Crash Testing and Coverity
The Numbers

Caolán McNamara, Red Hat
2015-09-25
• Coverity
  • Examples
  • Defect Density
  • Trends
• Crash Testing
  • Process
  • Trends
Examples
class SQLCommandPropertyUI : public ISQLCommandPropertyUI
{
    protected:
        SQLCommandPropertyUI( const Reference< XPropertySet >& _rxObject )
            : m_xObject( _rxObject )
        {
            1. Condition "this->m_xObject.is()", taking false branch
                if ( !m_xObject.is() )
                    throw ::rtl::CppRuntimeException();
        }
    virtual oslInterlockedCount SAL_CALL acquire()
    {
        return osl_atomic_increment( &m_refCount );
    }
    virtual oslInterlockedCount SAL_CALL release()
    {
        if ( 0 == osl_atomic_decrement( &m_refCount ) )
        {
            delete this;
            return 0;
        }
        return m_refCount;
    }
    protected:
        Reference< XPropertySet >& m_xObject;
private:
    2. memberDecl: Class member declaration for "m_refCount".
        oslInterlockedCount m_refCount;
};
bool ImplGetInvalidAsciiMultiByte(sal_uInt32 nFlags, char *pBuf, sal_Size nMaxLen)
{
    if (nMaxLen == 0)
        return false;
    switch (nFlags & RTL_UNICODETEXT_FLAGS_UNDEFINED_MASK)
    {
        case RTL_UNICODETEXT_FLAGS_INVALID_DEFAULT:
            *pBuf = 0x00;
            break;
        case RTL_UNICODETEXT_FLAGS_INVALID_QUESTIONMARK:
            *pBuf = 0x3F;
            break;
        case RTL_UNICODETEXT_FLAGS_INVALID_UNDERLINE:
            *pBuf = 0x5F;
            break;
    }
    return true;
}

dead_error_condition: The switch value nFlags & 0xfU cannot be 80U.

dead_error_begin: Execution cannot reach this statement case 80U.

Copy and Paste from previous ImplGetUndefinedAsciiMultiByte without corresponding change of UNDEFINED_MASK to INVALID_MASK
That doesn't actually specify what it throws
Somebody got confused on checking the result of dynamic_cast
CID#704127 CONSTANT_EXPRESSION_RESULT

```c
void WWBDopTypography::WriteToMem(sal_uInt8 *pData) const
{
    sal_uInt16 a16Bit = fKerningPunct;
    a16Bit |= (iJustification << 1) & 0x0006;
    a16Bit |= (iLevel0fKinsoku << 3) & 0x0018;

    // CID 704127 (#1 of 1): Wrong operator used (CONSTANT_EXPRESSION_RESULT)
    // operator_confusion: "(this->f2on1 << 5) & 2" is always 0 regardless of the values of its operands. This occurs as the bitwise operand of '|='. Did you intend to use right-shift ('>>') in "this->f2on1 << 5"?

    a16Bit |= (f2on1 << 5) & 0x0002;
    a16Bit |= (reserved1 << 6) & 0x0030;
    a16Bit |= (reserved2 << 10) & 0xFC00;
    Set_UInt16(pData, a16Bit);

    Set_UInt16(pData, cchFollowingPunct);
    Set_UInt16(pData, cchLeadingPunct);

    sal_Int16 i;
    for (i=0; i < rMaxFollowing; ++i)
        Set_UInt16(pData, rgxchFPunct[i]);
    for (i=0; i < rMaxLeading; ++i)
        Set_UInt16(pData, rgxchLPunct[i]);
}
```

typo, should be 0x0020 not 0x002, wrong for 14 years
Defect Density

Open Source Defect Density

LibreOffice: 7,102,667 line of code and 0.00 defect density

Open Source Defect Density By Project Size

<table>
<thead>
<tr>
<th>Line of Code (LOC)</th>
<th>Defect Density</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 100,000</td>
<td>0.35</td>
</tr>
<tr>
<td>100,000 to 499,999</td>
<td>0.5</td>
</tr>
<tr>
<td>500,000 to 1 million</td>
<td>0.7</td>
</tr>
<tr>
<td>More than 1 million</td>
<td>0.65</td>
</tr>
</tbody>
</table>

Note: Defect density is measured by the number of defects per 1,000 lines of code, identified by the Coverity platform. The numbers shown above are from our 2013 Coverity Scan Report, which analyzed 250 million lines of open source code.

Last Years density at conference time was 0.08
Defects over time

Here, “ignored” third party module warnings are counted.
Process integration

- Now run about twice a week
  - Those are the nums of slots coverity makes available to a project of this size
- Typically back to back
  - One to collect warnings
  - One after warnings fixed
- Results now mailed to the list
- Takes about 4-6 hours to build
- Takes about 12+ hours to analyze server-side
What it does

- Loads a bunch of documents
  - 118 different columns for formats in output
  - Some are now sort of pointless, e.g. staroffice binary format
  - See if anything crashes or triggers an assert
- Saves a bunch of documents
  - Exports to 12 different formats from all the compatible import formats
  - Export to doc, docx, odb, odg, odp, ods, odt, ppt, pptx, rtf, xls, xlsx
Process integration

- Typically run once or two a week
  - Takes about two days to complete
- Approx 80,000 documents in the document horde
  - Mostly populated from get-bugzilla-by-mimetype
  - + cloudon test documents
  - + w3c svg test documents
  - + various interesting documents that have caused trouble for some app or other in the past
Horde Updating

- Typically fairly rarely
- Full update takes about 12/13 hours
- Downloads are cached, so only new documents are updated
- Bugzilla is trusted wrt the mime-type
  - Lots of miscategorized stuff
  - Doesn't really matter, rtf's pretending to be docs, etc
  - Just made doc import filter look a little worse than it was
Import Failure Trends

Build 1 is 31 Oct 2013, final build was 16 Sep 2015
Export Failure Trends

Build 1 is 31 Oct 2013, final build was 16 Sep 2015
Triple 0 week

- 20 – 27 August 2015
- 0 coverity warnings
- 0 import failures
- 0 export failures

Then everyone came back from their Summer holidays
This week

- 4 (fixed) coverity warnings, pending next build
- 0 import failures
- 4 export asserts (2 unique asserts)
- Fairly typical
Taking the battle onwards
Generating troublesome documents

- Fuzzing
- Played with CERT bff for a while, some small results
- American Fuzzy Lop is much more fun
  - Build with afl-clang/afl-clang++
  - “coverage-assisted fuzz testing tool”
  - Generates new documents that trigger new internal states in the target
  - Got to love the UI
### American Fuzzy Lop 1.86b (png)

<table>
<thead>
<tr>
<th>Process Timing</th>
<th>Overall Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run time: 0 days, 0 hrs, 0 min, 38 sec</td>
<td>Cycles done: 0</td>
</tr>
<tr>
<td>Last new path: 0 days, 0 hrs, 0 min, 0 sec</td>
<td>Total paths: 119</td>
</tr>
<tr>
<td>Last uniq crash: none seen yet</td>
<td>Uniq crashes: 0</td>
</tr>
<tr>
<td>Last uniq hang: none seen yet</td>
<td>Uniq hangs: 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cycle Progress</th>
<th>Map Coverage</th>
<th>Findings in Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Now processing: 9 (7.56%)</td>
<td>Map density: 8517 (13.00%)</td>
<td>Favored paths: 11 (9.24%)</td>
</tr>
<tr>
<td>Paths timed out: 0 (0.00%)</td>
<td>Count coverage: 1.13 bits/tuple</td>
<td>New edges on: 102 (85.71%)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stage Progress</th>
<th>Path Geometry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Now trying: havoc</td>
<td>Levels: 3</td>
</tr>
<tr>
<td>Stage execs: 15.6k/80.0k (19.48%)</td>
<td>Pending: 117</td>
</tr>
<tr>
<td>Total execs: 142k</td>
<td>Pend fav: 9</td>
</tr>
<tr>
<td>Exec speed: 3787/sec</td>
<td>Own finds: 117</td>
</tr>
</tbody>
</table>

**Fuzzing strategy yields**

- Bit flips: n/a, n/a, n/a
- Byte flips: n/a, n/a, n/a
- Arithmetics: n/a, n/a, n/a
- Known ints: n/a, n/a, n/a
- Dictionary: n/a, n/a, n/a
- Havoc: 96/105k, 12/20.8k
- Trim: 14.87%/288, n/a

[CPU: 27%]
Speed #1

- Crucial thing is to be able to cycle **fast**
- under 100 execs a second is super cruddy
- soffice.bin is ponderous to startup
  - 0.18 executions a second for pngs
  - Configuration loading and parsing is expensive
- Custom no ui, no config, application
  - After much hacking
  - 40 executions a second for pngs
  - Approximately 200 times faster
Speed #2

- “Persistent mode”
- Don't exit after each document
- Just loop over the same document again and again
- SIGSTOP to afl controller to signal ready again
- Build with afl-clang-fast/afl-clang-fast++
- Makes something of a difference
- 3000-4000 executions per second with custom loader
  - So that's approx 20,000 faster
Process/Results to date

- Between stock crash testing runs afl runs
- 64 core box
- Currently 20+ instances running for the last month or so
- Mostly on a different file format, can run multiple for a single file format
- Crashes rare
- Rich source of hangs
- Using afl-cmin minimized corpus of crash testing as input
Thanks for your time