Editors: Martin R. Stytz, mstytz@att.net
James A. Whittaker, jw@cs.fit.edu

Why Security Testing Is Hard

Herbert H. Thompson
Security Innovation

Software testing is a discipline that has become pretty good at verifying requirements. Languages such as the Unified Modeling Language have made the process of moving from a specification (what the application should do) to test cases (verification that the application operates as specified) much easier. However, several types of bugs routinely escape testing. Many of these flaws are not specification violations in the traditional sense, meaning that the application might behave correctly according to requirements, but it might perform some additional, unspecified task in the process. Bugs like these would necessarily escape most automated testing because testers craft test cases to look for the *presence* of some correct behavior and not the absence of additional behavior. In this department, I examine the subtle nature of most security bugs and why testing for them can be difficult.

Side-effect behavior

A typical functional test case could take this form: when we apply input *A*, look for the presence of result *B*. What if, though, in producing result *B*, the application also performs some other action (which we'll call *C*)? If *C* were something overt, like an unexpected dialog box appearing onscreen, testers likely would notice it. But if it were something subtler, such as writing a file or opening a network port, testers might not detect it, and it could occur undetected repeatedly during testing.

The RDISK utility, for example, which administrators can run in Windows NT 4.0, helps create an Emergency Repair Disk for a machine. For the most part, this utility works as specified, and when run, it creates a backup of machine information, including the Windows Registry. During execution, however, the RDISK utility creates a temporary file with universal read permissions—meaning that even a guest user remotely logged in to the system could read its contents. The registry contains sensitive data about a system's configuration, and allowing a potential attacker access to this data creates a severe security vulnerability (see www.microsoft.com/technet/treeview/default.asp?url=/technet/security/bulletin/MS00-004.asp for more details). It's likely that hundreds of test cases were executed with this utility, while testers meticulously examined the backup's integrity and scrutinized its performance. Each time the utility might have performed as specified, but it's also likely to have exhibited the same insecure behavior. Testers routinely miss these hidden actions and the result is dangerous side-effect behaviors that ship with their products.

To illustrate the side-effect nature of most software vulnerabilities, consider Figure 1. The circle represents an application's intended functionality, which is usually defined by the specification and the tester's notion of what the application is intended to do. The amorphous shape superimposed on the circle represents the application's actual, implemented functionality. In a perfect world, the coded application would completely overlap with its specification, but in practice, this is hardly ever the case. Behavior gets coded incorrectly, features interact with each other in unintended ways, and bugs inevitably enter the system. The areas of the circle that the coded application does not cover (shown in blue in the figure) represent what we think of as bugs: behavior that was coded incorrectly and does not conform to the specification. Once the incorrect behavior manifests, these failures are fairly easy to detect. For example, if RDISK were to crash when we try to back up the registry, there is a pretty good chance that testers would notice the bug. From a testing perspective, tests designed to verify that this feature works correctly are fairly easy to construct: run RDISK to back up system data and then verify the backup's integrity by restoring the system. Tests like this tell us that the application is behaving according to the specification.

Consider again the temporary file that the RDISK utility creates. Its existence could let a malicious user view the registry's entire contents—Windows 98 and NT notoriously included passwords in clear text in the registry—and thus represents a security concern of the highest severity. This side effect is masked because the application actually did what it was supposed to. In Figure 1, areas of the application figure that fall outside the circular region (shown in red) repre-

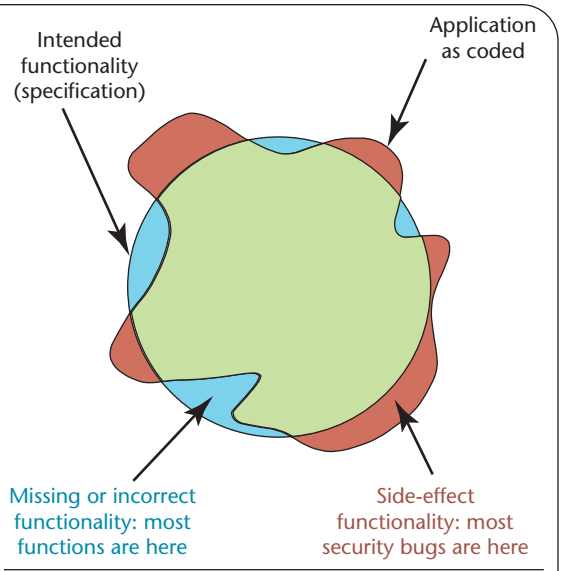

Figure 1. Intended versus implemented software behavior in applications. The circle represents the software's intended behavior as defined by the specification. The amorphous shape superimposed on it represents the application's true behavior. Most security bugs lay in the areas of the figure beyond the circle, as side effects of normal application functionality.

sent this unintended and potentially dangerous software functionality.

The state of security testing

The phrases *security testing*, *penetration testing*, and *red-teaming* have traditionally referred to executing a suite of scripted tests that represent known exploits. Many companies that offer security-testing services base their business essentially on maintaining exploit libraries. They know these exploits have worked in the past, but they would like to see if the exploits still work on the newest application. We liken this to librarians testing software. Whenever there is a new product, companies that use this approach check test cases out of the library, run them, and hope for the best. This strategy finds old, reemergent vulnerabilities without any hope of ever finding new ones. Such testing thus leaves the application vendor at a severe disadvantage as determined attackers are likely to meticulously search for novel vulnerabilities.

The problem with this argument, however, is that the librarian testing strategy actually works! Many security testing vendors have built businesses on this model because developers repeatedly make the same mistakes. Add to this the other piece of the equation—that current software is really buggy—and you have a winning test strategy.

But this will not remain the case for long. The increasing level of security awareness and the rash of books, conferences, and magazines (like this one) that have emerged recently signal that the librarian method of security testing is nearing an end. Applications will eventually become immune to these test cases as better coding methodologies take hold.

The need for techniques

To truly test for security concerns, we must test like detectives, following clues to insecure behavior and then zeroing in on vulnerabilities. This is not to say that we can't learn from past mistakes. Published bugs are probably the best source for learning about how vulnerabilities sneak into our applications. The key to success is extracting techniques to find these bugs instead of simply translating the bug into a scripted test case to add to the library.

In 2001, James Whittaker of the Florida Institute of Technology and I set out on a quest to study 10,000 security bugs. For each vulnerability we investigated, we asked the following three key questions:[1]

- *What fault would have caused this vulnerability?* In some cases, we could categorically identify the vulnerability's root cause because we could access the source code. For others, we had to infer what we thought to be the most likely set of faults that could cause such behavior.
- *What were the failure symptoms that should have alerted a tester to the vulnerability's presence?* Because we studied security vulnerabilities in released software products, we surmised that the original testers did not observe the failure symptoms or that the causal fault never executed. We wanted to understand why testers missed these failures and discover which tools and techniques could have alerted them to the vulnerability's presence.
- *What testing technique would find this vulnerability?* This is the ultimate question for security testers: exactly how do we go about testing for security vulnerabilities? We weren't surprised to discover that no existing testing technique could have readily uncovered many of the vulnerabilities. Thus, we created several new testing techniques that would find the vulnerabilities we examined in our study.

Our study resulted in a set of generalized techniques to find software security vulnerabilities that were published in the book *How to Break Software Security* (Addison-Wesley, 2003). In it, we characterized four general classes of testing techniques: dependencies, unanticipated user input, techniques to expose design vulnerabilities, and techniques to expose implementation vulnerabilities.

Dependency insecurities and failures

Software operates in a highly codependent environment in which modern applications load dozens of libraries and interface with several third-party and operating-system components. For security, two issues are of concern. First, the application might inherit insecurities from one of the components on which it depends. For example, consider a library that processes data from a file the application reads. If that library contains a buffer overflow vulnerability that the file's data could exploit, it creates a severe vulnerability in the root application even though the flaw

doesn't exist in the application's code.

Another concern is that an external resource that provides some security service to an application might become unavailable or fail. For libraries that provide some security-related service to the application, testers must conduct tests to ensure that the application would respond securely, if these libraries or components were to fail. Consider a vulnerability in Internet Explorer's Content Advisor feature.[2] Content Advisor password protects classes of Internet sites from being viewed by the user. Internet Explorer loads the library `MSRATING.DLL` when the feature is on. Given its name, we can speculate that this library provides some security-related content verification for Internet Explorer; we might also suspect that if that library failed to load, Internet Explorer would respond by raising an error message or shutting down. If we do, however, deny the browser access to this library, Internet Explorer then permits access to any previously blocked site. Without access to the source code, we can only speculate as to what happened. A likely scenario is that when Content Advisor is invoked, `MSRATING.DLL` loads, and overwrites the Internet Explorer functions that fetch page contents. When this library failed to load, Internet Explorer probably did not check return values of the `LoadLibrary` system call, and the call was assumed to have succeed. Dependency failures such as these are usually severely under-applied inputs to software, and as a result, error-handling code gets little testing scrutiny. These failures must be examined to have any real-world perspective on an application's security.

Unanticipated user input

Some inputs such as reserved words, escape characters, long strings, and boundary values can cause undesirable side effects and require special testing attention. The most notorious side-effect behavior is the buffer overrun, which is caused by processing strings entered into input fields that are longer than the memory allocated to hold them. In certain situations, the overflow can allow parts of the long string to be executed. If the string contains shell code (instructions in machine language) then the user can sometimes force arbitrary commands to execute on the system. This is a severe security concern when these inputs are accepted from an untrusted, remote user.

Long strings are only part of the problem. Applications might constrain input data with respect to length, but might not consider characters and character combinations that the application could interpret as commands. Many exploitation techniques, such as SQL injection and Cross Site scripting, take advantage of this.

Design insecurities

Many security vulnerabilities are designed into an application. Obviously, no ethical or prudent development organization would ever do this intentionally, but foreseeing high-level design decisions' impact on an application's or its host system's security is often very difficult. Test instrumentation—programmatic interfaces to the application which are added for testing purposes—is one example: these interfaces are commonly designed into an application from the beginning so that automation harnesses and other testing tools can efficiently execute tests with the intention of removing them before the product is released. Many applications ship with test instrumentation, either because some legitimate application features have come to depend on them or because long-abandoned testing hooks are forgotten in the frenzy of development modifications leading up to an application's release. These interfaces can sometimes bypass security controls to allow easy testing of product features that are not security related and can, thus, create gaping security holes. Other design errors can cause problems too, such as ports left open and unsecured, or insecure default values and configurations. Errors such as these can give attackers a clear and often unimpeded entry point into the application, its host system, or user data.

Implementation insecurities

Imperfect implementation can make even the most perfect designs insecure. Specifications can outline security meticulously and yet be implemented in a way that causes insecurity. The best example of this is the so-called man-in-the-middle attack.

In this attack, the attacker gets between the time that the application checks security on a piece of sensitive information and when the application actually uses that information. The ideal situation is that every time an application performs sensitive operations, it checks to ensure that they will succeed securely. If too much time lapses between time of check and time of use, the possibility for the attacker to get in the middle of such a transaction must be considered. It is the old "bait and switch" con applied to computing: bait the application with legitimate information and then switch that information with illegitimate data before it notices.

Specifications can outline security meticulously and yet be implemented in a way that causes insecurity.

Probably the best-known vulnerability of this type exists in xterm, a terminal emulation application for Unix and Linux.[3] This application runs as the root user and thus has access to write to any file on the system.

Most users, however, usually are more restricted. The application allows users to write information to a user-supplied log file and then checks to see if the user has write permission to that file before appending data to it. The flaw is that a restricted user of xterm can put a hard link to another file in place of the log file they provided. This can be done after xterm has done its permissions check on the file. Exploiting this flaw would let a restricted user append data to the password file (which is only accessible to someone with root permissions) and create their own account with root privileges.

The need for tools

The software community desperately needs tools that address the peculiarities of security vulnerabilities and bring their symptoms into plain view during development and testing. The industry has invested much time and money in processes to help organize functional testing efforts producing bug-severity scales, coverage metrics, and other generalized benchmarks for software's functional testing. Many of these tools and processes, however, work counter to security testers' needs. Consider standard bug-severity rankings, in order of severity[4]:

- Urgent—System crash, unrecoverable data loss, jeopardizes personnel
- High—Impairment of critical system functions and no workaround exists
- Medium—Impairment of critical system functions and workaround exists
- Low—Inconvenience, annoyance
- None—None of the above or an enhancement

With such metrics, it's easy to imagine why no one would notice the writing of a temporary file or the sending of extra network packets. Testers are traditionally rewarded for both the quantity and severity of the bugs they report. Because side-effect functionality does not equate to broken functionality, testers might not even notice these behaviors; even if they did, these bugs are likely to receive a rating of low or below—managers would likely dismiss these bugs out of hand if the product is near release. Equipped with the proper tools, though, testers would notice odd behavior and be alerted to its security implications.

A few new tools offer some promise for the future. Sysinternals (www.sysinternals.com) produces Regmon and Filemon, a line of free tools that help monitor application interactions with the registry and file system, respectively. Identify software (www. identify.com) produces AppSight, a tool that helps monitor environmental interactions. If used properly, tools like these can help alert developers and testers to the presence of potentially unsafe, hidden behavior.

The second piece of the equation is the ability to not only monitor for side effects and environmental interactions but to manipulate them as well. Many vulnerabilities result from an application's inability to handle unexpected data from its interfaces. Library, disk, memory, and network failures often force applications to execute error-handling routines that are grossly insecure. Because many of these error handlers are written as afterthoughts when error cases occur during testing or in the field, they rarely have the benefit of the security that functional routines do. In some cases, an application might not have any error handlers in place for many failure situations, and a failure of an external resource could cause carefully conceived security measures to break down. To make these situations practical enough to test in a lab situation, developers need tools that allow fine-grain control over interactions between an application and its environment. To that end, we at Security Innovation (www.sisecure.com) have been working on Holodeck, a tool that offers some promise in this area. It lets users manipulate environmental actions and control operating system responses. For example, instead of filling up the hard disk with large files, Holodeck can simulate the `disk full` error to system calls (such as `CreateFile`) that access the disk and thus simulate this failure. Such tools can make the task of testing error handlers much more feasible in practice.

Security testing has a long way to go. It's clear that the old way of doing things and the librarian mentality of application security testing must change. If a library of known exploits fails to find a gap in our application's armor, we have only the illusion of security. The development community's recent focus on security is a good sign, but we must apply new methods in practice if we ever hope to ship secure code with confidence. □

References

1. J. Whittaker and H. Thompson, *How to Break Software Security*, Addison-Wesley, 2003.
2. H. Thompson, J. Whittaker, and F. Mottay, "Software Security Vulnerability Testing in Hostile Environments," *Proc. 2002 ACM Software Applications Conf.* (ACM-SAC 02), ACM Press, 2002, pp. 260–264.
3. M. Bishop, "Vulnerabilities Analysis," *Proc. 2nd Int'l Symp. Recent Advances in Intrusion Detection* (RAID 99), 1999, pp. 125–136.
4. T. Dyes, "Tracking Severity: Assessing and Classifying the Impact of Issues (a.k.a. Defects)," *Software Test and Quality Engineering Magazine*, vol. 1, no. 2, March/April 1999, pp. 43–46.

Herbert H. Thompson *is director of security technology at Security Innovation. He earned his PhD in mathematics from the Florida Institute of Technology. He is coauthor of* How to Break Software Security *(Addison-Wesley, 2003). At Security Innovation, he develops security testing techniques and strategies that help test security for clients and is principal investigator on grants from the US Department of Defense. Contact him at hthompson@sisecure.com.*