An Approach for Certifying Security in Software Components*

Anup K. Ghosh & Gary McGraw
Reliable Software Technologies
21515 Ridgetop Circle, #250, Sterling, VA 20166
phone: (703) 404-9293, fax: (703) 404-9295
email: {aghosh,gem}@rstcorp.com
http://www.rstcorp.com

Abstract

The growth of Internet-based electronic commerce, with its potential to create new business markets and streamline corporate operations, has been hindered over the past three years by concerns over the security of the system. While several secure transaction protocols have emerged to allay concerns, most security violations in practice are made possible by flaws in e-commerce client/server software. The approach outlined in this paper develops a certification process for testing software components for security properties. The anticipated results from this research is a process and set of core white-box and black-box testing technologies to certify the security of software components. The manifestation of the product is a stamp of approval in the form of a digital signature.

1 Introduction

Component-based Internet technologies such as Java and ActiveX are making the use of software components easier and more pervasive than ever before. Today, the Internet is being harnessed by main-stream businesses of all sizes for group collaboration, communication, and inexpensive dissemination of information. The medium of choice is the Web. Component-based technologies such as Java applets, JavaBeans, and ActiveX controls make it possible for businesses to design Web-based information processing systems. The next step in the evolution of business on the Internet is electronic commerce.

The idea of electronic commerce—using the Internet and the Web for commercial purposes—is taking hold in both corporate board rooms and American homes. Component-based technologies designed for distributed networks, including the Internet, make widespread e-commerce possible and thus have the potential to expand business markets considerably.

The infancy of e-commerce can be likened to the pre-industrial era in the United States. Before parts became standardized, master craftsmen would custom design and build each hammer, each saw, each chair, each wheel, and at a larger scale, each bridge. The industrial era made possible the standardization of parts. Thus a bike manufacturer could reliably build a bike from standard bolts, wheels, seats, and handlebars. Without standardization, bolts would not fit in their sockets, seats would not fit snugly, and wheels on different bikes would even be differently sized.

Today, there are myriad protocols for e-commerce transactions: SSL, PCT, SET, S-HTTP, S/MIME, Cybercash, and Digicash, among others. Unfortunately, most of these protocols are not interoperable, and consumers must choose one protocol over another. If a merchant is not a subscriber to Cybercash, then a Cybercash consumer will not be able to purchase wares from the merchant. Similarly, if a consumer does not have a browser client that supports S-HTTP, then the consumer will not be able to engage in a secure transaction with a merchant that uses S-HTTP. Development of secure components for use in building commerce applications is an important step in the maturation and acceptance process. Objective and scientific security assessment is essential to this step.

Despite the great potential to connect businesses, merchants, and consumers anywhere at anytime, affordably and easily, the dangers of e-commerce loom large. The e-commerce systems of today are composed of a number of components including: a commerce server, data transaction protocols, and client software from which transactions originate. While most of the attention in e-commerce security has been focused on encryption technology and protocols for securing the data transaction, it is critical to note that a weak-

*This work is sponsored by the Advanced Technology Program (ATP), administered by the National Institute for Standards and Technology (NIST) Cooperative Agreement # 70NANB7H3049.

ness in *any one of the components* that comprise an e-commerce system may result in a security breach. For example, a flaw in the Web server software may allow a criminal access to the complete transaction records of an online bank without forcing the criminal to break any ciphertext at all. Similarly, vulnerabilities in security models for mobile code may allow insecure behavior to originate from client-side software interaction. Until the security issues of software-component–based commerce are adequately addressed, electronic commerce will not reach mass market acceptance.

2 Approach

One of the most popular approaches today to assessing computer security is using penetrate-and-patch tactics. Security is assessed by exploiting well-known vulnerabilities in an attempt to break into an installed system. 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 public domain security tools like Satan, COPS, ISS, and TAMU [2, 3, 7, 6]. These tools have been hailed as bringing computer security analysis to the average desktop computer user. 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. It is exactly these sorts of tools that will be applied against e-commerce systems.

Penetrate and patch, and the tools that help automate it, will always have a place in the security tool box. But there are several drawbacks to relying solely on the penetrate-and-patch approach: it happens too late leaving crackers one step ahead, patches are often ignored, and patches, themselves, sometimes introduce new vulnerabilities.

The approach developed here employs dynamic software analysis techniques to certify software components for secure behavior. The approach draws on years of research in software engineering analysis that has been employed in other areas of software assurance: testing, safety, reliability, and testability [9, 12, 11].

The approach is based on the premise that a significant portion of computer security violations occur because of errors in software design and coding. Cheswick and Bellovin state (page 7, [1]):

> ... 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?) We thus have a firm belief that everything is guilty until proven innocent.

The emphasis on security assessment during development stems directly from the relationship between bugs and security holes. Bugs are also the root cause of dependability and reliability problems. That brings up the issue of how dependability, reliability, and security are interrelated.

Dependability of a component measures a component's resilience to changes in the operational profile. A high dependability component could be placed in about any application environment and still work. Reliability, on the other hand, is the probability of failure-free operation of the component for a particular fixed environment. The definition implies that though a reliable component may be well-suited for at least one environment, it may not be very reliable in another.

Component models in and of themselves are neither secure nor insecure, just as at a lower level, programming languages are neither secure nor insecure. The security of a component is the degree to which the component thwarts malicious attacks that may endanger the functionality or information of the entire system. Reliability and dependability are quantified using non-malicious test data. Security is a function of malicious test data. Thus, it is logically possible to have a highly dependable component that is at the same time very vulnerable, and vice versa.

3 Certifying security

Security assessment must occur at two levels: the component level and the system level. If individual components behave insecurely than the security of the system can fall like a house of cards falls when a card is removed. Component certification involves assuring that a component will not behave dangerously in its operating environment. One class of component that is commonly understood to be dangerous is Trojan horses or computer viruses. These malicious components including hostile applets, malicious ActiveX controls, and voyeur JavaScripts violate security and privacy by intention of their designers. Another class of dangerous component not commonly recognized is composed of components that pose security hazards incidentally. For example, numerous versions of `sendmail`, a Unix e-mail server, have posed security hazards to systems that run it due to flaws in the implementation. Regardless of whether a software

component is designed to be malicious or whether its design is benign but its implementation is flawed in a way that permits exploitation, systems that execute these components are at risk. The latter category of dangerous components are the root cause of the vast majority of security violations in practice.

The second level at which security must be assured is the system level. Even in cases where components may exhibit secure behavior individually, the composition of components in systems may result in unexpected and possibly insecure behavior. Even so, it is much easier to build secure systems from secure components than building secure systems from insecure components.

The proposed approach to certifying security of software components is illustrated by the Component Security Certification (CSC) pipeline in Figure 1. The CSC pipeline is an architecture for providing security-oriented testing processes to a software component. The pipeline consists of several processes including the construction of test plans, analysis using white-box testing techniques, black-box testing techniques, and the stamping with a digital signature of the relative security rating based on the metrics evaluated through the testing.

The processes are broken out into sub-pipes of test plans, white-box testing, and black-box testing. The first stage to component certification is the development of a test plan. The application in which the JavaBean component will be used will influence the security policy, test suites, assertions, and fault perturbations used in both white-box and black-box testing processes. Based on the security policy, input generation will make use of test suites delivered from the applicant for certification as well as malicious inputs designed to violate the security policy. The definition of the security policy is used to code security assertions that dynamically monitor component vulnerability during security analyses. Finally, perturbation classes are generated for white-box fault injection analysis according to the application in which the component will be used.

The white-box and black-box dynamic analysis techniques yield a set of relative security metrics that can be used to determine how secure a given component is relative to its peers in a particular class of component given the testing processes and environment. These metrics can be used by certification authorities to digitally sign a component for use in particular applications. Next, a brief overview of the certification technologies is presented.

4 White-box certification technologies

Three white-box certification technologies have been identified. The pipeline architecture permits inclusion of other security-oriented testing methods depending on their contribution and the component being tested. The three white-box technologies briefly discussed here are: code coverage, fault injection, and assertion monitoring.

Code coverage provides a measure for how well a given component has been tested by identifying which statements and branches in the code have not been executed. If the code coverage analysis identifies section of the code that has not been executed (*e.g.*, a function that was not called or a branch that was not followed), then more test cases can be constructed to exercise those portions of the program. The more code that is tested, the higher confidence the analyst will have on the results of the certification process.

The code coverage metric by itself will not reveal any security properties of the component. Rather, code coverage analysis is most useful when combined with security-oriented testing such as fault-injection analysis and property-based testing [5, 4]. Fault injection analysis can simulate both programmer flaws as well as malicious attacks against programs while they are executing. Using fault injection analysis for security-oriented testing, the effect of programmer flaws on the security of the program and the system it is executing in can be observed. One common programmer error made in programs written in C is reading unconstrained input into a buffer. Fault injection analysis can overflow buffers with a calculated string that will execute system commands if the program stack can be reached from the buffer.

Observing security violations either through fault injection analysis or through property-based testing is made possible through the use of assertions. Assertions are conditional statements placed in the program code that codify the security policy of the program. Property-based testing is simply the process of analyzing the behavior of a program to determine if it adheres or violates some property. For the certification process, the property that is analyzed is the secure behavior for software components. One example of using assertions internally to the code is to determine if a user is granted access to a privileged resource when the use has not been authenticated. Assertions can also be employed external to the program to monitor system-wide properties. For example, an assertion can be used to determine if a portion of the file system has been accessed by an unprivileged process.

Combining assertions with fault injection analysis

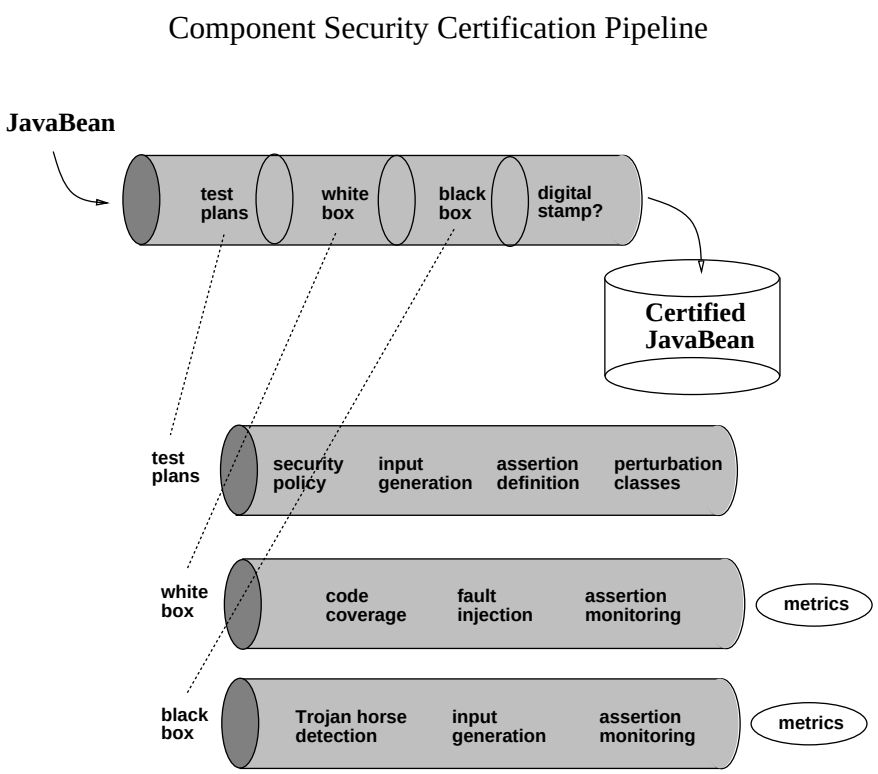

Figure 1: The Component Security Certification pipeline. The main pipeline is shown at the top of the figure. Three sub-sections of pipe making up the main pipeline are shown below. Both the white-box and black-box pipes result in metrics that can be used in the certification decision. Components that pass the rigorous testing process will be certified and digitally signed.

and coverage analysis, property-based testing of the software component can be performed until a required degree of confidence is reached when the component will be certified. The degree of confidence necessary will be determined by the application in which the component will be employed.

5 Black-box certification technologies

In order to assess the security of components for which the source code is not available, the certification pipeline employs black-box testing techniques. The simplest of the black-box testing techniques involves input generation.

A fundamental difference between standard testing and the analysis proposed here is the ability to test explicitly for security. Standard testing, which occurs in good software development labs, generally tests for functional correctness. Security is another story. Software can be correct yet still be insecure, just as software can be correct yet still be unsafe. The use of security assertions (described above) provides the ability to determine if a security policy violation has occurred as the result of input sampled from the expected user distribution. Thus, employing assertions on program outputs can enable monitoring of the security properties for a software component where source code is not available.

The generation of an input stream can be conceived as sampling from different input spaces. Figure 2 shows the black-box testing of a software component that employs security-based assertions. Most software application testing samples input from the expected user input distribution, or *operational profile*. In cases where the operational profile is unknown, the input generation functions provided will support sampling from a wide range of potential user profiles. In cases where user profile data has been collected, the input generation module will support sampling from the customized user profile. Generally, the most glaring (and perhaps most dangerous) vulnerabilities will be detected by sampling expected input distributions.

The malicious input space consists of the inputs that may be used to try to subvert a component. This space will vary from application to application, how-

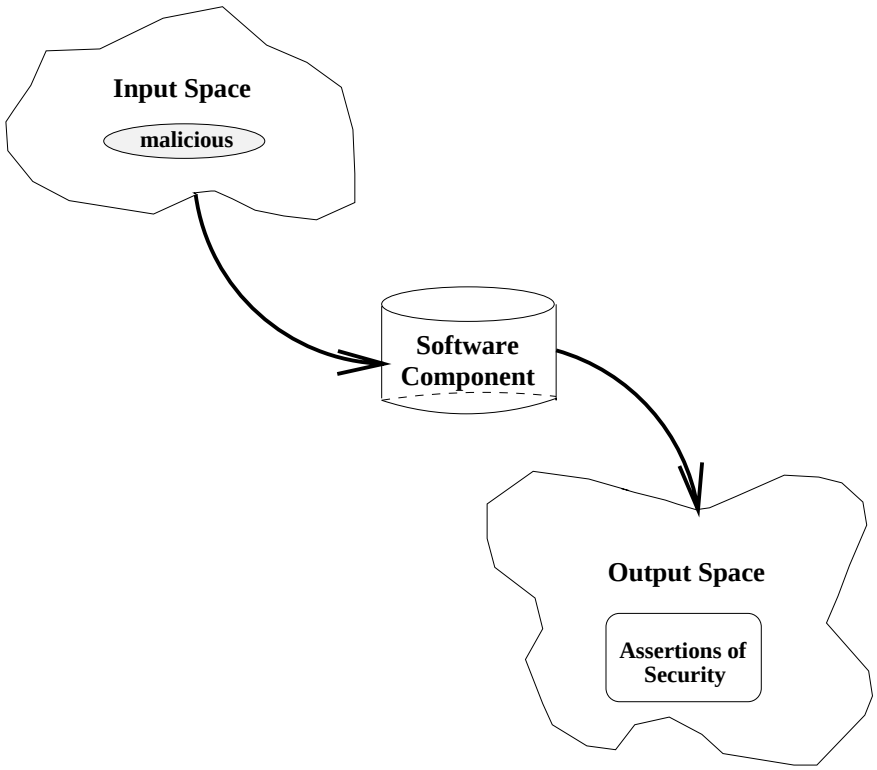

Figure 2: Black-box security testing. The black-box analysis involves generating inputs from both the expected operational profile as well as potentially malicious inputs. The software component is exercised with these test cases while the output space is monitored with security-based assertions.

ever certain characteristics will prevail. For example, the list of system commands that can be executed on the operating system platform may form a subset of the malicious input space. Another example is a list of non-alphanumeric control or "meta" characters that can potentially result in the execution of shell commands. The test cases used for black-box testing can be enhanced with malicious input using perturbation functions. These functions include the ability to truncate input streams, to overflow input buffers, to append garbage input and malicious commands, to perturb numerical constants, and to garble strings.

Analyses that use inputs perturbed in this way can be thought of as a variant on "stress testing". Stress testing typically refers to measuring the performance and availability of operating system and other multi-user capabilities in times of particularly heavy workload requests, where (in this case) the heavy workloads represent classes of unexpected and rare inputs. Many vulnerabilities result from design errors where the developer did not properly account for unexpected inputs. These errors are amenable to discovery by perturbation testing.

In addition to input generation and assertion monitoring, a third component of the certification testing will test black-box components for the presence of Trojan horses.

A Trojan horse is a program that surreptitiously performs some malicious function while at the same time appearing only to perform expected functions. Trojan horses can either execute malicious functions immediately upon startup or perform malicious functions at a later time when prompted into action by special input. The techniques discussed here address the latter class of Trojan horses. This is a particularly important class of programs because Trojan horses can easily be used in offensive attacks against components for electronic commerce.

The approach for detecting Trojan horses does not require a precise program specification. Instead the approach is based on the premise that program code (statements, functions, modules, libraries, etc.) that is *not executed* after extensive testing using inputs sampled from the expected operational profile may pose a security hazard to a system where the program may be installed.

The basic idea is to analyze software components to determine which sections of the application are not executed in spite of extensive testing using inputs from the expected operational profile. Those functions of the code not covered through testing will be flagged as potentially dangerous and in need of additional inspection/analysis. The flagged code may represent some portion of a Trojan horse which will only execute based on some unexpected or anomalous input.

The approach discovers idle code fragments by making use of *expected* input scenarios during extensive component testing. For the class of Trojan horses detected by this approach, Trojan horse code is executed only by unexpected or rare event input. The class of Trojan horses specifically excluded from this approach are those that execute during normal usage (*e.g.*, a logic bomb). These Trojan horses will likely be detected using input generation and assertion monitoring.

6 Signing components

Once a component has been thoroughly tested and some level of assurance is reached, it can be approved for use in electronic commerce systems. Without the application of formal methods which prove correctness (and can prove some things about security), there is no strict guarantee that a component will always behave in a secure fashion. However, careful application of the techniques we sketched above can lead to high levels of assurance.

This implies that formal methods are in some sense more powerful than extensive testing. This is certainly true in some cases. But the reality is that formal methods, though very powerful, cannot be economically applied to today's large and complex software systems. Formal analysis does not scale well for large systems, and formal techniques applicable component-based software have yet to be devised. Lacking formal analysis, security-based testing is a viable and economically-feasible alternative.

Once a component has been thoroughly tested and shown to be secure, it will be considered "approved." But approval by itself is not sufficient to sustain security. A potential user of the component needs some assurance that the approved component has been endorsed by the certifier and has not been altered since the testing process was complete. Digital signatures provide an answer to this problem.

One particular kind of cryptography tool allows digital information to be "signed" by their authors or distributors. Because a digital signature has special mathematical properties, it is irrevocable and unforgeable. That means a program like a Web browser can verify a signature, allowing a user to be *absolutely certain* where a piece of code came from. Better yet, browsers can be instructed always to accept code signed by some trusted party, or always to reject code signed by some untrusted party.

The key to certification and authentication is the use of *digital signatures*. The idea is simple: provide a way for people to "sign" components so that these signatures can be used in the same way we use signatures on paper documents.

7 System-level security analysis

As stated earlier, composing secure components in a system does not guarantee secure system behavior. Conventional engineering of large systems follows the doctrine of "divide and conquer". That is, large systems are broken into smaller subsystems and each is individually engineered.

Component-based software is aimed at building systems from the ground up from software components. One consequence of building large systems from components is the loss of system-wide robustness properties such as security due to the increase in the number of components that must be maintained and the number of interfaces that must be robust. A component designed and built for one application might behave remarkably different when employed in a different application. Even when component interfaces match (which is a difficult enough problem without universal acceptance of component standards), the system-wide behavior of components hooked together is as unpredictable as strange bedfellows. Unintended interactions between components can result in emergent system behavior that is unpredictable and possibly insecure.

In order to assess system-wide security properties, the approach here will employ interface propagation analysis (IPA) [10]. IPA is a fault injection analysis technique that perturbs the data at component interfaces and observes its subsequent effects. IPA determines to what extent the failure of one component will corrupt other components in a system — and ultimately the entire system itself.

Given a system composed of components A and B, questions that IPA can be used to address are: (1) will corrupted inputs to B result in corrupted outputs from B? (2) will corrupted outputs from B result in a corrupted input to A? and (3) will corrupted outputs from B corrupt subsequent outputs from A? Combining IPA with system level assertion monitoring, the security properties of the system can be analyzed.

The certification pipeline will be applied to certify and sign individual software components. IPA will be

used to certify the security of a given system composed of signed components. Further research on the application of IPA for security-based testing is necessary before concluding to what extent this analysis will be useful for system-wide security certification.

8 Conclusion

This paper describes a new approach for certifying software components for security using both tried and true software engineering analysis techniques applied to security problems as well as novel security analysis techniques. The objective of this research is to invent a process methodology that can be used to certify the security of software components used in e-commerce applications. By providing a means for assessing the security of software components, the old practice of "security through obscurity" will no longer be a viable technique, and fly-by-night software development organizations will not get away with selling supposedly secure products.

Bruce Schneier has an interesting commentary on this topic [8]:

> Billions of dollars are spent on computer security, and most of it is wasted on insecure products. After all, weak security looks the same on the shelf as strong cryptography. Two e-mail encryption products may have almost the same user interface, yet one is secure while the other permits eavesdropping. A comparison chart may suggest that two programs have similar features, although one has gaping security holes that the other doesn't. An experienced cryptographer can tell the difference. So can a thief.

The same can be said for security in general. Electronic commerce components may all appear to be equally secure on the surface. Only objective assessment can ascertain the true story. The role of the certification process will be to objectively evaluate the security of software components.

References

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

[2] D. Farmer and E.H. Spafford. The COPS security checker system. In *USENIX Conference Proceedings*, pages 165–170, Anaheim, CA, Summer 1990.

[3] D. Farmer and W. Venema. Improving the security of your site by breaking into it. Available by ftp from `ftp://ftp.win.tue.nl/pub/security/admin-guide-to-cracking.101.Z`, 1993.

[4] G. Fink and M. Bishop. Property-based testing: A new approach to testing for assurance. *ACM SIGSOFT Software Engineering Notes*, 22(4), July 1997.

[5] A.K. Ghosh, T. O'Connor, and G. McGraw. An automated approach for identifying potential vulnerabilities in software. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, Oakland, CA, May 3-6 1998.

[6] C. Klaus. Internet security scanner. Available by ftp from `ftp://ftp.iss.net/pub/iss`, 1995.

[7] D.R. Safford, D.L. Schales, and D.K. Hess. The TAMU security package: An ongoing response to Internet intruders in an academic environment. In *Proceedings of the Fourth Usenix UNIX Security Symposium*, pages 91–118, Santa Clara, CA, October 1993.

[8] B. Schneier. Cryptography, security, and the future. *Communications of the ACM*, 40(1):138, January 1997.

[9] J. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman. Predicting how badly "good" software can behave. *IEEE Software*, 14(4):73–83, July 1997.

[10] J. Voas, G. McGraw, A. Ghosh, and K. Miller. Gluing together software components: How good is your glue? In *1996 Proceedings of the Pacific Northwest Software Quality Conference*, October 1996.

[11] J. Voas, C. Michael, and K. Miller. Confidently assessing a zero probability of software failure. *High Integrity Systems Journal*, 1(3):269–275, 1995.

[12] J. Voas and K. Miller. Predicting software's minimum-time-to-hazard and mean-time-to-hazard for rare input events. In *Proc. of the Int'l Symp. on Software Reliability Eng.*, pages 229–238, Toulose, France, October 1995.