Software Security Testing

Bruce Potter
Booz Allen Hamilton

Gary McGraw
Cigital

Security testing has recently moved beyond the realm of network port scanning to include probing software behavior as a critical aspect of system behavior (see the sidebar). Unfortunately, testing software security is a commonly misunderstood task. Security testing done properly goes deeper than simple black-box probing on the presentation layer (the sort performed by so-called application security tools)—and even beyond the functional testing of security apparatus.

Testers must use a risk-based approach, grounded in both the system's architectural reality and the attacker's mindset, to gauge software security adequately. By identifying risks in the system and creating tests driven by those risks, a software security tester can properly focus on areas of code in which an attack is likely to succeed. This approach provides a higher level of software security assurance than possible with classical black-box testing.

What's so different about security?

Software security is about making software behave in the presence of a malicious attack, even though in the real world, software failures usually happen spontaneously—that is, without intentional mischief. Not surprisingly, standard software testing literature is only concerned with what happens when software fails, regardless of intent. The difference between software safety and software security is therefore the presence of an intelligent adversary bent on breaking the system.

Security is always relative to the information and services being protected, the skills and resources of adversaries, and the costs of potential assurance remedies; security is an exercise in risk management. Risk analysis, especially at the design level, can help us identify potential security problems and their impact.[1] Once identified and ranked, software risks can then help guide software security testing.

A vulnerability is an error that an attacker can exploit. Many types of vulnerabilities exist, and computer security researchers have created taxonomies of them.[2] Security vulnerabilities in software systems range from local implementation errors (such as use of the `gets()` function call in C/C++), through interprocedural interface errors (such as a race condition between an access control check and a file operation), to much higher design-level mistakes (such as error handling and recovery systems that fail in an insecure fashion or object-sharing systems that mistakenly include transitive trust issues). Vulnerabilities typically fall into two categories—bugs at the implementation level and flaws at the design level.[3]

Attackers generally don't care whether a vulnerability is due to a flaw or a bug, although bugs tend to be easier to exploit. Because attacks are now becoming more sophisticated, the notion of *which* vulnerabilities actually matter is changing. Although timing attacks, including the well-known race condition, were considered exotic just a few years ago, they're common now. Similarly, two-stage buffer overflow attacks using trampolines were once the domain of software scientists, but now appear in 0day exploits.[4]

Design-level vulnerabilities are the hardest defect category to handle, but they're also the most prevalent and critical. Unfortunately, ascertaining whether a program has design-level vulnerabilities requires great expertise, which makes finding such flaws not only difficult, but particularly hard to automate.

Examples of design-level problems include error handling in object-oriented systems, object sharing and trust issues, unprotected data channels (both internal and external), incorrect or missing access control mechanisms, lack of auditing/logging or incorrect logging, and ordering and timing errors (especially in multithreaded systems). These sorts of flaws almost always lead to security risk.

Risk management and security testing

Software security practitioners perform many different tasks to manage software security risks, including

- creating security abuse/misuse cases;
- listing normative security requirements;
- performing architectural risk analysis;

- building risk-based security test plans;
- wielding static analysis tools;
- performing security tests;
- performing penetration testing in the final environment; and
- cleaning up after security breaches.

Three of these are particularly closely linked—architectural risk analysis, risk-based security test planning, and security testing—because a critical aspect of security testing relies on probing security risks. Last issue's installment[1] explained how to approach a software security risk analysis, the end product being a set of security-related risks ranked by business or mission impact. (Figure 1 shows where we are in our series of articles about software security's place in the software development life cycle.)

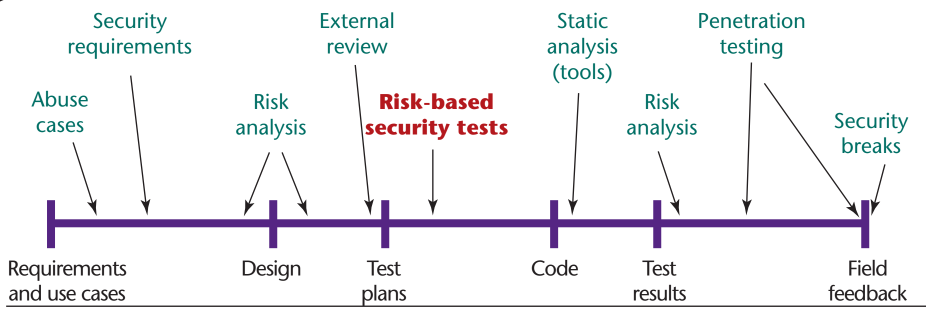

Figure 1. The software development life cycle. Throughout this series, we'll focus on specific parts of the cycle; here, we're examining risk-based security testing.

The pithy aphorism, "software security is not security software" provides an important motivator for security testing. Although security features such as cryptography, strong authentication, and access control play a critical role in software security, security itself is an emergent property of the entire system, not just the security mechanisms and features. A buffer overflow is a security problem regardless of whether it exists in a security feature or in the noncritical GUI. Thus, security testing must necessarily involve two diverse approaches:

1. testing security mechanisms to ensure that their functionality is properly implemented, and
2. performing risk-based security testing motivated by understanding and simulating the attacker's approach.

Many developers erroneously believe that security involves only the addition and use of various security features, which leads to the incorrect belief that "adding SSL" is tantamount to securing an application. Software security practitioners bemoan the over-reliance on "magic crypto fairy dust" as a reaction to this problem. Software testers charged with security testing often fall prey to the same thinking.

How to approach security testing

Like any other form of testing, security testing involves determining who should do it and what activities they should undertake.

Who

Because security testing involves two approaches, the question of who should do it has two answers. Standard testing organizations using a traditional approach can perform functional security testing. For example, ensuring that access control mechanisms work as advertised is a classic functional testing exercise.

On the other hand, traditional QA staff will have more difficulty performing risk-based security testing. The problem is one of expertise. First, security tests (especially those resulting in complete exploit) are difficult to craft because the designer must think like an attacker. Second, security tests don't often cause direct security exploit and thus present an observability problem. A security test could result in an unanticipated outcome that requires the tester to perform further sophisticated analysis. Bottom line: risk-based security testing relies more on expertise and experience than we would like.

How

Books like *How to Break Software Security* and *Exploiting Software* help educate testing professionals on how to think like an attacker.[4,5] Nevertheless, software exploits are surprisingly sophisticated these days, and the level of discourse found in books and articles is only now coming into alignment.

White- and black-box testing and analysis methods both attempt to understand software, but they use different approaches depending on whether the analyst or tester has access to source code. White-box analysis involves analyzing and understanding source code and the design. It's typically very effective in finding programming errors (bugs when automatically scanning code and flaws when doing risk analysis); in some cases, this approach amounts to pattern matching and can even be automated with a static analyzer (the subject of a future installment of this department). One drawback to this kind of testing is that it might report a potential vulnerability where none actually exists (a false positive). Nevertheless, using static analysis methods on source code is a good technique for analyzing certain kinds of software. Similarly, risk analysis is a white-box approach based on a deep understanding of software architecture.

Black-box analysis refers to analyzing a running program by probing it with various inputs. This kind of testing requires only a running program and doesn't use source-code analysis of any kind. In the security paradigm, malicious input can be supplied to the program in an effort

From outside→in to inside→out

Traditional approaches to computer and network security testing focus on network infrastructure, firewalls, and port scanning. The notion is to protect vulnerable systems (and software) from attack by identifying and defending a perimeter. In this paradigm, testing focuses on an outside→in approach. One classic example is the use of port scanning with tools such as `nmap <http://www.insecure.org/nmap/>` to probe network ports and see what service is listening. Figure A shows a classic outside→in paradigm focusing on firewall placement.

By contrast, we advocate an inside→out approach to security, whereby software inside the LAN (and exposed on LAN boundaries) is itself subjected to rigorous risk management and security testing.

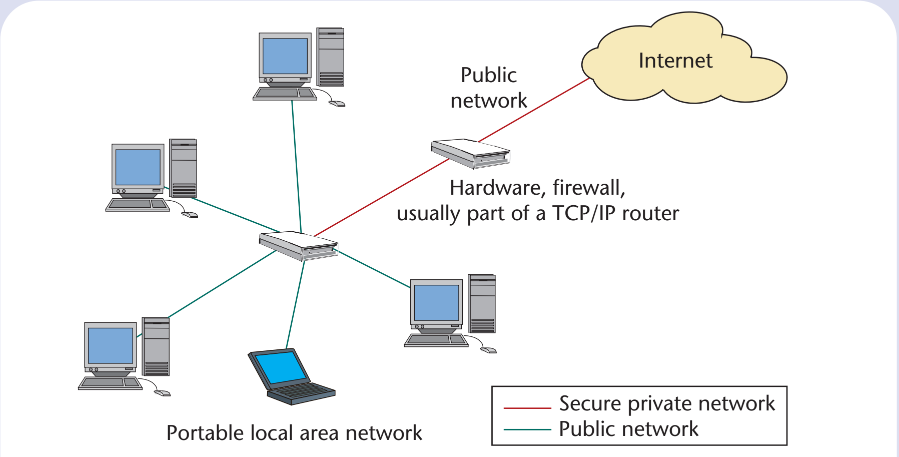
Public network
Internet
Hardware, firewall, usually part of a TCP/IP router
Portable local area network
Secure private network
Public network

Figure A: The outside→in approach. A firewall protects a LAN by blocking various network traffic on its way in; outside→in security testing involves probing the LAN with a port scanner to see which ports are "open" and what services are listening on those ports. A major security risk associated with this approach is that the services traditionally still available through the firewall are implemented with insecure software.

to break it: if the program breaks during a particular test, then we might have discovered a security problem. Black-box testing is possible even without access to binary code—that is, a program can be tested remotely over a network. If the tester can supply the proper input (and observe the test's effect), then black-box testing is possible.

Any testing method can reveal possible software risks and potential exploits. One problem with almost all kinds of security testing (regardless of whether it's black or white box) is the lack of it—most QA organizations focus on features and spend very little time understanding or probing nonfunctional security risks. Exacerbating the problem, the QA process is often broken in many commercial software houses due to time and budget constraints and the belief that QA is not an essential part of software development.

An example: Java Card security testing

Doing effective security testing requires experience and knowledge. Examples and case studies like the one we present now are thus useful tools for understanding the approach.

In an effort to enhance payment cards with new functionality—such as the ability to provide secure cardholder identification or remember personal preferences—many credit-card companies are turning to multiapplication smart cards. These cards use resident software applications to process and store thousands of times more information than traditional magnetic-stripe cards.

Security and fraud issues are critical concerns for the financial institutions and merchants spearheading smart-card adoption. By developing and deploying smart-card technology, credit-card companies provide important new tools in the effort to lower fraud and abuse. For instance, smart cards typically use a sophisticated crypto system to authenticate transactions and verify the identities of the cardholder and issuing bank. However, protecting against fraud and maintaining security and privacy are both very complex problems because of the rapidly evolving nature of smart-card technology.

The security community has been involved in security risk analysis and mitigation for Open Platform (now known as Global Platform, or GP) and Java Card since early 1997. Because product security is an essential aspect of credit-card companies' brand protection regimen, these companies spend plenty of time and effort on security testing. One central finding emphasizes the importance of testing particular vendor implementations according to our two testing categories: adherence to functional security design and proper behavior under particular attacks motivated by security risks.

The latter category, risk-based security testing (linked directly to risk analysis findings) ensures that cards can perform securely in the field even when under attack. Risk analysis results can be used to guide manual security testing. As an example, consider the risk that, as designed, the object-sharing mechanism in Java Card is complex and thus is likely to suffer from security-critical implementation errors on any given card. Testing for this sort of risk involves creating and manipulating stored objects where sharing is involved. Given a technical description of this risk, building specific probing tests is possible.

Automating security testing

Over the years, we (the authors) have been involved in several projects that have identified architectural risks in the GP/Java Card platform, suggested several design improvements, and designed and built automated security tests for final products (each of which had multiple vendors).

Several years ago, Cigital began developing an automated security test framework for GP cards built on Java Card 2.1.1 and based on extensive risk analysis results. The end result is a sophisticated test framework that runs with minimal human intervention and results in a qualitative security testing analysis of a sample smart card.

The first test set, the functional security test suite, directly probes low-level card security functionality. It includes automated testing of class codes, available commands, and crypto functionality. This test suite also actively probes for inappropriate card behavior of the sort that can lead to security compromise.

The second test set, the hostile applet test suite, is a sophisticated set of intentionally hostile Java Card applets designed to probe high-risk aspects of the GP on a Java Card implementation.

Results: Nonfunctional security testing is essential

Most cards tested with the automated test framework pass all functional security tests, which we expect because smart-card vendors are diligent with functional testing (including security functionality). Because smart cards are complex embedded devices, vendors realize that exactly meeting functional requirements is an absolute necessity for customers to accept the cards. After all, they must perform properly worldwide.

However, every card submitted to the risk-based testing paradigm exhibited some manner of failure when tested with the hostile applet suite. Some failures pointed directly to critical security vulnerabilities on the card; others were less specific and required further exploration to determine the card's true security posture.

As an example, consider that risk analysis of Java Card's design documents indicates that proper implementation of atomic transaction processing is critical for maintaining a secure card. Java Card has the capability of defining transaction boundaries to ensure that if a transaction fails, data roll back to a pre-transaction state. In the event that transaction processing fails, transactions can go into any number of possible states, depending on what the applet was attempting. In the case of a stored-value card, bad transaction processing could allow an attacker to "print money" by forcing the card to roll back value counters while actually purchasing goods or services.

When creating risk-based tests to probe transaction processing, we directly exercised transaction-processing error handling by simulating an attacker attempting to violate a transaction—specifically, transactions were aborted or never committed, transaction buffers were completely filled, and transactions were nested (a no-no according to the Java Card specification). These tests were not based strictly on the card's functionality—instead, security test engineers intentionally created them, thinking like an attacker given the results of a risk analysis.

Several real-world cards failed subsets of the transaction tests. The vulnerabilities discovered as a result of these tests would let an attacker terminate a transaction in a potentially advantageous manner, a critical test failure that wouldn't have been uncovered under normal functional security testing. Fielding cards with these vulnerabilities would let an attacker execute successful attacks on live cards issued to the public. Because of proper risk-based security testing, the vendors were notified of the problems and corrected the code responsible before release.

There is no silver bullet for software security; even a reasonable security testing regimen is just a start. Unfortunately security continues to be sold as a product, and most defensive mechanisms on the market do little to address the heart of the problem, which is bad software. Instead, they operate in a reactive mode: don't allow packets to this or that port, watch out for files that include this pattern in them, throw partial packets and oversized packets away without looking at them. Network traffic is not the best way to approach this predicament, because the software that processes the packets is the problem. By using a risk-based approach to software security testing, testing professionals can help solve security problems while software is still in production. ▢

References

1. D. Verndon and G. McGraw, "Risk Analysis in Software Design," *IEEE Security & Privacy,* vol. 2, no. 4, 2004, pp. 79–84.
2. C.E. Landwehr et al., *A Taxonomy of Computer Program Security Flaws, with Examples*, tech. report NRL/FR/5542—93/9591, Naval Research Laboratory, Nov. 1993.
3. G. McGraw, "Software Security," *IEEE Security & Privacy*, vol. 2, no. 2, 2004, pp. 80–83.
4. G. Hoglund and G. McGraw, *Exploiting Software*, Addison-Wesley, 2004.
5. J. Whittaker and H. Thompson, *How to Break Software Security*, Addison-Wesley, 2003.

***Bruce Potter** is a senior associate with*

Booz Allen Hamilton. He is also the founder of the Shmoo Group of security professionals. His areas of expertise include wireless security, large-scale network architectures, smartcards, and promotion of secure software engineering practices. Potter coauthored the books 802.11 Security *(O'Reilly and Associates, 2003) and* Mac OS X Security *(New Riders, 2003); he's currently coauthoring* Master FreeBSD *and* OpenBSD Security *(O'Reilly and Associates, summer 2004). He was trained in computer science at the University of Alaska, Fairbanks.*

***Gary McGraw** is chief technology officer of Cigital. His real-world experience is grounded in years of consulting with major corporations and software producers. He serves on the technical advisory boards of Authentica, Counterpane, Fortify, and Indigo. He also is 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.*