State of Application Assessment

Many companies have started building a code review capacity into their existing software security groups. Although they see value in skilled security professionals looking at source code, they still struggle with problems related to quick turnaround and consistent depth of quality.

To address this challenge, some organizations purchase static analysis tools as a cost-effective means of scaling some portion of the code review’s visibility. However, after they complete their static analysis tool pilot projects, they must then grapple with a broader deployment model and customization. The ultimate goal is to rapidly distribute these tools and get high-confidence feedback (in the form of code review and static analysis findings and recommendations) into the hands of developers while still retaining a deeper centralized scanning capability.

An additional complication is that organizations often attempt to automate governance of their corporate standards via static analysis tools—for example, many organizations use static analysis to check for approved cryptographic cipher use or for unauthorized storage of particular customer-specific sensitive information. Because corporate security often writes standards that focus on the use of sanctioned APIs, the correct and safe use of organizational libraries, and specific container or framework configurations, static analysis tools’ customization interfaces do a good job of facilitating or even automating such checks. Early-adopter organizations have been able to contract or build internal expertise to handle such customizations, but smaller, later adopters don’t possess the staff or capability to accomplish this task in house.

A Dirty Little Secret

The recent announcement on the Spring Framework’s vulnerability (www.springsource.com/security/advisory) underscores static analysis’s dirty secret: without special engineering, tools can’t follow the flow of control or data when it’s not explicit in the code. Frameworks such as Struts, Spring, Hibernate, and others use configuration to dynamically link and dispatch functionality or grant access to data, which might trip up or outright allude a tool’s detection capabilities. As adoption of these frameworks becomes more prolific, organizations might find that their result accuracy suffers. For Web 2.0, mashups, and other technologies, don’t even ask—the news is much worse.

Both rule-pack updates from tool vendors and organizational customization efforts increase an organization’s ability to produce findings, but changes in the programming paradigm, nonexplicit flow, and new frameworks decrease its likelihood of doing so. This phenomenon also impacts an organization’s ability to collect reliable measurement about whether its software possesses more or less vulnerability than it did previously.

These and other issues impose so much effort that existing IT security budgets can’t bear the cost of scaling software security’s “white box” assessment techniques (such as code review) across the entirety of the software portfolio developed in house, let alone to the software outsourced or purchased and integrated. The mounting costs of code review, paired with existing penetration testing tools and activities, have pushed organizations to reduce cost as much as possible. Specifically, organizations now ask whether they can reduce the “cost per finding” inherent in identifying security vulnerabilities by reducing overlap in detection techniques and relying more heavily on automation. The fact that security management can’t explain to management their expenditures in terms of cost per finding (let alone cost to remediation) and can’t draw a clear relationship between cost and a particular level of assurance against vulnerability plays an even larger role in recent software security budget reductions than market effects.

While grappling with this issue, other questions have emerged:

- What will penetration testing find?
- What will static analysis find?
- What will I find if I manually review code after using a tool?
When will I reach a diminished return from manual review?

Without answers, I predict broader IT management will increasingly lack confidence in application security. They’ve already begun to reach down into their application security departments and twiddle the knobs and dials themselves, which has led to consolidation, outsourcing, regression through reassignment of software security budgets to reactive audit approaches, and greater reliance on superficial (but extremely automated) tools. But to be clear, automation isn’t the enemy here—automation of the wrong abstractions is.

**Philosophy**

To address these newly emerging questions, organizations will have to let go of their current micro-optimizing tactics and revisit the assumptions they used to build the model in the first place. It’s important to distinguish between code review and application assessment, for example—whereas code review requires a static analysis tool and then demands a human reviewer, application assessment simply seeks to uncover the greatest number of potential security vulnerabilities through whatever means prove most efficient. These means could involve customizing a static analysis tool, combining open source tools and handcrafted scripts, selecting dynamic analyses, and even facilitating a targeted manual review. Ideally, an organization’s application assessment must be:

- quick (taking fewer than five days to conduct),
- consistent (high confidence that any reviewer will produce the same results),
- self-evolving (staffing includes cycles for improved automation and accuracy),
- accurate (matches findings from existing industry or internal reviews), and
- targeted (identifies high-value vulnerabilities).

Some organizations have in fact started to consider their application assessment strategies in this manner. Those with existing code review and penetration testing practices will immediately find inefficiencies that give them immediate cost savings, and those just beginning the journey can benefit by installing lean programs from the start. However, reducing application assessment costs rests on three pivotal assumptions:

- Although reviewed, low-hanging fruit (vulnerabilities) will remain in an application.
- Detection of most previously uncovered findings can be automated.
- Similar application implementations allow for leverage during review.

In my experience, these assumptions have held without exception. The first means that improvement doesn’t have to come at the hands of increased depth of analysis. This is important—it means improvement can focus on increasing coverage and decreasing the effort of identifying each finding rather than inventing new or difficult techniques for uncovering problems. Second, most organizations have found that they can automate most of what they previously found, regardless of what manual technique they used to initially identify the finding. This offers another opportunity to make an appreciable cost cut. Third, most organizations (with the exception of some very large software vendors) build very similar applications—for example, most financial, banking, and insurance companies predominantly use three-tier Java EE or .NET applications in front of middleware that connects to legacy mainframe systems. In contrast, gaming, telecom, and logistics companies usually build custom application platforms on top of languages such as C/C++ in a Unix environment.

Because of architecture similarities, we can make important generalizations about large swaths of an organization’s portfolio of...
Improvement can focus on increasing coverage and decreasing the effort of identifying each finding rather than inventing new techniques for uncovering problems.

sets might differ, but rest assured they all live in accessible databases. This and other similarities have important implications for manual and automated code review: a maturing organization can instruct manual code reviewers to consider whether particular security principals (such as encapsulation, enforcement of authorization decisions, and validation of untrusted input) hold within the architecture in which we expect to find them. For instance, when reviewing a Java EE application, we seldom trust input from Web-based clients. We also know that Web input will come in the form of HTTP headers, URLs, and form data. This narrowly targets the places where security reviewers must search to determine whether development correctly applied input validation. Similar gains appear in automation as well: organizations can develop custom rules within static analysis tools that codify important elements of the architecture outside the scanned code. For instance, these rules can indicate that, for a subset of URLs, a proxy server will demand authentication prior to allowing access to the application code.

Approach
To arrive at a quick, consistent, accurate, self-evolving, and targeted application assessment process, an organization must index its tool sets’ and reviewers’ findings. As it indexes these findings, it should unify assessment tools and activities under a common assessment process step and a single technology framework. Finally, the organization must facilitate both automating the detection of some findings and facilitating the detection of others. This will mean incorporating a variety of tools from commercial, open source, and in-house sources.

Index Findings
Indexing a review’s findings will provide essential data for planning. Reviewers always find new types of vulnerabilities within their organization’s software, and business logic is always ripe for application- and context-specific vulnerability, but most vulnerabilities in today’s Web applications are common to Web frameworks and their use. For organizations without vulnerability data, the OWASP top 10 (www.owasp.org/index.php/Top_10_2007) provides a starting point, but it’s designed to increase security awareness, not serve as a benchmark. From my experience with analyzing select sets of client data from financial organizations, I see the overwhelming majority of vulnerabilities centered around failure to demand authentication (or in more mature organizations, failure to conduct an authorization check after authentication), failure to validate untrusted input, and failure to apply comprehensive checks. Knowing that finding these (or other) vulnerabilities automatically would cover 60 to 80 percent of what both code reviewers and penetration testers uncover gives automation efforts a very tight focus that dramatically decreases manual effort.

When indexing findings, organizations should annotate each with the tools and techniques required to find them. This will help prioritize which tools were most successful in uncovering specific vulnerabilities. Almost immediately, the need to normalize these findings will become abundantly clear—especially those stemming from manual code review efforts. Using taxonomies such as Mitre’s Common Weakness Enumeration (http://cwe.mitre.org) will help organizations that don’t have their own internal classifications, but regardless of whether findings are normalized to an internal or industry standard, they should be cross-referenced with corporate security standards where applicable. This helps increase developer awareness of these resources, the state of their code’s compliance, and the motivating reasons why compliance is important.

Set Up a Tool Chain
Having indexed their findings, organizations will quickly realize that certain techniques more readily identify certain classes of vulnerability. The next step is to bring the various vulnerability detection tools and techniques under a single umbrella so that the security group, as a whole, uses only the most effective ones in each case.

If an organizational structure prevents a single group from being responsible for both penetration testing and code review because of reporting, skills, or other circumstances, separating penetration testing from code review isn’t a non-starter. Separate groups simply demand greater coordination, especially for self-evolution.

Organizations setting up a common tool chain have seen tremendous benefit from building a portal through which their development teams can submit applications for assessment. Getting the application to work with commonly available static analysis and penetration testing tools takes, on average, eight to 24 hours to pro-
duce first time, portals should solve the following problems:

• Question submitters about the submission’s key design or structural elements.
• Validate the submission’s inventory (source, configuration, and environment).
• Resolve forgotten dependencies (libraries, open source, and so forth).
• Retain submission versioning and version control.
• Support self-service for business units.

This last point is important. Organizations have long struggled with whether to deploy tools within business units (in a distributed fashion) or to keep them under the security group’s central control and have teams submit applications for review. Each approach has its advantages, but organizations have increasingly benefited from central-control-only deployment with highly accurate, immediate turnaround self-service that development teams can use as often as they want. Although it isn’t as thorough, a development team can integrate self-service usage of a centrally deployed tool (often through a portal or Web service) with continuous integration tools, such as cruise control. Organizations that opt for this combination do so without giving up on a more thorough, manually intensive tool used centrally as well.

The tool chain can also help automate presentation, reporting, and findings retention. Some organizations have integrated application assessment tools with their business units’ bug tracking or change management software—where successful, this can shave another eight to 16 hours of effort off the average application assessment. Thus, by focusing on submission reception and finding reporting, an organization can take 16 to 40 hours from an assessment effort. The time saved from avoiding these menial tasks can give reviewers time to scrutinize code more deeply or help trim the overall cost of review.

Incorporate Tools

When incorporating tools, organizations shouldn’t be afraid to combine the tools they’ve purchased with other freely available or homegrown tools. Centrally managing these tools makes this prospect tractable and avoids the nightmare of maintaining a bevy of tools and configurations distributed to every business unit.

Merging results is the principal issue associated with integrating multiple tools. Each tool uses a different presentation format, and they all report results inconsistently, relative to each other. However, the ability to corroborate results between static analysis tools or to confirm static results with penetration tests can outweigh the effort of normalizing and merging results. Although it’s valuable to drive a dynamic testing effort from static analysis results, I haven’t experienced many organizations doing this. However, commercial tool vendors from both camps have started to hybridez their approach and tools.

Automation vs. Facilitation

Once they’ve begun an automation push, organizations commonly make the mistake of believing that they must now automate every vulnerability’s detection. This is folly. You simply can’t detect many vulnerabilities—even low-hanging fruit—without some human interaction (unless you’re willing to give up a bunch of accuracy or depth). Instead, organizations should consider what they can easily automate and where some amount of reviewer input can nudge automation along.

Input can come in the form of reviewer hints (such as what effective input validation routines are called or what objects do or don’t replicate sensitive data from a database); hints can also come from the submitter. This could also reduce false positives—for example, by indicating that a proxy server demands authentication prior to dispatching application functionality.

Tools can provide assessment benefits even when there’s no hope of uncovering a finding, hinted or otherwise. For instance, static analysis tools can quickly scan a code base of hundreds of thousands of lines of code and determine where any interesting system calls will take place. Likewise, scripts built from freely available development tools determine what code has changed since the last submission.

Correlating these data provides reviewers with a “heat map” of the code base that helps them prioritize where to find something sharp in the haystack. Even if tooling picks out every module that appears to be conducting session management, organizations can rest more assured that their reviewers have considered a code base’s important aspects rather than having randomly poured through its entirety in an ad hoc fashion.

Software security initiatives must begin to manage themselves differently and, consequently, sell themselves differently. By producing the cost-of-finding measurement by vulnerability and showing innovation in decreasing their numbers, security initiatives will drive executive management to increase the security budget—rather than slash it—despite these troubled economic times.

John Steven is a senior director at Cigital. His research interests include architectural analysis, static analysis, and metrics around software assessment. Steven has an MS in computer science from Case University. Contact him at jsteven@cigital.com.