don't have any suggestions about how to protect oneself against an error like this using coding conventions. It's just a typo.

By the way, it may happen sometimes that code contains an error but it doesn't affect the program's correct execution in any way. Here is one of such from LibreOffice:

```
bool PolyPolygonEditor::DeletePoints(....)
{
  bool bPolyPolyChanged = false;
```

```
  std::set< sal_uInt16 >::const_reverse_iterator
    aIter;( rAbsPoints.rbegin() );
  for( aIter = rAbsPoints.rbegin();
       aIter != rAbsPoints.rend(); ++aIter )
  ....
}
```

PVS-Studio's diagnostic message: V530 The return value of function 'rbegin' is required to be utilized. polypolygoneditor.cxx 38

The error is in the line aIter;(rAbsPoints.rbegin());

The programmer intended to initialize an iterator but wrote a semicolon by mistake. The iterator remained uninitialized while the "(rAbsPoints.rbegin());" expression was left hanging about idle.

What saves it all is that the iterator is still luckily initialized to the necessary value inside the 'for'. So there's no error in fact, but the excessive expression still should be removed. By the way this loop was multiplied through the Copy-Paste technique, so the developers should also check lines 69 and 129 in the same file.

And finally a typo inside a class constructor:

```
XMLTransformerOOoEventMap_Impl::XMLTransformerOOoEventMap_Impl(
        XMLTransformerEventMapEntry *pInit,
        XMLTransformerEventMapEntry *pInit2 )
{
  if( pInit )
    AddMap( pInit );
  if( pInit )
    AddMap( pInit2 );
}
```

PVS-Studio's diagnostic message: V581 The conditional expressions of the 'if' operators situated alongside each other are identical. Check lines: 77, 79. eventoootcontext.cxx 79

The second 'if' operator must check the 'pInit2' pointer.

Code probably written on purpose but still looking suspicious

I found a few code fragments that seem to contain typos. But I'm not sure about that - perhaps it was purposely written that way.

```
class VCL_DLLPUBLIC MouseSettings
{
  ....
  long GetStartDragWidth() const;
  long GetStartDragHeight() const;
```

```
  ....
}

bool ImplHandleMouseEvent( .... )
{
  ....
  long nDragW  = rMSettings.GetStartDragWidth();
  long nDragH  = rMSettings.GetStartDragWidth();
  ....
}
```

PVS-Studio's diagnostic message: V656 Variables 'nDragW', 'nDragH' are initialized through the call to the same function. It's probably an error or un-optimized code. Consider inspecting the 'rMSettings.GetStartDragWidth()' expression. Check lines: 471, 472. winproc.cxx 472

It's not clear whether or not the variables nDragW and nDragH should be initialized to one and the same value. If yes, then a comment about that is missing. Or, it would have been even better in the following way:

```
long nDragW  = rMSettings.GetStartDragWidth();
long nDragH  = nDragW;
```

A similar issue:

```
void Edit::ImplDelete(....)
{
  ....
  maSelection.Min() = aSelection.Min();
  maSelection.Max() = aSelection.Min();
  ....
}
```

V656 Variables 'maSelection.Min()', 'maSelection.Max()' are initialized through the call to the same function. It's probably an error or un-optimized code. Consider inspecting the 'aSelection.Min()' expression. Check lines: 756, 757. edit.cxx 757

Those working on the project would see right away if the code is OK or not. I'm not among them, so I don't know exactly if there is an error here or not.

And the last case. A class contains three functions:

- GetVRP()
- GetVPN()
- GetVUV()

But in the following place, the GetVRP() function is used to initialize the 'aVPN' constant.

```
void ViewContactOfE3dScene::createViewInformation3D(....)
```

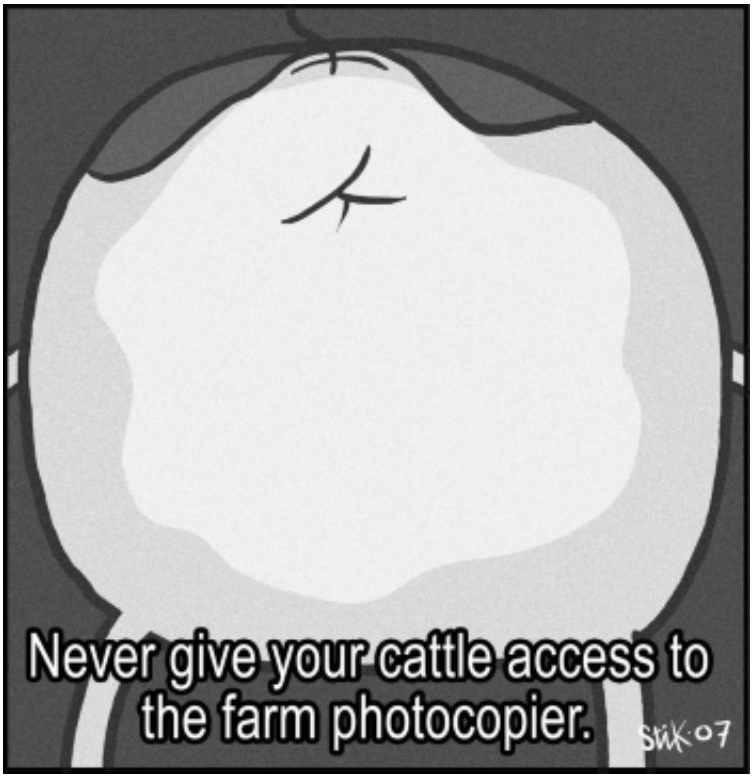
```
{
  ....
  const basegfx::B3DPoint aVRP(rSceneCamera.GetVRP());
  const basegfx::B3DVector aVPN(rSceneCamera.GetVRP());  <<<---
  const basegfx::B3DVector aVUV(rSceneCamera.GetVUV());
  ....
}
```

PVS-Studio's diagnostic message: V656 Variables 'aVRP', 'aVPN' are initialized through the call to the same function. It's probably an error or un-optimized code. Consider inspecting the 'rSceneCamera.GetVRP()' expression. Check lines: 177, 178. viewcontactofe3dscene.cxx 178

The analyzer generated one more V656 warning. I'm almost sure there's a genuine bug there. But I won't cite that code as it is too lengthy. So I suggest that the developers take a look at the following:

- V656 Variables 'oNumOffset1', 'oNumOffset2' are initialized through the call to the same function. It's probably an error or un-optimized code. Check lines: 68, 69. findattr.cxx 69

Copy-Paste



I'm forced to admit that programming would be extremely tiresome and boring at times without Copy-Paste. It's impossible to program without Ctrl-C and Ctrl-V, however strongly one may wish these shortcuts to be banned. That's why I won't preach dropping the copy-paste technique. But I do ask everyone: Please be careful and alert when copying and pasting code!

```
uno::Sequence< OUString >
SwXTextTable::getSupportedServiceNames(void)
{
```

```
  uno::Sequence< OUString > aRet(4);
  OUString* pArr = aRet.getArray();
  pArr[0] = "com.sun.star.document.LinkTarget";
  pArr[1] = "com.sun.star.text.TextTable";
  pArr[2] = "com.sun.star.text.TextContent";
  pArr[2] = "com.sun.star.text.TextSortable";
  return aRet;
}
```

PVS-Studio's diagnostic message: V519 The 'pArr[2]' variable is assigned values twice successively. Perhaps this is a mistake. Check lines: 3735, 3736. unotbl.cxx 3736

It's the classic last line effect. I'm almost sure the last line was derived from the one before it. The programmer replaced "Content" with "Sortable" but forgot about the index '2'.

Another very similar case:

```
Sequence<OUString> FirebirdDriver::getSupportedServiceNames_Static()
{
  Sequence< OUString > aSNS( 2 );
  aSNS[0] = "com.sun.star.sdbc.Driver";
  aSNS[0] = "com.sun.star.sdbcx.Driver";
  return aSNS;
}
```

PVS-Studio's diagnostic message: V519 The 'aSNS[0]' variable is assigned values twice successively. Perhaps this is a mistake. Check lines: 137, 138. driver.cxx 138

The most terrible thing, however, is that errors can sometimes quickly multiply throughout the code thanks to Copy-Paste. Here's an example. Unfortunately, the code I'm going to cite is somewhat difficult to read. So be patient.

So, we have the following function:

```
static bool GetPropertyValue(
  ::com::sun::star::uno::Any& rAny,
  const ::com::sun::star::uno::Reference<
                  ::com::sun::star::beans::XPropertySet > &,
  const OUString& rPropertyName,
  bool bTestPropertyAvailability = false );
```

Notice that the last argument 'bTestPropertyAvailability' is optional.

I should also explain what 'sal_True' is:

```
#define sal_True ((sal_Bool)1)
```

Now the bug itself. Notice how the GetPropertyValue() function is called:

```
sal_Int32 PPTWriterBase::GetLayoutOffset(....) const
{
  ::com::sun::star::uno::Any aAny;
  sal_Int32 nLayout = 20;
  if ( GetPropertyValue(
          aAny, rXPropSet, OUString( "Layout" ) ), sal_True )
    aAny >>= nLayout;

  DBG(printf("GetLayoutOffset %" SAL_PRIdINT32 "\n", nLayout));
  return nLayout;
}
```

PVS-Studio's diagnostic message: V639 Consider inspecting the expression for 'GetPropertyValue' function call. It is possible that one of the closing ')' brackets was positioned incorrectly. pptx-epptbase.cxx 442

If you look close, you'll see that one of the closing parentheses is in a wrong place. It results in the GetPropertyValue() function receiving the default argument value (equal to 'false') instead of 'sal_True' as the last argument.

But it's just half the trouble. The 'if' operator's work was also spoiled. The condition looks as follows:

```
if (foo(), sal_True)
```

The comma operator returns its right operand. As a result, the condition is always true.

The error in this code doesn't relate to Copy-Paste. It's just an ordinary typo - a parenthesis in a wrong place. It happens sometimes.

The sad thing about it is that this error was multiplied through other program parts. So even if the bug is fixed in one fragment, it may well remain unnoticed and unfixed in other fragments.

Thanks to Copy-Paste, this issue can be found in 9 more places:

- epptso.cxx 993
- epptso.cxx 3677
- pptx-text.cxx 518
- pptx-text.cxx 524
- pptx-text.cxx 546
- pptx-text.cxx 560
- pptx-text.cxx 566
- pptx-text.cxx 584
- pptx-text.cxx 590

To finish this section, here are 3 last non-critical warnings. Just one excessive check:

```
#define CHECK_N_TRANSLATE( name ) \
  else if (sServiceName == SERVICE_PERSISTENT_COMPONENT_##name) \
```

```
    sToWriteServiceName = SERVICE_##name

void OElementExport::exportServiceNameAttribute()
{
  ....
  CHECK_N_TRANSLATE( FORM );      <<<<----
  CHECK_N_TRANSLATE( FORM );      <<<<----
  CHECK_N_TRANSLATE( LISTBOX );
  CHECK_N_TRANSLATE( COMBOBOX );
  CHECK_N_TRANSLATE( RADIOBUTTON );
  CHECK_N_TRANSLATE( GROUPBOX );
  CHECK_N_TRANSLATE( FIXEDTEXT );
  CHECK_N_TRANSLATE( COMMANDBUTTON );
  ....
}
```

PVS-Studio's diagnostic message: V517 The use of 'if (A) {...} else if (A) {...}' pattern was detected. There is a probability of logical error presence. Check lines: 177, 178. elementexport.cxx 177

It's nothing serious but still a defect. Two other excessive checks can be found in the following places:

- querydesignview.cxx 3484
- querydesignview.cxx 3486

Brave use of the realloc() function

The realloc() function is used so obviously unsafely that I don't even dare call it a bug. It must be a conscious decision of the authors to use it that way: If memory fails to be allocated through malloc()/realloc(), then the program had better crash right away - no use "floundering about". Even if the program manages to make it through and keep working, it'll be no good. But it would be unfair to ignore the analyzer's warnings for this code as if they were false positives. So let's see what our tool didn't like in about it.

As an example, let's take the implementation of the add() function in the FastAttributeList class:

```
void FastAttributeList::add(sal_Int32 nToken,
  const sal_Char* pValue, size_t nValueLength )
{
  maAttributeTokens.push_back( nToken );
  sal_Int32 nWritePosition = maAttributeValues.back();
  maAttributeValues.push_back( maAttributeValues.back() +
                               nValueLength + 1 );
  if (maAttributeValues.back() > mnChunkLength)
  {
     mnChunkLength = maAttributeValues.back();
     mpChunk = (sal_Char *) realloc( mpChunk, mnChunkLength );
  }
```

```
  strncpy(mpChunk + nWritePosition, pValue, nValueLength);
  mpChunk[nWritePosition + nValueLength] = '\0';
}
```

PVS-Studio's diagnostic message: V701 realloc() possible leak: when realloc() fails in allocating memory, original pointer 'mpChunk' is lost. Consider assigning realloc() to a temporary pointer. fastattribs.cxx 88

The main problem with this code is that the realloc() function's return result is not checked. Of course, the situation when memory fails to be allocated is very rare. But suppose it has happened. Then realloc() returns NULL and an alarm condition occurs as the strncpy() function starts copying data god knows where:

```
   mpChunk = (sal_Char *) realloc( mpChunk, mnChunkLength );
}
strncpy(mpChunk + nWritePosition, pValue, nValueLength);
```

But it's really another thing that the analyzer didn't like. Suppose we have some error handler in the program that will keep it executing. But from that moment, we will be dealing with memory leak. NULL will be written into the mpChunk variable and memory freeing will become impossible. I'll explain this bug pattern in more detail. Many people don't stop to think twice about how to use realloc(), so they tend to do it in a wrong way.

Let's examine an artificial code sample:

```
char *p = (char *)malloc(10);
....
p = (char *)realloc(p, 10000);
```

If memory cannot be allocated, the 'p' variable will get "spoiled". Now there's no chance left to free the memory pointed to by the pointer previously stored in 'p'.

The bug is clear and prominent in this form. But such code is still pretty frequent in practice. The analyzer generates 8 more warnings of the same kind but we won't discuss them. LibreOffice believes it can get memory anytime, anyway.

Logical errors

I came across a few funny errors in conditions. I guess there are different reasons behind them - carelessness, typos, poor language knowledge.


```
void ScPivotLayoutTreeListData::PushDataFieldNames(....)
{
  ....
  ScDPLabelData* pLabelData = mpParent->GetLabelData(nColumn);

  if (pLabelData == NULL && pLabelData->maName.isEmpty())
    continue;
  ....
}
```

PVS-Studio's diagnostic message: V522 Dereferencing of the null pointer 'pLabelData' might take place. Check the logical condition. pivotlayouttreelistdata.cxx 157

It's a logical error in the condition: If the pointer is null, let's dereference it. As far as I can get it, the programmer should have used the || operator here.

A similar error:

```
void grabFocusFromLimitBox( OQueryController& _rController )
{
  ....
  vcl::Window* pWindow = VCLUnoHelper::GetWindow( xWindow );
  if( pWindow || pWindow->HasChildPathFocus() )
  {
    pWindow->GrabFocusToDocument();
  }
  ....
}
```

PVS-Studio's diagnostic message: V522 Dereferencing of the null pointer 'pWindow' might take place. Check the logical condition. querycontroller.cxx 293

In this fragment, on the contrary, '&&' should have been written instead of '||'.

Now a bit more complex condition:

```
enum SbxDataType {
  SbxEMPTY    =  0,
  SbxNULL     =  1,
  ....
};

void SbModule::GetCodeCompleteDataFromParse(CodeCompleteDataCache& aCache)
{
  ....
  if( (pSymDef->GetType() != SbxEMPTY) ||
      (pSymDef->GetType() != SbxNULL) )
  ....
}
```

PVS-Studio's diagnostic message: V547 Expression is always true. Probably the '&&' operator should be used here. sbxmod.cxx 1777

To make it simpler, I'll rewrite the expression for you:

```
if (type != 0 || type != 1)
```

You see, it is always true.

Two similar bugs can be found in the following fragments:

- V547 Expression is always true. Probably the '&&' operator should be used here. sbxmod.cxx 1785
- V547 Expression is always false. Probably the '||' operator should be used here. xmlstylesexporthelper.cxx 223

I also saw two fragments with excessive conditions. I believe these are errors:

```
sal_uInt16 ScRange::ParseCols(....)
{
  ....
  const sal_Unicode* p = rStr.getStr();
  ....
  case formula::FormulaGrammar::CONV_XL_R1C1:
    if ((p[0] == 'C' || p[0] != 'c') &&
        NULL != (p = lcl_r1c1_get_col(
                            p, rDetails, &aStart, &ignored )))
```

```
    {
  ....
}
```

PVS-Studio's diagnostic message: V590 Consider inspecting the 'p[0] == 'C' || p[0] != 'c'' expression. The expression is excessive or contains a misprint. address.cxx 1593

The (p[0] == 'C' || p[0] != 'c') condition can be reduced to (p[0] != 'c'). I'm sure it's an error and the condition should have really looked as follows: (p[0] == 'C' || p[0] == 'c').

An identical bug can be found in the same file a bit further:

- V590 Consider inspecting the 'p[0] == 'R' || p[0] != 'r'' expression. The expression is excessive or contains a misprint. address.cxx 1652

I guess we can also call it a logical error when a pointer is first dereferenced and only then checked for being null. This is a very common bug in every program. It usually occurs due to carelessness while doing code refactoring.

Here's a typical example:

```
IMPL_LINK(....)
{
  ....
  SystemWindow *pSysWin = pWindow->GetSystemWindow();
  MenuBar      *pMBar   = pSysWin->GetMenuBar();
  if ( pSysWin && pMBar )
  {
    AddMenuBarIcon( pSysWin, true );
  }
  ....
}
```

PVS-Studio's diagnostic message: V595 The 'pSysWin' pointer was utilized before it was verified against nullptr. Check lines: 738, 739. updatecheckui.cxx 738

The 'pSysWin' pointer is dereferenced in the 'pSysWin->GetMenuBar()' expression and then is checked for being null.

I suggest that LibreOffice's authors also review the following fragments: LibreOffice-V595.txt.

And the last error, a more complicated one. If you feel tired, you may skip to the next section. In the code below, we are dealing with a typical enumeration:

```
enum BRC_Sides
{
    WW8_TOP = 0, WW8_LEFT = 1, WW8_BOT = 2,
    WW8_RIGHT = 3, WW8_BETW = 4
};
```

Notice that the named constants are not a power of two - they are just numbers. And there's 0 among them.

But the programmer is working with them as if they were a power of two - trying to select and check single bits by mask:

```
void SwWW8ImplReader::Read_Border(....)
{
  ....
  if ((nBorder & WW8_LEFT)==WW8_LEFT)
    aBox.SetDistance(
      (sal_uInt16)aInnerDist.Left(), BOX_LINE_LEFT );

  if ((nBorder & WW8_TOP)==WW8_TOP)
    aBox.SetDistance(
      (sal_uInt16)aInnerDist.Top(), BOX_LINE_TOP );

  if ((nBorder & WW8_RIGHT)==WW8_RIGHT)
    aBox.SetDistance(
      (sal_uInt16)aInnerDist.Right(), BOX_LINE_RIGHT );

  if ((nBorder & WW8_BOT)==WW8_BOT)
    aBox.SetDistance(
      (sal_uInt16)aInnerDist.Bottom(), BOX_LINE_BOTTOM );
  ....
}
```

PVS-Studio's diagnostic message: V616 The 'WW8_TOP' named constant with the value of 0 is used in the bitwise operation. ww8par6.cxx 4742

The programmer shouldn't have done that. For example, the ((nBorder & WW8_TOP)==WW8_TOP) condition appears to be always true. To make it clear, I'll substitute the numbers: ((nBorder & 0)==0).

The check for WW8_LEFT doesn't work right either when the nBorder variable stores the value WW8_RIGHT equal to 3. Let's substitute the numbers: ((3 & 1) == 1). It turns out that WW8_RIGHT will be mixed up with WW8_LEFT.

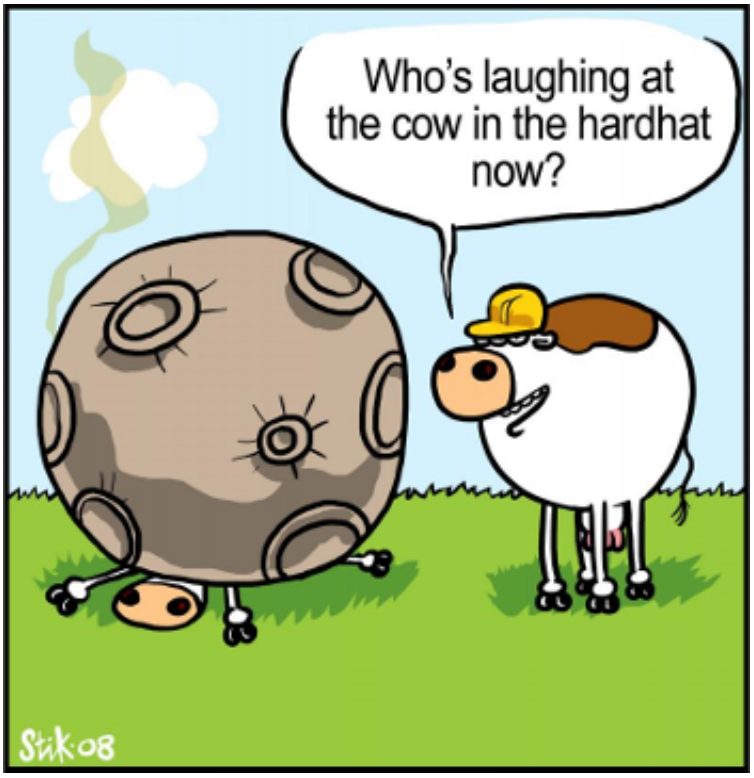
Skeleton in the closet

Every now and then, the analyzer would detect abnormal fragments in the code. These are not errors but programmer's clever tricks. It's no use touching them but they can be just interesting to study. Here's one of such cases where the analyzer didn't like the argument of the free() function:

```
/* This operator is supposed to be unimplemented, but that now leads
 * to compilation and/or linking errors with MSVC2008. (Don't know
 * about MSVC2010.) As it can be left unimplemented just fine with
 * gcc, presumably it is never called. So do implement it then to
 * avoid the compilation and/or linking errors, but make it crash
```

```
 * intentionally if called.
 */
void SimpleReferenceObject::operator delete[](void * /* pPtr */)
{
  free(NULL);
}
```

Safety rules



Among other things, the analyzer detected a few issues that make the program's code dangerous. The nature of these dangers varies but I decided to group them all in one section.

```
void writeError( const char* errstr )
{
  FILE* ferr = getErrorFile( 1 );
  if ( ferr != NULL )
  {
    fprintf( ferr, errstr );
    fflush( ferr );
  }
}
```

PVS-Studio's diagnostic message: V618 It's dangerous to call the 'fprintf' function in such a manner, as the line being passed could contain format specification. The example of the safe code: printf("%s", str); unoapploader.c 405

If the 'errstr' string contains control characters, any kind of trouble may occur. The program may crash, write rubbish into the file, and so on (details here).

The correct way of writing it would be as follows:

```
fprintf( ferr, "%s", errstr );
```

Here are two more fragments where the printf() function is used in a wrong way:

- climaker_app.cxx 261
- climaker_app.cxx 313

Now a dangerous use of dynamic_cast.

```
virtual ~LazyFieldmarkDeleter()
{
  dynamic_cast<Fieldmark&>
    (*m_pFieldmark.get()).ReleaseDoc(m_pDoc);
}
```

PVS-Studio's diagnostic message: V509 The 'dynamic_cast<T&>' operator should be located inside the try..catch block, as it could potentially generate an exception. Raising exception inside the destructor is illegal. docbm.cxx 846

When working with references, the dynamic_cast operator throws the std::bad_cast exception when conversion is impossible.

If an exception occurs in the program, stack unwinding begins which causes the objects to be destroyed by calling their destructors. If the destructor of an object being destroyed during stack unwinding throws another exception and this exception leaves the destructor, the C++ library will immediately trigger a program crash by calling the terminate() function. Therefore, destructors should never spread exceptions and if an exception is thrown it should be processed inside the same destructor.

Due to the same reason, it is dangerous to call the new operator inside destructors. When the program is short of memory, this operator will generate the std::bad_alloc exception. A good coding style is to wrap it in a try-catch block.

Here's an example of dangerous code:

```
WinMtfOutput::~WinMtfOutput()
{
  mpGDIMetaFile->AddAction( new MetaPopAction() );
  ....
}
```

PVS-Studio's diagnostic messages: V509 The 'new' operator should be located inside the try..catch block, as it could potentially generate an exception. Raising exception inside the destructor is illegal.

winmtf.cxx 852

Here's the list of all the other dangerous issues inside destructors:

- V509 The 'dynamic_cast<T&>' operator should be located inside the try..catch block, as it could potentially generate an exception. Raising exception inside the destructor is illegal. ndtxt.cxx 4886
- V509 The 'new' operator should be located inside the try..catch block, as it could potentially generate an exception. Raising exception inside the destructor is illegal. export.cxx 279
- V509 The 'new' operator should be located inside the try..catch block, as it could potentially generate an exception. Raising exception inside the destructor is illegal. getfilenamewrapper.cxx 73
- V509 The 'new' operator should be located inside the try..catch block, as it could potentially generate an exception. Raising exception inside the destructor is illegal. e3dsceneupdater.cxx 80
- V509 The 'new' operator should be located inside the try..catch block, as it could potentially generate an exception. Raising exception inside the destructor is illegal. accmap.cxx 1683
- V509 The 'new' operator should be located inside the try..catch block, as it could potentially generate an exception. Raising exception inside the destructor is illegal. frmtool.cxx 938

By the way, since we have started talking about the new operator, I'd like to speak about the following code and the danger hidden in it:

```
extern "C" oslFileHandle
SAL_CALL osl_createFileHandleFromOSHandle(
  HANDLE     hFile,
  sal_uInt32 uFlags)
{
  if ( !IsValidHandle(hFile) )
      return 0; // EINVAL

  FileHandle_Impl * pImpl = new FileHandle_Impl(hFile);
  if (pImpl == 0)
  {
    // cleanup and fail
    (void) ::CloseHandle(hFile);
    return 0; // ENOMEM
  }
  ....
}
```

PVS-Studio's diagnostic message: V668 There is no sense in testing the 'pImpl' pointer against null, as the memory was allocated using the 'new' operator. The exception will be generated in the case of memory allocation error. file.cxx 663

The 'new' operator throws an exception when the program is short of memory. So checking the pointer returned by the operator won't make sense: It will always be not equal to 0. When there's not enough memory, the CloseHandle() function won't be called:

```
FileHandle_Impl * pImpl = new FileHandle_Impl(hFile);
if (pImpl == 0)
{
  // cleanup and fail
  (void) ::CloseHandle(hFile);
  return 0; // ENOMEM
}
```

Keep in mind that I may be wrong as I'm not familiar with the LibreOffice project. Perhaps the developers use some special library versions where the 'new' operator returns nullptr instead of throwing an exception. If so, then please just ignore the V668 warnings. You can turn them off so they don't bother you.

But if the new operator does throw an exception, please check the following 126 warnings: LibreOffice-V668.txt.

The next danger is found in the implementation of one of the DllMain functions:

```
BOOL WINAPI DllMain( HINSTANCE hinstDLL,
                     DWORD fdwReason, LPVOID lpvReserved )
{
  ....
  CreateThread( NULL, 0, ParentMonitorThreadProc,
                (LPVOID)dwParentProcessId, 0, &dwThreadId );
  ....
}
```

PVS-Studio's diagnostic message: V718 The 'CreateThread' function should not be called from 'DllMain' function. dllentry.c 308

A large number of functions can't be called inside DllMain() as it may cause an application hang or other errors. CreateThread() is among those prohibited functions.

The trouble with DllMain is well described at MSDN: Dynamic-Link Library Best Practices.

This code may work well but it is dangerous and will fail you one day.

I also encountered an issue when the wcsncpy() function may cause a buffer overflow:

```
typedef struct {
  ....
  WCHAR wszTitle[MAX_COLUMN_NAME_LEN];
  WCHAR wszDescription[MAX_COLUMN_DESC_LEN];
} SHCOLUMNINFO, *LPSHCOLUMNINFO;

HRESULT STDMETHODCALLTYPE CColumnInfo::GetColumnInfo(
  DWORD dwIndex, SHCOLUMNINFO *psci)
{
  ....
```

```
  wcsncpy(psci->wszTitle,
          ColumnInfoTable[dwIndex].wszTitle,
          (sizeof(psci->wszTitle) - 1));
  return S_OK;
}
```

PVS-Studio's diagnostic message: V512 A call of the 'wcsncpy' function will lead to overflow of the buffer 'psci->wszTitle'. columninfo.cxx 129

The (sizeof(psci->wszTitle) - 1) expression is wrong: The programmer forgot to divide it by the size of one character:

```
(sizeof(psci->wszTitle) / sizeof(psci->wszTitle[0]) - 1)
```

The last bug type we will discuss in this section is about malfunctioning memset() calls. For example:

```
static void __rtl_digest_updateMD2 (DigestContextMD2 *ctx)
{
  ....
  sal_uInt32 state[48];
  ....
  memset (state, 0, 48 * sizeof(sal_uInt32));
}
```

PVS-Studio's diagnostic message: V597 The compiler could delete the 'memset' function call, which is used to flush 'state' buffer. The RtlSecureZeroMemory() function should be used to erase the private data. digest.cxx 337

I already wrote a lot about this bug pattern. So now I'll just briefly describe it, and you may check the links below for details.

The compiler has the right to remove a call of the memset() function when zeroed memory is not used in any way after that call. And this is exactly what happens in the code cited above. It will result in retaining some of the private data in memory.

References:

1. V597. The compiler could delete the 'memset' function call, which is used to flush 'Foo' buffer.
2. Overwriting memory - why?
3. Zero and forget -- caveats of zeroing memory in C .

Here's the list of other fragments where private data fail to be cleared: LibreOffice-V597.txt.

Miscellaneous

```
Guess::Guess()
{
  language_str = DEFAULT_LANGUAGE;
```

```
  country_str = DEFAULT_COUNTRY;
  encoding_str = DEFAULT_ENCODING;
}

Guess::Guess(const char * guess_str)
{
  Guess();
  ....
}
```

PVS-Studio's diagnostic message: V603 The object was created but it is not being used. If you wish to call constructor, 'this->Guess::Guess(....)' should be used. guess.cxx 56

The programmer who wrote this code is not very good at the C++ language. They intended to call one constructor from another. But actually they created a temporary unnamed object. Because of this error, some class fields will remain uninitialized. Details here.

Another poorly implemented constructor: camera3d.cxx 46

```
sal_uInt32 readIdent(....)
{
  size_t nItems = rStrings.size();
  const sal_Char** pStrings = new const sal_Char*[ nItems+1 ];
  ....
  delete pStrings;
  return nRet;
}
```

PVS-Studio's diagnostic message: V611 The memory was allocated using 'new T[]' operator but was released using the 'delete' operator. Consider inspecting this code. It's probably better to use 'delete [] pStrings;'. profile.hxx 103

The correct code: delete [] pStrings;.

There was another warning about incorrect memory freeing:

- V611 The memory was allocated using 'new T[]' operator but was released using the 'delete' operator. Consider inspecting this code. It's probably better to use 'delete [] pStrings;'. profile.hxx 134

```
static const int kConventionShift = 16;
static const int kFlagMask = ~((~int(0)) << kConventionShift);
```

PVS-Studio's diagnostic message: V610 Undefined behavior. Check the shift operator '<<'. The left operand '(~int (0))' is negative. grammar.hxx 56

There is also an issue with undefined behavior because of a negative number shift (details here).

```
sal_Int32 GetMRest() const {return m_nRest;}
```

```
OUString LwpBulletStyleMgr::RegisterBulletStyle(....)
{
  ....
  if (pIndent->GetMRest() > 0.001)
  ....
}
```

PVS-Studio's diagnostic message: V674 The '0.001' literal of the 'double' type is compared to a value of the 'long' type. Consider inspecting the 'pIndent->GetMRest() > 0.001' expression. lwpbulletstylemgr.cxx 177

Something is not right here. It doesn't make sense to compare an integer number to 0.001.

An annoying mess with a return value's type:

```
BOOL SHGetSpecialFolderPath(
  HWND hwndOwner,
  _Out_  LPTSTR lpszPath,
  _In_   int csidl,
  _In_   BOOL fCreate
);

#define FAILED(hr) (((HRESULT)(hr)) < 0)

OUString UpdateCheckConfig::getDesktopDirectory()
{
  ....
  if( ! FAILED( SHGetSpecialFolderPathW( .... ) ) )
  ....
}
```

PVS-Studio's diagnostic message: V716 Suspicious type conversion: BOOL -> HRESULT. updatecheckconfig.cxx 193

The programmer decided that SHGetSpecialFolderPath() would return the HRESULT type. But actually it returns BOOL. To fix the code, we should remove the FAILED macro from the condition.

Another error of this kind: updatecheckconfig.cxx 222

And here, on the contrary, we are lacking the FAILED macro. One can't check an HRESULT status like this:

```
bool UniscribeLayout::LayoutText( ImplLayoutArgs& rArgs )
{
  ....
  HRESULT nRC = ScriptItemize(....);
  if( !nRC ) // break loop when everything is correctly itemized
    break;
```

```
  ....
}
```

PVS-Studio's diagnostic message: V545 Such conditional expression of 'if' operator is incorrect for the HRESULT type value 'nRC'. The SUCCEEDED or FAILED macro should be used instead. winlayout.cxx 1115

In the following code, I guess, the comma should be replaced with a semicolon:

```
void Reader::ClearTemplate()
{
    if( pTemplate )
    {
        if( 0 == pTemplate->release() )
            delete pTemplate,
        pTemplate = 0;
    }
}
```

PVS-Studio's diagnostic message: V626 Consider checking for misprints. It's possible that ',' should be replaced by ';'. shellio.cxx 549

Some trifle:

```
void TabBar::ImplInit( WinBits nWinStyle )
{
  ....
  mbMirrored = false;
  mbMirrored = false;
  ....
}
```

PVS-Studio's diagnostic message: V519 The 'mbMirrored' variable is assigned values twice successively. Perhaps this is a mistake. Check lines: 415, 416. tabbar.cxx 416

And another one: V519 The 'aParam.mpPreviewFontSet' variable is assigned values twice successively. Perhaps this is a mistake. Check lines: 4561, 4562. output2.cxx 4562

An incorrect magic constant specifying the string length:

```
static bool CallRsc2(....)
{
  ....
  if( !rsc_strnicmp( ....,  "-fp=", 4 ) ||
      !rsc_strnicmp( ...., "-fo=", 4 ) ||
      !rsc_strnicmp( ...., "-presponse", 9 ) ||   <<<<----
      !rsc_strnicmp( ...., "-rc", 3 ) ||
      !rsc_stricmp( ...., "-+" ) ||
```

```
        !rsc_stricmp( ...., "-br" ) ||
        !rsc_stricmp( ...., "-bz" ) ||
        !rsc_stricmp( ...., "-r" ) ||
        ( '-' != *.... ) )
    ....
}
```

PVS-Studio's diagnostic message: V666 Consider inspecting third argument of the function 'rsc_strnicmp'. It is possible that the value does not correspond with the length of a string which was passed with the second argument. start.cxx 179

The length of the "-presponse" string is 10 characters, not 9.

A strange 'break' inside a loop:

```
OUString getExtensionFolder(....)
{
  ....
  while (xResultSet->next())
  {
    title = Reference<sdbc::XRow>(
      xResultSet, UNO_QUERY_THROW )->getString(1 /* Title */ ) ;
    break;
  }
  return title;
}
```

PVS-Studio's diagnostic message: V612 An unconditional 'break' within a loop. dp_manager.cxx 100

Three other strange loops:

- V612 An unconditional 'break' within a loop. svdfppt.cxx 3260
- V612 An unconditional 'break' within a loop. svdfppt.cxx 3311
- V612 An unconditional 'break' within a loop. personalization.cxx 454

Unlikely null pointer dereferencing:

```
BSTR PromptNew(long hWnd)
{
  ....
  ADOConnection* piTmpConnection = NULL;

  ::CoInitialize( NULL );

  hr = CoCreateInstance(
                CLSID_DataLinks,
                NULL,
                CLSCTX_INPROC_SERVER,
                IID_IDataSourceLocator,
```

```
                (void**)&dlPrompt
                );
  if( FAILED( hr ) )
  {
    piTmpConnection->Release();
    dlPrompt->Release( );
    return connstr;
  }
  ....
}
```

PVS-Studio's diagnostic message: V522 Dereferencing of the null pointer 'piTmpConnection' might take place. adodatalinks.cxx 84

If the CoCreateInstance() function happens to return the error status, the 'piTmpConnection' pointer which is equal to NULL will be dereferenced.

Microoptimizations

A static analyzer in no way can serve as a substitute for profiling tools. Only a profiler can suggest what fragments of your program should be optimized.

Nevertheless, a static analyzer can point out those places than can easily and safely be improved. It doesn't necessarily mean that the program will run faster but it definitely won't make it worse. I think we should rather treat it as a way to improve the coding style.

Let's see what recommendations about microoptimizations PVS-Studio has to offer.

Passing objects by reference

If an object passed into a function doesn't change, it would be nicer to pass it by reference, not by value. Of course, it doesn't concern each and every object. But when we are dealing with strings, for example, there's no sense allocating memory and copying the string's contents to no purpose.

For example:

```
string getexe(string exename, bool maybeempty) {
  char* cmdbuf;
  size_t cmdlen;
  _dupenv_s(&cmdbuf, &cmdlen, exename.c_str());
  if(!cmdbuf) {
    if (maybeempty) {
      return string();
    }
    cout << "Error " << exename << " not defined. "
      "Did you forget to source the environment?" << endl;
    exit(1);
  }
  string command(cmdbuf);
  free(cmdbuf);
  return command;
}
```

The 'exename' object is read-only. That's why the analyzer generates the following message: V813Decreased performance. The 'exename' argument should probably be rendered as a constant reference. wrapper.cxx 18

The function declaration should be changed in the following way:

```
string getexe(const string &exename, bool maybeempty)
```

Passing complex objects by a constant reference is usually more efficient and allows avoiding the "slicing" problem. Those who are not quite familiar with the issue, please see "Item 20. Prefer pass-by-reference-to-const to pass-by-value" from the book:

Effective C++, Third Edition: 55 Specific Ways to Improve Your Programs and Designs, by Scott Meyers. Copyright © 2005 by Pearson Education, Inc. ISBN: 0-321-33487-6.

Another related diagnostic is V801. The analyzer generated total 465 warnings where it suggested that objects should be passed by reference: LibreOffice-V801-V813.txt.

Using the prefix increment

For iterators, the prefix increment operation is a bit faster. To learn more about it, see "Item 6. Distinguish between prefix and postfix forms of increment and decrement operators" from the book:

More Effective C++: 35 New Ways to Improve Your Programs and Designs, by Scott Meyers. ISBN 0-201-63371-X

You may find this recommendation farfetched, for there's presumably no difference between 'A++' and '++A' in practice. I have investigated this question and carried out some experiments and believe that this recommendation should be followed (details here).

For example:

```
typename InterfaceMap::iterator find(const key &rKey) const
{
  typename InterfaceMap::iterator iter = m_pMap->begin();
  typename InterfaceMap::iterator end = m_pMap->end();

  while( iter != end )
  {
    equalImpl equal;
    if( equal( iter->first, rKey ) )
      break;
    iter++;
  }
  return iter;
}
```

PVS-Studio's diagnostic message: V803 Decreased performance. In case 'iter' is iterator it's more effective to use prefix form of increment. Replace iterator++ with ++iterator. interfacecontainer.h 405

The "iter++" expression should be replaced with "++iter". I don't know if the developers find it worthy to spend some time on it, but if they do, here's 257 more places where the postfix increment can be replaced with the prefix one: LibreOffice-V803.txt.

Checking for an empty string

To find out if a string is empty, you don't need to calculate its length. Here's an example of inefficient code:

```
BOOL GetMsiProp(....)
{
  ....
  char* buff = reinterpret_cast<char*>( malloc( nbytes ) );
  ....
  return ( strlen(buff) > 0 );
}
```

PVS-Studio's diagnostic message: V805 Decreased performance. It is inefficient to identify an empty string by using 'strlen(str) > 0' construct. A more efficient way is to check: str[0] != '\0'. sellang.cxx 49

What makes it inefficient is that the program has to sort through all the characters in the string until it encounters the terminal null. But it is actually enough to check one byte only:

```
return buff[0] != '\0';
```

This code doesn't look neat, so we'd better create a special function:

```
inline bool IsEmptyStr(const char *s)
{
  return s == nullptr || s[0] == '\0';
}
```

Now we've got an extra check of a pointer for null. I don't like it, so you may think of some other ways to implement it. But still, even in this form, the function will run faster than strlen().

Other inefficient checks: LibreOffice-V805.txt.

Miscellaneous

There were a few other warnings that may be interesting: LibreOffice-V804_V811.txt.

The number of false positives

I mentioned 240 warnings that I found worthy. In total, the analyzer generated about 1500 general warnings (GA) of the 1-st and 2-nd levels. Does it mean that the analyzer generates too many false positives? No, it doesn't. Most warnings point out real issues and are quite relevant but I didn't find them interesting to discuss in the article.

Every now and then, we get positive replies from our users, telling us, "The PVS-Studio analyzer produces pretty few false positives, and that's very convenient." We, too, believe our tool doesn't generate too many false positives. But how come? We only told you about 16% of the messages. What's the rest? Aren't they false positives?

Well, of course there is some amount of false positives among them. You just can't avoid them all completely. In order to suppress them, we offer a number of mechanisms in our analyzer. But most of the warnings, though not pointing to real errors, revealed code with a smell. I'll try to explain it by a few examples.

The analyzer generated **206 V690** V690 warnings about a class containing a copy constructor but missing an assignment operator. Here's one of these classes:

```
class RegistryTypeReader
{
public:
  ....
  inline RegistryTypeReader(const RegistryTypeReader& toCopy);
  ....
};

inline RegistryTypeReader::RegistryTypeReader(const RegistryTypeReader& toCopy)
  : m_pApi(toCopy.m_pApi)
  , m_hImpl(toCopy.m_hImpl)
  { m_pApi->acquire(m_hImpl); }
```

There's hardly any error here. It is most likely that the = operator is not used in all the 206 classes. But what if it is?

The programmer has to make a choice.

If they believe the code is dangerous, then they should implement an assignment operator or forbid it. If they don't think the code is dangerous, the V690 diagnostic may be disabled and the list of the diagnostic messages will immediately become 206 warnings shorter.

Another example. Earlier in the article, I mentioned the following suspicious fragment:

```
if( pInit )
  AddMap( pInit );
if( pInit )
  AddMap( pInit2 );
```

It was diagnosed by the V581 rule. But, to be honest, I just briefly scanned through the V581 warnings and could have missed a lot. You see, there are 70 more of them. And the analyzer is not to blame. How does it know why the programmer would like to write code like this:

```
static bool lcl_parseDate(....)
{
  bool bSuccess = true;
  ....
  if (bSuccess)
  {
    ++nPos;
```

```
  }

  if (bSuccess)
  {
    bSuccess =
      readDateTimeComponent(string, nPos, nDay, 2, true);
  ....
}
```

'bSuccess' is checked twice. What if it is some other variable that should have been checked for the second time?

Again, it's up to the programmer to decide what to do with these 70 warnings. If they like to have a sequence of identical checks to reveal some logical blocks, then the analyzer is certainly wrong. Then the V581 diagnostic should be turned off to get rid of 70 warnings at once.

If the programmer is not that confident, they will have to do something about that. For example, refactor the code:

```
static bool lcl_parseDate(....)
{
  bool bSuccess = true;
  ....
  if (bSuccess)
  {
    ++nPos;
    bSuccess =
      readDateTimeComponent(string, nPos, nDay, 2, true);
  ....
}
```

So, the basic idea I'm trying to communicate to you is that there is no serious problem with false positives. If you think some group of warnings is not relevant for your particular project, you can simply disable them, thus making the list of the diagnostic messages you'll have to examine much shorter. If, on the contrary, you think the code should be reviewed and fixed, then these are in no way false messages but absolutely true and relevant ones.

Note. You can get started with the analyzer without having to review hundreds or thousands of messages. Just use our new message marking mechanism. It is useful when you need to hide all the present warnings to work only with those generated for freshly written code. You can return to bugs in the old code at any moment when you have time for that.

Conclusion

Although the number of errors, defects, and slip-ups discussed in this article is, as usual, great, the LibreOffice project's code is still very high-quality. And it does bear the evidence of being regularly checked by Coverity, which indicates the authors' serious approach to the development. The number of bugs found by PVS-Studio is pretty small for such a large project like LibreOffice.

What did I mean to say by this article? Well, nothing special, really. It's a bit of advertising for our tool, and that's all. Use the PVS-Studio static analyzer regularly to find and fix piles of errors at the earliest development stages.

I'm like the cow in the last picture - laid a pile of errors and ran away. And LibreOffice's authors will now have to sort it all out. Sorry for that. It's just my job.