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 design, 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!

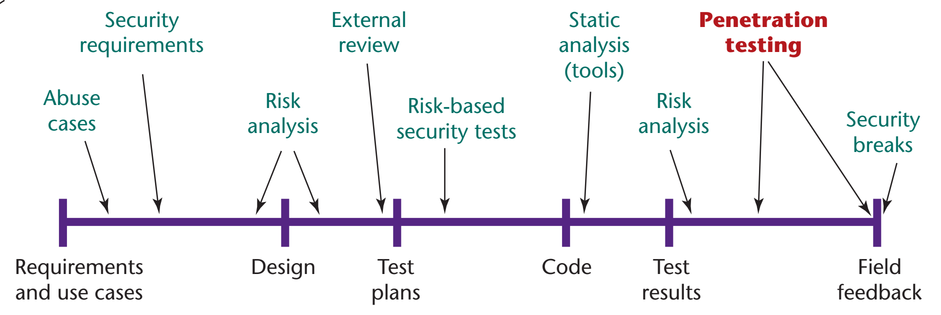

Figure 1. The software development life cycle. Throughout this series, we'll focus on specific parts of the cycle; here, we're examining penetration testing.

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 reasonable 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.

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.

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 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.*