Analysis Tool Evaluation: Coverity Prevent

Final Report
May 1, 2006

Ali Almossawi
Kelvin Lim
Tanmay Sinha

Carnegie Mellon University

1. Introduction

As a software engineer, you are probably used to having “bug hunter” in your job description. Although computers are deterministic machines driven by precise logic, there is a fly in the ointment: they function only as well as their human creators design them to. And humans are somewhat prone to error, to put it mildly. Software systems, in particular, seem to be an attractive arena for spectacular displays of human error. Bugs get introduced all the time despite our best intentions. Should you resign yourself to a lifetime of bug hunting?

Not if Coverity Prevent does what it claims.

Coverity Prevent is one of the leading commercial static code analysis tools on the market today. Code analysis techniques apply the computer’s logical precision and computational power to automatically and quickly discover code defects resulting from human error. This saves developers time and reduces overall development costs. Although many of these analysis techniques are relatively new, recent years have seen a number of analysis tools mature sufficiently to start establishing a presence in the marketplace. Coverity Prevent is one such tool.

Derived from Stanford’s Metal/xgcc research project, Prevent’s developers claim that it is the “world’s most advanced static analysis tool” in existence. According to the company, major users of the tool include software industry giants such as IBM, Oracle, Veritas, and NASA. Prevent has also been applied to a few public open source projects, including Linux.

All this sounds very good. But how much value is Coverity Prevent likely to actually bring you as a software engineer in your everyday work? What will using it be like? And what drawbacks are there? To answer these questions, we explored the tool in detail and experimented with it on a number of real software projects. We hope that our findings will prove useful to you as you decide whether Coverity Prevent represents a viable means for you to find relief from your bug hunting chores.

2. How Coverity Prevent Works

2.1. Summary of Analysis Techniques

Coverity Prevent discovers code defects using a combination of inter-procedural data flow analysis and statistical analysis techniques.

- *Inter-procedural data flow analysis.* Prevent analyzes each function and generates a context-sensitive summary for it. Each summary describes all characteristics of the function that have important implications for externally-observable behavior. Coverity calls these summaries "function models". During data flow analysis, Prevent uses these summaries to determine the effects of function calls whenever possible. Prevent also performs false path pruning to eliminate infeasible paths from consideration. This reduces computation costs and lowers the chances of generating false positives.

- *Statistical analysis.* Prevent uses statistical inference techniques to detect important trends in the code. Using these conclusions, it then tries to discover statistically significant coding anomalies, which may represent deviations from the software developers' intentions.

The exact details of its algorithms are proprietary and therefore are generally not known outside the company. As Prevent remains under active development, its techniques will undoubtedly continue to be refined with each new release.

2.2. Applicable Development Environments

Is Coverity Prevent applicable to your development environment? First and foremost, it is designed to analyze only C and C++ source code. If your projects do not involve any substantial C/C++ components, then you will not gain any value from using Prevent.

However, this programming language requirement is the only major constraint. Prevent supports most popular development operating systems (Windows, Mac OS, Linux, FreeBSD, NetBSD, Solaris, HP-UX) and compilers for both traditional and embedded systems (GCC/G++, Microsoft Visual Studio, Intel Compiler, ARM CC, Sun CC, Green Hills Compiler, etc.). It fully supports parallel compilation systems.

Project scale is unlikely to pose any problems. Prevent has been shown to work well even with code bases containing millions of lines of production code. Prevent's user interface also contains flexible multi-user features, making it easy to use even if your organization uses large project teams.

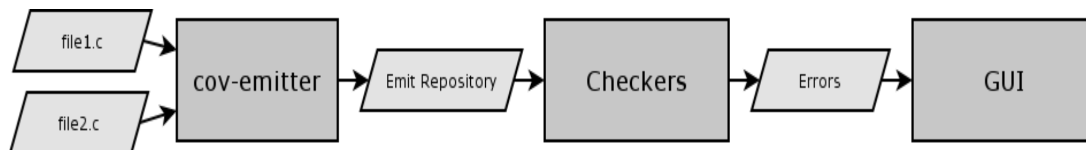
2.3. Process

When you use Coverity Prevent to analyze your code, you will generally go through this three-step process:

1. *Emitter.* After configuring Prevent for your compiler(s), it will integrate itself with your existing build process. When you build your project (e.g. using a "make" command), it will listen to all compiler calls and simultaneously process your code through its own built-in C/C++ parser. This allows it to generate internal binary representations of your project code, stored in what Coverity calls "emit repositories". Prevent handles subsequent build runs incrementally, processing only changed code whenever possible.

2. *Checkers.* After creating the emit repositories, Prevent's "inference engine" analyzes the repositories using the inter-procedural data flow and statistical analysis techniques described previously. It stores its generated function summaries and derived inferences in a database. After this, the "analysis engine" searches for defects by applying a number of checkers to the database generated by the inference engine. Each checker tries to match for a specific category of potential defects. We will describe these checkers in the next section.

3. *Output and Graphical User Interface.* Finally, Prevent aggregates the checkers' results and converts them to XML format. The resultant output can then either be used as inputs to other programs, or viewed through the included graphical web (HTTP) interface. Prevent allows you to set up user accounts so that multiple users can access the analysis results and have their changes and comments tracked.



Coverity Prevent process steps (source: Coverity Prevent manual)

2.4. Defect Checkers

The version of Prevent we tested (2.2.6) includes these nineteen defect checkers. Appendix A provides brief descriptions of the defects that each checker is designed to detect.

- NULL_RETURNS
- FORWARD_NULL
- UNUSED_VALUE
- REVERSE_NULL
- REVERSE_NEGATIVE
- RETURN_LOCAL
- SIZECHECK
- CHECKED_RETURN
- STACK_USE
- RESOURCE_LEAK
- USE_AFTER_FREE
- DEADCODE
- UNINIT
- DELETE_ARRAY
- INVALIDATE_ITERATOR
- PASS_BY_VALUE
- OVERRUN_STATIC
- OVERRUN_DYNAMIC
- NEGATIVE_RETURNS

As already mentioned, these checkers work using inter-procedural data flow and statistical analysis techniques. For instance, RESOURCE_LEAK checks for leaks of system resources such as allocated memory and file descriptors by tracking aliases to these resources and searching for control flow paths that result in a resource having no remaining in-scope aliases despite still being allocated. In contrast, CHECKED_RETURN uses statistical analysis to infer when a given function's return value is usually explicitly checked, and then flags anomalous cases where it is not checked.

Why might finding these defects be useful to you? In our opinion, there are at least three distinct reasons that you may find relevant:

- *Correctness.* Obviously, many of these defects can produce incorrect behaviors, such as wrong outputs or system crashes. Some of these faults may be hard to detect because they appear only under certain inputs or operating conditions. CHECKED_RETURN and NULL_RETURNS are some examples of defects that may cause the program behave incorrectly.

- *Security.* Some of these defects may not necessarily violate behavioral specifications, but may nevertheless open the system up to security vulnerabilities. For instance, OVERRUN_STATIC may lead to buffer overrun exploits, while DEADCODE may highlight normally inactive (but linked) code that a malicious party planted to gain backdoor access.

- *Performance.* Other defects may indicate inefficiencies or issues that can result in degraded system performance over time. Examples of these include RESOURCE_LEAK and SIZECHECK.

3. Experiment Setup

3.1. Purpose

To help you gain insight into how well Coverity Prevent actually performs in the field (as opposed to artificial test cases for each of its defect checkers), we decided to obtain detailed quantitative and qualitative assessments of its performance for real software projects.

We conducted our experiments on a variety of project types in order to make the results as broadly representative as possible. Our intention was to provide you with a basis for deciding how much value Prevent will probably give you for each of your projects.

3.2. Setup

We conducted our experiments using the Linux version of Coverity Prevent version 2.2.6. This was a fully functional version that Coverity kindly provided to Carnegie Mellon for evaluation.

We used the following existing software projects as our test cases. We chose these to represent as broad a range of development contexts, application domains, and code base size as we could feasibly accommodate for a small-scale study like this.

- *OS-P3* – An academic Operating Systems project that implements basic Unix-like kernel system call and thread handling functionality.
- *Networks-P2* – An academic computer networks project that implements a basic client for a BitTorrent-like peer-to-peer file sharing protocol.
- *DirList* – A single-developer open-source Web CGI providing access to a user directory database.
- *Apache HTTP* – The Apache project's HTTP server, one of the most popular and well-supported open source web servers currently in use.
- *GAIM* – An open-source instant messaging client.
- *GLib* – The GNOME C function library, a large repository of algorithm and data structure implementations that developers can use.
- *Vim* – A very popular interactive text editor for UNIX systems with a broad feature set.

- *Thunderbird* – The Mozilla Foundation's open source email client, which was recently distributed as a stable release for the first time.

- *MySQL* – A mature and full-featured database system supporting SQL queries, developed by a large open-source developer network.

We used the latest stable code versions of each project at the time we conducted the experiments.

3.3. Method

Each experiment trial involved running the entire Coverity Prevent analysis process of one of the test projects. For each test case, we noted the following quantitative information reported by Prevent:

- Number of lines of code (excluding comments and blank lines)
- Total number of detected defects
- Defect density (average number of defects per thousand lines of code)
- Breakdown of defects by defect category
- Time taken to complete the analysis

In addition, because Prevent is an unsound analysis tool, not all the defects it reports are guaranteed to be genuine. We therefore studied each individual reported defect and assigned it a rating indicating how confident we were that it was a genuine code defect. A high confidence rating meant that we were very sure that it was a real defect. Conversely, a low confidence rating meant that we were very sure that it was a false positive. We assigned medium confidence to all defects that we could not come to a strong conclusion about after expending a reasonable amount of effort on manual analysis. Although this is clearly a subjective measure, we tried to make our evaluations as consistent as possible.

We also qualitatively assessed our experience with each project. We tried to do so from the perspective of a typical developer in the midst of a real project development process, so this should reflect what it would be like for you when you use the tool.

4. Empirical Findings

4.1. Result Data

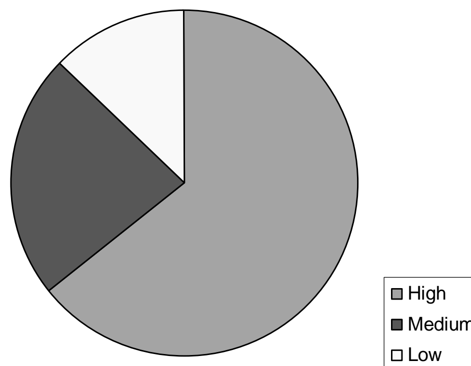
This table summarizes the key results we obtained from our experiments. Appendix B provides more detailed breakdowns for each test case.

Project	LOC	Defects	Defects/ KLOC	Our Confidence Rating		
				High	Med	Low
OS-P3	12,636	14	1.108	9	-	5
Networks-P2	14,716	6	0.408	6	-	-
DirList	17,459	17	0.974	14	-	3
Apache HTTP	66,382	16	0.241	8	6	2
GAIM	93,733	3	0.032	3	-	-
Glib	121,691	34	0.279	26	7	1
Vim	171,459	80	0.467	49	29	2
Thunderbird	246,810	75	0.304	65	7	3
MySQL	607,457	233	0.384	127	61	45

Summary of experiment results

In total, Coverity Prevent reported 478 defects over 1,352,343 lines of code, yielding an average defect density of 0.353 defects per thousand lines of code. Of the 478 defects, we rated 307 (64.2%) as high confidence, 110 (23.0%) as medium confidence, and 61 (12.7%) as low confidence (as shown in the chart below).

Depending on whether we consider medium confidence defects to be false positives, we therefore estimate the overall false positive rate to be somewhere between 12.7% and 35.7%, with a mean of 24.2%.



Breakdown of reported defects by confidence rating

4.2. Implications of Results

Coverity claims that Prevent's average false positive rate is around 20% ("one false positive to every four genuine errors"). Our results appear to be consistent with this claim.

Compared to many other static analysis tools, we consider this to be a very low false positive rate, especially considering the fact that these results were obtained without providing any additional annotations to aid Prevent's analysis. The key implication of this statistic for you as a developer is that you will spend most of your time fixing real bugs in your program, instead of wasting time attempting to trace spurious error messages.

The overall average defect density of 0.353 defects (or 0.227 high confidence defects) per thousand lines of code suggests that Coverity Prevent does a decent job of surfacing defects. Industry studies suggest that developers typically introduce somewhere on the order of one to three defects (of all possible categories) per thousand lines of code during initial development. Considering that we ran these analyses on fairly mature code bases, this suggests that Prevent provides a means to automatically catch a very significant proportion of all defects in your code.

One reason why we indicated nearly a quarter of all defects as medium confidence was the fact that some of the defects were very hard to trace by hand. This was because these defects often occurred across complex interactions between multiple functions, and only on some data flow paths. If these defects were genuine, this suggests that Prevent will help you greatly in surfacing non-trivial bugs that are very difficult to detect through code inspection. Although testing may eventually find these defects, the great advantage of static analysis tools like Prevent is that you can apply it throughout the entire development lifecycle instead of only when you have testable deliverables. This can save you substantial cost by finding quality problems earlier on, when they are usually cheaper to fix.

We should also note that even the false negative reports were often surprisingly insightful. For instance, one reason for the high number of false positives with the MySQL code was that it used its own custom debugging macros, which would terminate the program when an assertion failed. Prevent did not recognize these termination mechanisms and therefore continued to trace the data flow beyond them. However, the false negative reports essentially provided a list of all the instances where these debugging macros actually serve useful functions (i.e. they were not redundant checks). Prevent's analysis results and function models can therefore help you in ways beyond their primary defect detection role. (Note also that Prevent's models can in fact be manually configured to recognize MySQL's debugging macros if so desired.)

In particular, we found Prevent's analysis results very useful in helping us better understand the logic and structure of the large public open source code bases we tested. As none of us had any experience working on these projects, we were essentially thrust in the role of developers coming into a previously existing project that we did not originally work on—a realistic scenario that you may well face in the future. Many of the defect reports helped highlight hidden assumptions in the code that were not explicitly reflected in the accompanying comments and documentation. We therefore suggest that Prevent can be a very valuable aid in architecture discovery and code refactoring efforts.

Overall, we were very pleased with Prevent's performance on our experiment trials. Considering the minimal setup work we had to do to perform these analyses, we consider the returns we received to be excellent, and we believe that most developers will derive similar value from the tool.

5. Qualitative Evaluation

5.1. Usability

Coverity Prevent's user interface clearly reflects its original Linux/Unix heritage. As a user, you will initiate most of the analysis steps through the shell command line. Indeed, everything that can be done with it is done by running command-line executables, except viewing the final analysis output.

Surprisingly, we found that this does not really hinder the usability of the tool. The user gets a satisfying sense of progress as each analysis module is run in turn, readying the environment for the next one. During the different stages of the analysis, a progress bar shows exactly how much more time is left before completion (and a twirling animation even helps keep the user entertained during the wait!). If the user prefers not to run each executable in sequence for each code base, Coverity also provides an executable that runs the entire process with a single command.

The graphical user interface works through a standard web browser interface, with pages served up by a built-in version of Apache HTTP Server. Using the GUI is one of the more enjoyable aspects of the tool. After logging in, the screen cleanly shows all the different analysis runs that were committed, together with the line count and defect statistics for each run. Clicking on a particular run displays a table with all the defects that were detected. Clicking on a run brings up a very cleanly organized window (pictured below) with details of the defect and the exact source lines involved in the defect trace, annotated with Prevent's analysis inferences.

Sample trace view for one defect.

As a user, you can easily jump between different parts of the code base by clicking on function or variable names. We found this highly interactive feature extremely useful while trying to understand the reason behind why a line has been flagged as a defect. The ease with which one can move between different parts of the code using hypertext links makes the manual analysis process much more enjoyable than it would have been if we had been expected to take on the burden of browsing through files outside the fold of the tool.

Prevent also gives us the ability to tag defects as being actual defects, false positives, pending, and so on. This is much like one can do with other popular bug-tracking systems such as Bugzilla. This, coupled with the color coding used throughout the GUI, is especially useful for the novice.

5.2. Performance

Prevent performed its analysis runs remarkably quickly even for large code bases. From our experiments, we found that it took as little as 15 seconds to process a 12,000 line code base, and even MySQL (over 600,000 lines of code) took just under an hour. This was significantly faster than most other static analysis tools we used in the past.

Coverity Prevent's excellent performance will therefore probably serve most organizations' efficiency needs. It minimizes the amount of time you and your team have to wait before you can engage in productive work based on its analysis results.

5.3. Scalability

As previously mentioned, we ran Prevent on projects ranging from small academic ones with about 12,000 lines of code to large projects with 600,000 lines of code. In all cases, it did not crash or report any kinds of unexpected errors. To accommodate the needs of very large projects, Coverity's documentation also indicates that Prevent can perform parallel builds and analyses, although we did not test this functionality in our experiments. In general, we would probably not hesitate to apply this tool to code bases of any size.

5.4. Limitations

Perhaps the biggest limitation of the tool is its high monetary cost. Coverity charges a yearly fee for Prevent according to the number of lines of code in the code base you use it with. Although Coverity does not publish standard rates on its publicity materials, one online source indicated that a project with 500,000 lines of code will cost approximately $50,000. Had we been

charged for the privilege of using the program, even our relatively small-scale experiments could well have cost us over $100,000!

You must therefore carefully consider whether Prevent provides enough benefits to justify its cost in your development environment. Fortunately, Coverity apparently provides a free trial program that involves them demonstrating the tool on your existing code bases. This can provide a valuable source of input in helping you decide whether Prevent makes sense to your organization.

Another significant limitation, as we mentioned at the outset, is that Prevent works only with C and C++ code. The tool's brochure, available on its website, mentions the defects that it can detect include some that may lead to denial-of-service-type attacks and SQL injections, so it would be extremely helpful if this tool can be extended to Web-based projects written in languages such as Java and PHP. Until this happens, however, you are essentially out of luck if your project involves sizeable non-C/C++ modules.

5.5. Documentation

Coverity Prevent's included documentation is extremely well-written. It covers everything from setting up the tool on different operating systems to how we can interpret the results of analyses. For each of the defects listed in section 2.4, the manual explains the defect in detail and also discusses common cases where the tool might produce a false positive. It also suggests ways of suppressing each of these false positive cases with annotations and other built-in mechanisms. Other sections include detailed overviews of how the tool works, as well as troubleshooting tips that describe possible pitfalls that a user may encounter and the likely causes.

Overall, the well thought-out documentation reflects how well thought-out the overall product is. Prevent clearly gives the impression of being a very mature and professional tool.

6. Conclusions and Recommendations

So what's the bottom line? Does Coverity Prevent live up to its claim of being "the world's most advanced static analysis tool"? It would be unfair to answer that question based only on our small-scale study. Nevertheless, we are confident that our conclusions generally point us in the right direction. What is certain is that Prevent is an excellent static analysis tool around for C and C++ code.

In terms of its defect detection efficacy, Prevent does an excellent job of surfacing important defects while keeping the false positive rate low. This is surely good news for software engineers everywhere. As an incomplete tool, it does not do everything, of course. So Prevent will not remove bug hunting from your job description, at least not in its current iteration. But it will probably make you a better and more productive bug hunter, and will free up more time for you to attend to other aspects of your job.

Prevent is also a very practical tool. It has a lot of features that were clearly designed with end users in mind. Its web-based user interface is very intuitive, and it provides an excellent way for teams to keep track of bugs found. Its multi-user features help managers track bug fixes to individual developers and avoid redundancies and unnecessary overlap. By not requiring code annotations or changes to the build process, it also caters to most developers' preferences by leaving as small a footprint on their code as possible—while still providing very useful results in spite of the minimal input it requires.

Indeed, the only factor deterring us from wholeheartedly recommending Coverity Prevent to everyone is the high monetary cost of using the tool. If your organization can justify the cost of purchasing this product, we are confident that you will not just find more bugs more easily, but—believe it or not—actually enjoy the bug-hunting process a lot more.

Appendix A: Defect Checker Descriptions

- NULL_RETURNS: A function that can return NULL must be checked before it is used. This checker identifies for such dereferences of NULL return values.

- FORWARD_NULL: A program will normally crash when a NULL pointer is dereferenced. One situation this can happen is when the pointer has been checked against NULL and is dereferenced later. This check identifies such situation by checking all possible paths where such NULL dereferences can occur.

- REVERSE_NULL: A program will normally crash when a NULL pointer is dereferenced. Another situation this can happen is when the pointer is dereferenced before it has been checked against NULL. If the dereference is NULL, the check programmer should be warned to place the check against NULL before dereference. This check identifies such situation by checking all possible paths where such NULL dereferences can occur.

- UNUSED_VALUE: When a variable is assigned a pointer value returned from a function call and is never used anywhere else in the source code, it can not only cause inefficient use of resources but can also result in undetermined behavior. This checker identifies all variables that are never used anywhere else in the program after a value is assigned to them.

- REVERSE_NEGATIVE: Sometimes a negative value is not advisable to use. One way to avoid such use is to check for negative value after a possible dangerous use. In this situation there could be a problem when the value should not be negative before use. This checker identifies such conditions by checking all possible paths where such usage of negative value can occur.

- RETURN_LOCAL: If a function returns a pointer to a local stack variable, there could is a possibility of memory corruption and un-deterministic behavior. This checker identifies such returns and marks them as defects.

- SIZECHECK: Incorrect amount of memory allocation can lead to undetermined behavior and program crashes. This checker identifies such memory allocations that are assigned to a pointer to a type that are bigger than amount of memory allocated to them.

- CHECKED_RETURN: If might be necessary for the developer in many cases to check for the value returned by a function call. This checker uses static analysis to determine whether all the error conditions are handled when the function call is made.

- STACK_USE: Some applications have limitations on the overall stack allocation, e.g. device drivers. This checker can be used to determine whether there is a violation of overall stack use. This is a special checker that is required to be enabled. The analysis does not use this checker by default.

- RESOURCE_LEAK: Resource leaks can have all kinds of bad effects on the program. A memory leak can lead to program crashes. A leak of file descriptors, and socket can cause crashes and also have other harmful effects on the program. This checker identifies all these kinds of leaks in the source code and marks them as errors. Apart from usual memory leak checks, it checks for interesting situations like aliasing as well.

- USE_AFTER_FREE: It is not advisable to use a memory after it has been freed, as it might lead to non-deterministic results when it is used. This checker identifies such conditions where a memory location is dereferenced after it has been freed. This also includes checks for double freeing of a pointer.

- DEADCODE: If a certain part of source code can never be reached during the execution of program, it’s called dead code. Usually harmless, it can cause problems when the dead code contains some code that is important for the proper functioning of the program. This checker identifies such pieces of code in the program using static analysis.

- UNINIT: The use of un-initialized variables can often result in non-deterministic behavior. Under some situations, it can also cause security vulnerabilities. This checker identifies such variables and points their usage.

- DELETE_ARRAY: This checker simply checks if there is a use of “delete” instead of “delete [ ]” to free an array.

- INVALIDATE_ITERATOR: A C++ specific checker that identifies a wrong usage of iterators in Standard Template Libraries (STL), as it might not work across operating environments and can cause crashes.

- PASS_BY_VALUE: This checker warns if the value being passed in a function is larger than 64 bytes, as it might result in poor performance under certain situations.

- OVERRUN_STATIC: This checker identifies invalid accesses to a static array, as it can cause buffer overruns that can ultimately lead to security vulnerabilities and program crashes.

- OVERRUN_DYNAMIC: This checker, unlike the static overrun, identifies invalid accesses to a dynamic array, as it can cause buffer overruns that can ultimately lead to major security vulnerabilities and program crashes.

- NEGATIVE_RETURNS: If a value that is returned from a function can be negative and is used inappropriately, it can cause multiple errors such as memory corruption, crashes, infinite loops and so on. This checker identifies such situations and marks such usage of a negative value as an error.

Appendix B: Experiment Results by Project

- **OS-P3**
  Analysis time: 00:00:57

	Total	Hi	Med	Low
UNINIT	3	2	-	1
RESOURCE_LEAK	6	2	-	4
NULL_RETURNS	1	1	-	-
FORWARD_NULL	3	3	-	-
CHECKED_RETURN	1	1	-	-

- **Networks-P2**
  Analysis time: 00:00:15

	Total	Hi	Med	Low
RESOURCE_LEAK	5	5	-	-
OVERRUN_STATIC	1	1	-	-

- **DirList**
  Analysis time: 00:02:00

	Total	Hi	Med	Low
DEADCODE	2	2	-	-
FORWARD_NULL	3	3	-	-
RESOURCE_LEAK	5	5	-	-
REVERSE_INULL	3	3	-	-
SIZECHECK	3	-	-	3
USE_AFTER_FREE	1	1	-	-

- **Apache HTTP**
  Analysis time: 00:05:58

	Total	Hi	Med	Low
REVERSE_INULL	1	1	-	-
NULL_RETURNS	2	-	-	2
FORWARD_NULLS	7	2	5	-

DEADCODE	4	2	1	1
CHECKED_RETURN	1	1	-	1

- **GAIM**
  Analysis time: 00:15:00

	Total	**Hi**	**Med**	**Low**
UNUSED_VALUE	3	3	-	-

- **gLib**
  Analysis time: 00:29:34

	Total	**Hi**	**Med**	**Low**
USE_AFTER_FREE	1	1	-	-
UNUSED_VALUE	5	5	-	-
UNINIT	1	1	-	-
RESOURCE_LEAK	6	2	4	-
OVERRUN_STATIC	4	3	1	-
FORWARD_NULL	13	11	2	-
DEADCODE	1	-	-	1
CHECKED_RETURN	3	3	-	-

- **Vim**
  Analysis time: 00:43:45

	Total	**Hi**	**Med**	**Low**
CHECKED_RETURN	22	-	20	2
DEADCODE	5	5	-	-
FORWARD_NULL	13	13	-	-
NEGATIVE_RETURNS	21	21	-	-
NULL_RETURNS	2	2	-	-
OVERRUN_STATIC	4	-	4	-
RESOURCE_LEAK	6	-	6	-
REVERSE_INULL	2	2	-	-
UNINIT	3	-	3	-
USE_AFTER_FREE	2	2	-	-

- **Thunderbird**
  Analysis time: 00:45:22

	Total	Hi	Med	Low
USE_AFTER_FREE	1	1	-	-
UNUSED_VALUE	4	4	-	-
UNINIT	9	9	-	-
REVERSE_INULL	3	3	-	-
RESOURCE_LEAK	6	2	4	-
OVERRUN_STATIC	7	7	-	-
NULL_RETURNS	3	3	-	-
NEGATIVE_RETURNS	1	1	-	-
FORWARD_NULL	34	31	3	-
DEADCODE	3	2	-	1
CHECKED_RETURN	4	2	-	2

- **MySQL**
  Analysis time: 00:59:29

	Total	Hi	Med	Low
USE_AFTER_FREE	2	2	-	-
UNUSED_VALUE	3	3	-	-
UNINIT	38	3	15	20
REVERSE_INULL	3	-	-	3
RESOURCE_LEAK	25	19	3	3
OVERRUN_STATIC	3	1	-	2
NULL_RETURNS	11	8	3	-
NEGATIVE_RETURNS	5	-	1	4
FORWARD_NULL	34	10	16	8
DEADCODE	23	14	8	1
CHECKED_RETURN	86	67	15	4