OPEN SOURCE REPORT

2008

Acknowledgements

On behalf of the Scan site and Coverity, we would like to acknowledge the following organizations for their contribution to this report. It is our intention that developers, both open source and commercial, will benefit from the findings in this report, which is possible thanks to the following:

Open Source Developers – Around the world, open source developers have invested their time and energy to utilize results from the Scan site to improve the quality and security of their code. It has been a pleasure to work with fellow developers who are so passionate about their work. We thank you for your time, your feedback, and your trust in our organization. Moreover, we applaud your drive and success at building better software. A special word of thanks goes to the developers/projects that assisted with the creation of this report, or have provided valuable input and feedback on the Scan project, including: The Samba Team, The FreeBSD Project, der Mouse, Mark Arsenault, and the developers from ntp, Linux, Perl, Python, and PHP.

U.S. Department of Homeland Security – We extend our gratitude to the U.S. Department of Homeland Security for their continued support of the Scan site. In particular, we thank the members of the DHS responsible for the success of the U.S. Government's Open Source Hardening Project.

Executive Summary

The Scan site (www.scan.coverity.com) is sponsored by Coverity™ with support from the U.S. Department of Homeland Security. This report presents historical trend data collected by Scan researchers over the past two years.

Findings are based on analysis of over 55 million lines of code on a recurring basis from more than 250 open source projects, representing 14,238 individual project analysis runs for a total of nearly 10 billion lines of code analyzed. In summary, this report contains the following findings:

- **The overall quality and security of open source software is improving** – Researchers at the Scan site observed a 16% reduction in static analysis defect density over the past two years
- **Prevalence of individual defect types** – There is a clear distinction between common and uncommon defect types across open source projects
- **Code base size and static analysis defect count** – Research found a strong, linear relationship between these two variables
- **Function length and static analysis defect density** – Research indicates static analysis defect density and function length are statistically uncorrelated
- **Cyclomatic complexity and Halstead effort** – Research indicates these two measures of code complexity are significantly correlated to codebase size
- **False positive results** – To date, the rate of false positives identified in the Scan databases averages below 14%

Introduction

In January 2006, Coverity Inc. was awarded a contract from the U.S. Department of Homeland Security as part of the agency's 'Open Source Hardening Project' designed to improve the security and quality of open source software.

As part of this project, Coverity Prevent™, the industry leading static analysis tool, was made available to qualified open source software projects via the Scan website. Through the Scan site, open source developers can retrieve the defects identified by Prevent analyses through a portal accessible only by qualified project developers. The Scan site is located at: http://scan.coverity.com.

The site divides open source projects into rungs based on the progress each project makes in resolving defects. Projects at higher rungs receive access to additional analysis capabilities and configuration options. Projects are promoted as they resolve the majority of defects identified at their current rung. More information on the criteria for the Scan ladder is available at: http://scan.coverity.com/ladder.

Since 2006, the Scan site has analyzed over 55 million lines of code on a recurring basis from more than 250 open popular source projects such as Firefox, Linux, and PHP. This represents 14,238 individual project analysis runs for a total of nearly 10 billion lines of code analyzed. Of these 250 projects, over 120 have developers actively working to reduce the number of reported defects in their code. The efforts of these developers have resulted in the elimination of more than 8,500 defects in open source programs over a period of 24 months.

The collection of such a large, consolidated set of data regarding the security and quality of source code provides a unique opportunity to examine coding trends from a diverse collection of code bases.

The Scan Report on Open Source Software 2008 was created to provide an objective presentation of code analysis data from the Scan site. The goal of this report is to help software developers understand more regarding:

- The relationships between software defects and fundamental elements of coding such as function lengths, code complexity, or the size of code bases
- The progress the open source community continues to make in improving the integrity of their code by leveraging static analysis

Objective Measurement

In the past, evaluating source code has only been attempted at a very coarse level. Individual inspection by programmers or group code reviews can reach valid conclusions about the properties of a particular piece of code. However, these conclusions are often made through a lens of subjectivity regarding code characteristics such as coding style, the use of lengthy functions, large source files, individual programmer opinion, and a host of other factors. An additional issue with these types of reviews is that results can vary significantly due to the subjective nature of programming style and correctness.

Static Analysis Defect Density:

The number of Coverity Prevent discovered defects per 1,000 lines of code in a given project or set of projects.

The fundamental technology behind the Scan site and the data contained in this report is Coverity Prevent, a commercial static analysis tool that can provide an objective measure of the quality of software. Previous research from Microsoft has proven static analysis can be an accurate predictor of defect density (Source: *Static Analysis Tools as Early Indicators of Pre-Release Defect Density*, Microsoft Research).

This objectivity provides the opportunity to compare and contrast different pieces of code, as well as the same piece of code at different stages of development. Prevent identifies defects in software and while doing so, gathers a great deal of information about the structure and complexity of the source code it is analyzing. However, Prevent does not collect information regarding code maintainability or comments in code, which are not the focus of this report.

Because software packages can consist of hundreds of thousands to millions of lines of code, this report will use defect density to provide a common unit of measurement for discussion purposes. Static analysis defect density, for the purposes of this report, is defined as the number of Coverity Prevent discovered defects per 1,000 lines of code in a given project or set of projects.

When referring to static analysis defect density it is important to understand what exactly is included in this measure. Lines of comments and whitespace in code are not included in density measures because they do not affect the operation of the code. Density measures in the Scan Report on Open Source Software 2008 count lines of code. For the purposes of this report, a line of code with multiple statements is still considered a single line.

Scan Report Benchmarks

Because computer scientists and developers at Coverity continually add new defect detection capabilities to Prevent with each successive product release, the number of defects found in a given piece of software will vary significantly depending on the version of Prevent and the product's configuration settings. For example, significant product innovations during 2007 included the first use of Boolean satisfiability in static analysis and sophisticated race condition defect detection. These capabilities were not available to the projects discussed in this report, but they will be available via the Scan site at a later date and will be discussed in future installments of this report.

Defects identified by Prevent are classified according to the checker that identified them. While a single checker may often employ dozens of different methods for identifying a type of software defect, all of the defects found by that checker will be similar in nature.

When Prevent is configured, individual checkers may be enabled/disabled, and most have optional parameters to tune their aggressiveness or to toggle options in their behavior. Configuring checkers to conduct more aggressive analysis will result in more defects found, but potentially more false positives as well.

It is important to note that all open source projects discussed in this paper have identical checkers enabled/disabled in addition to identical aggressiveness and options settings for every individual checker. This allows us to provide an accurate measure of comparison between code bases based on consistent capabilities. Technical details of the tools and statistical methods used are provided in Appendix A. In broad terms the product configuration is:

- **Scan Benchmark 2006**: Coverity Prevent 2.4.0 with 12 checkers and default settings.

For the reader's reference, Coverity was shipping version 3.10 of Prevent at the time this report was published.

Comparing Scan Report Results to Other Defect Density Ratios

There are two main areas where the reader may be tempted to compare the results from this report with other defect density measures.

The first is in the area of other static code analysis measurements. For example, in Coverity's experience with analyzing over 2 billion lines of code with most of our checkers enabled and properly configured, we find that a typical static analysis defect density is approximately 1 per 1000 lines of code.

Other static analysis vendors have performed analyses to give alternative "typical" ratios. It is a useless exercise to compare the static analysis defect densities reported in this report with any other defect densities based on static analysis because the analyses are so different (different amount of false positives, types of defects discovered, etc.). The goal within this report is to simply provide a measure that is internally consistent such that all projects analyzed under that measure can be compared not only with each other but with themselves over time.

For example, consider a situation where two people compete to get the best gas mileage in identical cars with full gas tanks, and drive until they run out of gas. Clearly, it would be erroneous for one driver to claim victory because his 600 kilometer trip is 'further' than his opponent's 373 mile trip.

The second type of data the reader may be tempted to compare these findings to are defect densities reported by the Software Engineering Institute (SEI) or other organizations that study software. Those defect densities can be much higher per thousand lines of code as they try to take into account all defects in a code base, not simply the ones discoverable by a static analysis technology.

Overall Code Improvement by Open Source Projects

Data regarding the continually improving quality and security of open source projects at the Scan site have been collected across 14,238 individual project analysis runs. The results have been vetted by the open source developers who are responsible for the code being analyzed.

By comparing the number of defects identified in the first analysis of each open source project to the number of defects found in the most recent analysis, we can measure the overall progress of participating open source projects at the Scan site. These statistics exclude any defect identified as a false positive as determined by the project's developers. Even if the defect was not marked false until recently, if that defect occurred in the earliest analysis, it has been removed from these statistics.

Note that this statistic will naturally change as new projects are added to the Scan site, or existing projects change the status of a defect. Changes in the status of defects to or from false-positive would also impact this figure.

Based on the Scan 2006 Benchmark, the initial static analysis defect density averaged across participating projects is **0.30, or roughly one defect per 3,333 lines of code.**

The current average number of individual defects per project, based on the Scan 2006 Benchmark (as of March 2008) is 283.49. Based on the consolidated results of the most recent analysis for each project, the current static analysis defect density averaged across all the participating projects is **0.25, or roughly one defect per 4,000 lines of code.**

These findings represent an overall reduction of static analysis defect density across 250 open source projects of a total of 23,068 individual defects, lowering the average static analysis defect density in these open source projects by 16%.

Average Number of Defects Identified per Project

Low	1
High	4967
Average	283.49

Initial Static Analysis Defect Density— Scan Benchmark 2006

Low	0.02
High	1.22
Average	0.30

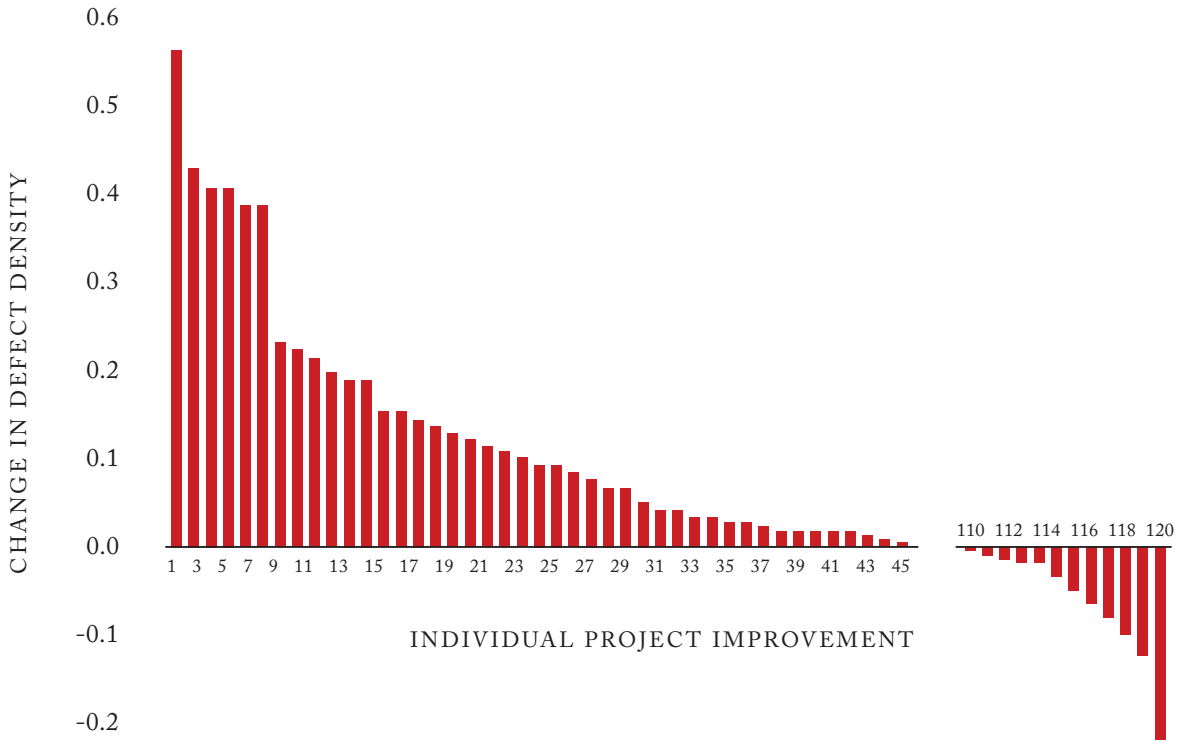
Represents the first analysis for each project at the Scan site

Current Static Analysis Defect Density — Scan Benchmark 2006

Low	0.00
High	1.22
Average	0.25

Represents the latest analysis for each project at the Scan site.

Change in Defect Density Across All Open Source Projects



Projects above the horizontal zero axis have a reduction in defect density, while projects below the axis have a increase in defect density. Improvement generally reflects action on the part of the project to resolve identified defects, while an increase indicates ignoring the analysis while also introducing new defects during code changes. Projects with no change include projects with too few analyses to capture progress/regress, or projects whose forward progress and introduction of new defects are balanced.

Note that this statistic changes as developers integrate fixes for outstanding defects into their code bases, and the analysis confirms that the defects have been resolved. Changes in the status of defects to or from false-positive would also impact this figure.

Projects with Exceptionally Low Defect Density

While we applaud all open source projects that make use of the Scan site for their efforts to deliver code of the highest quality, we would like to individually recognize the project teams responsible for writing the cleanest code among all participating projects. In particular, we wish to recognize the accomplishments of two groups of projects.

The following projects advanced to Rung 2 of the Scan ladder, as announced in January, 2008.

- Amanda
- NTP
- OpenPAM
- OpenVPN
- Overdose
- Perl
- PHP
- Postfix
- Python
- Samba
- TCL

All of these projects eliminated multiple classes of potential security vulnerabilities and quality defects from their code on the Coverity Scan site. Because of their efforts to proactively ensure software integrity and security, organizations and consumers can now select these open source applications with even greater confidence

The following projects are among the largest codebases that have eliminated all their Prevent-identified defects, and will be included in the next set of projects that are upgraded to Rung 2 of the Scan ladder.

- courier-maildir
- curl
- libvorbis
- vim

Each of these projects includes over 50,000 lines of code, and the developers have used the Scan results to eliminate all their identified defects.

Frequency of Individual Code Defect Types

To protect open source projects, sensitive information regarding specific projects is not presented on the Scan site and will not be discussed in this report. This includes the frequency of a given defect type for any specific open source project. For more descriptive information on these defect types, please see Appendix B. To provide insight into general trends regarding the frequency of specific defect types, consolidated totals across all open source projects are presented in the following table.

Defect Type	Number of Defects	Percentage
NULL Pointer Dereference	6,448	27.95%
Resource Leak	5,852	25.73%
Unintentional Ignored Expressions	2,252	9.76%
Use Before Test (NULL)	1,867	8.09%
Buffer Overrun (statically allocated)	1,417	6.14%
Use After Free	1,491	6.46%
Unsafe use of Returned NULL	1,349	5.85%
Uninitialized Values Read	1,268	5.50%
Unsafe use of Returned Negative	859	3.72%
Type and Allocation Size Mismatch	144	0.62%
Buffer Overrun (dynamically allocated)	72	0.31%
Use Before Test (negative)	49	0.21%

The most frequent type of defect, representing 28% of total defects found is the NULL pointer dereference. Use of pointers in C/C++ is widely understood to be error prone. This type of error often occurs when one code path initializes a pointer before its use, but another code path bypasses the initialization process. Pointers are a notoriously challenging programming concept that many languages elide altogether (e.g., Java). Senior developers know that new programmers frequently have trouble understanding pointers. Because pointers are often used to pass data structures by reference between pieces of program logic, they may be the most commonly manipulated data objects due to repeated copying, aliasing and accessing. Therefore, it is not surprising that the most frequently used artifacts will incur the most errors in manipulation.

Resource leaks are the second most frequent type of code defect found. Some resource leaks are pointer related, while others may be the result of misusing an API. Libraries may use pointers internally, but expose a different type of handle to the calling functions, and while this isn't the same as mishandling pointers, it can still result in a leak. Resource leaks often involve failure to release resources when the initial allocation succeeds, but a subsequent additional required resource is not available. Since most allocated resources are allocated once, accessed multiple times, then freed, it makes sense that resource leaks are less common errors than handling errors like NULL pointer dereferences.

Uninitialized data and use of memory after it is freed together comprise fewer than 12% of the issues identified by the Scan. These issues can be particularly troublesome to eliminate using traditional debugging techniques because the behavior of a program can vary each time it is executed depending on the contents of memory at the particular moment the defect occurs. These types of issues often change when a program is run in the debugging environment, giving them the tongue-in-cheek nickname 'Heisenbugs' because they cause the program behavior to change when observed in a different execution environment (e.g., debugging, profiling, etc.).

Slightly over 9% of total defects involve unsafe handling of return values from functions, either negatives or NULLs. In this situation, a program likely performs correctly most of the time, but the programmer has not accounted for some of the potential return values from functions being called. Negative return values are often defined to signal error conditions, and if the code does not handle the error and also uses the value in an unsafe way, unintentional consequences can occur. One example of unsafe usage would be an array index, where a negative value will address a piece of memory outside of the array. These types of errors sometimes occur when calling functions that can return a negative value or a NULL, so their frequency seems reasonable, since calls to such functions normally compose a smaller part of the code than operations on data.

Buffer overflows comprise over 6% of the total issues identified by the analysis. These occur when assignment to dynamically or statically allocated arrays occurs without checking boundary conditions causing memory reserved for other data to be overwritten. Examples of this include the str* group of functions, with either no boundary checking, or incorrectly calculated boundary limits.

What about Security or Crash-Causing Defects?

Readers of this report would understandably be curious about information related to 'security defects' and 'crash-causing defects.' However, presenting details about the average density of these defect 'types' would require too many presumptions about the relationship of a particular defect type and its consequences in a given code base. For example, a resource leak would most likely be considered a quality issue if it is infrequent and cannot be externally initiated in a short-lived program. A resource leak may be a crash-causing denial of service opportunity for an attacker if it is frequent, and/or can be initiated by externally supplied data, and/or in a long-lived program such as a persistent daemon process for a Web server or other network service.

Because of the difficulties involved in reliably categorizing individual defects based on their symptoms as opposed to root cause in the code, this report will not speculate regarding the potential densities of these types of software issues.

Function Length and Defect Density

Many developers have heard or believe that extremely long functions can lead to programmer errors. If long functions do indeed lead to errors then a correlation should exist between projects with long functions and higher defect densities. Similarly, projects with the shortest functions should have the fewest defects, and any increase in average function length should be proportional to increasing defect density.

The open source projects discussed in this report range in size from 6,493 lines of code to a little over 5,000,000 lines of code with a mean average of 425,179 lines of code. The following tables provide some of the basic data for a discussion of function length and defect density.

It is interesting to note the wide variance in these values for open source projects analyzed by the Scan site. For example, the project with the largest average number of lines in each function, 345.72, has functions that are almost 25 times as long as the shortest ones, while the smallest average number of lines per function is 13.97.

The correlation between average function length and static analysis defect density is +1.49%, indicating that these two variables can be considered completely unrelated. Therefore, looking at average project length, we see no relationship between static analysis defect density and function length in code.

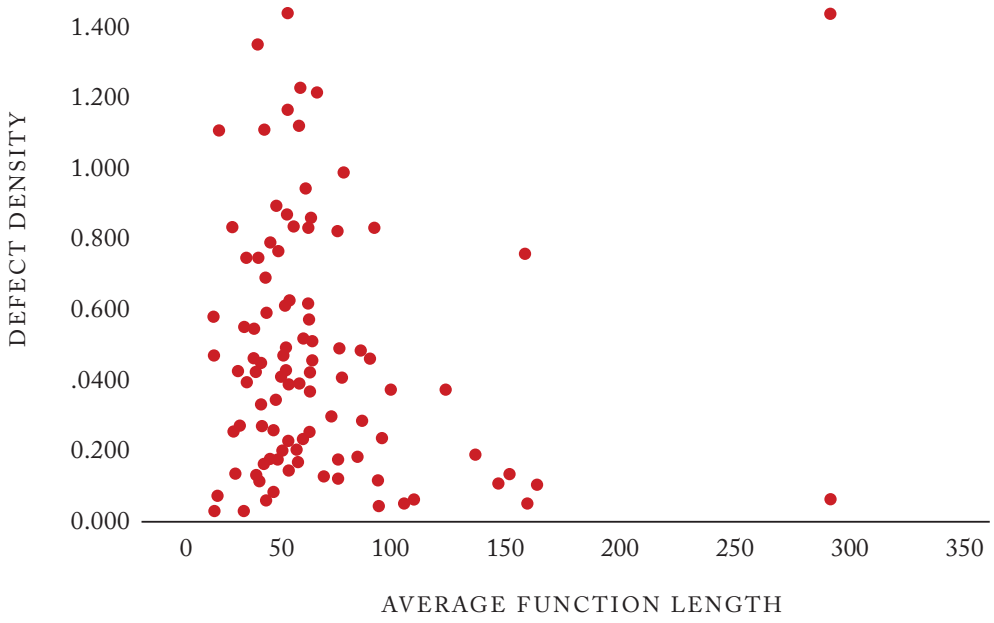
Lines of Code	
Minimum	6,493
Maximum	5,050,450
Mean	425,179

Number of Functions	
Minimum	47
Maximum	215,925
Mean	12,880

Average Function Length	
Minimum	13.97
Maximum	345.72
Mean	66

There are certainly justifications for an occasional function to be long, but the data in these tables represent the average across all functions in a project. A popular programming adage states that an entire function should fit on the programmer's screen at once. Interestingly, the average function length of 66 comes close to this measure. In fact, the average function length has a median value of 48.04, so half of the projects examined have an average function length of 48 lines or more.

Static Analysis Defect Density and Function Length



It is important that the unit of measure for comparison here is density based, because longer functions contain more code, and consequently more errors if the static analysis defect density throughout a long function is constant. If the number of defects were to increase in relationship to function length and not more quickly than that, we could assert that the common assumption about long functions being error-prone is true. However, the dataset shows that many codebases with the same average function length vary widely in static analysis defect density, and codebases with longer average function lengths do not show a consistent increase in defect rates, and in fact often show decreasing defect rates. So, based on this data, there is no statistically significant relationship between function length and static analysis defect density.

Possibilities do remain to prove a relationship between function length and defect density. For example, it may be the case that some specific type of error that is not included in the Scan 2006 Benchmark may occur more frequently in longer functions. It could also be the case that looking at errors in individual functions might show errors occur more often in longer functions, but averaging over the entire codebase hides this. Longer functions could also make code harder to maintain. The current report does not address these possibilities. Still, based on the analysis of available data, the overall assumption that longer functions lead to more bugs is untrue.

Finally, it is important to note that while the findings of this research show there is no correlation between function length and static analysis defect density, there may still be other correlations between function length and defect types that are not identified by the static analysis testing conducted for the Scan 2006 Benchmark.

Code Base Size and Defect Density

SCAN.COVERITY.COM

Intuitively, one might presume that better programmers write better code. If on average a programmer makes mistakes at a consistent rate, the sheer amount of code they produce would be a predicator of the number of defects they would create. However, a task like coding, which involves creativity and original work is not likely to have a consistent rate of errors. Though averaged over a large enough sample set, the results may be consistent enough to be informative. Given this, examining the relationship between code base size and defect density merits exploration. The tables here provide data for such a discussion.

Lines of Code	
Minimum	6,493
Maximum	5,050,450
Mean	425,179

Number of Files	
Minimum	53
Maximum	28,856
Mean	1,576

The correlation between the number of lines of code in a project and the number of defects is +71.9%. This represents a very significant relationship between the number of defects and the size of a code base. Note that this does not imply that the larger the code base, the higher the static analysis defect density, but as intuition might suggest, as the codebase grows in size, so does the number of defects. What may not be intuitive is how strongly correlated the two values are, and many developers may expect a greater degree of variation between projects of a given size with the least defects, and the most defects.

Because of this correlation, it may be possible for developers to anticipate the number of defects in a given piece of software predicated on the size of the code base, and to do so with almost 72% accuracy in that estimate. By extension, developers may be able to track defects in a codebase, as they are fixed, and use this estimate to judge roughly how many defects remain in the code.

The remaining 28.1% variance most likely represents the extent to which individual programmers or the nature of a particular project result in the code having a greater or lesser number of defects than projected by this linear prediction. A number of potential factors could explain why more defects occur in larger code bases, including:

- **Legacy code** – Larger code bases are almost always more mature, and therefore more prone to developer turnover
- **Conflicts between code branches** – When individual developers are responsible for specific branches of code in an application, inconsistencies between branches can lead to developer confusion, and increase defect density
- **Uncertain accountability** – For open source developers on large projects, due to time, distance, language and the volunteer nature of open source, clearly delineating accountability is often a challenge
- **Inconsistent review processes** – Again, for large, geographically distributed development teams, not all testing and review processes may be applied consistently by each developer

Open Source & Commercial Code

Over the entire history of the Scan site and six years of work with commercial source code, Coverity researchers and engineers have reviewed over 2 billion unique lines of source code. This represents a total of 250 million lines of open source code, and a massive amount of code from commercial and government sources.

Many developers have an opinion about the differing quality and security of open source versus commercial software, and a number of theories have been hypothesized to justify the superiority of one class of code over another. However, comparing these two classes of code is impossible for the purposes of this report, primarily due the difficulty involved in obtaining comparable datasets. Additional issues with making such comparisons are privacy and disclosure requirements surrounding commercial code that restrict or eliminate altogether the ability to discuss that class of code in a public report.

Ultimately, we believe that most developers are inherently interested in creating quality products, because excellent software will be adopted and relied upon just as quickly as defective software will be avoided and discarded. From an industry perspective, both open source and commercial software continue to mature and thrive. In fact, many of the developers with open source projects at the Scan site also have existing responsibilities developing commercial code in a variety of capacities.

The groups of developers that work on open source and commercial code overlap, and best practices for software development cross-pollinate as well. Differences between the code with the most defects and the code with the least defects may be driven by other aspects of a project, such as prioritization, a tendency to reward success, and the method for prioritizing tasks.

Cyclomatic Complexity and Halstead Effort

As previously stated, the Scan site examines source code using Coverity Prevent, which leverages Coverity's Software DNA Map™ analysis system. This analysis system captures a massive amount of detail about the software during the analysis process because it conducts a full parsing of the source code, including the assembly of dictionaries of identifiers and functions. Prevent accomplishes this without any change to the build environment or code.

Two standard software complexity algorithms computed during Prevent analysis runs are Cyclomatic complexity and Halstead effort. Each of these algorithms provide a way to mathematically represent the complexity of a piece of software, which allows us to draw objective comparisons of code complexity between two unrelated pieces of software.

Lines of Code	
Minimum	6,493
Maximum	5,050,450
Mean	425,179

Cyclomatic Complexity	
Minimum	158
Maximum	816,066
Mean	53,035

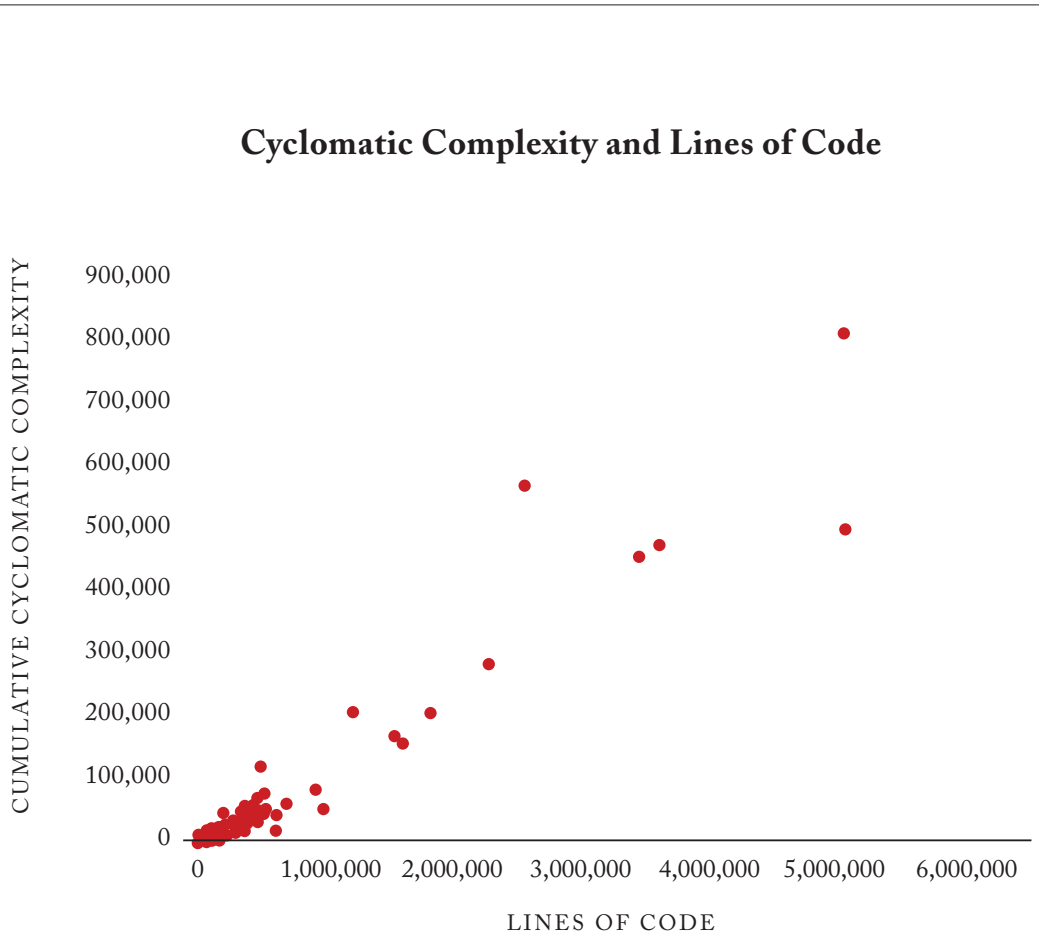
Halstead Effort	
Minimum	2,276
Maximum	71,949,783
Mean	6,399,178

Cyclomatic Complexity:

Measures the amount of decision logic in a single software module. It is used for two related purposes in the structured testing methodology. First, it gives the number of recommended tests for software. Second, it is used during all phases of the software lifecycle, beginning with design, to keep software reliable, testable, and manageable. Cyclomatic complexity is based entirely on the structure of software's control flow graph. (Source: NIST, Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric)

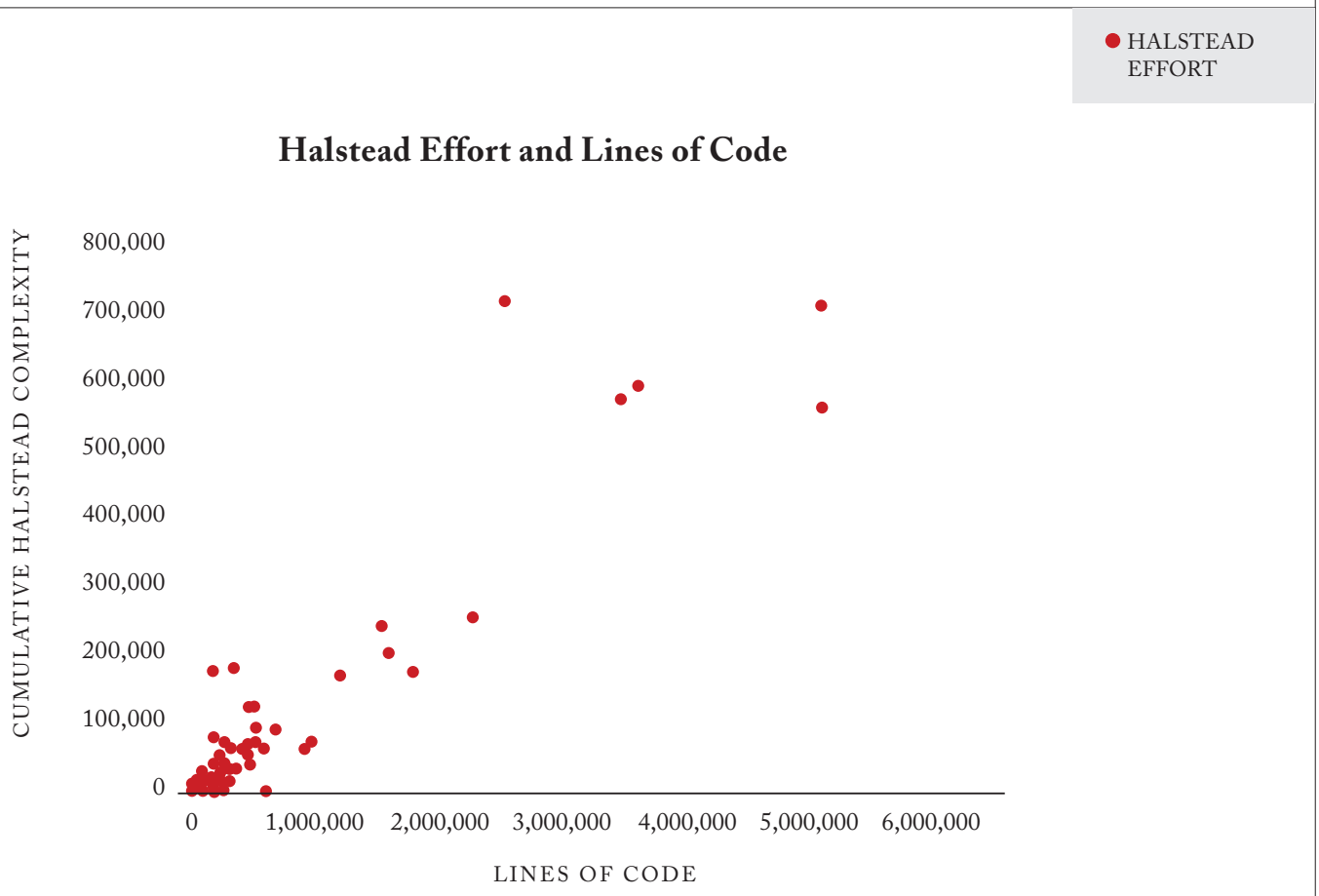
Halstead Effort:

Measures a program module's complexity directly from source code, with emphasis on computational complexity. The measures were developed by the late Maurice Halstead as a means of determining a quantitative measure of complexity directly from the operators and operands in the module. Among the earliest software metrics, they are strong indicators of code complexity. (Source: Carnegie Mellon University, Edmond VanDoren, Kaman Sciences, Colorado Springs)



Coverity Prevent captures the Cyclomatic complexity and Halstead effort calculations for each function that it analyzes. Since the algorithms for both of these metrics include factors that increase with the amount of code, it is not surprising that the total complexities for both metrics are significantly related to the size of the code base that they are calculated for.

Cyclomatic complexity and the number of lines of code in Scan projects are correlated at +91.94%. This nearly perfect correlation means that a large part of the total complexity number comes directly from the size of the codebase. This implies that it might be more interesting for developers to ask "By how much does our complexity metric differ from the expected value, for this much code, and is it larger or smaller?" For example, if developers have agreed to refactor code, one measure of success would be to observe how much lower their complexity metrics are for their codebase than expected by this linear estimation.



Total Halstead effort and the number of lines of code in Scan projects are correlated at +87.29%. This is also a very strong correlation which indicates that, like Cyclomatic complexity, a large part of this measure provides information about how much code is analyzed, rather than just its complexity. Since the correlation is not as high as for Cyclomatic complexity, there is more room for variation which is likely due to implementation complexity or challenges inherent to what Halstead effort measures in a given software system.

The Halstead totals here were calculated as a total of the complexity of the functions in a codebase. Since the Halstead algorithm allows using the program call flow graph to generate an overall number for the program, the correspondence of that number with program size could be weaker This means that two codebases with a similar quantity of code could have very different call flow graphs, and the Halstead measurement generated using the call graph could reflect this.

Based on these results, the question may be asked whether complexity calculations provide any significant information at an individual project level. While it would be interesting to compare this result at a more specific level by correlating complexity and length of individual functions, it may also be the case that looking at the total complexity of a project averages away valuable information identifying complex and simple functions.

False Positives

A common concern from developers regarding static analysis pertains to the accuracy of results, specifically false positive rates that can compromise the usefulness of a given tool. A false positive result can be defined as what occurs when a test claims something to be positive, when that is not the case. For example, a pregnancy test with a positive result (indicating a pregnancy) would produce a false positive in the case where the woman is not actually pregnant. False positive results in this report were individually flagged as such by open source developers as they reviewed their analysis runs. Observed false positive rates vary from project to project, depending on various elements in the design of the code being analyzed.

The average false positive rate of the 250 open source projects participating at the Scan site is 13.32%, which represents a total of 28,916 defects and 3,853 false positives.

This overall false-positive average is based on an out-of-the-box configuration according to the Scan 2006 Benchmark, which relies on version 2.4 of Prevent's analysis engine (Note: Coverity is currently shipping version 3.10 which delivers a significantly lower false positive rate).

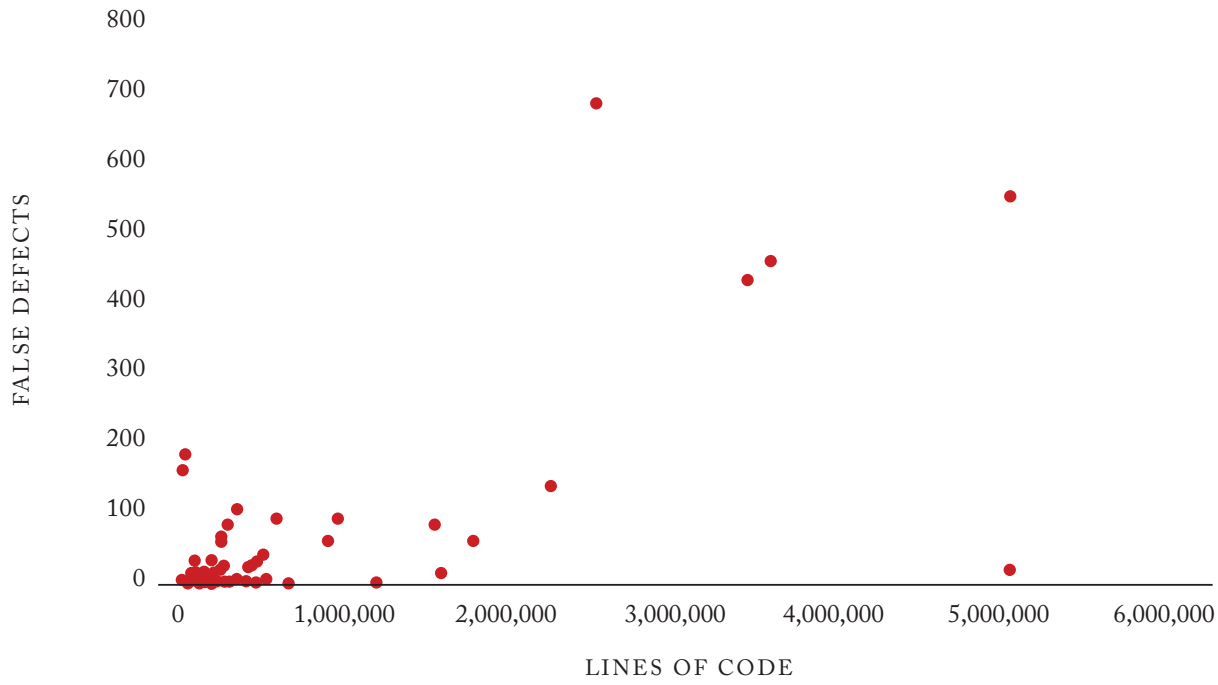
Initial Scan Benchmark 2006 defect rate

- .3480 average across all projects
- .2531 weighted
- 13,329 defects (1812 false)

Latest Scan Benchmark 2006 defect rate

- .2823 average across all projects
- .1967 weighted
- 10,685 defects (1733 false)

False Positive Results and Lines of Code



The +54.3% correlation between false positive and lines of code can be described as a large correspondence, indicating that the larger the code base, the greater number of false positive defects we can expect to see flagged.

At 71.90%, the correlation between lines of code and number of defects found is much stronger than the relationship between false positives and lines of code. Taken together, these two numbers show that Prevent identifies more bugs and more false positives in larger codebases, but the frequency of false positive results decrease as the size of a codebase increases.

As the Scan site continues to evolve, open source developers will have access to advanced new technology designed to further lower false positive results. One example of this is Coverity's Boolean satisfiability based false path pruning solver, which has proven to lower false positive results by an additional 30% in testing.

It is also Coverity's intent to provide access to tuning parameters so that open source developers with projects on the Scan site can further improve the accuracy of the analysis for their own code base. As additional defect checkers are made available on new rungs of the Scan Ladder, the overall false-positive rate will vary depending on the presence of coding idioms, specific to each checker, and the frequency with which those idioms occur in each open source project.

Finally, one aspect of false-positive calculation should be discussed since it is sometimes confusing. As the analysis runs over a code base and developers identify some defects as false positives, subsequent runs will remember those defects have been flagged and they will remain marked as a false-positive result.

For example, if a code base starts with 100 defects, 10 of which are false positives, then it has a 10% false positive rate. If developers then fix the 90 true positive defects and introduce no new defects as a result of their fixes or other development on the code, then the next analysis will show 10 defects, of which 10 are false positives.

The above interpretation could lead an uninformed reader to view a given false-positive rate as 100%—when clearly this is not the case. Therefore, to calculate the overall false-positive rate, one needs to look at the list of unique defects identified, and the percentage of them that were false positives—which has been done for the data contained in this report.

Final Notes on Interpreting Data

Due to the uniqueness of open source projects, programming experience, and coding styles of open source developers themselves, there are some important considerations to keep in mind when reviewing the results of this report.

Developer interpretation - Every group of open source development team that participates in the Scan site will have their own view regarding the analysis results of their code. Specifically, a "human factor" exists in the data presented herein because developers sometimes have different ideas about what constitutes a defect, what is an intentional use of code that registers as a defect, and what is a false positive to name just a few possibilities. Because of this, the number of reported defects and false positives may be off by a small quantity in individual projects, but this type of error would occur in both positive and negative directions. Therefore, if the number of defects is understated in this regard, it is by only a small margin.

Frequency of analysis – It is also worth noting the role that the frequency of project analysis may play in the data contained in this report. Projects that are scanned each time a new official version is released will show a continual progression of defects being eliminated over time. While projects scanned only at release time will have defects identified for developers to fix, the number of defects eliminated will not be reflected by the Scan site until the project is analyzed again. As a result of this, the progress made by projects with infrequent scans may be slightly understated in the report.

Although these factors may complicate the analysis of data in this report, it is still valuable to use the information available through this massive collection of data to learn more about the ongoing evolution of open source software with respect to security and quality issues.

APPENDIX A:
Tools and Statistical Methods

Scan Benchmark 2006

Coverity Prevent is updated on a regular basis with new commercial releases. New releases are gradually migrated into the Scan project service. Efforts are currently underway to enable this process to proceed more rapidly.

Since the largest set of analysis runs have been performed with Prevent 2.4.0, and that version was the current release in 2006, this data has been collected under the heading 'Scan Benchmark 2006'.

The following C-language checkers were enabled with default settings: DEADCODE, FORWARD_NULL, NEGATIVE_RETURNS, NULL_RETURNS, OVERRUN_DYNAMIC, OVERRUN_STATIC, RESOURCE_LEAK , REVERSE_INULL, REVERSE_NEGATIVE, SIZECHECK, UNINIT, USE_AFTER_FREE

The default C++ checkers have also been enabled using the '-cxx' option to Prevent, but their results have not been discussed in this report because C++ projects are a subset of the projects on Scan, and do not represent a large enough sample set to provide a high level of confidence in any statistical analysis.

Statistical Methods

For all calculations in this report that refer to a correlation percentage, the techniques were applied as described here.

Gather the set of data points to be compared, and ensure that the units of measure to be compared contain no conflicting or redundant terms.

Plot the scattergraph for the comparison, and review outliers to determine if the project in question had anomalies that affected its data. For example, partially broken builds during the first or last analysis of a project could lead to an incorrect count for lines of code, and would affect defect density calculations as well. Where possible, obtain a corrected value for the data point, or if not possible, exclude that data point from the set to avoid polluting the results.

Calculate the Pearson Correlation Coefficient 'r', and square it for use as a confidence value. Retain the sign of 'r' to simplify the text in the one case where an insignificant inverse relationship was found.

We received feedback on the statistical analysis in this report that it may be more appropriate to use logarithmic scales for the calculation of correlation values, and on scatterplots. We reviewed that correlations using logarithmic scales, and found that the r-squared confidence values varied by less than ten percent from the linear ones. For some correlations, such as codebase size versus defect count, the correlation increased (71.9% -> 79.9%) while for some others, the correlation decreased. We have retained the linear scales in this report for the sake of simplicity.

APPENDIX B:
Defect Types

There are a variety of defect types of defects identified by the analysis at the Scan site. The 12 checkers in use can be grouped into nine categories of defect types. Where different types of defects fit into the same category, they will be explained together.

Unintentional Ignored Expressions

Unintentional Ignored Expressions indicates that part of the source code is not reachable on any execution path. One example involves if statements and return statements. Consider when a programmer makes a mistake in the if() statement's test value, and the code block for the statement returns from the current function. If the test value is always true then all the code after the if() block can never be reached. Similarly if the test value is always false then the code in the if() block can never be reached.

When more than one programmer works on the code it is even easier for a change be introduced far away from the if() block that will have an impact on it. Manual code reviews are unlikely to spot this type of error because the cause and effect could be distant, and even in different files.

This type of defect can be a problem because the unreachable code was meant to perform some needed action such as an initialization function. The code can also be an indicator that another portion of the code is not behaving as intended, and in that case the fix is not to change the conditional closest to the ignored code, but rather to change the portion of code where the variables are assigned that leads to the always or never taken branch condition.

NULL Pointer Dereference

In C/C++, pointers may contain values that refer to memory locations, or have a special value NULL. The term 'dereference' means to use a pointer and access the memory location it points to in order to obtain the data stored there. Dereferencing a pointer with the special value NULL is an invalid operation, because by definition it has no meaning. An operating system will terminate a program that attempts this operation. Termination is a useful behavior because the programmer will more quickly discover the error if the program crashes than if the program continues to run but produce unpredictable output.

This analysis looks for pointers that are assigned NULL, or successfully compared to NULL, and a path with a program subsequently dereferenced as the pointer without it having been assigned a valid value in between the two operations.

NULL pointer dereferences are normally very easy to correct in the code once you know they are there. Whether or not the program crashes at test time depends on having the required test configuration that will cause the troublesome code path to be executed when the program is run. Often, test suites do not provide coverage of a significant percentage of the possible paths in a program. Static analysis has the advantage of not requiring test suites to inspect all the possible paths.

Use Before Test

There are two special types of values that programmers might test for in order to handle them correctly. The two types are NULL values and negative values. NULL values need to be tested for in order to avoid NULL pointer dereferences. Negative values need to be tested for in order to avoid using them in unsafe places, such as the index to an array.

When programmers test the value of a variable this implies that they believe it is possible for the variable to have the undesired value at the point of the test. If a code path exists where the variable is used in an unsafe way before this tested, then the test does not accomplish its objective. For example if a pointer variable is dereferenced before testing it for NULL, and the pointer variable is not assigned between the use and the test, then if the test was ever true, it occurs too late because the program already would have crashed.

Buffer Overrun

Buffer overruns are one of the best-known causes of security issues. When a program is copying data, it needs to understand the limits on the destination it is writing to and the amount of data it is writing. If the source data is larger than the destination data storage area and the program does not check for this, then the destination area will be filled and the remaining data will be written into the memory beyond the destination area. Since the memory beyond the destination area has been assigned a different purpose, the program has now overwritten it with what is effectively garbage, since the two areas may not even store the same type of data.

When the source data has been provided by a user, from an input file or the network, there is a potential that a malicious user can plan their input in advance so that a portion overwrites beyond the destination boundary causing the program to misbehave to the attacker's advantage.

Buffers can be statically allocated by declarations in the program, or dynamically allocated based on calculations made if the program is running. Coverity calls these two types static buffers and dynamic buffers. When data is copied to either type of buffer without assuring the source is smaller than the destination, the analysis will flag a defect.

Resource Leak

Many programs allocate a variety of resources at run time. Often, memory requirements are driven by the input supplied to the program. Additional resources such as file handles may be allocated when accessing data on disk. When a program allocates resources it has the responsibility to return those resources when they are no longer needed. In order to return allocated resources a program must retain a handle associated with the allocation. The function to release the allocation takes the handle as an argument so that the operating system can look up the handle and perform the correct operations to release the resource for reuse.

When programs leak memory, the operating system may terminate them for exceeding limits, or they may cause resource starvation which affects other programs on the computer. When programs leak file handles, they may fail in unanticipated ways when a later attempt to open a file is refused by the operating system because they have exceeded their limits. The analysis looks for resources that are allocated, whose handles are then discarded without the resources being freed.

Unsafe Use of Returned Values

In C/C++, when a program calls a function the results are normally passed back to the calling code in a return value. In many cases a function reports errors in its operation, or complains about the arguments it was called with, by returning a negative or a NULL value.

When the analysis sees that a function can return a negative or NULL value, and a code path exists where that value is subsequently used unsafely, then a defect will be flagged.

Type and Allocation Size Mismatch

When memory is allocated, the address of the space assigned is stored in a pointer. Although any pointer contains a memory address, pointers do have different types. The type of the pointers used to access the data referred to when the pointer is dereferenced.

If an allocation statement requests a particular amount of storage space, and the address of the allocation will be stored in a pointer that has a type that is larger than the requested space, a defect will be noted. Attempts to access the assigned pointer will also access memory outside of the allocated range, either reading garbage data, or corrupting memory.

Uninitialized Values Read

When variables are defined they do not need to be given an initial value. However, if variables are not initialized when they're defined, then the first operation on them must be to assign a value. Performing an operation that reads the contents of the uninitialized memory will result in random behavior, and is almost never what is desired.

If the analysis sees that a variable is read before it is first assigned a value, then a defect will be noted at that point.

Use After Free

Related to the allocation and freeing of memory, this type of error is almost exactly the opposite of a resource leak. Once a resource is freed, the application must not continue to use the handle that has been released. In the case of memory, that space may have been assigned for another program to use, or assigned for another purpose within the current program. Writing to memory that has been freed will overwrite its contents, and if that memory is currently assigned for some use, the effect is equivalent to memory corruption.

A defect occurs if a resource handle is released, and then without that handle being assigned by a new allocation, the handle is used as if it were still valid.