Testing for Security During Development: Why we should scrap penetrate-and-patch

Gary McGraw[*]
Reliable Software Technologies
21515 Ridgetop Circle, Suite 250
Sterling, VA 20166
`http://www.rstcorp.com`
`gem@rstcorp.com`

Abstract

In the commercial sector, security analysis has traditionally been applied at the network system level, after release, using tiger team approaches. After a successful tiger team penetration, specific system vulnerabilities are patched. I make a case for applying software engineering analysis techniques that have proven successful in the software safety arena to security-critical software code. This work is based on the generally held belief that a large proportion of security violations result from errors introduced during software development.

1. Moving security analysis into development

Most computer security analysis is currently performed using penetrate-and-patch tactics. Security is assessed by attempting to break into an installed system by exploiting well-known vulnerabilities. If a break-in attempt is successful, the vulnerability that permitted the security breach is patched. Traditionally, penetrate-and-patch tactics were the domain of elite security professionals and consultants whose methods and tools were as secretive as their services were expensive. More recently, many of their methods and tools have been captured in several public domain security tools. These tools have been hailed as bringing computer security analysis to the average desktop computer user. However, they have also been criticized for putting years of security experience into the hands of computer crackers in the form of simple point-and-click tools.

Penetrate and patch, and the tools that help automate it, will always have a place in the security tool box. But there are several reasons why a penetrate-and-patch approach is bad: it happens too late leaving crackers a step ahead, patches are often ignored, and patches sometimes introduce new vulnerabilities themselves. I expand on these points below.

I am a member of a team researching a different approach to security assessment that employs dynamic white-box software analysis. Our approach draws upon years of research and development into software engineering analysis techniques in the areas of software safety, reliability, and testability [7, 3]. The project is an ongoing DARPA-funded research effort investigating the application of software engineering analysis techniques to the assessment of software vulnerability. We contend that rigorous software analysis techniques will play an increasingly important role in assessing the vulnerability of security-critical applications. Our hope is that such methods will find their way into the software development process itself.

Our approach relies on the well-known fact that a significant portion of computer security violations occur because of errors in software design and coding. Cheswick and Bellovin state (page 7, [2]):

> . . . any program, no matter how innocuous it seems, can harbor security holes. (Who would have guessed that on some machines integer divide exceptions could lead to system penetrations?) I thus have a firm belief that everything is guilty until proven innocent.

Recognizing that software errors can lead to security violations, we employ software-based fault injection and automatic input generation to determine the effect of program state corruptions on the security of the system. The fault injection technique perturbs program states by *instrumenting* software source code to force changes into the state of

[*]The author's work is sponsored under the Defense Advanced Research Projects Agency (DARPA) Contracts F30602-95-C-0282 and DAAH01-97-C-R095, and National Science Foundation grant number DMI-9661393.

the program as it executes. Instrumentation is the process of non-intrusively inserting code into the software that is being analyzed and then compiling and executing the modified (or instrumented) software. Assertions are added to the code to raise a flag when a violation condition is encountered.

Problems in information security are different from those in other fields of software assurance, like safety. The fact that a security threat is *malicious* adds important subtleties and challenges. Furthermore, the characteristic threats usually encountered change from one application to another. What may be considered a threat in one application, may be an attribute of another. For example, creating a shell with *SUID* root as an application-level program may constitute a violation, while executing root shells at the operating system kernel layer may be perfectly acceptable.

The Adaptive Vulnerability Analysis algorithm we employ provides quantitative estimates of the vulnerability of software relative to a set of fault injection experiments and test cases [5]. Unlike other source-code–based measures, AVA measures how software behaves when it is forced into anomalous circumstances. That is, AVA measures a *dynamic* characteristic of software. AVA attempts to determine if a program has weaknesses that can be leveraged into security violations.

2. Penetrate and patch and the status quo

A number of software tools have been developed in recent years to support *post facto* security analysis of installed computer systems. Perhaps the most popularly known network scanning tool is SATAN. The Internet Security Scanner (ISS) is another popular network scanning tool. ISS scans network ports and attempts to find and exploit known vulnerabilities. The Computer Oracle and Password System (COPS) is a collection of about a dozen programs and shell scripts that each attempt to detect different problem areas in Unix security (*e.g.,* system directories with loose permissions and poor password files, among others). A similar tool developed by Texas A&M University and called Tiger Scripts is based on the tiger team concept.

Each of the tools listed above performs *post facto* security analysis on installed systems by attempting to exploit well-known vulnerabilities. The role of these tools is to provide a “base check” for system administrators to ensure they have not overlooked any of the basic vulnerabilities that can be exploited by crackers (who, incidentally,use all of these tools). While post-installation analysis has its place, it also has some serious drawbacks:

The tiger team approach to security analysis occurs too late. Security analysis needs to be performed as part of the software development process, *before* software is released. Releasing a product without performing security analysis puts customer’s sites at risk and exposes the software development company to liability. Penetrate and patch is not a true solution to the security problem. What is actually required is a way to identify the *root causes* of vulnerabilities and look for them during development.

Crackers are always one step ahead of system administrators. Crackers often know about and exploit vulnerabilities before system administrators become aware of them. This problem follows directly from the first point. Security analysis is often postponed until after software is released and is left to the discretion of the client. The ad hoc manner in which vulnerabilities are patched today ensures that many Internet sites will remain vulnerable.

Resistance against patching installed systems. Most system administrators live by the philosophy “if it ain’t broke, don’t fix it.” This philosophy exacerbates problems raised by the preceding point. System administrators have neither the time nor the inclination to patch software if they have not noticed any security breaches. But once an attack is noticed, it is usually too late.

Patches can introduce new vulnerabilities. Most code is not originally designed with security as a goal and vulnerabilities are the unanticipated result of buggy code. While a patch may close one security hole, it may simultaneously open up others. Most software vendors respond very quickly to software vulnerabilities that have been discovered after release. The pressure to cook up a solution fast is very high. Instead of applying a software engineering design process (which would be a good way to build secure code in the first place), code is patched after release. Patches are often coded under intense (sometimes public) pressure, typically in an ad hoc fashion, and often without regard to their subsequent impact. Worse yet, patches are often released without adequate testing. Compounding these problems, patched code often doesn’t make it into a subsequent version of the software. This results in the same vulnerability cropping up all over again in subsequent releases, despite the existence of a patch distributed for a previously found bug.

Software engineering and computer security researchers have recently come to recognize that detecting and eliminating bugs in software can nip security violations in the bud. In a study of the reliability of Unix utilities to random streams of input data, Miller et al. found that “…the failure rate of utilities on the commercial versions of UNIX …tested (from Sun, IBM, SGI, DEC, and NeXT) ranged from 15-43%” [6]. Most of these utilities failed because of coding errors. The class of errors that caused the most failures had their root causes in the use of pointers and array subscripts. For example, incrementing the pointer past the end of an array was a common coding error. Using dangerous input functions, such as the `gets` call, turned out to be the second most common cause of errors that crashed system utilities. Besides being a cause of reliability errors, the `gets` call gained notoriety during the Morris Internet Worm incident. The reason this call and other related input functions are dangerous is that they do not limit or check the length of the input they read. In the case of the Internet worm, supplying the `gets` call with more than 512 bytes of data overruns the stack frame, thus enabling arbitrary input data to be executed [4]. This example emphasizes the dangers of using input functions and system calls that do not check or limit input lengths.

The work by Miller et al. managed to crash several system utilities, including `ftp` and `telnet`, by testing the bounds on input functions. In *Practical Unix & Internet Security*, Garfinkel and Spafford recognize the daunting potential for security violations in standard software distributed by vendors in light of the random black-box testing results from the Miller et al. study (pg 705,[4]):

> What is somewhat frightening about the study is that the tests employed by Miller's group are among the least comprehensive known to testers — random, black-box testing. Different patterns of input could possibly cause more programs to fail. Inputs made under different environmental circumstances could also lead to abnormal behavior. Other testing methods could expose these problems where random testing, by its nature, would not.

Our testing-based approach is far superior to random black box testing, and is meant to address the need for sophisticated tools pointed out by Garfinkel and Spafford.

Bishop and Dilger studied a class of race condition flaws called time-of-check-to-time-of-use (TOCTTOU) flaws [1]. Their research attempts to identify a coding error in which a program checks for a particular characteristic of an object, then takes some action while assuming that characteristic still holds, when in fact it does not. This type of problem is particularly critical in SUID-root programs that attempt to verify that a user has access permissions to a given file, then go on to modify it. A cracker can exploit this flaw by creating a link from the file that has been granted access, *e.g.* `/usr/spool/mail/john`, to another file that requires higher privilege for access, like `/etc/passwd`. A clever enough cracker can create the link *after* access has been granted and *before* the program accesses the file. This sleight of hand can fool the program into modifying a file it would not otherwise been allowed to modify. Bishop and Dilger's research has focused on a source-code–based technique for identifying patterns of code which could have this programming condition flaw. A prototype tool was built as a Perl script which uses pattern matching to locate potential instances of this vulnerability. Current research by Bishop's group is focusing on detecting other desirable (or conversely undesirable) properties in software source code. One of the limitations of this technique reported in the paper is that static analysis cannot determine if the environmental conditions necessary for this class of TOCTTOU binding flaws exist [1]. The authors conclude that only a dynamic analyzer will be able to test the environment *during execution* and warn when an exploitable TOCTTOU binding flaw occurs. Our tool is just such a dynamic analyzer.

3. Conclusion

Having the ability to perform white-box analysis of software code together with dynamic execution of the software in a real environment will provide developers and security analysts alike a powerful tool for determining the relative vulnerability of a given software application. Our work involves developing just such a tool. The philosophy we espouse is a new kind of *design for security*. In order to develop security-critical systems, software application code must be developed with security built in rather than patched on after development. This philosophy has been recognized as a sound policy within the safety-critical applications community and should likewise become become a part of the development process for security-critical systems. With these goals in mind, we are developing a vulnerability analysis prototype tool to support developers of security-critical code in detecting vulnerabilities *prior* to release [5].

References

[1] M. Bishop and M. Dilger. Checking for race conditions in file accesses. In *The USENIX Association, Computing Systems*, pages 131–152, Spring 1996.

[2] W. Cheswick and S. Bellovin. *Firewalls and Internet Security*. Addison-Wesley, 1994.

[3] M. Friedman and J. Voas. *Software Assessment: Reliability, Safety, Testability*. John Wiley and Sons, New York, 1995. ISBN 0471-01009-X.

[4] S. Garfinkel and G. Spafford. *Practical Unix & Internet Security*. O'Reilly & Associates, Inc., 2nd edition, 1996.

[5] A. Ghosh, G. McGraw, F. Charron, and M. Schatz. Towards analyzing security-critical software during development. Technical Report RSTR-96-023-01, RST Corporation, December 1996.

[6] B. Miller, D. Koski, C. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz revisted: A re-examination of the reliability of UNIX utilities and services. Technical report, University of Wisconsin, Computer Sciences Dept, November 1995.

[7] J. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman. Predicting how badly "good" software can behave. To appear in *IEEE Software*., 1997.