# Security Testing in Software Engineering Courses

Andy Ju An Wang
Department of Software Engineering
School of Computing and Software Engineering
Southern Polytechnic State University
Marietta, GA 30060, jwang@spsu.edu

*Abstract* - **Writing secure code is at the heart of computing security. Unfortunately traditional software engineering textbooks failed to provide adequate methods and techniques for students and software engineers to bring security engineering approaches to software development process generating secure software as well as correct software. This paper argues that a security testing phase should be added to software development process with systematic approach to generating and conducting destructive security test sets following a complete coverage principle. Software engineers must have formal training on writing secure code. The security testing tasks include penetrating and destructive tests that are different from functional testing tasks currently covered in software engineering textbooks. Systematic security testing approaches should be seamlessly incorporated into software engineering curricula and software development process. Moreover, component-based development and formal methods could be useful to produce secure code, as well as automatic security checking tools. Some experience of applying security testing principles in our software engineering course teaching is reported.**

*Index Terms* - Security testing, Software quality, Software security, Software engineering education.

## 1. INTRODUCTION

Security did not matter much at early days of computing when mechanical computers (1642 – 1945) were the major computing devices without stored software to drive those computers. Security did not matter much neither when we had batch processing and machine operators thirty years ago. Typically at that time computers were not networked at all. With the Moore's Law, computing processing power doubles every eighteen months, and as a consequence, software size and complexity grow rapidly to consume all available memory and processing power. Software engineering has been relatively successful in programming in the large, producing large software effectively, but it has not been so successful in producing secure software immune to abuse and malicious attacks, as demonstrated recently by many headline news about computer security attacks, spreading viruses, e-mail spams, and economic loss due to these security breaches.

Security vulnerabilities are approximately linear in the number of program bugs. For the last 30 years, software size has been growing in an astonishing pace. For instance, V6 (AT&T) in 1976 had about 9K lines of code, while Windows NT (Microsoft) in 2000 had 30M lines of code. The growing complexity of software contributes to the disturbing security concerns nowadays. Building software to be secure and trustworthy is qualitatively different from typical software construction covered by traditional software engineering textbooks. The current practice in building software concerns functional requirements – what outputs must be produced for given valid inputs. Security requirements are either treated as non-functional, or not covered at analysis and design phases at all. Functional requirements are relatively easy to be specified, designed, implemented, and tested. Requirements involving security breaches and malicious attacks, on the other hand, are decidedly nonfunctional. Software engineers are not told what attacks to expect so the specification of the problem is inherently incomplete. By their very nature security attacks are unpredictable and difficult to formalize.

Most security problems are due to buggy code. Buffer overflow, for instance, is mainly due to the programming language fails to provide index bound checking and programmers fail to foresee the consequences of overflowing their bounded data structures. Another example is the DoS (Denial of Services) attack. In a simpler case, DoS via locally exhausting resources could be prevented if the operating system had been designed and implemented in such a way that a user process would never be able to fill up the process table by repeatedly self-reproducing or to fill up the communication link by repeatedly sending outbound traffic. Many malformed packet DoS attacks, Smurf attacks, and remotely exhausting resources via SYN flood etc. take advantages of bad implementations of TCP/IP protocols.

Secure software refers to programs without security bugs or exploits vulnerable to abuse and malicious attack. These security exploits are defects inside programs that will lead to damages to users or the environment. Secure programs should be rigorously developed and analyzed. The characteristics of secure software include:

- Enforcement of confidentiality
- Enforcement of data integrity
- Minimized privilege
- Confinement or compartment
- Enforcement of MAC (Mandatory Access Control) instead of DAC (Discretionary Access Control)
- Keeping code secure from unauthorized or malicious use

- Fail securely and promote privacy

Software testing is an important phase in software development. There are two fundamental approaches to software testing. The first approach consists of experimenting with the behavior of a software product to see whether the product performs as expected, i.e., conforms to the specification. The other approach consists of analyzing the product to deduce its correct operation as a logical consequence of the design decision. Unfortunately both approaches do not work well for security testing, and we will elaborate this in detail in Section 2. Computing security is divided into two major categories: hardware security and software security. Hardware security is more oriented towards physical security, for instance, device protection and secure environment. Software security, on the other hand, is more complex due to the inherent complexity of software. Software security includes operating system security, database security, network security, and Web Services security. In Section 3 below, we will discuss how to design and conduct a security testing following the Relatively Complete Coverage principle. Section 4 discusses how component-based development and formal methods can help enhance software security. Section 5 discusses security testing in a classroom setting and some useful tools useful in security testing. Final summary and further research topics are provided in Section 6.

## 2. WHY DOES FUNCTIONAL TESTING APPROACH FAIL?

There are two aspects of secure code: (1) develop code without bugs and security holes, and (2) develop code immune to abuses and security attacks. However, there is no guaranteed solution available from the current software engineering technology as how to develop code satisfying both of these two criteria. The best thing we can do is to test the code dynamically or statically to verify that the behaviors of the code conform to the user requirements specification. Functional testing is the way of verifying the code by operating it in some representative situations to see whether its behavior is as expected. It is well-known that exhaustive testing is not feasible and software testing cannot show the absence of errors and defects. Software testing can show only that software errors and defects are present. It is less-known, however, that traditional functional testing fails to verify code is secure – either to verify it has no security holes, or to verify it is immune to security attacks.

What is a *security* requirements specification? A requirements specification is an agreement between the end users and the system developers. Software requirements are often classified as functional or non-functional requirements. Functional requirements are statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations [9]. Non-functional requirements are constraints on the services or functions offered by the system. Security requirements, on the other hand, are not well-understood at present. Therefore, they may either appear in non-functional requirements only, or may not appear at all in the software requirements specification. Even in the case of users pay special attention to security, it is not clear how to document the security requirements specification in effective way. For instance, we might say in the requirements specification that every user of the software must be authenticated, and every transaction must be encrypted. However, this won't stop the authenticated users abuse the system, nor malicious attackers break into the system with buffer overflow or other methods.

Software can be correct without being secure. The symptoms of security vulnerabilities are very different from those of traditional bugs [12]. Various testing methods have been discussed in software engineering literature including black-box testing, white-box testing, equivalence partitioning, structural testing, path testing, and integration testing. Unfortunately none of these testing methods work well for software security testing. The key reason for this is that traditional testing methods assume a perfect world for software to run in concerns of security: the users of the software are perfect, who never attempt to penetrate for revenge their former employers or for illicit personal gain; the environment of the software is perfect, it never interact with the software with hostile return values; the API or library functions are perfect, they never refuse the request from the software under testing and they always deliver the correct values in the right temporal order. Let's take buffer overflow as one example. From January to August 2003, CERT Coordination Center (http://www.cert.org/advisories/) published 22 security advisories and 9 of them were directly related to buffer overflow. Unfortunately buffer overflow vulnerability can hardly be tested with the current testing methods targeting program correctness. A buffer overflow occurs when a computer program attempts to stuff more data into a buffer that it can hold. The excess data bits then overwrite valid data and can even be interpreted as program code and executed [6]. Here is a piece of code looks harmless in C:

```
void functionA (char *string) {
   char buffer[16];
   strcpy(buffer, string);
   return;
}
void main() {
   char bigBuffer[256];
   int i;
   for (i=0; i<256; i++)
         bigBuffer[i] = 'A';
   functionA(bigBuffer);
}
```

As long as the buffer is overflowed with harmless data, it will not affect the correctness of the program. However, this program could be exploited to launch a severe attack to the computing system running this program. Widely used operating systems like Solaris, Linux, and Windows had been thoroughly tested using traditional software testing methods before they released. However, security breaches and new

vulnerabilities in these operating systems are reported every month. Software security vulnerabilities were identified even in several products used to increase computer security or manage passwords, including Zone Lab's ZoneAlarm personal firewall, McAfee Security ePolicy Orchestrator, and the open-source Password Safe [10]. In 2002, Microsoft Corp. stopped all coding for a month to retrain its programmers and examine old code for security problems [4].

Software vulnerabilities root in bugs and defects. Some bugs, like buffer overflow in C, reflect poorly designed language features and can be avoided by switching to a safer language, like Java. However, safer programming languages alone cannot prevent many other security bugs, especially those involving higher level semantics [2]. As expressed in [12], traditional software bugs can be found by looking for behaviors that do not work as specified. Security bugs are mostly found by looking those additional behaviors, their side effects, and the implications of interactions between the software and its environment. These additional behaviors are not specified in software requirements specification, therefore, they must be verified using non-traditional testing method – security testing, which will be elaborated in the next section.

## 3. How to Conduct Security Testing

In this section we propose a relatively complete coverage (RCC) principle for security testing, which is an adapted version of the complete coverage (CC) principle for functional testing presented in [3]. Then we discuss the process of conducting security testing following this relatively complete coverage principle. The discussion in this section is in a precise manner with formal notations, so that we could present security testing in detail without much wording overhead.

Let $P$ be a program, and let $D$ and $R$ denote its input domain and its range, respectively. $RC$ represents the run-time constraints for $P$. For simplicity, we assume that $P$ behaves as a partial function with domain $D \times 2^{RC}$ and range $R$. The behavior of $P$ can be fully captured by a finite state machine $F_P$, as described by [5], but it will be too expensive to test all the computing paths in $F_P$ for all the domain elements. Let $SR$ denote the security requirements for $P$. Note that these security requirements are not just first-order logic formulas defined on $R$ but a set of temporal logic formulas defined on a subset of computing paths in the state transition diagram of $F_P$. In other words, security specification is not just statements about the output values of a program, but behavior constraints with temporal requirements on the computation activity involved in delivering an output. For a given $d$ in $D$ and some run-time constraints $rc$ from $2^{RC}$, $P$ is said to be *relatively secure* for $d$ with respect to $rc$ if any computing path leading to $P(d)$ in $F_P$ satisfies $SR$. $P$ is said to be *(completely) secure* with respect to $RC$ if and only if it is relatively secure for every $d$ in $D$ and every $rc$ in $2^{RC}$ .

The presence of a security defect or vulnerability is demonstrated by showing that $P$ is not relatively secure for some $d$ in $D$ under some constraints $rc$ in $2^{RC}$, that is, it does not satisfy the security requirements. We call such a situation a security *exploit*. One security exploit is a manifest symptom of the presence of a security breach, even though a security defect does not necessarily cause a security exploit. Thus security testing tries to increase the likelihood that program security defects cause security exploits, by selecting appropriate test cases.

A *test case* is an element in $D \times 2^{RC}$. A *test set* $T$ is a finite set of test cases, that is, a finite subset of $D \times 2^{RC}$. $P$ is relatively secure for $T$ if it is relatively secure for all elements in $T$. In such a case, we also say that $T$ is *successful* for $P$. A test set $T$ is said *ideal* if, whenever $P$ is insecure, there exists a *(d, rc)* in $T$ such that $P$ is insecure for *(d, rc)*. In other words, an ideal test set always shows the existence of a security defect in a program, if such a security defect exists. Obviously, for a secure program, any test set is ideal. Also, if $T$ is an ideal test set and $T$ is successful for $P$, then $P$ is secure. If the input domain can be divided into classes such that the elements of a given class are expected to behave in exactly the same way with respect to a particular run-time constraint, then we could select one single test case from each class as representative and form all these test cases into a test set $C$, called a *test criterion*. We will say $C$ satisfies *relative complete coverage* (RCC) principle.

Based on the discussion above, a security testing for a given program $P$ can be described with the following algorithm:

(1) Based on the security specification, form a run-time constraints set $RC$ and an output security requirements set $SR$.
(2) Select one subset of run-time constraints $rc$ from $RC$ and calculate the new value of runtime constraints: $RC = RC - \{rc\}$.
(3) Divide the input domain $D$ into classes $D_i$ (i = 1, 2, …, n) such that the elements of a given class are expected to behave in exactly the same way for any given $rc$ in $RC$ and $D = \bigcup D_i$ $(i = 1, 2, \dots, n)$.
(4) Form a test criterion $C = \{ <d1, d2, \dots, dn> \mid di \text{ in } Di, i = 1, 2, \dots, n\}$.
(5) Test $P$ against $C$ and $rc$ to see whether $SR$ is satisfied. If not, stop. Otherwise go to Step 2.
(6) Repeat Step 2 to 5 until $RC = \emptyset$.

If the algorithm stops at step 5, it means we have found a security defect, thus the algorithm terminates. If the algorithm terminates successfully when $RC = \emptyset$, then $P$ is relatively secure for the test set determined by the test criterion $C$. Given a program $P$, the most difficult tasks in security testing include the generation of test criteria $C$ satisfying the relative complete coverage principle and identifying run-time constraints that related closely with security exploits. Note that in step 3 of the algorithm, the divide of the input domain $D$ into classes does not have to be a partition. However, it would be more effective to create a test criterion $C$ if we can have a partition, that is, $D = \bigcup D_i$ and $\cap D_i = \emptyset$, $(i = 1, 2, \dots, n)$. Also the complexity and scale of $RC$ and $SR$ might be high for large software thus security testing tools and automatic support would be needed. We will discuss an example of applying RCC in Section 5.

### 4. CBD Approaches and Formal Methods

Component-based software development [13, 14, 15] has been recognized as a new programming paradigm to simulate "software integrated circuits" to resolve the software crisis. The key idea of component-based development (CBD) is to design and implement independently deployable software units and use them over and over again in different contexts. This will realize large productivity gains, taking advantage of best-in-class solutions, the consequent improved quality, and so forth. From the perspective of security testing, CBD lends a great deal in developing secure software.

Component-based development provides a new security checking at component level. For instance, Java security model and security policy enhance the security assurance of Java-based software components like JavaBeans and EJBs. Microsoft .NET component adds security attributes to components. The programmer of .NET components can request permissions for the components through the use of security attributes which modify methods or classes. At compile time, security attributes are emitted into metadata that is then stored in the assembly manifest. The assembly manifest is then examined by the common language runtime, and the requested permissions are examined and applied against the local computer security policy. If the policy allows, the requested permissions are granted and the assembly is allowed to run. Other component architecture like CORBA component model (CCM) and Web Services also take security into their specifications and implementations.

Recently there are a lot of research works on enhancing software security through formal methods. These research works could be divided into mainly two categories: (1) static analysis, and (2) model checking. Static analysis of programs is a proven technology in the implementation of compilers and interpreters. It has been applied in areas such as software validation and software re-engineering. Static analysis aims at determining properties of programs by inspecting their code, without executing them. These properties can be used to verify or optimize programs as well as enhance the security of the programs. A key idea of static techniques for source code analysis is that they allow us to proactively eliminate or neutralize security bugs before they are deployed or exploited.

Model checking, on the other hand, is to use model checkers to verify program properties. Model checkers are decision procedures for temporal prepositional logic. They are particularly suited to show properties of finite state-transition systems [8]. Some security properties of a program can be described using temporal safety properties [2]. A temporal safety property dictates the order of a sequence of security-relevant operations. If a program and its security properties are modeled as finite state automata, model checking techniques will help to identify whether any state violating the desired security goal.

### 5. Security Testing Practice in Classrooms

In School of Computing and Software Engineering at Southern Polytechnic State University, both "Component Based Software Development" and "Formal Methods in Software Engineering" are offered as core courses at graduate level with "Software Engineering I" and "Software Engineering II" as prerequisites. The author has experimented component testing in the Component Based Software Development, and more security testing in Embedded Systems Construction and Testing, which is offered for both graduate students and undergraduate students. We have applied RCC principle for security testing different software applications. Students conducted functional testing first and then security testing on assigned programs. They found these two steps of testing help them understand the importance of security testing and gain insights on developing secure software.

Let's take Microsoft Internet Explorer (IE) as a software application to demonstrate how to apply RCC principle to test its security. As we have discussed in Section 3, the first step is to build a run-time constraints set, *RC* for short, and an output security requirements set denoted by *SR*. The run-time constraints could be any environmental conditions that must be satisfied at execution time to run the application successfully. They include, for instance, memory size, disk space, library accessibility, correct registry, etc. Assume that we have decided the following constraints and requirements:

*SR = { IE provides a graphical interface that lets users access to World Wide Web or the local file systems. If network connection is available, a user can navigate a Web page by supplying a URL. Otherwise, IE allows a user to browse local file systems. A "Content Advisor" should always be enabled to provide content-rating service, that is, help to control the Web content that can be viewed on the host computer. }*

*RC = {mc, dc, nc, al, re, cf, rf},* where:

- *mc*: memory constraints, say, 510 MB.
- *dc:* disk space constraints, say, 1024 MB.
- *nc:* network constraints, that is, the network connection might not be available.
- *al:* access library, certain dynamic link library (DLL) might not be available.
- *re:* registry vales, the data in the Windows registry database might be tampered.
- *cf:* corrupt files, that is, some files used by IE might be corrupted.
- *rf:* replace files, that is, the files that IE creates might be replaced.

Next we select some run-time constraints *rc* from *RC* and design test criterion *C*, to test IE against *C* and *rc* to check whether *SR* is satisfied. Suppose IE behave normally for the run-time constraints *mc*, *dc*, and *nc*. Then we have a new run-time constraint set *RC = {al, re, cf, rf},* and a new iteration of the algorithm starts with select *al* in *RC*. Now the key is to select a particular library that has the highest possibility to identify a possible security breach in IE. For instance, the library `msrating.dll` was loaded both when IE was first started and again when the user enabled the Tool – Internet

Options – Content – Content Advisor – Enable feature. If this library is not loaded, the buttons for Content Advisor are disabled and no error message is raised. Thus the algorithm stopped as the security vulnerability has been found and the particular run-time constraint was identified.

The RCC principle discussed in Section 3 provides only a guideline for systematic security testing. The application of RCC would be difficult without tool support. Fortunately [12] provided a security testing tool called Holodeck that helps to conduct security testing. As shown in Figure 1 below, this tool provides a number of run-time constraints checking wizards including networking, memory, and disk.

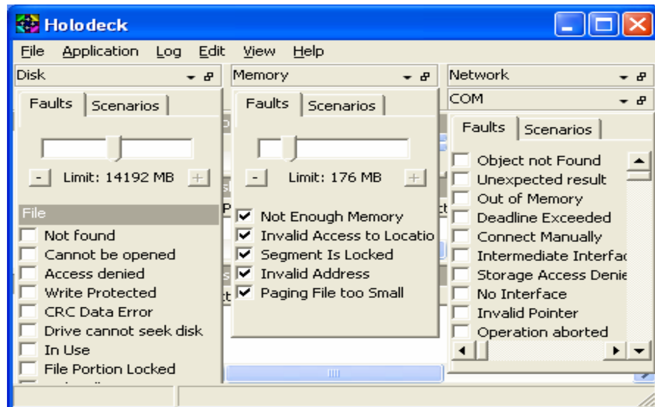


FIGURE 1
A WINDOW CAPTURE OF HOLODECK

Other security testing tools are also available for software security testing. Some of them are listed below:

**(1) MOPS**: MOPS is a tool for finding security bugs in C programs and for verifying conformance to rules of defensive programming. It models the C program as a pushdown automaton and uses model checking techniques to determine the reachability of risky states. MOPS is available at http://www.cs.berkeley.edu/~daw/mops/.

**(2) ESC/Java**: This is a programming tool for finding errors in Java programs. It detects common programming errors at compile time. ESC/Java conducts program verification with an underlying automatic theorem prover. More detailed information can be found at http://www.research.compaq.com/SRC/esc/Esc.html.

**(3) Splint**: Splint is a tool for statically checking C programs for security vulnerabilities and coding mistakes. More information can be found at http://splint.org/.

**(4) FlawFinder**: This is a tool for C/C++ programs, which is avilable at http://www.dwheeler.com/flawfinder/.

**(5) ITS4**: ITS4 is again a software security tool for C/C++ programs. It scans source code, looking for function calls that are potentially dangerous. More information is available at http://www.cigital.com/its4/.

**(6) RATS**: This is a security scanning tool identifying a list of potential security defects and suggesting possible remedies. More information on RATS is available at http://www.securesoftware.com/download_form_rats.htm.

## 6. SUMMARY AND DISCUSSION

There are a number of key skills for software engineers in the information age, for instance, information and communication technologies, the skills of analysis, evaluation, and synthesis, coding techniques, design patterns, refactoring, etc. This paper has argued that software security should be among the top skills for software engineers in the next decade. The information age is characterized by the pervasive software-intensive systems which include the Internet, large-scale heterogeneous distributed systems, systems for avionics and automotive applications, massive sensor networks and information management systems, etc. The scope, scale and complexity of these systems continue to increase faster than our ability to design them to meet our requirements. Complexity is the primary source of bugs, and bugs lead to security vulnerabilities. Thus the key solution to secure software is to handle complexity effectively. We believe that component-based design and component-oriented programming provide a systematic approach to the design and implementation of software-intensive systems, while security testing is very important for software quality in addition to functional testing. It is vital for software engineering teachers to develop ways of incorporating security testing into software engineering courses.

The advancement of hardware technology, software technology, embedded systems, consumer electronics etc. requires software engineers have adequate information security knowledge to develop secure software in a disciplined way. Unfortunately current software engineering textbooks failed to provide adequate methods and techniques for software engineers to write secure code or to conduct security testing. More research is demanded in this area. Security testing is an important phase to software development process with systematic approach to generating security test sets following a complete coverage principle. Software engineers must have formal training on writing secure code. Moreover, component-based development and formal methods could be useful to produce secure code, as well as automatic security checking tools.

The security testing methods discussed in this paper represents a very limited and preliminary experiment seeking for systematic approaches in software security validation and verification. We believe that security testing is vital to software engineering especially for embedded software construction and testing. This is due to the characteristics of embedded systems, which are often operating under extreme environmental conditions, have far fewer system resources than desktop computing systems. The implication of embedded software is much more severe than desktop computing systems. Embedded systems are usually cost sensitive, have real-time constraints, often have power constraints, and small form factors. All these constraints might be specified and tested using the RCC principle discussed in section 3.

As a final note, it might be reasonable for software engineering educators to consider developing a new course called "Security Software Engineering" in a degree program. As a minimum, we should introduce security testing as an integrated component in our existing software engineering courses.

## REFERENCES


[1] Computer Emergency Response Team (CERT) Coordination Center, http://www.cert.org/, August 2003.

[2] Hao Chen and David Wagner, MOSPS: an Infrastructure for Examining Security Properties of Software, *Proceedings of CCS'02*, November 18 -- 22, 2002, Washington, DC, USA.

[3] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli, *Fundamentals of Software Engineering*, Prentice Hall, 1991.

[4] Frank Hayes, The Story So Far, *ComputerWorld*, Vol. 37, No 28, July 14, 2003.

[5] John E. Hopcroft, Rajeev Motwani, and Jeffery D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 2nd edition, Addison Wesley, 2001.

[6] Russell Kay, Buffer Overflow, *ComputerWorld*, Vol. 37, No 28, July 14, 2003.

[7] Roger S. Pressman, *Software Engineering, A Practitioner's Approach*, 5th edition, McGraw Hill, 2001.

[8] Johnn M. Schumann, *Automated Theorem Proving in Software Engineering*, Springer-Verlag, Berlin, 2001.

[9] Ian Sommerville, *Software Engineering*, 6th edition, Addison Wesley, 2001.

[10] SECURITY WIRE DIGEST, VOL. 5, NO. 59, AUGUST 7, 2003.

[11] John Viega and Gary McGraw, Building Secure Software, Addison Wesley, 2002.

[12] James A. Whittaker and Herbert H. Thompson, *How to Break Software Security*, Addison Wesley, 2003.

[13] Clemens Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, 2nd eds., 2002. ISBN: 0-201-74572-0.

[14] Ivica Crnkovic and Magnus Larsson, *Building Reliable Component-Based Software Systems*, Artech House, Norwood, MA 02062, 2002. ISBN: 1-58053-327-2.

[15] George T. Heineman and William T. Councill eds: *Component-Based Software Engineering, Putting the Pieces Together,* Addison-Wesley, 2001. ISBN 0-201-70485-4.