
Effective Management of Static Analysis Vulnerabilities and Defects

Introduction

According to a recent industry study, companies are increasingly expanding their development testing efforts to lower their costs and deliver products faster to the market. One of the primary methods used for development testing is static code analysis which helps organizations find critical defects in their code earlier in the lifecycle. To be most successful, organizations must carefully consider the process changes that will be required to manage an array of newly identified defects in their code and how development testing will fit into their overall Software Development Lifecycle.

With over 1,100 customers and over 5 billion lines of source code under management, Coverity has helped our customers implement highly successful development testing adoption plans. A key component of the plans is a well-considered defect management and integration strategy. This paper reviews what we have seen to be the most consistently successful static analysis defect management and integration strategies across development teams of varying size and sophistication.

Challenges of Successful Static Analysis Defect Management

Regardless of the industry the organization is building products and services for, or why an organization acquires a static analysis product, every development team using static analysis for the first time will be faced with similar challenges. This is because an accurate static tool represents a new source of information that developers, QA and managers alike will become responsible for processing prior to the next release of any application. Depending on the size of a given code base, development team, and the average experience of developers and application engineers – the volume and impact that the results of a static analysis tool will have on an organization varies significantly. First time users of static analysis will be faced with two new challenges immediately following deployment:

1. How to inspect, prioritize, and resolve the large number of legacy defects detected within their code base
2. How to inspect, prioritize, and resolve a constant stream of newly introduced defects as development progresses

Effective defect management is not a challenge faced solely by new users of static analysis. Even experienced developers who are part of seasoned teams that have successfully installed and configured a static analysis solution can struggle to create an effective and efficient defect resolution process. A common pitfall is to underestimate the importance of the process surrounding the management of static analysis defects. To do so may create inefficiencies within development teams that ultimately limits static analysis adoption and compromises the return on investment the team has anticipated. Outlining and implementing a defect resolution process early in the deployment of a static analysis tool can help developers significantly increase the probability of successful developer adoption. Based on Coverity's experience, there are five steps required to create a successful defect resolution process:

1. Determine goals and metrics
2. Develop a project plan
3. Assign ownership of defects
4. Notify owners of defects
5. Integrate defect resolution workflow with SDLC

With these five steps, development teams can accelerate their ability to achieve their quality and security goals, often within months or even weeks. By quantifying and tracking these variables, development teams gain the ability to present objective data regarding the integrity of their code to both internal and external audiences. Particularly for organizations using static analysis for the first time, the lack of this data can impair the ability to report on their progress, based on objective criteria, to management and other audiences.

Step 1: Determine Goals and Metrics

The introduction of static analysis will provide developers with hundreds, or possibly thousands, of newly detected defects to inspect and resolve. Unlike customer reported defects, statically detected defects have no external pressure demanding they be repaired. Because of this, it is critical that the internal goals and policies of a development organization are related to static analysis as early as possible in the deployment phase.

Agreeing upon early goals for static analysis provides development teams with a common objective for their newly acquired technology and provides an objective means to measure the both the improvement of your code, and the value derived from your investment in static analysis technology.

When considering software quality and security, there are many reasonable goals that can be pursued with static analysis. Before agreeing upon any proposed goal, it is important to evaluate:

- Is this goal aligned with other preexisting software quality goals?
- Can progress towards this goal be easily measured?

Selecting the appropriate goals for static analysis requires an understanding of the overall goals and challenges for a given development team. For example, if a company is heavily focused on code security, the development team may choose extremely stringent goals regarding the elimination of potential vulnerability types. Conversely, if an organization is focused on developer productivity and the release of new features, developers may be asked to focus on defects that cause protracted debugging efforts, such as memory corruption or the use of uninitialized data.

Coverity® Integrity Control, which is the latest addition to Coverity's development testing portfolio, allows teams to set policy thresholds for defect density, total outstanding defects, outstanding and uninspected high impact defects, and various other quality and security thresholds, and monitor violations within every code component and raise a warning or an alert as appropriate. Coverity Integrity Control aligns various defect metrics with predefined software quality goals and provides an out-of-the-box solution to measure progress towards defined goals. Next we will discuss some of the most common frameworks that merit consideration along with a sample of potential goals:

Sample Goal 1: Maintain Quality and Security Thresholds

One common goal setting approach with static analysis is to track code quality and security based on defect density. Defect density is the number of static analysis defects identified per 1,000 lines of code. Maintaining quality and security thresholds via defect density means that a code base will not be allowed to reach a set number of defects based on its size. Common threshold metrics for this type of goal framework include:

- Outstanding defects per line of code
- Total outstanding

These types of metrics have the advantage of being easily measured; in fact both of these metrics are directly reported within Coverity® Static Analysis's user interface as well as included in the Software Integrity Report that presents a snapshot of the overall quality and security vulnerability assessment of your entire codebase. For example, if a code base has 1 million lines of code and a defect density threshold of 0.1, then at any given time it should always have fewer than 1,000 static analysis identified defects. If the defect density ever exceeds 1,000, developers would not be allowed to submit their code to the central build system until there were less than 1,000 defects in that code base.

Several refinements on this goal framework are possible. For example, a development team may set goals based on the defect severity, probability of runtime impact or density within a specific code component. By this token, a possible set of defect thresholds could be:

- Less than 5 total outstanding memory corrupting defects
- Less than 1 outstanding major severity defects per 10,000 lines of code
- Less than 1 outstanding moderate severity defects per 2000 lines of code
- Less than 1 outstanding minor severity defects per 1000 lines of code

While another team's threshold could be as simple as:

- Less than 1 outstanding quality defect per 2,500 lines of code, and less than 1 outstanding security defect per 10,000 lines of code

Sample Goal 2: Reach Zero Uninspected Defects

Another goal which many organizations find useful is that all statically detected defects should be inspected by a developer prior to release. A benefit of this absolute goal is that it ensures each defect will be reviewed and prioritized by a developer. As a result, development teams can prioritize their efforts to ensure the most crucial potential quality defects and security vulnerabilities are identified in their code.

As stated above, this goal relates only to the inspection of defects, and not necessarily their resolution. It presumes that developers will correctly prioritize potential issues prior to release, and that they will be given time to resolve them. Teams that select this as a goal should also select a secondary metric surrounding the number or type of defects that should be repaired.

For users of Coverity Static Analysis, it is simple to measure progress towards this goal, because Coverity Integrity Manager, the central defect database and management interface, maintains the status of each individual defect or vulnerability, including its classification, severity, and owner. These capabilities are critical for developers, because it allows you to quickly classify defects and search for issues which are not yet uninspected.

Coverity Integrity Manager also tracks the status of identified issues as a given code base evolves. This persistent tracking of analysis results helps you avoid the redundant inspection of issues that have not changed between analysis runs. Because of this, even as your code base matures or changes, the insights your developers have stored within the defect database remains associated with each individual defect – so issues are only inspected once.

Managers and Executives in charge of multiple teams can utilize Coverity Integrity Control to establish and enforce the appropriate policies regarding uninspected defects. Managers can quickly see areas of risk in their projects or teams based on the violation of the established policy.

Sample Goal 3: Reach Zero Outstanding Defects

The ultimate goal, and in some ways simplest goal to measure, is to achieve zero outstanding static analysis defects at release time. At first, this goal may seem a difficult challenge, but for some code bases, the time required to simply fix all outstanding defects can be less than the time required for multiple cycles that entail defect review, prioritization and repair.

For many companies, the most challenging part of this goal is not reaching zero defects at a given point in the Software Development Life Cycle (SDLC), but in retaining the zero defect status as a release date approaches. Before selecting this goal for your organization, you must first consider the quality and security assurance procedures that already exist in your release process and explore whether they can be revised to support this goal.

Recommendations for Software Quality Goals

Regardless of the software quality goal(s) you select, all defects should be inspected prior to release. A static analysis tool will provide developers with the root causes of a runtime defect, rather than the symptoms of that defect when it occurs at runtime. Because of this, statically detected defects can be resolved with minimal diagnostic effort. However, this also means that statically detected defects require input from a developer to accurately assess the severity of a statically detected defect at runtime.

As we have observed at numerous customer sites, once trained, it takes a developer less than five minutes on average to inspect, classify, and assign severity to a potential vulnerability or defect. By requiring that all defects are inspected prior to release, an objective assessment of release integrity can be made based on the classification and severity of outstanding defects.

If the release process can be made to accommodate this goal, Coverity recommends selecting the goal of zero outstanding defects. If time pressures around the release date may result in a given release shipping with statically detected defects present, it is better to create threshold goals that can be more easily met. This allows developers to benefit from static analysis by improving overall code integrity while still meeting release commitments to management and line of business leaders.

Step 2: Develop a Project Plan

Once your development team agrees upon its static analysis goals, it should be simple to estimate the effort required to achieve them. As mentioned previously, on average it takes less than five minutes for a trained developer to inspect and classify a defect. The time to repair a specific defect depends on a variety of factors. However, because static analysis identifies the root cause of a defect, the time required to repair a statically identified defect will be dramatically less than the time required to repair a customer reported issue.

With this in mind, as development organizations create their project plan, you should be confident your efforts will generate a significant benefit to both your company and your customers. A defect management project plan will require two components:

- How to address existing legacy defects
- How to address newly introduced defects

Addressing Legacy Defects

When Coverity Static Analysis is run for the first time, new users may feel that overwhelmed by the initial number of defects that may be reported. Coverity Static Analysis typically finds between .5 and 2 defects per thousand lines of code in large, mature code bases. By this token, you should expect between 500 and 2,000 initial defects for a legacy code base of approximately one million lines of code.

The most successful strategy for effectively managing the initial backlog of legacy defects tends to be a divide and conquer approach that contains two distinct phases. In the first phase, responsibility for legacy defects is distributed between small teams of developers who inspect defects, assign defect classification and severity. They can then document any reasoning that accompanies those settings with text comments.

After defects are initially triaged, they should be assigned to developers familiar with the code that generated the defect. Randomly assigning defects within a development team is inefficient because developers asked to evaluate a code component they are unfamiliar with will require extra time to accurately evaluate the context and severity of defects. A sample defect resolution plan for the first phase of static analysis is presented in Table 1.

Table 1: Example project plan for first phase of legacy defect removal

Size of the code base	1,342,000 lines of code
Outstanding Defects	1,200
Number of Developers	20
Defects per developer after distribution (avg., min., max.)	60, 6, 120
Time to inspect defect	5 minutes
Time investment required per developer to inspect all defects (avg., min., max.)	5 hours, 30 minutes, 10 hours
Time scheduled per week for triage per developer	5 hours
Weeks until project completion	2*

*Note: most results will be inspected at the end of week 1

After processing the legacy static analysis defects in the first phase, a development team will need to decide how to resolve and remove these legacy static analysis defects on an ongoing basis. To do so will require a project plan that outlines how these legacy defects should be treated. A sample project plan for the second phase of eliminating legacy defects is in Table 2 below.

Table 2: Example project plan for removal of legacy defects

Size of the code base	1,342,000 lines of code
Outstanding Defects	1,200
Number of Developers	20
Defects per developer after distribution (avg., min., max.)	60, 6, 120
Time to inspect defect	30 minutes
Time investment required per developer to inspect all defects (avg., min., max.)	30 hours, 3 hours, 60 hours
Time scheduled per week for triage per developer	5 hours
Weeks until project completion	10*

*Note: most results will be inspected at the end of week 6

This plan should leverage the quality and security goals discussed earlier in this paper, as well as the severity information from the initial inspection of defects. By utilizing this data, you can formulate a plan that will direct your team to address the most severe defects and vulnerabilities first.

Addressing Newly Introduced Defects

As a developer, you know the cost to repair defects increases the further in the software development lifecycle a defect is allowed to persist. For this reason, there is value in addressing newly detected defects as they are identified because they cost less time and effort to repair. In order for developers to prioritize eliminating new defects, you should budget a specific amount of time during development for them to address newly identified quality and security issues.

By lowering defect density during development, organizations can lower the cost of QA by delivering higher integrity code to testers that require less test case generation for effective coverage. As proof of this, Coverity customers have documented individual developer productivity improvements ranging from 12.5 to 30 percent.

Step 3: Assign Ownership of Defects

Static analysis defects that are not assigned to a specific individual are unlikely to be resolved. For this reason, the ability to assign ownership is a critical feature to require of your static analysis tool. Moreover, configuring your static analysis product to automatically assign defects is an important part of this process to streamline workflow and drive developer adoption. Done properly, defects can be automatically associated with the developer responsible for introducing them. This prevents defects from escaping from one development phase to the next before an owner can be identified. It can also prevent additional code from being built around defective code, which ultimately makes a repair more difficult.

The challenge with manually assigning defects, beyond simple time consumption, is that this function can be quickly abandoned during crunch time prior to a product release – precisely when it is most critical to capture and correct new defects to avoid a dangerous field issue. For these reasons, the assignment of initial ownership for statically detected defects must be automated. There are two recommended approaches for the automated assignment of defects:

- Component based ownership
- SCM based ownership

Component Based Ownership

Every code base can be divided into components using Coverity Static Analysis's web based user interface. For every defect, Coverity Integrity Manager displays comprehensive detail surrounding any given defect including file, function, and line numbers among other elements. A component within the Integrity Manager is defined by a set of regular expressions which are applied against the full path name of a given source file. This component mechanism is both powerful and flexible, because it allows the use of regular expressions against absolute path names to deliver control at the file level. It also provides the ability to match the existing organization of a given code base - no matter how complex. Once components are defined, the task of assigning every new defect within a given component to a specific developer or development team can be automated.

Many development organizations already consider the structure of directories when assigning ownership of select development tasks. For those organizations, establishing a component based ownership model will be a natural approach. A component based ownership model is the simplest, and often the best way of ensuring each detected defects is assigned to the responsible party. A static analysis tool administrator should be able to establish this type of ownership model in a few minutes.

SCM Based Ownership

Instead of automatically assigning ownership, many development organizations elect to integrate their static analysis tool with their existing Source Code Management (SCM) systems. Most SCM systems can be queried for metadata such as file ownership, or the identity of the last individual to modify a particular line of code.

By creating a custom script to interface between the SCM system's file ownership and modification data, and the defect information in Coverity Integrity Manager, many sensible ownership policies may be applied. Typically, a straightforward policy can be applied by which the last developer to modify the line of code that generated a defect becomes the assigned owner. For example, whoever edits a file or line of code last is responsible for the initial inspection of the defect it. In such a situation, if after evaluating the defect the first developer to triage it determines it is not their problem, they would reassign the defect to the developer responsible for the root cause of the error.

For example, developer A could add a line of code that calls a function developer B wrote. Perhaps developer B's function has a bug in it, which is exposed by developer A's call. It's possible that this bug will get assigned to developer A, even though it will ultimately need to be repaired by developer B. With this common understanding, a developer that changes code in such a way that causes the introduction of a new static defect is the developer initially assigned to inspect it - even if the developer who is actually responsible for the root cause and the eventual fix of the problem is someone else on their team.

The SCM based ownership approach works well in development organizations which have no notion of file ownership based on a hierarchical directory structure. This approach has some challenges with regard to incorrect assignments; occasionally the individual who edited a given file or line most recently will not be responsible for the introduction of the defect in question, or may not even have access to the static analysis results (in the case of a contractor or outsourced development). A sample workflow for SCM system integration is presented in Figure 1 below.

Figure 1: Common SCM and Static Analysis Integration



The SCM based ownership approach requires development of custom ownership policy logic, and scripted connectors to both the SCM system and Coverity Integrity Manager using a well-defined Web Services API. Coverity's professional services team has substantial experience in quickly creating such integrations.

Measuring the Progress of Your Defect Resolution Effort

For senior developers and team managers to prove the value of the investment in static analysis, you need to track more than the overall progress of defect resolution efforts. You also need to ensure your team receives the credit they deserve for proactively improving the quality and security of your code. Often, developers use defect density statistics regarding a given piece of software to prove the integrity of their code to technology partners, to support individual performance reviews, or to justify additional resources and headcount for their teams (because the most efficient teams should be the best funded teams). As you measure your progress towards your quality goals, there will generally be two possible metrics:

- The number of defects fixed to date
- The number of defects still outstanding

It is natural for many new users of static analysis to use the second metric to measure their progress; this is because many of the final goals will depend on the number of defects outstanding at the time of a release. Instead of selecting this goal out of hand, Coverity recommends monitoring and reporting the number of defects fixed to date to measure progress. There are two distinct benefits of this approach:

First, simply tracking the defects repaired is an inaccurate assessment of the team effort in the context of static analysis. Defects found by static analysis existed before your purchase, so to prove the value of static analysis technology, it's best to track progress forward from your implementation date.

Second, whenever the new checkers and testing depth are enabled, or your deployment is upgraded to include new defect detection capabilities, the number of outstanding defects will rise in response to the improved analysis capabilities. Such an increase in outstanding defects could be erroneously attributed to declining code quality, when that is clearly not the cause of this type of change in defect density. This type of change in defect density is solely attributable to the evolving analysis capabilities of Coverity Static Analysis.

Based on these reasons, measuring the value and progress of your static analysis deployment based on the number of repaired defects will provide a more accurate representation of your teams effort than tracking the number of outstanding defects.

Step 4: Notify Owners of Defects

Assigning ownership to each defect is a good first step, but the value of that first step is amplified immensely if the owners are then automatically notified about defects which have been assigned to them. If there is no notification mechanism in place, developers must be required to periodically check to see if any new issues have been assigned to them. However, this manual process requires extra effort on the part of the developer and slows the overall defect removal time and increases the chance that an issue may be ignored.

Automation of this notification mechanism along with automation of ownership assignments creates a reliable process that will help your team remove defects as they are introduced. Email is the most common means of notifying developers a defect has been assigned to them because it is a lightweight, effective and utilizes existing communication channels. Coverity recommends automating email reports that:

- Notify owners on a daily basis of any newly detected issues
- Notify owners regularly, perhaps once a month, of all outstanding issues assigned to them

This combination allows developers to address issues quickly, and provides a safety net to prevent issues from being overlooked. Coverity Static Analysis provides a scriptable interface for sending email notifications. Alternatively, customized scripts can be written which extract defect information and produce email reports in a particular format.

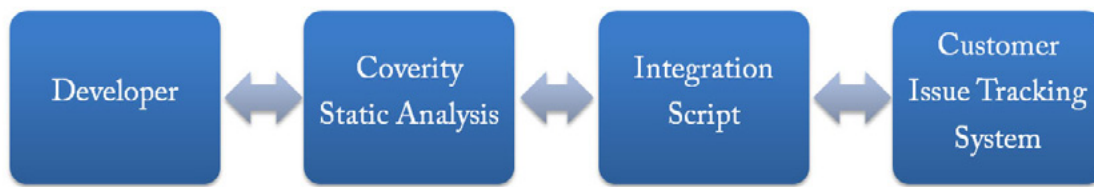
Step 5: Integrating Defect Resolution Workflow with SDLC

Applied properly, a new static analysis tool will not disrupt the existing SDLC. To derive full value from static analysis, organizations must consider how to properly incorporate it into their SDLC.

As previously mentioned, developers should budget time to address incoming static analysis defects as they are introduced throughout the software development cycle. Simply automating ownership assignment and notification will go a long way towards ensuring successful adoption of static analysis. To facilitate developer adoption, you should consider adding requirements that all newly introduced defects be eliminated at preexisting development checkpoints, such as during code reviews.

Integration between the Coverity Integrity Manager and existing issue tracking systems can provide a bridge to your existing SDLC defect resolution process. By developing custom scripts, Coverity can be configured to allow developers to “promote” a defect in the Integrity Manager into an issue within the primary issue tracking system. This powerful integration allows your existing issue tracking system to consume critical defects or vulnerabilities identified by static analysis with a single mouse click. Once inserted into primary issue tracking system the Coverity detected issue will undergo the same notification, escalation, and workflow mechanisms of any other issue. An example of this type of workflow is diagramed in Figure 2 below:

Figure 2: Example of One-click Creation of a Defect for Issue Tracking



After deciding upon your static analysis defect management goals as outlined earlier in this paper, you should revise your release criteria so both measurements are in alignment. The best release goals are ones that are meaningful, but do not create undue stress at release time. It is important that you select a release goal that reflects the nature of the software project to which you are assigning it. For example, if a release process permits a zero outstanding defects goal, that goal would be appropriate. However, for large, complex code bases, consider setting threshold quality goals, such as less than 1 defect per 10,000 lines of code. This type of goal permits the code base to enter the release period well ahead of the defect management goal, ensuring your overall goals are upheld - even if last minute defects are introduced that are challenging to remove.

Summary

If you demand the highest quality and security in the software applications and services that your organization creates, development testing using static analysis can be a valuable ally for you and your team. Understanding that development testing is not just a single tool or technology, but rather includes a set of processes is critical to derive the most value from your implementation. For static analysis to be successful, it requires more than simply properly installing and integrating the tool. The most impactful uses of static analysis involve a concerted plan surrounding how the defects and vulnerabilities the tool identifies are managed by development teams. Based on Coverity's experience with customers around the globe, sometimes the biggest challenge faced by development organizations in adopting static analysis is not in the resolution of defects, but in effectively managing identified defects within an organizations existing SDLC. Coverity recommends the following course of action for development organizations that want to maximize the value of their investment in static analysis:

1. Determine goals and metrics
2. Develop a project plan
3. Assign ownership of defects
4. Notify owners of defects
5. Integrate defect resolution workflow with SDLC

As discussed in this paper, to effectively implement and measure the success of your static analysis tool, development and QA teams need to define goals and metrics early. With consensus on these key goals, teams can then install a project plan that will enable them to immediately start making progress towards those goals. Automation of certain processes surrounding defect management must play a critical role to ensure defects are assigned and resolved in a timely fashion. The successful combination of integration planning and team goals will create a strong foundation for your team to improved developer efficiency while delivering innovative products and services to the market using software.

For More Information:
sales@coverity.com

Coverity Inc. Headquarters
185 Berry Street, Suite 6500
San Francisco, CA 94107 USA

U.S. Sales: (800) 873-8193
International Sales: +1 (415) 321-5237
www.coverity.com

Coverity and the Coverity logo are trademarks or registered trademarks of Coverity, Inc. in the U.S. and other countries.
All other company and product names are the property of their respective owners. © 2008-2011 Coverity, Inc. All rights reserved.

