Security Compliance as an Engineering Discipline

By Brad Hill | February 2010

As a result of new initiatives and requirements like the Payment Card Industry Data Security Standard (PCI-DSS), many organizations are building comprehensive application security programs for the first time. To do so, a number of those concerns are looking to the proven success of the Microsoft Security Development Lifecycle (SDL). This can be a very smart business move, but it's important to understand how the engineering focus of the SDL makes it different from the typical security-compliance effort. This month I'm going to address some of the ways to harmonize compliance-focused programs with security engineering to improve your software development practices.

By integrating secure engineering practices into the entire software lifecycle, the SDL lets you achieve application security assurance using a holistic approach. The process requires direct engagement with software developers and testers, and unlike the typical compliance program, it's incremental, iterative and should be customized to every organization.

In this article, I'll focus on some of the strategies and best practices for deploying SDL and integrating it with security compliance regimes.

Compliance and the SDL

Building a software security program from scratch can seem daunting. Few organizations have a program or team dedicated specifically to secure application development. The portfolio of the typical security group includes networks and firewalls, policy and identity management, systems configuration, and general operations and monitoring. Securing the ever-growing set of custom-built, mission-critical applications is a new and challenging addition to these responsibilities.

While a mature software security practice will facilitate compliance efforts like PCI and the Common Criteria for Information Technology Security Evaluation, looking at the SDL as just another compliance effort will keep you from getting the best results from your program. Unlike compliance programs, which have externally defined goals and activities, the SDL recognizes that the business goals and abilities of attackers are constantly

changing. Many organizations achieve compliance through large up-front efforts followed by minimal maintenance. In contrast, the SDL starts incrementally and focuses on using constant effort for continuous improvement. In addition, compliance often encourages separation of duties between implementation and enforcement. The SDL is built on close coordination with developers and testers. Given these differences, if your experience is only in managing compliance efforts, how should you approach an SDL implementation?

First, consider how your organization has managed other major software industry changes in the past decade, such as moving from procedural to object-oriented development, client-server to Web-based software, or waterfall to agile development methods. These changes weren't made across the entire organization with the stroke of a pen. They were introduced incrementally and grew organically. A similar approach, as outlined below, will work best for the SDL.

- Start small with one or two pilot projects.
- Even though you'll introduce the new way of doing things with a capable team, bring in outside help where needed.
- Study and learn from the results at the completion of a project. Adapt the standard practices to fit your particular organization. Get rid of the things that didn't work.
- Let coverage expand organically, nurturing internal experts and spreading the culture change at a pace the organization can manage without disrupting the business.

Second, recognize that being in compliance is just one goal for your program—the first step, not the finish line. More-secure and higher-quality software provides its own business and competitive value. Measure your program by how successful you ultimately are at delivering more trustworthy products and what that means for your bottom line.

Adding Value with Secure Engineering

Often a team needs help optimizing its software security program because the project is in the early stages and the implementers require guidance and encouragement. But sometimes even established programs somehow fail to gain traction. Although the organization may have recognized the

value of security and implemented a wide variety of activities and best practices—training, design review, static analysis, dynamic scanning, code review and manual penetration testing—the results end up reflecting poorly on the large investment made.

The compliance-oriented approaches of traditional IT security frequently bring with them biases that prevent these programs from being truly successful in changing software quality. By reorienting programs to avoid some common compliance pitfalls and by including the development organization, thus integrating engineering approaches into compliance efforts, organizations can often identify simple changes that help their security effort reach its full potential.

In the following sections I'll highlight four of the most common pitfalls and explain how to avoid or address them.

Don't Do Everything—Do a Few Things Well

Compliance efforts and their results tend to be well-defined. Individual requirements are compartmentalized and have standard measures, and overall compliance status is a matter of having enough of the right boxes checked. From the abstract goal of becoming compliant, the correct management strategy is to use the available resources to cover as many areas as possible, and to spend as little as necessary to become compliant with any individual requirement.

Quality software engineering is much less amenable to this sort of measurement, except at the very finest granularity. Spreading resources too thin is a common pitfall. What's worth doing is worth doing well. A cursory or unskilled penetration test may give a false sense of security—overconfidence in the capabilities of a code-scanning tool may lead to under-allocating resources to manual code review, and limited training in areas such as cryptography may give developers just enough knowledge to be dangerous. It's better to do a few things really well than everything poorly.

Consider one example from the code review process used in security effort my company consulted on. A security code review by a dedicated team was a mandatory part of the development process for all Internet-facing

applications—a rarely seen practice and capability that's usually indicative of a very strong program.

Code review by humans can be highly effective at finding difficult-to-spot and important classes of security flaws, but what did this organization's process look like? Because it was mandatory, the reviewers had a great deal of work to do and little time to complete it. Almost no time was allocated for the development team to collaborate in the process, so the code reviewers were working with little business context. Additionally, they had no access to previous bug reports from tools or manual testing, nor did they have access to a live instance of the system against which to prove their findings.

What were the results? Findings from code review were delivered to the development team mere days before scheduled go-live dates, and false-positive rates were as high as 50 percent. It was rare that these bugs could even be triaged, let alone fixed, during the same release cycle in which they were found. Furthermore, identification of logic flaws—perhaps the most important benefit of manual analysis—was unattainable due to the lack of business context and collaboration.

A better approach—one unlikely to be heard of often in the world of compliance—is do less, and do it less often. Instead of devoting two days each to code review and black-box penetration testing on 10 releases a quarter, eliminate code review as a distinct activity. Instead, perform eight days of source-code-informed, white-box penetration testing on five releases per quarter. The reduced number of targets and broader context would allow enough time to find more flaws and to actually fix them before release.

Target Projects, Not Policies

Generally, when a compliance program is undertaken, its requirements are mandated for the entire organization and activities often are structured to have few dependencies. This usually means that compliance efforts proceed horizontally, with new policies made effective for everyone in an organization simultaneously. While some SDL activities are amenable to this approach, it makes much more sense to pick a few teams or projects to target with a vertical rollout.

To understand this, consider an analogy to another methodology change that many software organizations have undertaken recently: the use of agile development methods. No company approaches getting all its teams on agile methods by eliminating formal specifications for all its projects in Q1, increasing iteration speeds in Q3, and starting to write unit tests in Q2 of the following year. The practices of agile development enhance and support each other and should be rolled out together, one team and one project at a time.

Approach the SDL similarly: Don't exhaust your initial hard-won resources by implementing a universal mandate for threat modeling without follow-up that uses the threat models, measures their success, and iteratively improves. No SDL activity reaches its potential in isolation from the others, and there's no surer path to a security initiative's failure than a stack of expensive security documentation with no actionable next steps.

This isn't to say you should do everything at once. Pick a few activities to start with, such as threat modeling, code review, and automated scanning and roll them out together for a limited set of projects. This approach allows you to gather data on how accurate and useful your code scanner was in reducing vulnerabilities. It also lets you see what kinds of bugs the scanner missed, but that manual testing was able to find or that threat modeling was able to prevent. This is important information that helps you fine-tune each process, understand the assurance levels you're achieving, and guide the effective allocation of future resources.

Always Measure Effectiveness

Too often when it comes to compliance, the thinking is, to paraphrase Tennyson, "Ours is not to reason why. Ours is but to do and die." If a requirement is on the list, you won't be compliant without it. With such an iron-clad mandate, all incentive to measure the benefit of the activity is lost. So, for example, while most organizations have password rotation policies, few quantify how much security benefit such guidelines provide. Not many concerns attempt a cost-benefit analysis to see if changing passwords on 15- or 45-day cycles would be better than 30, because there's no reason to question the standard.

Avoid making this mistake when quality is your real goal. Typical training regimens provide a great example of this error. Often the only two measurements we see for evaluating instructional programs are: did everyone get security training and did the training mention the major vulnerability classes? If your business goal is more-secure software, though, have those two hard-earned (and probably expensive) checked boxes actually improved your software quality?

Measuring effectiveness comes back to the core idea of relating software security directly to business goals. Before you start any activity, understand what you're trying to achieve and find a way to measure whether it works. Are developers and testers retaining what they learn? Can they apply it to the real problems they have to solve? Are teams that have been trained producing software with fewer bugs or eliminating bugs earlier in the lifecycle than teams that haven't been trained?

Metrics don't have to be expensive to collect—a simple survey might be enough to tune a training course—but they have to be there. Developers are savvy enough to tell apart meaningful activities tuned to produce results and hoops to be jumped through. The difference between successful and unsuccessful security programs is often one of mistaking the hoops for the results.

Setting principled goals, measuring the program's progress against those goals, and improving the process at each step will help motivate developers and change the internal culture of quality. Showing progress against meaningful metrics also provides a good justification for the effort and will support the procurement of additional resources required for the maintenance and growth of the program.

Just Because Everyone Else Is Doing It ...

Just as there are no silver bullets for the general software-development process, there can be none for developing secure software. Be wary of anyone telling you they have the one universal solution for all your security problems. Start small, adopt practices that are appropriate to your organization, and learn from the experiences of others with similar security needs and resources.

But how do you know where to start? The list of things to do seems huge.

A recent survey of the world's best software security programs observed that, essentially, all of them performed the same set of nine activities. At last: best practices! Shouldn't you set as your target the procedures the top 10 programs in the world agree on?

If you look at the world's top 10 navies, you'll find that they all deploy aircraft carriers and submarines, and their admirals would be quick to tell you that these technologies are the most effective way to project force at sea. Of course, as you keep looking, you'll find that nobody *but* the top 10 navies has submarines and aircraft carriers. Such fleets require vast expenditures, support structures and technical capabilities that are only within the reach of a major power. Many smaller countries would find little use for equivalent resources and might very well be bankrupted trying to acquire and maintain such systems.
Similarly, a small software-development organization may well bankrupt itself following practices and trying to manage tools whose purpose is scaling secure engineering to thousands of developers managing many millions of lines of code.

The big-ticket, silver-bullet solutions for security compliance tend to be better at killing budgets and credibility than bugs—unless an organization is well-prepared for the introduction of such products. The most successful organizations aren't those that check all the big boxes, but those that start small and that grow sustainable and manageable security programs closely related to the goals of the business.

Before you buy that expensive software package, build your expertise with the growing arsenal of free and built-in security tools available on the major enterprise software platforms. Get comfortable triaging and fixing all the bugs FxCop and C++ /analyze (or FindBugs if your organization uses Java) can turn up, and get your code to compile cleanly against the SDL banned API list before you spend a penny doing anything more complex.

Once you're used to managing and running cleanly with these tools, you're much more likely to succeed with something more comprehensive. You may find that the free tools are just what you need, or they may leave you eager for bigger and better. You haven't lost anything by starting with free.

Harmonizing Quality and Compliance

While it may be starting to sound like the SDL is antithetical to compliance, that's definitely not the case. The two target the same basic needs, and harmonizing and integrating them can produce significant cost savings. When extending a compliance program to improve quality through the SDL, you can avoid common pitfalls and maximize your success with the four strategies discussed.

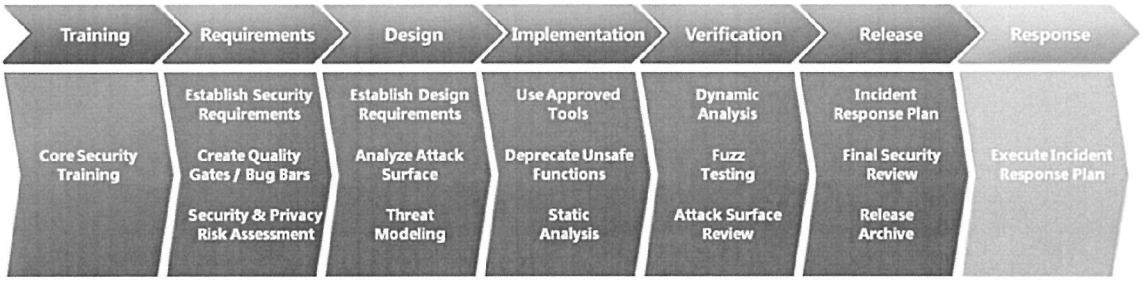
Don't try to do everything at once. Start with what you can do well. Increment your efforts project by project, not policy by policy. Always measure the effectiveness of your efforts for your own business bottom line. Grow into your program, don't buy into it.

Most importantly, get started now! There's no minimum increment for quality. Start by taking advantage of the free tools and guidance at the Microsoft SDL Web site. Download and try out the SDL Optimization Model to evaluate the current state of your organization's capabilities, and use the professional services and training available through the Microsoft SDL Pro Network to start improving quality and delivering more secure software while you meet your audit and compliance goals.

Announcing SDL for Agile Development Methodologies

By Bryan Sullivan | November 2009

Hi everyone, Bryan here. There is a common misconception that because the SDL was originally created for Microsoft's big showcase box products like Windows and SQL Server, that it only works for those kinds of products. This is of course patently false: virtually every Microsoft product and online service, large or small, follows the SDL. Many other organizations outside of Microsoft are also successfully implementing the SDL. However, while the content of the SDL – its requirements and recommendations – may be universal, the structure of the SDL as originally designed is more suited to long-running waterfall- or spiral-style development methodologies. Consider the classic "chevron" SDL graphic:



As you can see, the SDL prescribes certain activities to take place during certain phases of the development lifecycle – threat modeling for example happens during the Design phase, and static analysis is performed during the Implementation phase. But not every development methodology has well-defined lifecycle phases like this. Specifically, Agile development methodologies do not have distinct phases and instead follow an iterative, time-boxed approach. How can the SDL be applied successfully in these environments?

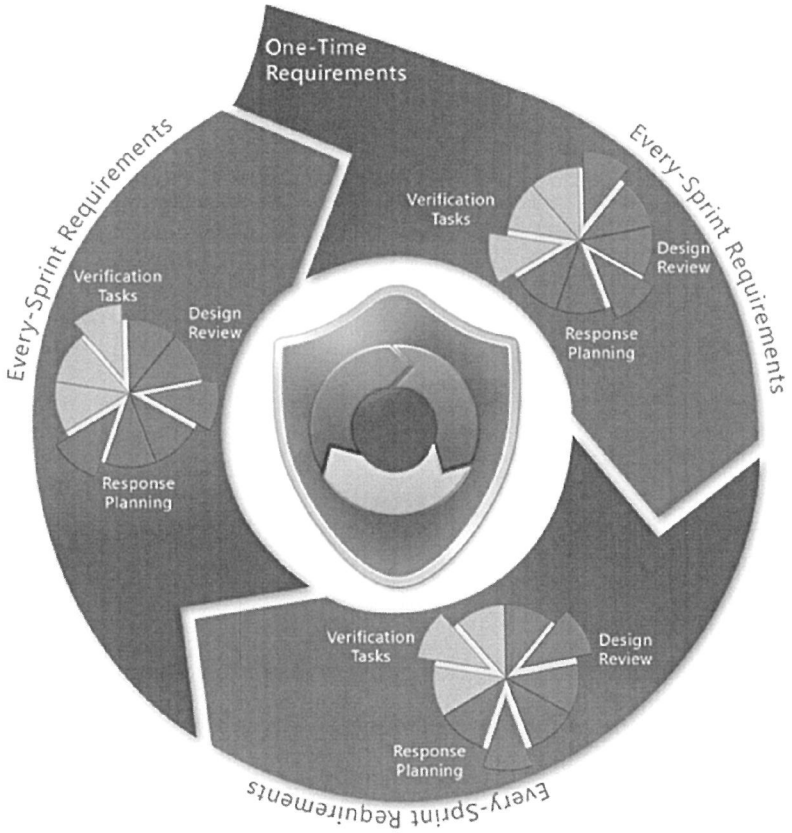
One solution might be to take all the SDL requirements and put them into the product backlog, then pull them into the active queue (aka the sprint backlog, if you're using Scrum) just like any other user story. This might work adequately for box products with well-defined product lifecycles that use Agile; for example, the Visual Studio teams that follow Scrum would fall into this category. However, the majority of internal teams (and very likely the majority of all development teams outside Microsoft too) that follow Agile use it to build web applications. This is important because web

applications often don't have a defined "end"; they just keep building and growing indefinitely. If we put the SDL requirements into the product backlog, it might take a year or more for a team to complete them all, but all the features added to the product after that date would go unsecured.

An alternative solution might be to just apply the entire SDL to every iteration. This would solve the problem of unsecured functionality being added after the SDL requirements have been completed, but it would create a whole new problem just as big, namely: how to complete all that SDL work in such a short amount of time! Per the Agile Manifesto, Agile projects should have short iterations, lasting from one month to a few weeks or less. There are online services teams here at Microsoft with one week long sprints. There's no way these teams could complete the entire SDL in a sprint that short. And even if they could, there would be no time left to actually develop new features.

Another alternative would be to pare back the SDL, to cut out the "unnecessary" SDL requirements and just complete a smaller, core subset of the SDL each iteration. Unfortunately, this approach is flawed too, because none of the SDL requirements are unnecessary. Every requirement has been proven to prevent vulnerabilities or to reduce the impact of a successful exploit. Leaving requirements out of Agile projects would jeopardize their security, and that's simply not an acceptable solution.

However, although none of these approaches solves the problem of adapting the SDL to Agile, that doesn't mean the task is impossible. Over the last year, a team of security professionals throughout the Trustworthy Computing Security and Online Services Security & Compliance teams (including myself and Michael Howard from SDL) have worked to find a solution to the problem. Our resulting process has been in internal beta since the spring, has just recently released internally, and now I'm happy to announce that we're releasing the details of the SDL for Agile Development Methodologies process today.



In brief, SDL-Agile breaks the SDL into three categories of requirements: every-sprint requirements, the requirements so important that they must be completed every iteration; one-time requirements, the requirements that only have to be completed once per project no matter how long it runs; and bucket requirements, the requirements that still need to be completed regularly but are not so important that they need to be completed every sprint.

Over and above the reorganization of requirements into a more Agile-friendly structure, SDL-Agile also provides guidance for adapting many of the core SDL activities to Agile. Threat modeling is a perfect example: a team could easily spend an entire week-long sprint performing threat modeling, but this may not be the best use of their time. SDL-Agile describes how a team can spend an appropriate amount of time modeling new features as well as how to build up a baseline of threat models for existing functionality.

Instead of getting into an in-depth discussion of SDL-Agile in the limited space I have here, I ask that you download and read the complete SDL-Agile guidance here, included as part of the SDL 4.1a Process Guidance document. We believe we've developed a process that is faithful to both Agile and to SDL, in which teams can innovate and react quickly to changing customer needs but in which the products they create are still more resilient to attack. As always, we welcome your feedback.

Software Penetration Testing

Brad Arkin
Symantec

Scott Stender
Information Security Partners

Gary McGraw
Cigital

Quality assurance and testing organizations are tasked with the broad objective of assuring that a software application fulfills its functional business requirements. Such testing most often involves running a series of dynamic functional tests to ensure proper implementation of the application's features. However, because security is not a feature or even a set of features, security testing doesn't directly fit into this paradigm.[1]

Security testing poses a unique problem. Most software security defects and vulnerabilities aren't related to security functionality—rather, they spring from an attacker's unexpected but intentional misuses of the application. If we characterize functional testing as testing for positives—verifying that a feature properly performs a specific task—then security testing is in some sense testing for negatives. The security tester must probe directly and deeply into security risks (possibly driven by abuse cases and architectural risks) to determine how the system behaves under attack.

One critical exception to this rule occurs when the tester must verify that security functionality works as specified—that the application not only doesn't do what it's not supposed to do, but that it does do what it's supposed to do (with regard to security features).

In any case, testing for a negative poses a much greater challenge than verifying a positive. Quality assurance people can usually create a set of plausible positive tests that yield a high degree of confidence a software component will perform functionally as desired. However, it's unreasonable to verify that a negative doesn't exist by merely enumerating actions with the intention to produce a fault, reporting if and under which circumstances the fault occurs. If "negative" tests don't uncover any faults, we've only proven that no faults occur under particular test conditions; by no means have we proven that no faults exist. When applied to security testing, where the lack of a security vulnerability is the negative we're interested in, this means that passing a software penetration test provides very little assurance that an application is immune to attack. One of the main problems with today's most common approaches to penetration testing is misunderstanding this subtle point.

Penetration testing today

Penetration testing is the most frequently and commonly applied of all software security best practices, in part because it's an attractive late lifecycle activity. Once an application is finished, its owners subject it to penetration testing as part of the final acceptance regimen. These days, security consultants typically perform assessments like this in a "time boxed" manner (expending only a small and predefined allotment of time and resources to the effort) as a final security checklist item at the end of the life cycle.

One major limitation of this approach is that it almost always represents a too little, too late attempt to tackle security at the end of the development cycle. As we've seen, software security is an emergent property of the system, and attaining it involves applying a series of best practices throughout the software development life cycle (SDLC; see Figure 1).[1] Organizations that fail to integrate security throughout the development process often find that their software suffers from systemic faults both at the design level and in the implementation (in other words, the system has both security flaws and security bugs). A late lifecycle penetration testing paradigm uncovers problems too late, at a point when both time and budget severely constrain the options for remedy. In fact, more often than not, fixing things at this stage is prohibitively expensive.

An ad hoc software penetration test's success depends on many factors, few of which lend themselves to metrics and standardization. The most obvious variables are tester skill, knowledge, and experience. Currently, software security assessments don't follow a standard process of any sort and therefore aren't particularly amenable to a consistent application of knowledge (think checklists and boilerplate techniques). The upshot is that only skilled and experienced testers can successfully perform penetration testing.

The use of security requirements, abuse cases, security risk knowledge, and attack patterns in application de-

sign, analysis, and testing is rare in current practice. As a result, security findings can't be repeated across different teams and vary widely depending on the tester. Furthermore, any test regimen can be structured in such a way as to influence the findings. If test parameters are determined by individuals motivated not to find any security issues (consciously or not), it's likely that the penetration testing will result in a self-congratulatory exercise in futility.

Results interpretation is also an issue. Typically, results take the form of a list of flaws, bugs, and vulnerabilities identified during penetration testing. Software development organizations tend to regard these results as complete bug reports—thorough lists of issues to address to secure the system. Unfortunately, this perception doesn't factor in the time-boxed nature of late lifecycle assessments. In practice, a penetration test can only identify a small representative sample of all possible security risks in a system. If a software development organization focuses solely on a small (and limited) list of issues, it ends up mitigating only a subset of the security risks present (and possibly not even those that present the greatest risk).

All of these issues pale in comparison to the fact that people often use penetration testing as an excuse to declare victory. When a penetration test concentrates on finding and removing a small handful of bugs (and does so successfully), everyone looks good: the testers look smart for finding a problem, the builders look benevolent for acquiescing to the test, and the executives can check off the security box and get on with making money. Unfortunately, penetration testing done without any basis in security risk analysis leads to this situation with alarming frequency. By analogy, imagine declaring testing victory by finding and removing only the first one or two bugs encountered during system testing!

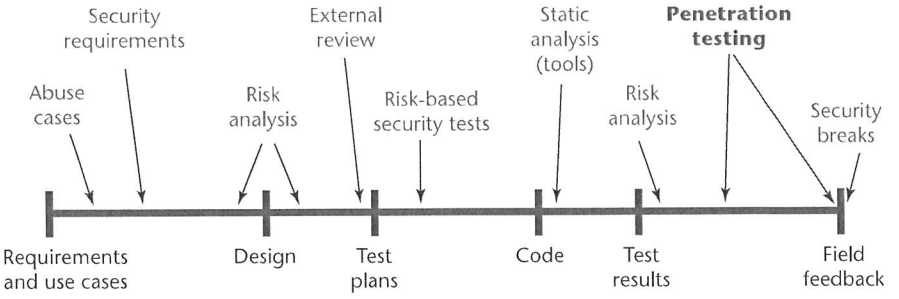


Figure 1. The software development life cycle. Throughout this series, we'll focus on specific parts of the cycle; here, we're examining penetration testing.

A better approach

All is not lost—security penetration testing can be effective, as long as we base the testing activities on the security findings discovered and tracked from the beginning of the software lifecycle, during requirements analysis, architectural risk analysis, and so on. To do this, a penetration test must be structured according to perceived risk and offer some kind of metric relating risk measurement to the software's security posture at the time of the test. Results are less likely to be misconstrued and used to declare pretend security victory if they're related to business impact through proper risk management.

Make use of tools

Tools should definitely be part of penetration testing. Static analysis tools can vet software code, either in source or binary form, in an attempt to identify common implementation-level bugs such as buffer overflows.[2] Dynamic analysis tools can observe a system as it executes as well as submit malformed, malicious, and random data to a system's entry points in an attempt to uncover faults—a process commonly referred to as fuzzing. The tool then reports the faults to the tester for further analysis.[3] When possible, use of these tools should be guided by risk analysis results and attack patterns.

Tools offer two major benefits. First, when used effectively, they can perform most of the grunt work needed for basic software security analysis. Of course, a tool-driven approach can't be used as a replacement for review by a skilled security analyst (especially because today's tools aren't applicable at the design level), but such an approach does help relieve a reviewer's work burden and can thus drive down cost. Second, tool output lends itself readily to metrics, which software development teams can use to track progress over time. The simple metrics commonly used today don't offer a complete picture of a system's security posture, though, so it's important to emphasize that a clean bill of health from an analysis tool doesn't mean that a system is defect free. The value lies in relative comparison: if the current run of the tools reveals fewer defects than a previous run, we've likely made some progress.

Test more than once

Today, automated review is best suited to identifying the most basic of implementation flaws. Human review is necessary to reveal flaws in the design or more complicated implementation-level vulnerabilities (of the sort that attackers can and will exploit), but such review is costly. By leveraging the basic SDLC touchpoints described in this series of articles, penetration tests can be structured in such a way as to be cost effective and give a reason-

able estimation of the system's security posture.

Penetration testing should start at the feature, component, or unit level, prior to system integration. Risk analysis performed during the design phase should identify and rank risks as well as address intercomponent assumptions.[4,5] At the component level, risks to the component's assets must be mitigated within the bounds of contextual assumptions. Tests should attempt unauthorized misuse of, and access to, target assets as well as try to violate any assumptions the system might make relative to its components.

Testers should use static and dynamic analysis tools uniformly at the component level. In most cases, no customization of basic static analysis tools is necessary for component-level tests, but a dynamic analysis tool will likely need to be written or modified for the target component. Such tools are often data-driven tests that operate at the API level. Any tool should include data sets known to cause problems, such as long strings and control characters.[6] Furthermore, the tool design should reflect the security test's goal: to misuse the component's assets, violate intercomponent assumptions, or probe risks.

Unit testing carries the benefit of breaking system security down into several discrete parts. Theoretically, if each component is implemented safely and fulfills intercomponent design criteria, the greater system should be in reasonable shape (although this problem is much harder than it seems at first blush[7]). By identifying and leveraging security goals during unit testing, we can significantly improve the greater system's security posture.

Penetration testing can be effective, as long as we base the testing activities on the security findings discovered and tracked from the beginning of the software lifecycle.

Penetration testing should continue at the system level and be directed at the integrated software system's properties such as global error handling, intercomponent communication, and so on. Assuming unit testing has successfully achieved its goals, system-level testing shifts its focus toward identifying intercomponent issues and assessing the security risk inherent at the design level. If, for example, a component assumes that only trusted components have access to its assets, security testers should structure a test to attempt direct access to that component from elsewhere. A successful test can undermine the system's assumptions and could result in an observable security compromise. Dataflow diagrams, models, and high-level intercomponent documentation created during the risk analysis stage can also be a great help in identifying where component seams exist.

Tool-based testing techniques are appropriate and encouraged at the system level, but for efficiency's sake, such testing should be structured to avoid repeating unit-level testing. Accordingly, they should focus on aspects of the system that couldn't be probed during unit testing.

If appropriate, system-level tests should analyze the system in its deployed environment. Such analysis should be targeted to ensure that suggested deployment practices are effective and reasonable and that external assumptions can't be violated.

Integrate with the development cycle

Perhaps the most common problem with the software penetration testing process is the failure to identify lessons to be learned and propagated back into the organization. As we mentioned earlier, it's tempting to view a penetration test's results as a complete and final list of bugs to be fixed rather than as a representative sample of faults in the system.

Mitigation strategy is thus a critical aspect of the penetration test. Rather than simply fixing identified bugs, developers should perform a root-cause analysis of the identified vulnerabilities. If most vulnerabilities are buffer overflows, for example, the development organization should determine just how these bugs made it into the code base. In such a scenario, lack of developer training, misapplication (or nonexistence of) standard coding practices, poor choice of languages and libraries, intense schedule pressure, or any combination thereof could ultimately represent an important cause.

Once a root cause is identified, developers and architects should devise mitigation strategies to address the identified vulnerabilities and any similar vulnerability in the code base. In fact, best practices should be developed and implemented to address such vulnerabilities proactively in the future. Going back to the buffer overflow example, an organization could decide to train its developers and eliminate the use of potentially dangerous functions such as `strcpy()` in favor of safer string-handling libraries.

A good last step is to use test result information to measure progress against a goal. Where possible, tests for the mitigated vulnerability should be added to automated test suites. If the vulnerability resurfaces

in the code base at some point in the future, any measures taken to prevent the vulnerability should be revisited and improved. As time passes, iterative security penetration tests should reveal fewer and less severe flaws in the system. If a penetration test reveals serious severity flaws, the "representative sample" view of the results should give the development organization serious reservations about deploying the system.

Penetration testing is the most commonly applied mechanism used to gauge software security, but it's also the most commonly misapplied mechanism as well. By applying penetration testing at the unit and system level, driving test creation from risk analysis, and incorporating the results back into an organization's SDLC, an organization can avoid many common pitfalls. As a measurement tool, penetration testing is most powerful when fully integrated into the development process in such a way that findings can help improve design, implementation, and deployment practices. □

References

1. G. McGraw, "Software Security," *IEEE Security & Privacy*, vol. 2, no. 2, 2004, pp. 80–83.
2. B. Chess and G. McGraw, "Static Analysis for Security," *IEEE Security & Privacy*, vol. 2, no. 6, 2004, pp. 76–79.
3. B.P. Miller et al., *Fuzz Revisited: A Re-Examination of the Reliability of Unix Utilities and Services*, tech. report CS-TR-95-1268, Department of Computer Science, Univ. Wisconsin, Apr. 1995.
4. D. Verndon and G. McGraw, "Software Risk Analysis," *IEEE Security & Privacy*, vol. 2, no. 5, 2004, pp. 81–85.
5. F. Swidersky and W. Snyder, *Threat Modeling*, Microsoft Press, 2004.
6. G. Hoglund and G. McGraw, *Exploiting Software*, Addison-Wesley, 2004.
7. R. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems*, John Wiley & Sons, 2001.

Brad Arkin *is a technical manager for Symantec Professional Services. His primary area of expertise is helping organizations improve the security of their applications. Arkin has a dual BS in computer science and mathematics from the College of William and Mary, an MS in computer science from George Washington University, and is an MBA candidate at Columbia University and London Business School. Contact him at barkin@atstake.com.*

Scott Stender *is a partner with Information Security Partners. His research interests are focused on software security, with an emphasis on software engineering and security analysis methodology. Stender has a BS in computer engineering from the University of Notre Dame. Contact him at scott@isecpartners.com.*

Gary McGraw *is chief technology officer of Cigital. His real-world experience is grounded in years of consulting with major corporations and software producers. McGraw is the coauthor of* Exploiting Software *(Addison-Wesley, 2004),* Building Secure Software *(Addison-Wesley, 2001),* Java Security *(John Wiley & Sons, 1996), and four other books. He has a BA in philosophy from the University of Virginia and a dual PhD in computer science and cognitive science from Indiana University. Contact him at gem@cigital.com.*