

Methods For The Prevention, Detection And Removal Of Software Security Vulnerabilities

Jay-Evan J. Tevis

Department of Computer Science and Software Engineering
Auburn University
Auburn, Alabama 36849
(334) 844-6300
tevisjj@auburn.edu

John A. Hamilton, Jr.

Department of Computer Science and Software Engineering
Auburn University
Auburn, Alabama 36849
(334) 844-6300
hamilton@eng.auburn.edu

ABSTRACT

Over the past decade, the need to build secure software has become a dominant goal in software development. Consequently, software researchers and practitioners have identified ways that malicious users can exploit software and how developers can fix the vulnerabilities. They have also built a variety of source code security checking software applications to partially automate the task of performing a security analysis of a program. Although great advances have been made in this area, the core problem of how the security vulnerabilities occur still exists. An answer to this problem could be a paradigm shift from imperative to functional programming techniques. This may hold the key to removing software vulnerabilities altogether.

Categories and Subject Descriptors

D.1.1 [Programming Techniques]: Applicative (Functional) Programming.

D.4.6 [Operating Systems]: Security and Protection – *invasive software*.

General Terms

Security, Languages.

Keywords

Static code analysis, functional programming.

1. INTRODUCTION

The announcement of another computer virus threatening our business and home computers is starting to be a regularly expected occurrence. Moreover, we have come to accept the installation of a software patch as the preferred means to stopping such viruses. Companies have even devoted large amounts of resources to anti-virus teams and defense strategies to combat these problems. Such strategies include firewalls, intrusion detection mechanisms, honey pots, port

monitors, system security scanners, and e-mail content scanners [12].

Instead, we should ask the question, "How did the security vulnerability get there in the first place?" The typical answer is that the developers were unaware that the vulnerability was created when they implemented the software design. Another answer is that the security vulnerability already existed in the source code or object code that the developers used with their program. We should keep in mind that the domino effect is an important issue in computer security. Crackers cover their tracks by breaking into insecure systems and using them to launch attacks against other systems [36].

Simply including some standard data validation techniques in the source code can prevent many software security vulnerabilities. These techniques are based on the principle that all data should be filtered and then either accepted or rejected. [33] recommends several data validation rules such as assuming all input is guilty until proven otherwise, preferring to reject data rather than filter it, performing data validation both at input points and at the component level, not accepting commands from the user unless they are parsed by the software, and making policy decisions based on a "default deny" rule.

Along with good practices to follow in source code development, there are certain insecure coding practices that a software developer should avoid. [11] contains an in-depth list of Do's and Do Not's to follow when developing secure software.

Security researchers and practitioners have developed other defensive strategies targeted at decreasing the security vulnerabilities of computer systems. These solutions involve auditing all source code, authenticating all software, giving security concerns a higher priority than increased functionality during software development, preventing any unauthorized changes in the code base on a system, performing formal software verification, and applying experience-based validation based on analysis of known attacks and how the software will react to them [12, 28].

This paper focuses on source code analysis. Security experts have developed several software tools that provide a security audit of C and C++ source code. These tools, in short, check for known security vulnerabilities. The researchers in this area are concentrating on the collection of information about functions and data constructs that pose a risk to the security of a computer system. These insecure items can be considered a

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ACM Southeast Conference '04, April 2–3, 2004, Huntsville, AL, USA.
Copyright 2004 ACM 1-58113-000-0/00/0004...\$5.00.

doorway through which viruses and other malicious code enter to attack a system. The auditing tools search for these "doorways" in a source code file and alert the programmer to their presence.

These auditing tools fill a void in the prevention of security vulnerabilities; however, the common weakness among them is the reactive approach to problem detection. A tool containing the best of all the criteria identified in this paper would be a valuable resource, but it would still fall short of the need for proactive security measures. A software security checker with proactive capabilities would go beyond the standard auditing steps. It would identify general coding practices that are inherently insecure in the source code and recommend alternative approaches. Such alternative approaches may involve specific secure design patterns or algorithms. They may also include a paradigm shift from imperative, assignment-oriented programming methods to a purely functional approach.

This paper begins by looking at the current software security vulnerabilities and corresponding defense techniques. It then gives a brief description of the source code security checkers available to partially automate the security analysis of software. It finishes with a discussion of functional programming techniques and how those techniques may be a better answer to ensuring secure software.

2. SPECIFIC SOFTWARE VULNERABILITIES TO AVOID IN SOURCE CODE

Attacks on software vulnerabilities vary from year to year as old bugs are fixed and new ones are found. Nevertheless, the one bug that invariably holds the top position in the list is buffer overflow. In a June 2000 study of the ten top vulnerabilities, cases of buffer/stack overflow were in the #1, #3, and #6 positions [2]. An interesting history of the birth of buffer overflow know-how is described in [24]. It also describes how easy it is to use a web browser to modify a login page in an effort to abort a web server. [25] contains a detailed scenario called Jack and Jill that recounts how an actual attack occurred on the Internet Information Server software using buffer overflows.

Buffer overflow attacks occur when a string of characters of unchecked length is entered into a program. This allows user-supplied input to overwrite other variables, thereby changing their values. Such attacks can change the value of a return address from a function call and cause control to jump to malicious code that was also entered via the buffer overflow. Some solutions are declaring all local variables in C as static to keep them off of the stack. Patches can be added to an operating system to make code in the stack non-executable. Canary values can be declared right next to string variables. The value of a canary value can then be checked after each string write using the assert function to see if its value has changed [14]. A version of the gcc compiler has been modified to automatically add canary values to functions [36]. A variation of the buffer overflow attack is the exploitation of the mismatch between the sizes of Unicode characters and ANSI characters. The vulnerable function is MultiByteToWideChar() which has a buffer length parameter that can be changed by exploiting the stack [14].

Heap overflow attacks are also possible. A common way is to manipulate the bits maintained for each memory block in the free list. By doing so, a user can get calls to the free() function to overwrite memory locations with specific malicious data [14].

Array indexing attacks can allow a malicious user to write data to an arbitrary location in the data segment of a software application. Such data could change the constant value used in a conditional expression for example, thereby allowing the expression to return a true value to a larger range of use inputs [14]. This vulnerability exists because of the semantics of the array operator in C and C++. Such a vulnerability does not exist in Ada or Java because of the implicit bounds checking that occurs in the runtime environment [18].

Format string attacks using the "%n" specifier can make the printf() function write an integer value to an arbitrary location in memory. Because the printf() function allows a variable number of arguments, it doesn't know what number of arguments have been passed. A simple solution is to always pass a constant string as the format string, but then the values in this constant string could be changed [14].

Standard C functions that do no range checking of character string inputs are vulnerable to function algorithm attacks. These functions include scanf(), gets(), sprintf(), vsprintf(), strcpy(), and strcat() [26]. Alternatives are available on some operating systems for each of these functions. These alternatives, such as fgets(), require an additional string length parameter to counter any buffer overflow attempt. Such measures are effective; however, a long string can still be entered into one function and the remaining unread part of the string will be input by the next function that reads input. [32] contains a two-page table of standard C functions that should be used with caution or avoided altogether.

Several types of system software applications can be exploited. Setuid programs have the setuid or setgid bits set, thus giving the program all the privileges of the file owner, which may be root. Network servers (daemons) can be continually attacked with data until one breaks and the attack is successful. Network clients are normally built with a lower concern for security than network servers. Clients such as browsers many times allow a server to execute code on the client machine. This code can easily be malicious software. Mail user agents are targets for buffer overflow attacks and malicious attachments. CGI programs, which are run on a server, have the same if not more vulnerabilities as the server software because they are usually written in insecure scripting languages. Utilities, such as those commonly available on a UNIX system, can be exploited through the use of special patterns of characters that may take advantage of buffer overflow or be interpreted in a special way by a shell program. Specific user applications such as office productivity software are vulnerable to malicious macrocode embedded in documents [36].

Inside each of these applications, various code features can be exploited. The use of certain commands in shell scripts such as eval() or function calls such as the system() function call in programs allows a malicious user to possibly execute any arbitrary command. An alternative to the system() call is the

use of one of the functions in the `exec()` family. Although the `exec()` calls have fewer vulnerabilities, they should be chosen and used wisely [33].

Changes in system environment variables can cause unexpected changes in the behavior of a program. An example is the change of `LD_LIBRARY_PATH` that can cause a program to link to code in a malicious library. Symbolic links can be changed or added to introduce vulnerabilities into a computer system. After such changes are made, many standard C functions (e.g., `chmod()`, `chown()`, `link()`, `stat()`) are vulnerable to allowing unauthorized access to certain files or directories [36].

Host name attacks can occur. Information returned by the `gethostbyname()` function should not be trusted because a server can spoof the DNS response. A possible solution is to cross-check all responses using the `gethostbyaddr()` call [36].

Signals that occur when a program is in a privileged state can cause vulnerabilities. [36] lists over 50 functions whose operation can be interrupted midstream by a signal. Through the use of a signal, a malicious user can induce a race condition involving a system command that executes in two or more user modes. An interruption while the command is in kernel mode can allow unbridled access to system-level files [2].

One other area of exploitation to consider is core dumps. Malicious users can analyze a core dump to glean information on the value of program constants, variables, and registers. UNIX systems offer the `setrlimit()` function to disable memory dumps if an application crashes [33].

3. STATIC CODE SECURITY CHECKERS

Static code security checkers parse through and scan the source code, looking for potential security problems. The process is similar to virus scanners. The static code checker looks through the source code for any of the known and previously defined problem conditions. Both false positives and false negatives may occur, and should therefore be used in conjunction with other security auditing and testing methods [11].

The goal of static code security checkers is to focus the security analysis. Instead of the programmer searching the source with a utility program such as `grep()`, the checker software is aware of known potential problems and searches for them based on encoded rules and entries in a database. These checkers not only find problems, they many also describe the problem and suggest possible remedies. In addition, they provide an assessment of the potential severity of each problem for an auditor to use in his overall assessment [32].

Although source code checkers are very effective in detecting security vulnerabilities, they do have several weaknesses. The liberal syntax of C makes the language poorly suited to static analysis. The added object-oriented complexities of C++ make it difficult to analyze. Static analysis in a multi-threaded environment is difficult because of the potential interaction of data. Performing a better static analysis using more advanced algorithms is difficult and can cause an order of magnitude increase in scan time [31].

We have compiled a list of static code checkers available today. These checkers detect problematic code using proprietary heuristics to look for suspicious code segments, calls to specific utilities known to have vulnerability issues, or a combination of both. The following paragraphs give a brief description of each of these code checkers.

BOON stands for Buffer Overrun Detection. As the full name implies, the software searches for buffer overruns in C source code. The concept behind BOON is that buffer overflow detection is an integer range analysis problem. The algorithm first takes the allocated size and the actual length of each character string and builds a corresponding value pair. This approach is also taken with the parameters of the standard C library functions that handle character strings. A comparison is then made to see if the inferred allocated size of the string is at least as large as its maximum length [35].

CodeWizard is a proprietary general-purpose source code analyzer. It is not targeted specifically at security issues, and does not even advertise to do so, although the capabilities are there. Instead, it examines source code to locate violations of industry-accepted language-specific guidelines. This is done to reduce the opportunities for coding errors that could result in bugs. The analysis works by using a patented technology to search for patterns and then compare what is found to a set of rules [20].

FlawFinder searches through C/C++ code looking for potential security flaws. After the code analysis is complete, it produces a list of potential flaws sorted by risk. FlawFinder's database contains both general rules that affect any program and specific Windows and Unix functions that are very vulnerable to exploitation [37, 38].

Illuma is proprietary software that searches C/C++ source code for problems such as memory leaks, null pointer dereferences, bad memory deallocation, out-of-bounds array access, and uninitialized variables. It is used in conjunction with contracted services to assess the quality of a client's source code [23].

ITS4 stand for It's The Software Stupid (Security Scanner) [31]. It statically scans C and C++ code for vulnerabilities, but it does not do so by parsing the actual source code used in a single build configuration. Instead, ITS4 looks at several files to check for vulnerabilities in multiple builds of the software. This is done for many reasons. First, it reduces the false negatives to almost zero. Second, it avoids the complexities of real parsing that add no value to the security scanning requirement. Third, it allows ITS4 to be used in an integrated development environment to highlight potential errors from within an editor [31].

LDRA Testbed is proprietary software that performs a general-purpose static code analysis. It checks for such things as code complexity, unreachable code segments, variable interdependence, loop analysis, and correctness of procedure interfaces. It can also be used to verify a set of programming standards established by an organization; however, it does not specifically address security scanning as a possible use for the product [17].

MOPS stands for Model Checking Program for Security Properties. It checks for security vulnerabilities from a

sequence of operations viewpoint. It uses model checking together with specific rules to detect the violation of temporal safety properties. A user describes the rules in the form of a finite state machine. If the software finds any problems related to a property, it prints out the offending path found in the source code. Such techniques can find potential issues with buffer overflow, user privileges, and array indexing [5, 34].

PC-Lint is proprietary software that checks C/C++ source code to find such things as bugs, glitches, inconsistencies, non-portable constructs, and redundant code. The software can produce over 1900 distinct error messages. It does not specifically address security vulnerabilities in its findings; however, a security analyst could spot many of these vulnerabilities in PC-Lint's error report. FlexeLint is a version of the PC-Lint software extended to non-PC platforms [10].

PSCAN searches a C source code file for problematic uses of functions in the printf() and scanf() family, the syslog() function, and a variety of functions used to display warning and error messages. It does not scan for normal buffer overflows or general misuse of function parameters [7].

RATS stands for Rough Auditing Tools for Security. It checks a variety of different language source code files for security-related problems such as buffer overflows and time-of-check vs. time-of-use race conditions. The software uses greedy pattern matching to find potential errors; consequently, false positives are prone to occur more often [27, 32].

Splint stands for Secure Programming Lint. (It was previously known as LCLint.) The software checks that the source code is consistent with security properties stated in annotations. The annotations appear as comments and are associated with function parameters and return values, global variables, and structure fields. The annotations provide a way for the Splint software to use the preconditions to see if the function implementation ensures the postconditions. It resolves preconditions using postconditions from previous statements and annotated preconditions for the function [8, 9].

UNO is named after the three focus areas of the software: use of uninitialized variables, nil pointer references, and out of bounds index checking. It emphasizes these three areas to reduce the amount of false alarms produced by other static code checkers that try to look for everything. It also concentrates specifically on ANSI C source code. UNO has the ability to accept user-defined properties of application specific requirements, and then check the source code for strict compliance with these requirements [13].

WebInspect is proprietary software that automates the discovery of security vulnerabilities in both traditional and web-based applications. It can be used in an integrated development environment to do static code analysis at the click of a button. WebInspect also makes recommendations on how to fix any potential security flaws that are found. SPI Dynamics, the maker of WebInspect, is part of a technical committee working on the definition of an Application Vulnerability Description Language (AVDL). The goal of the committee is to form an XML standard to define, categorize, and classify application vulnerabilities that can be understood and used by a variety of security products [29].

4. CRITIQUE OF STATIC CODE SECURITY CHECKERS

Although the static code security checkers provide an excellent service, they still need to improve. First, many of them focus on vulnerabilities in UNIX applications. This obviously needs to expand into Windows and Macintosh software. Second, they still require a significant level of expert knowledge. In other words, they work well for novice programmers; however, an expert can do a better job at manually evaluating the potential security vulnerabilities in the source code. Third, even for experts, analysis is still time consuming. The static code checker only cuts down about $\frac{1}{3}$ of the static code analysis that needs to be performed. The rest must still be done manually.

Nevertheless, the checkers are useful because every little bit helps. These tools help to prevent the rush to check the security vulnerabilities of a piece of source code. Because of the prioritization and assessment features, they focus the analyst's attention on the more severe problems that may have manually been overlooked. Also, they can help find real bugs. These tools actually work to find problems in just a few minutes that may have taken much longer to detect [32].

[19] points out even more limitations in the current checkers. First, an automated scan has not been developed yet that catches many of the problems detected during manual analysis. Second, the scanners don't know the particulars of functions contained in libraries supplied by various domain-specific applications. Developers need to understand this so they don't think the checker looks at such things. Third, most checkers scan at most two languages. An exception is RATS, which can scan five. Fourth, the checkers perform no preprocessing that would expand macros or constant definitions.

5. SOFTWARE SECURITY ASSURANCE FROM A FUNCTIONAL PROGRAMMING PERSPECTIVE

The limitations discussed above point to a much deeper issue in providing secure source code. Imperative programming languages rely on assignment, control loops, and an environment state when implementing a program. They also have concepts such as array indexing and variable references that translate directly to memory addresses and the underlying hardware implementation [18].

Functional programming languages, on the other hand, operate in the world of mathematics [4, 15]. They offer the principle of referential transparency, which is the property of a function that its return value depends only on the values of its arguments. This provides a software developer with the assurance that a function will give the same answer to the same inputs every time. It also eliminates side effects and any nondeterministic behavior in the function [18]. In addition, it provides well-defined interfaces anchored on a firm foundation of functions with high cohesion and low coupling [21].

Functional programming languages also provide a more elegant way of programming. This means eliminating loops, pointers and variable assignments that deal directly with

computer memory. It also means using recursion, functions as first-class values, and pattern matching. This results in much less source code to write, audit, maintain, and understand. In addition, they offer the foundation for building correct programs. With preconditions, postconditions, and a formal specification given in advance, a function can be refined into an implementation that can be proven mathematically correct [3, 6].

Does this mean that traditional imperative algorithms need to be thrown out? Such a conclusion may be shortsighted. Instead we propose the need to first extract the essence of the algorithm from the language in which it was implemented and then examine the algorithm to see where imperative techniques can either be replaced by functional techniques or possibly removed altogether [22]. This dual imperative/functional approach of software development reveals the need for a security-assured library of C/C++ functions to implement many of the high-level list, map, and filter abstractions found in functional languages such as Scheme [1] and Haskell [16].

The idea of proving a software program correct has been researched for decades [3, 6]. These principles and techniques can be used to build purely functional programs that are correct. Such principles and techniques can then be applied to secure software. This refinement approach may reveal that incomplete and faulty security requirements result in incomplete and faulty security designs and programs [28]. Refining a requirement specification down to its functional implementation may reveal mathematically that the original specification is incorrect. This approach could identify a security vulnerability in a requirement long before the program is implemented and released to the public for possible virus attacks.

6. RELATED AREAS

Along with static code checking, runtime checkers have been developed also. Running in a layer between the application and operating system, these checkers work by intercepting system calls and screening each call for correctness before passing it to the operating system to be executed. Example products are Libsafe, PurifyPlus, and Immunix tools [11].

Another related method is use profiling. The concept works as follows. The behavior of a program's system calls and file activity is studied for a number of software executions and then defined. The profile is then used as a basis to monitor the software activity for any behavior anomalies. Such anomalies could indicate malicious actions by an application or the presence of virus software. Example use profiling tools are Papillon, Janus, and gprof [11].

Potential buffer overflow is undoubtedly the most searched for problem in static code checking. Run-time checking of this problem can also be done by executing destructive tests intentionally designed to detect the existence of a buffer overflow vulnerability. Example tools for testing buffer overflow are NTOMax and SendIP. Test tools can also be built to record and play back data submitted to a software application [30].

7. FUTURE WORK

Each of the static code checkers listed in this paper concentrate on specific security vulnerabilities in software. We plan to

correlate the available information in order to identify the most common insecure features in imperative languages. We then plan to build a prototype source code checker in a functional programming language that scans imperative source code. Upon doing so, the next step will be identification of functional programming techniques that could eliminate many of the imperative security vulnerabilities. Our eventual goal is to identify security vulnerabilities in imperative source code and then give a functional programming alternative to eliminating that vulnerability.

8. CONCLUSION

The production of secure software is a must in the interconnected electronic world of today. The threat from malicious users is real and the target is soft. Design and implementation methods exist for reducing the security vulnerabilities of software. In addition, a variety of source code security checkers are available that use automatic scanning to indicate potential security pitfalls in a software application. Nevertheless, the imperative nature of the majority of programming languages used now may lie at the heart of the security vulnerabilities.

Functional programming techniques and languages such as Haskell have been the domain of software researchers and enthusiasts for the past decade. The time has come for functional programming to prove its worth in solving the software problem that is probably the biggest threat to information assurance: the vulnerability to software viruses and malicious activity. As pervasive computing takes hold and networked electronic devices become the norm, software developers need to move out from the von Neumann paradigm embodied in imperative programming techniques, and move into a functional paradigm. Such a move may hold the key to building secure software that is provably secure.

9. REFERENCES

- [1] Abelson, H., et al. Revised Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation*, 11, 1 (Aug. 1998), 7-105.
- [2] Anderson, R. *Security Engineering*. Wiley Computer Publishing, New York, NY, 2001, 368.
- [3] Back, R. and von Wright, J. *Refinement Calculus*. Springer-Verlag, New York, NY, 1998, 20.
- [4] Bird, R. *Introduction to Functional Programming using Haskell, 2nd Edition*. Prentice Hall, New York, NY, 1998, 262.
- [5] Chen, H. and Wagner, D. MOPS: An Infrastructure for Examining Security Properties of Software. In *Proceedings of the 2002 Conference on Computer Communications and Security* (Washington, DC, Nov. 17-21, 2002). ACM Press, New York, NY, 2000.
- [6] Cohen, E. *Programming in the 1990s*. Springer-Verlag, New York City, NY, 1990, 2-7.
- [7] Dekok, A. PScan. *Striker-On-Line*. www.striker.ottawa.on.ca/~aland/pscan/, Oct 28, 2003.
- [8] Evans, D. Secure Programming Lint (SPLINT). *Department of Computer Science, University of Virginia*. www.splint.org/, Oct. 28, 2003.

- [9] Evans, D. and Larochelle, D. Improving Security Using Extensible Lightweight Static Analysis. *IEEE Software*, 2 (Jan-Feb 2002), 42-51.
- [10] Gimpel. PC-Lint Software. *Gimpel Software*. www.gimpel.com, Nov. 1, 2003.
- [11] Graff, M. and van Wyk, K. *Secure Coding: Principles and Practices*. O'Reilly and Associates, Sebastopol, CA, 2003, 110-117, 167.
- [12] Grimes, R. *Malicious Mobile Code*. O'Reilly and Associates, Sebastopol, CA, 2001, 464-469.
- [13] Holzmann, G. UNO Software. *Bell Labs*. www.spinroot.com/gerard/, Nov. 1, 2003.
- [14] Howard, M. and LeBlanc, D. *Writing Secure Code*. Microsoft Press, Redmond, WA, 2002, 63-88.
- [15] Hughes, J. Why Functional Programming Matters. *The Computer Journal* 32, 2 (1989), 98-107.
- [16] Jones, P. and Hughes, J. Report on the Programming Language Haskell 98. *Journal of Functional Programming* 31, 1 (Jan 2003).
- [17] LDRA. LDRA Testbed Software. *LDRA Software Technology*. www.ldra.co.uk/, Nov. 1, 2003.
- [18] Louden, K. *Programming Languages: Principles and Practice, 2nd Edition*. Brooks/Cole, Pacific Grove, CA, 2003, 206-210.
- [19] Nazario, J. Source Code Scanners for Better Code. *LinuxJournal.Com*. www.linuxjournal.com/article.php?sid=5673, Jan. 26, 2002.
- [20] Parasoft. CodeWizard. *Parasoft Inc.* www.parasoft.com, Nov. 1, 2003.
- [21] Pressman, R. *Software Engineering: A Practitioner's Approach, 5th Edition*. McGraw-Hill, Boston, MA, 2001, 527-528.
- [22] Rabhi, F. and Lapalme, G. *Algorithms: A Functional Programming Approach*. Addison-Wesley, Boston, MA, 1999, 7-10.
- [23] Reasoning. ILLUMA Software. *Reasoning Inc.* www.reasoning.com, Nov. 1, 2003.
- [24] Scambray, J., McClure, S., and Kurtz, G. *Hacking Exposed, 3rd Edition*. McGraw-Hill, Berkeley, CA, 2001, 590.
- [25] Schiffman, M. *Hacker's Challenge*. Osborne/McGraw-Hill, Berkeley, CA, 2001, 272-278.
- [26] Schildt, H. *C: The Complete Reference, 4th Edition*. McGraw-Hill, Berkeley, CA, 2000, 309-382.
- [27] Secure Software. Rough Auditing Tool for Security (RATS). *Secure Software Inc.* www.securesoftware.com, Oct. 28, 2003.
- [28] Sommerville, I. *Software Engineering, 6th Edition*. Addison-Wesley, New York, NY, 2001, 483.
- [29] SPI Dynamics. WebInspect Software. *SPI Dynamics Inc.* www.spidynamics.com, Nov. 1, 2003.
- [30] Splaine, S. *Testing Web Security*. Wiley Publishing, Indianapolis, IN, 2002, 165.
- [31] Viega, J., Bloch, J., Kohno, T., and McGraw, G. ITS4: A Static Vulnerability Scanner for C and C++ Code. *Cigital Inc.* www.cigital.com/its4/, 2000.
- [32] Viega, J. and McGraw, G. *Building Secure Software*. Addison-Wesley, Boston, MA, 2002, 127-133.
- [33] Viega, J. and Messier, M. *Secure Programming Cookbook for C and C++*. O'Reilly and Associates, Sebastopol, CA, 2003, 72-75.
- [34] Wagner, D. Modelchecking Program for Security Properties (MOPS). *Computer Science Division, University of California Berkeley*. www.cs.berkeley.edu/~daw/mops/, Oct. 28, 2003.
- [35] Wagner, D., Foster, J., Brewer, E., and Aiken, A. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proceedings of the 2000 Network and Distributed Security Symposium* (San Diego, CA, Feb. 3-4, 2000). ISOC.
- [36] Wall, K., Watson, M., and Whitis, M. *Linux Programming Unleashed*. Sams Publishing, Indianapolis, IN, 1999, 642-661.
- [37] Wheeler, D. Flawfinder. *David A. Wheeler*. www.dwheeler.com/flawfinder/, Oct. 28, 2003.
- [38] Wheeler, D. Secure Programming for Linux and Unix HowTo. *David A. Wheeler*. www.dwheeler.com/secure-programs/, Oct. 28, 2003.