



redhat

How to use data from static analysers efficiently?

Red Hat

Ondřej Vašík and Kamil Dudka

2014-02-07

Abstract

There is a lot of static analysers for searching programming issues in various languages. This presentation will focus on the most common ones for analyzing C/C++, especially Coverity, Cppcheck, CLang, and GCC warnings. We are going to cover efficient ways how to get the biggest benefits from using these tools, how to automate usage of these tools in your project and how to reduce the time spent on reviewing the results and fixing the real bugs introduced in the development.

Agenda

- 1 **Static Analysis – What Does It Mean?**
- 2 **How to Use Data from Static Analysers Efficiently?**
- 3 **How to Automate Static Analysis of RPM Packages?**

Static Analysis

- **generic definition**: analysis of code without executing it
- already done by the compiler (optimization, warnings, ...)
- we are interested in using static analysis to **find bugs**

Agenda

- 1 Static Analysis – What Does It Mean?
- 2 How to Use Data from Static Analysers Efficiently?**
- 3 How to Automate Static Analysis of RPM Packages?

Answer in 10 minutes?

- let's try that! :)

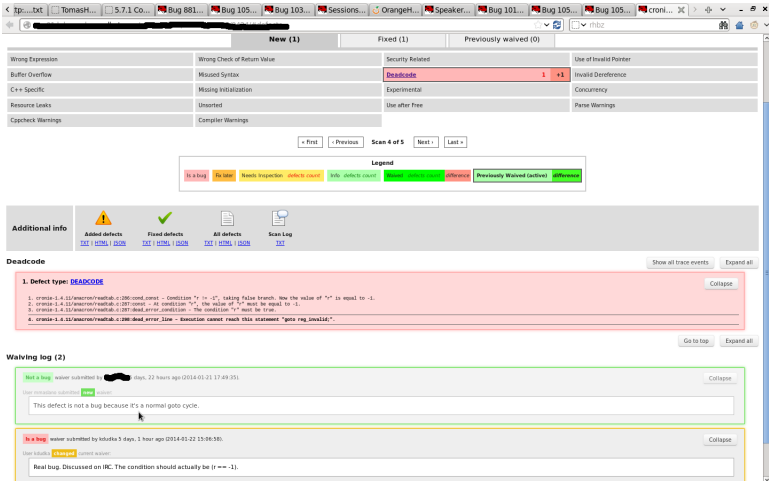
Use more static analyzers as frequently as possible

- continuous integration!
- don't use them in production build system
- be sure analysis takes some time - comparable to compilation time \implies balance is needed
- Fedora stats
 - 1500 C/C++ packages from 6500+ scanned
 - 150M lines of code \rightarrow 150k potential defects \rightarrow 50k+ real defects
 - Coverity+Cppcheck+Gcc warnings = 3+ weeks scan time
- You can join scan.coverity.com for free (if OSS)

Mark the false positives with annotations, use difference scan tools

- annotations → comments used to suppress the warnings
- difference scanning
 - 1st scan : 30 defects, 10 real, 20 FP
 - 2nd scan (after 6 months) : 5 new defects → 25 defects, 2 real, 23 FP
 - Using e.g. csdiff : 5 new defects : 2 real, 3 FP

Review the result carefully, think about the output twice!



The screenshot shows a web browser window displaying the Red Hat Insights interface for a static analysis. The browser tabs include 'Bug 681...', 'Bug 105...', 'Bug 103...', 'Sessions...', 'OrangeH...', 'Speaker...', 'Bug 101...', 'Bug 105...', 'Bug 105...', and 'cronl...'. The main content area is divided into sections:

- Summary:** Shows 'New (1)', 'Fixed (1)', and 'Previously waived (0)' items.
- Table:** A table with columns for 'Wrong Expression', 'Wrong Check of Return Value', 'Security Related', and 'Use of Invalid Pointer'. The 'Deadcode' item is highlighted in red.
- Navigation:** Buttons for '< First', 'Previous', 'Scan 4 of 5', 'Next', and 'Last >'.
- Legend:** A row of colored boxes with labels: 'Is a bug' (red), 'Is fixed' (green), 'Needs inspection' (orange), 'Defects count' (yellow), 'Info' (blue), 'Defects count' (green), 'Needs inspection' (orange), 'Difference' (red), and 'Previously Waived (active)' (green).
- Additional info:** A row of icons and links for 'Added defects', 'Fixed defects', 'All defects', and 'Scan Log'.
- Deadcode:** A section with a red background containing a list of 4 defect types. The first one is 'DEADCODE'. A 'Collapse' button is on the right.
- Waiving log (2):** A section with two entries. The first entry is green and says 'Not a bug' with a 'Collapse' button. The second entry is orange and says 'Is a bug' with a 'Collapse' button.

Focus on high priority defect types!

- Common "real bugs" defects:
 - use after free (71% marked real)
 - resource leak (65% marked real)
 - missing initialization (43% marked real, but highest "needs fix now" rate)
- In more details:

Defect type	Defect occurrence	Real defects	Needs fix
Wrong return value check	13.6 %	37.9 %	9.4 %
Deadcode	12.5 %	46.1 %	8.5 %
Invalid dereference	10.0 %	37.9 %	10.2 %
Resource leaks	8.1 %	65.5 %	10.9 %
Security related	8.0 %	40.2 %	18.0 %
Missing initialization	7.4 %	43.4 %	33.3 %

Focus on real-world defects?

- Might be possible in future! → a lot of crash data at <https://retrace.fedoraproject.org/faf/>
- tool to match crashes to static analyzer output in development (<https://github.com/mmilata/mock-with-analysis/tree/crash-correlation>) → still long run to get it more useful (hard to match backtrace to defects)

Agenda

- 1 Static Analysis – What Does It Mean?
- 2 How to Use Data from Static Analysers Efficiently?
- 3 **How to Automate Static Analysis of RPM Packages?**

How to Automate Static Analysis of RPMs?

- Static analysis already used by compilers (in our case GCC).
- How can we efficiently process **GCC warnings**?
- How can we plug **static analyzers** into our build process?
- How can we **fully automatically** analyze a given SRPM?
- How can we build a **differential static analysis** on top of it?

Processing GCC Warnings – Problems

- 1 Some projects produce a lot of warnings during build.
- 2 Some projects do not produce any warnings during build.
- 3 Compilers do not use **absolute paths** in diagnostic messages.
- 4 Some projects use obscure **build systems** (samba, ksh, etc.).
- 5 Compiler warnings are difficult to collect consistently when **building in parallel**.

Processing GCC Warnings – Solutions

- 1 We want to make sure that we are not introducing new warnings whenever changing the code (while not touching the code that is known to work).
⇒ We need an easy to use **diff tool** for compiler warnings.
- 2 We want to adjust **warning level** (ideally in a way that is fully transparent to the build process).
- 3 We need a tool to translate **relative** to **absolute** paths in diagnostic messages.
- 4 The easiest way is to put a **compiler wrapper** to \$PATH.
- 5 We can extend such a compiler wrapper to **synchronize** writes of diagnostic messages in order to get a usable output when building in parallel.

Plugging Static Analyzers into Build Process

- How to plug other static analyzers into the build process?
- We put another **compiler wrapper** to \$PATH because:
 - We do not want to scan code that is not going to run.
 - We use the exactly same configuration (header files, defines) as we use for build.
- Currently supported static analyzers:
 - **Coverity Analysis**
 - **Cppcheck**
 - **Clang**
 - GCC warnings (as mentioned above)

Coverity Analysis

- Enterprise static analyzer, closed source.
- Not for free, but available to open source project maintainers.
- During the build, Coverity only captures intermediate code to a so called **intermediate directory**.
- The static analysis itself runs in a separate step (it may even run on another machine).
- Provides its own **compiler wrapper**.

Cppcheck

- Based on **pattern matching** (very lightweight static analysis).
- Can be run blindly on a directory with sources, but then:
 - It tries several combinations of `-D` flags.
 - Cppcheck ignores missing include files in this mode.
 - This usually implies many false positives and false negatives.
- Upstream does not provide any compiler wrapper.
- We implemented our own **compiler wrapper** for Cppcheck:
 - Previously implemented as a shell script, now written in C.
 - Runs Cppcheck in parallel \implies significant **performance boost** when chained with other compiler wrappers.
 - It is going to be packaged for Fedora!

Clang

- A set of static analysis-based checkers running on top of **LLVM** (Low Level Virtual Machine).
- It is written in C++ and recent versions are difficult to compile with older tool chains, which we use for building RHEL packages.
- Upstream maintains a **compiler wrapper**:
 - The compiler wrapper is written in Perl.
 - Too slow to be chained with other compiler wrappers.
 - Currently bottleneck when running autoconf checks, etc.

Plugging Static Analyzers into Build Process

- We do not run static analyzers during production build.
- But we also use **mock** (chroot-based tool for building RPMs).
- The build environment is very close to what we use for building RPMs in production.
- On the other hand, we are able to make **destructive hacks** in the chroot in order to make the static analysis succeed (e.g. to simplify system header files for tools that would not parse them otherwise).

Overview of Helper Tools We Developed

- cswrap**
 - Generic **compiler wrapper**.
 - Makes it possible to add/remove compiler flags.
 - Translates relative to absolute paths in diagnostic messages.
 - <https://git.fedorahosted.org/cgit/cswrap.git>
- cscppc**
 - Compiler wrapper running **cppcheck** in background.
 - <https://git.fedorahosted.org/cgit/cscppc.git>
- csmock**
 - Mock-based tool for **automated static analysis of SRPMs**.
 - User only specifies a mock profile and list of analyzers to use.
 - Uses static analyzers and compiler wrappers mentioned above.
 - <https://git.fedorahosted.org/cgit/csmock.git>
- csdiff**
 - Set of command-line utilities for **comparing**, **filtering**, and **formatting** the list of defects.
 - <https://git.fedorahosted.org/cgit/codescan-diff.git>



Use of Compiler Wrappers – Process Tree (4/4)

```

*---- csmock -a cppcheck,clang -p fedora-20-x86_64 package.src.srpm
|
*---- mock -r fedora-20-x86_64 --chroot ...
|
*---- rpmbuild -bc ...
|
*---- make -j13
|
*---- ccc-analyzer ... unit0.c
|   |
|   *---- cscppc ... unit0.c
|   |
|   |   *---- cswrap ... unit0.c
|   |   |
|   |   |   *---- gcc ... unit0.c
|   |   |   |
|   |   |   *---- cswrap ... unit0.c
|   |   |   |
|   |   |   *---- cppcheck ... unit0.c
|   |   |
|   |   *---- ccc-analyzer ... unit1.c
|   |   |
|   |   *---- cswrap ... unit1.c
|   |   |
|   |   *---- clang-analyzer ... unit1.c
|   |
|   ...

```


Conclusion

- Using a single static analyzer is insufficient.
- We provide a user-friendly way to run **multiple static analyzers** (currently Coverity, Cppcheck, Clang, and GCC warnings).
- We maintain a set of easy to use command-line utilities for processing the results of static analyzers (**codescan-diff**).
- We are now getting all the helper tools into Fedora!