Metrics That Matter:

Quantifying Software Security Risk

Brian Chess
Fortify Software
2300 Geng Road, Suite 102
Palo Alto, CA 94303
1-650-213-5600

brian@fortifysoftware.com

Abstract

Any endeavor worth pursuing is worth measuring, but software security presents new measurement challenges: there are no established formulas or procedures for quantifying the security risk present in a program. This document details the importance of measuring software security and discusses the less-than-satisfying approaches that are prevalent today. A new set of metrics is then proposed for ensuring an accurate and comprehensive view of software projects ranging from legacy systems to newly deployed web applications. Many of the new metrics make use of source code analysis results.

1. Introduction: Why measure?

What would happen if your company cut its security budget in half? What if the budget was doubled instead? In most companies today, no one knows the answers to these questions. Security remains more art than science, and nothing is more indicative of this fact than the inability of security practitioners to quantify the effects of their work.

Software security is no exception: nearly every major business-critical application deployed today contains vulnerabilities—buffer overflow and cross-site scripting are commonplace, and so are many other, less well-known, types of vulnerabilities. These problems can be exploited to cause considerable harm by external hackers or malicious insiders. Security teams know that these errors exist, but are, for the most part, ill equipped to quantify the problem. Any proposed investment in improving this situation is bound to bring up questions such as:

- Are the applications more secure today than yesterday—or less secure?
- Does security training really make a difference?
- How will we know when our systems are secure?

This paper examines the current state of practice for measuring software security. It then suggests two new approaches to the problem: quantifying the secure development lifecycle, and focusing on the root cause of many vulnerabilities using metrics built with source code analysis results.

2. The State of Practice: Three Flawed Approaches to Measuring Security

1. Build then Break: Penetration Testing as a Metric

The de facto method that most organizations use for measuring software security today can be summarized as "build then break." Developers create applications with only a minimum of attention paid to security, and the applications are deployed. The operations team then attempts to compensate for the problematic software with perimeter security. When the team takes inventory of all of the ways that data moves through and around the perimeter defenses, it becomes clear that the perimeter security is insufficient. At this point, the operations team may bring in penetration testers to find the problems before hackers or malicious insiders do. The penetration testers generally have a fixed schedule for performing their work, and their goal is to find a small number of serious problems to justify their consulting fee. Once these problems are resolved, everyone is happy. But there's no reason to believe that the penetration test revealed all of the problems with the application. In fact, subsequent audits usually prove that it did not. There's also very little feedback to the developers, so penetration tests often find the same types of problems over and over again.

2. *Measure Software Security as Part of Software Quality*

A naive approach to software security calls for treating security as just another aspect of software quality. The problem is that traditional quality assurance is aimed at verifying a set of features against a specification. Software security, however, requires much more than well-implemented security features. The reality is that a typical process for achieving good results with respect to traditional quality issues does not guarantee good results with respect to security issues. In other words, you have to focus specifically on security in order to improve it. Good security is not a byproduct of good quality.

Further complicating this approach, the majority of Quality Assurance (QA) departments lack the requisite security expertise to carry out adequate security tests. Finally, as Figure 1 illustrates, any approach to quality that is based on the behavior of regular users will leave many untested opportunities for attackers.

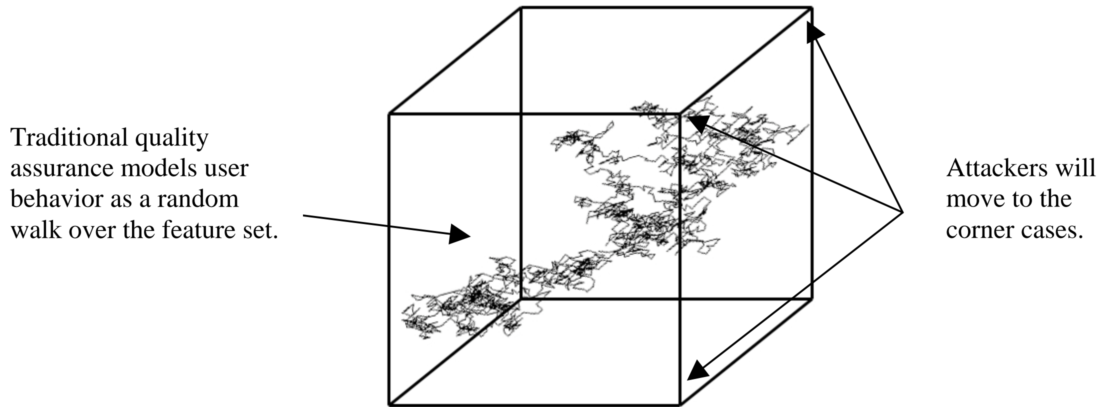

Figure 1: A quality-oriented approach to security leaves many opportunities for attackers.

3. *The Feel-Good Metric: If It Hasn't been Hacked Yet, It's Probably Okay*

Because security so often goes unquantified, the bottom-line measure for security is often gut-feel. Human nature and the nature of security are in conflict on this point: people and organizations tend to gain comfort with the status quo over time, but security may actually degrade as time passes. New types of attacks and new applications for old types of attacks can harm a program's security—even as an organization becomes more and more complacent because security "hasn't been a problem yet!"

A similar fallacy holds that the security of a program can be correlated to the breadth of its adoption. Interestingly, this line of reasoning always seems to work in favor of the status quo. For applications with a small user base, people assume that attackers will not take an interest. For applications with a large user base, people assume that any security issues will be flushed out of the system shortly after release. In truth, security is no more related to breadth of adoption than it is to longevity. The BugTraq mailing list (where news of many new vulnerabilities debuts) is filled with entries about small and obscure applications. Furthermore, the long history of buffer overflows in widely adopted programs as varied as SendMail and Internet Explorer shows that neither age nor a large install base prevents attackers from finding new exploits.

3. A Positive Trailing Indicator

There are encouraging signs that the longstanding neglect, ignorance, or apathy shown to software security is beginning to change. Microsoft has adopted the Trustworthy Computing Security Development Lifecycle (SDL) process for the creating software that needs to withstand malicious attack [4]. The process adds a series of security-focused activities and deliverables to each of the phases of Microsoft's software development process. These activities and deliverables include risk analysis during software design, the application of source code analysis tools during implementation, and code reviews and security testing during a focused "security push." Before software subject to the SDL can be released, it must undergo a final security review by a team independent from its development group. When compared to software that has *not* been subject to the SDL, software that *has* undergone the SDL has experienced a significantly reduced rate of external discovery of security vulnerabilities. Figure 2 shows the number of security bulletins for Windows 2000 in its first 12 months after release versus the number of security bulletins for Windows Server 2003 in its first 12 months after release. The number of issues has been reduced by more than 50%, even as the size and complexity of the operating system has increased.

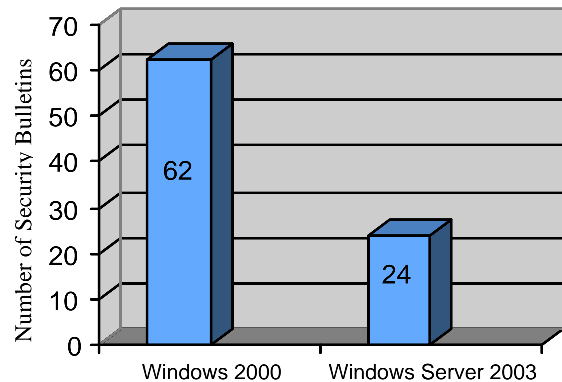

Figure 2. A measurable improvement in Microsoft OS security: the number of security bulletins issued in the first 12 months following two major OS releases.

However, Figure 2 is an example of a *trailing indicator*. It only demonstrates that security has been improved after the OS has been released. It provides strong evidence that the SDL has a beneficial effect on the security of the resulting operating system, but if Microsoft only releases an operating system every five or six years, it requires five or six years to know whether there is a measurable improvement in software security from the previous release. That is far too slow. Security must be measured on an ongoing basis throughout the software development lifecycle, and for that we need *leading indicators* for software security.

4. Software security metrics you can use now

Having explained the measurement problem and how *not* to solve it, we now turn to two practical methods for measuring software security.

1. *Quantify the Secure Development Lifecycle*

Software security must be addressed as part of the software development lifecycle [*1,2*]. There are practical steps that development groups can take during each phase of the lifecycle in order to improve the security of the resulting system. These steps include:

- **Evaluate** the current state of software security and create a **plan** for dealing with it throughout the development life cycle.
- **Specify** the threats, identify both business and technical risks, and plan countermeasures.
- **Review** the code for security vulnerabilities introduced during development.
- **Test** code for vulnerabilities based on the threats and risks identified earlier.
- **Build a gate** to prevent applications with vulnerabilities from going into production. Require signoff from key development and security personnel.
- **Measure** the success of the security plan so that the process can be continually improved. Yes, your measurement efforts should be measured!
- **Educate** stakeholders about security so they can implement the security plan effectively.

Each of these steps can be measured. For example, if your security plan includes educating developers, you can measure what percentage of developers have received software security training. [1]

Of course, not all organizations will adopt all steps to the same degree. By tracking and measuring the adoption of secure development practices, you will have the data to draw correlations within your organization. For example, you will likely find that the up-front specification of threats and risks correlates strongly to a faster and easier security signoff prior to release.

2. *Use Source Code Analysis to Measure Security*

All software organizations, regardless of programming language, development methodology, or product category, have one thing in common: they all have source code. The source code is a very direct embodiment of the system, and many vulnerabilities manifest themselves in the source [3]. It follows that the source code is the one key artifact to measure as part of assessing software security. Of course, source code review is useful for more than just metrics. The following sections discuss some source code analysis fundamentals and then look at how source code analysis results can provide the raw material for powerful software security metrics.

[1] It seems reasonable to assume that Microsoft also produces metrics related to their SDL, but they have published very little on the topic.

5. Source Code Analysis

Source code analyzers process code looking for known types of security defects. In an abstract sense, a source code analyzer searches the code for patterns that represent potential vulnerabilities and presents the code that matches these patterns to a human auditor for review. The three key attributes for good source code analysis are accuracy, precision, and robustness.

A source code analyzer should accurately identify vulnerabilities that are of concern to the type of program being analyzed. For example, web applications are typically at risk for SQL injection, cross-site scripting, and access control problems, among others. Further, the analysis results should indicate the likely importance of each result.

The source code analyzer must also be precise, pointing to a manageable number of issues without generating a large number of false positives. Furthermore, if a program is analyzed today, and subsequently re-analyzed tomorrow, it is likely that only a small amount of code will have changed. The source code analyzer must be able to give the same name to the same issue today and tomorrow, allowing for the ability to track when issues appear and disappear. This capability is critical for extracting meaningful metrics from source code analysis results.

Finally, the source code analyzer must be robust: it must be able to deal with large, complex bodies of code. Of course, not every issue the source code analyzer identifies will be a true vulnerability. Therefore, part of being robust is allowing human auditors to evaluate and prioritize potential issues. A preferred scenario has a human auditor classify the output from the analyzer into 1) severe vulnerabilities that must be corrected immediately, 2) bad practices, and 3) issues that are not relevant to the organization. An even better application of source code analysis allows developers to analyze their own code as they write it, making source code analysis part of the daily process of program development.

6. Security Metrics Based on Source Code Analysis

The best metrics that can be derived from source code analysis results are, to a certain extent, dependent upon the way in which an organization applies source code analysis. We will consider the following scenarios:

1. Developers use the source code analyzer on a regular basis as part of their development work. They are proactively coding with security in mind.
2. A software security team uses the source code analyzer as part of a periodic code review process. A large body of code has been created with little regard for security. The organization plans to remediate this code over time.

Of course, the first scenario is preferable, but most organizations cannot achieve that overnight. For the near future, it is likely that both scenarios will co-exist in most organizations.

Metrics for Secure Coding

After a development team adopts a source code analysis tool and tunes it for the security policies that are important for their project, they can use source code analysis results in aggregate

for trending and project comparison purposes. Figure 3 shows a comparison between two projects, one red and one blue, where the source code analysis results have been grouped by severity. The graph suggests a plan of action: eliminate the critical issues for the red project, then move on to the high-importance issues for the blue project.

It can also be useful to look at the types of issues found broken down by category. Figure 4 shows the results for the same two projects in this fashion. Here, the differences between the red and the blue project become pronounced: the blue project has a significant number of buffer overflow issues. A strategy for preventing buffer overflow is in order.

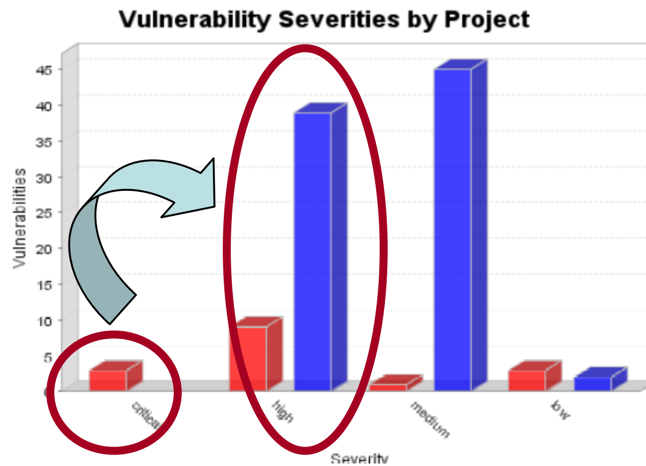

Figure 3: Source code analysis results broken down by severity for two projects.

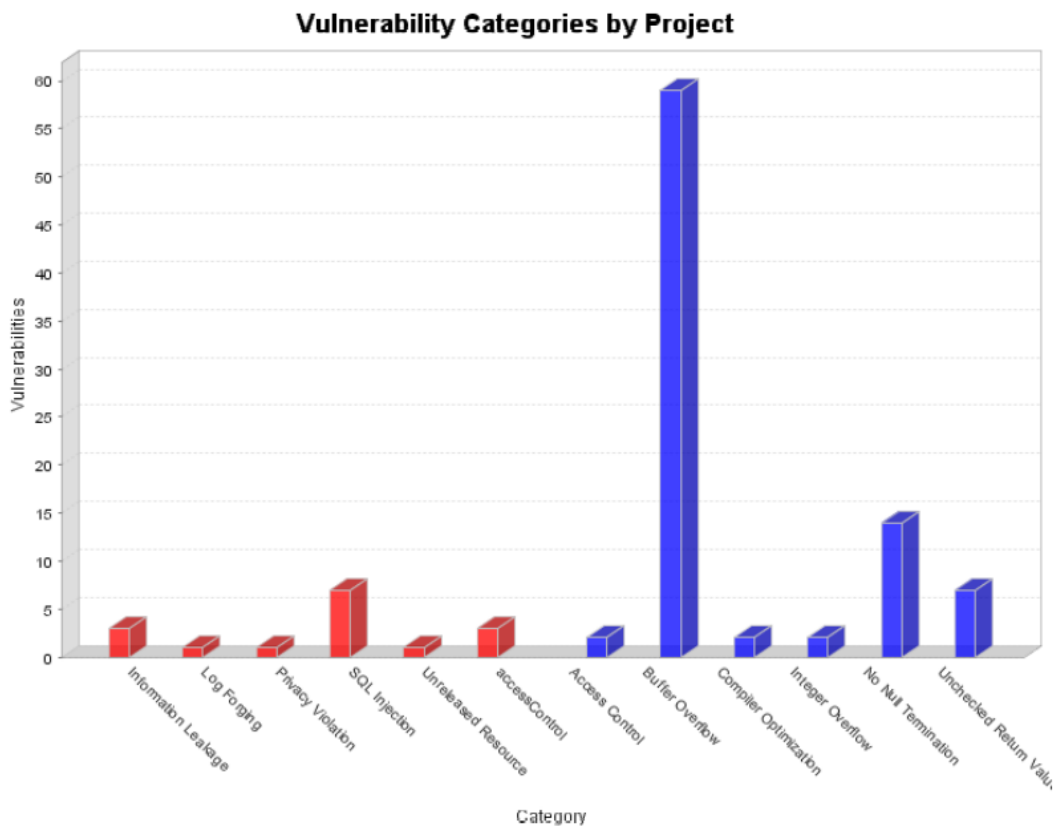

Figure 4: Source code analysis issues organized by vulnerability type.

Source code analysis results can also be used to examine vulnerability trends. Teams that are focused on security will decrease the number of source code analysis findings over time as they increasingly use the automation to mitigate security problems. A sharp increase in the number of issues found is likely to indicate a new security concern. Figure 5 shows the number of issues found during each nightly build. Trend indicators show how the project is evolving. In this case, the spike in the number of issues found is a result of the development group taking over a module from a group that has not been focused on security. This code represents a risk that will need mitigation throughout the remaining portion of the development life cycle.

Build Date (Y-M-D) & Time	Policy Violations	Violation Change from Previous	Vulnerabilities	Vulnerability Change from Previous
2005-01-07 @ 08:21:55	2	0%	16	-20%
2005-01-06 @ 08:21:55	2	0%	20	-16.7%
2005-01-05 @ 08:21:55	2	0%	24	0%
2005-01-05 @ 08:21:55	2	-33.3%	24	-7.7%
2005-01-04 @ 08:21:55	3	+50%	26	+85.7%
2005-01-03 @ 08:15:08	2	0%	14	-6.7%
2005-01-02 @ 08:15:08	2	0%	[illegible]	-11.8%
2005-01-01 @ 08:15:08	2	0%	17	-5.6%
2004-12-30 @ 08:15:08	2	0%	18	-18.2%
2004-12-29 @ 00:51:05	2	0%	22	0%
2004-12-29 @ 00:51:05	2	-33.3%	22	-8.3%

Figure 5: Source code analysis results over time.

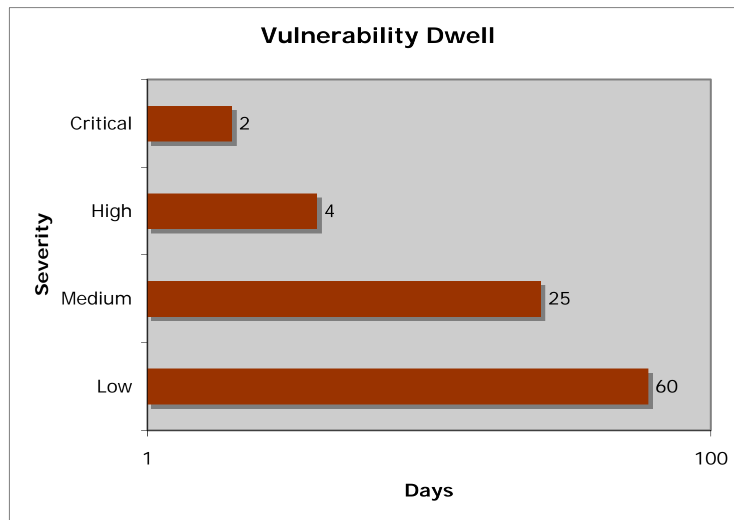

Figure 6: Vulnerability dwell as a function of priority.

Metrics for Legacy codebases

For large codebases where security has not historically been a priority, the security challenge has a different flavor. In most cases, it is not possible to instantaneously remodel the entire codebase for security purposes. Instead, an audit team needs to prioritize the problems and work to remove the worst ones. Of course, new development will continue even as the triage takes place.

Metrics for legacy codebases leverage the ability of the source code analyzer to give the same issue the same name across different builds. By following the same issue over time and associating it with the feedback provided by a human auditor, the source code analyzer can provide insight into the evolution of the project.

For example, the source code analysis results can reveal the way a development team responds to security vulnerabilities. After an auditor identifies a vulnerability, how long on average does it take for the developers to make a fix? This metric is named "Vulnerability Dwell." Figure 6 shows a project where the developers fix critical vulnerabilities within two days and take progressively longer to address less severe problems.

Because a legacy codebase often continues to evolve, auditors will need to return to the same projects again and again over time. But how often? Every month? Every six months? The

rate of auditing should keep pace with the rate of development, or rather the rate at which potential security issues are introduced into the code. By tracking individual issues over time, the output from a source code analysis tool can show an audit team how many unaudited issues a project contains. Figure 7 presents a typical graph. At the point the project is first audited, audit coverage goes to 100%. Then, as the code evolves over time, the audit coverage decays until the project is audited again.

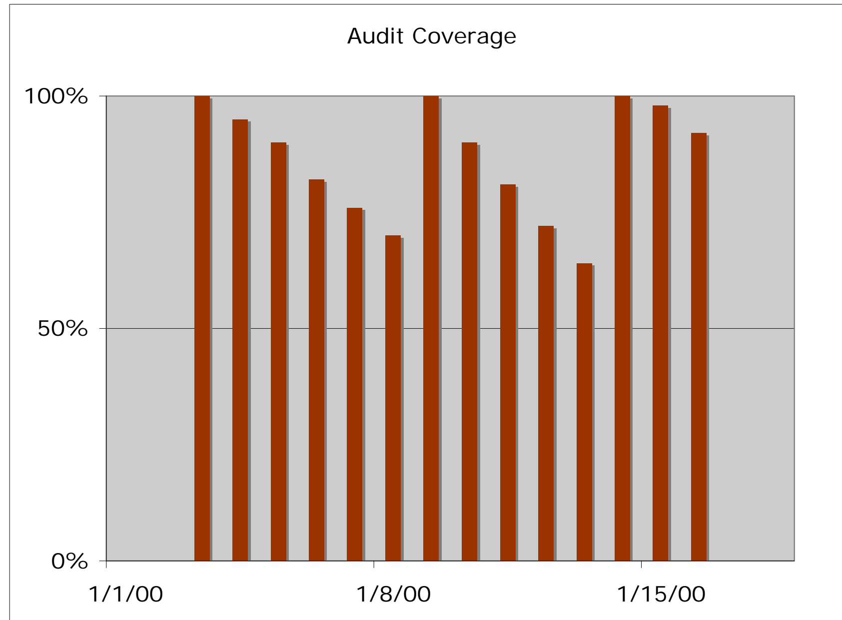

Figure 7: Audit coverage over time.

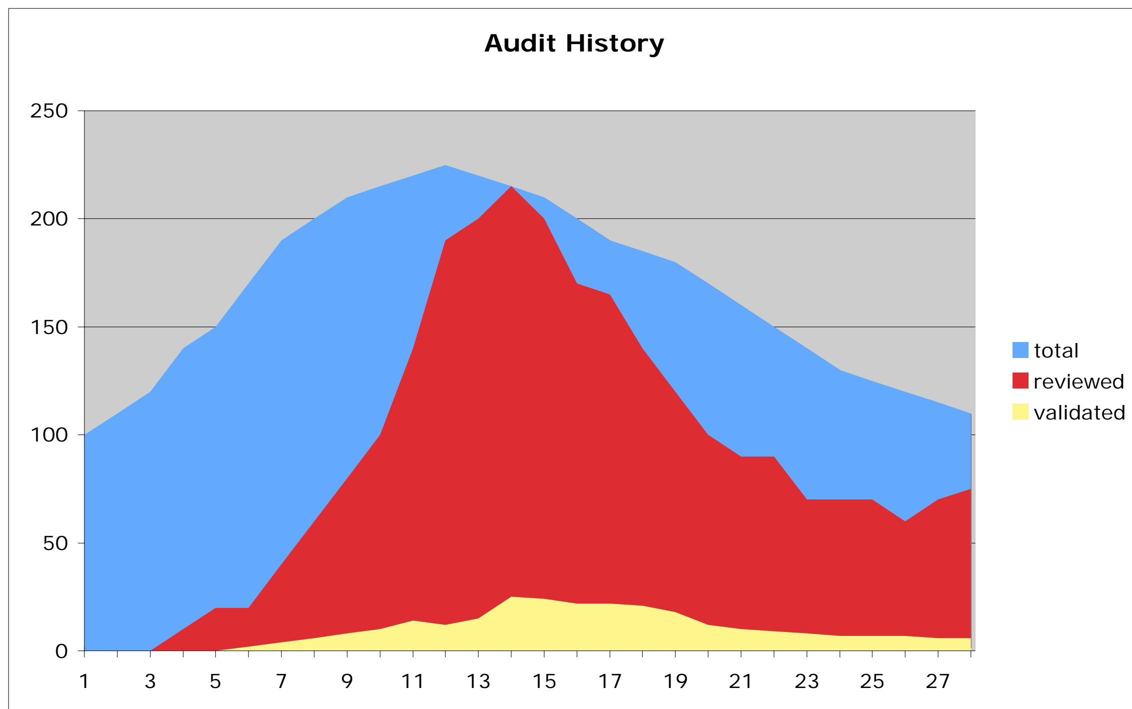

Figure 8: Audit history.

Another view of this same data gives a more comprehensive view of the project. An audit history shows the total number of issues, number of issues reviewed, and number of vulnerabilities identified as a function of time. This view takes into account not just the work of the auditors, but the effect the developers have on the project, too. Figure 8 shows an audit (shown in red) conducted over several product builds. At the same time the audit is taking place, the number of issues in the codebase (shown in blue) is growing. As the auditors work, they report vulnerabilities (shown in yellow). When the blue and red meet, the auditors have looked at all of the issues. Development work is not yet complete though, and soon the project once again contains unaudited issues. As the developers respond to some of the vulnerabilities identified by the audit team, the number of issues begins to decrease and some of the identified vulnerabilities are fixed. At the far right side of the graph, the uptick in the red indicates that another audit is beginning.

7. Conclusion

While software security has been a universally recognized risk, there has been an absence of established procedures for quantifying the security risk present software. Only by measuring can organizations conquer the software security problem.

The first step in this journey is the adoption of security-focused activities and deliverables throughout each phase of the software development process. These activities and deliverables include risk analysis during software design, code review during development, and security-oriented testing that targets the risks that are specific to the application at hand. By tracking and measuring the security activities adopted into the development process, an organization can begin to quantify their software security risk.

The data produced by source code analysis tools can be particularly useful for this purpose, giving insight into whether or not code review is taking place and whether or not the results of the review are being acted upon.

8. REFERENCES

[1] G. McGraw. *Software Security: Building Security In*. Addison-Wesley, to appear in 2006.

[2] G. McGraw et al. Building Security In. *IEEE Security and Privacy Magazine*, 2004-2005.

[3] C. E. Landwehr, A. R. Bull, J. P. McDermott, W. S. Choi. A Taxonomy of Computer Program Security Flaws, with Examples. *ACM Computing Surveys*, Vol. 26, No. 3, September 1994, pp. 211-254.

[4] S. Lipner and M. Howard. The Trustworthy Computing Security Development Lifecycle. In *Proceedings of the 20th Annual Computer Security Applications Conference* (ACSAC'04), 2004, pp. 2-13.