SECURE PROGRAMMING CONCEPTS IN SELECTED C++ AND Java™ TEXTBOOKS

David F. Wood, Robert Morris University, wood@rmu.edu
Robert Joseph Skovira, Robert Morris University, skovira@rmu.edu

ABSTRACT

Creating secure software applications and writing secure programs are difficult affairs. Secure programs are well designed software applications which meet specified requirements. Well designed software is reliable in its availability and accessibility. The paper's problem is whether the textbooks used in instruction of beginning programming students directly and actively discuss secure programming issues. A possible flaws model (buffer overrun, format string, integer overflow, SQL injection, command line injection, and exception handling) is constructed. Selected introductory textbooks on Java and C++ were reviewed.

Keywords: Security, Secure programming, Secure coding, Security flaws, C++, Java

INTRODUCTION

Creating secure software applications and writing secure programs are difficult affairs. They are difficult because applications and programs are complex affairs and side effects of interactions between and among software modules create exploitable openings. The complexity of programs, despite best efforts and practices in design [10, 22], allow for unrecognized places in the code that can be used to attack an information system [9, 23].

When people consider the security ecology of information systems [30], they seldom think about the hazards present in software code. There is an implicit trust, a mental model of sorts, which mitigates suspicion [12, 16, 17, 29]. This is possibly due to the state of affairs where security concerns are focused on networks or operating systems. Secure program code slips under the wire. And this is as it should be, to a degree. Nevertheless, software applications do the real work. Software applications use networks and operating systems. If program code is written without an awareness of its security, the most secure networks and operating system in the world will be unable to stop exploitations of an organization's information and information resources.

Another aspect of the security ecology of software creation [30] is the approaches which program designers and programmers habitually use in doing their jobs [9]. Habitual approaches are "mental models" which designate what the designers and programmers recognize and handle in a program's code [24, 25].

Secure programming is the implementation of well designed software applications which meet specified requirements and no more than these. Well designed software is reliable in its availability and accessibility. It is user-friendly so that people do not try to circumvent safety measures. Well designed and written code is modular. Encapsulating data and methods is fundamental to programming based on security awareness [26]. Software programs pose difficult challenges for the security conscious programmer [6, 15]. Software applications are a business' draft animals focused on accomplishing sometimes rather complex tasks. Applications are designed to work within the security environments of networks and operating systems on which they depend for their functionality. The designers and programmers must work with an awareness of complex environments of networks and operating systems and exploitable points of contact [9]. Consequently, software applications are to be designed and tested to requirements and nothing more (or less). Insecure code presents many possibilities for malicious exploitation [13]. Code designed with a consciousness of the importance of a secure environment affords safe use and access to other exploitable systems [31]. "Frankly, it doesn't matter what operating system or programming language you use,, and it doesn't matter how secure the underlying platform is. If your code is insecure, your customers could be open to attack" [13, p. xxi]. While it may be difficult to write secure code , the golden rule is that the software should do no more than required. A corollary might be that permission to use and access exploitable systems be only what is required [31].

The Problem

A situation facing any instructor of programming is how to enable programming students with an awareness of security contexts and to construct secure programs and software applications. Since this is an educational state of affairs, this paper's problem is if the textbooks used in instruction of beginning programming students directly and actively discuss secure programming issues. There are some general questions which can be considered: Is there any security awareness or discussion of secure programming in the texts? Are there any explicit warnings as part of the syntactical discussions? Is there any explicit method of code testing? Is there an explicit discussion of quality in design and implementation of code? Consequently, we analyze several C++ and Java™ texts, looking for fostering of security awareness and techniques.

To do this efficiently, we will discuss the known security flaws facing software designers and programmers. We will then look at some randomly selected textbooks which are fundamental and introductory in character to see what, if anything, they offer in the way of awareness and concern about security and secure programming to the instructor and the students.

POSSIBLE EXPLOITABLE FLAWS MODEL

There are many possible exploitable programming flaws, and one widely used text presents 19 of them [13]. For the purpose of this paper, we are interested in those possible flaws internal to a program's code, or exposed during a program's execution. It is a reasonable expectation that such flaws and their discussion might show up in texts used to teach programming.

Buffer Overrun Flaw

Buffer overrun flaws or overflows [6, 13, 18] are situations where code creating a buffer or an array permits a larger number of characters to be put into the buffer or array without checking the size of the input string. This can happen when the default buffer size is used, or when the size is calculated in the code. Attackers exploit this by using a string larger than the intended size to place their commands into memory space beyond the buffer space allocated. An example in C++ is expressed by the lines [13, p. 7] "`char buf[32]; strncpy (buf, data, strlen (data));`" This intrusion can overload stacks, heaps, as well as other kinds (print) of buffers. This exploitable is possible if array boundaries are not checked. [1, 32, 13, 18, 22]. Some questions that can be raised are: Do the texts raise security issues when they discuss buffer classes or functions, looping structures, array or vector structures, out-of-bounds checks, default buffer size or calculated buffer size allocations? Do the texts suggest input validation when dealing with buffers or arrays?

Format String Flaw

A format string flaw [6, 13] allows the input of commands etc as invalidated input rather than the expected information. A format string exploitable is present when an application expects that certain information input by users as a string argument and when it is displayed or saved in a particular manner or format as output, but it is not checked before hand. This attack is used to put malicious code in memory. An insecure example of this is `printf(user_input);` a secure version is `printf("%s", user_input);` [6, 9, 13, p. 21, 22]. Some questions that can be raised are: Do the texts raise security issues when discussing user input into functions or methods? Do the texts raise security issues around internationalization methods and formats?

Integer Overflow Flaw

An integer overrun flaw [13], overflow, or even underrun happens when an attacker inputs a number which goes beyond the memory boundaries used to store the number. Any algorithm used in any calculation is suspect as are casting operations when signed and unsigned numbers are being represented. Floating point errors are part of this flaw. [6, 9, 15, 13, 22]. Some questions that can be raised are: Do the texts raise security issues when discussing numeric types and variables, and memory allocation to store numeric types? Do the texts raise security issues when discussing, if they do, casting and signed and unsigned numbers? Do the texts raise security issues when discussing buffer size calculations?

SQL Injection Flaw

Because many applications interface with databases, an exploitable flaw [13] is possible when the database port is hard coded, default passwords are used at high levels of permission, or string concatation is used with "+" or even concat() and concatenate() functions or methods. An attacker can give malicious data, and the application builds an SQL query with the bad string, thus changing the semantics of the query. These are serious if there is no input validation. Usually, this is a serious problem because of the legal issues about privacy and protecting data confidentiality. [6, 9, 1, 13, 22]. Some questions that can be raised are: Do the texts raise security issues when discussing database connectivity? Do the texts raise security issues when discussing any kind of concatenation?

Command Injection Flaw

A command injection flaw [13] happens when the information entered at am application's prompt can be interpreted as a system command and calls an executable. This exploitation is possible anytime an input screen collects information and passes the information unvalidated to a system function. This frequently is a violation of the least privilege principle of security. [6, 9, 15, 13, 22]. Some questions that can be raised are: Do the texts raise security issues when discussing any Application Programming Interfaces (APIs) used to prompt for and gather user-supplied information? Do the texts raise security issues when discussing input validation? Do the texts discuss input validation in any detail?

Error (Exception) Handling Flaw

This is an exploitable situation if code does not properly deal with reporting of any errors or exceptions which may happen. The checking and catching of errors or exceptions are difficult because of the underlying complexities of the applications. The more complex a system is, the more untraceable an error or exception becomes. This exploitable shows up in the attempt to be more user-friendly by giving up too much information, especially what to do to correct the problem. The situation appears also if the error or exception is ignored, or misinterpreted as to its cause. It can happen if the application handles the wrong exception or error or if it tries to handle all possible errors or exceptions. [13]. Some questions that can be raised are: Do the texts raise security issues when discussing error or exception alerts and catches?

WHAT WAS FOUND

Java™ has security APIs, and a Cryptography Extension (JCE) for dealing with passwords and other things and makes it easier for programmers to deal with creating safe and secure code. It is also more difficult to impose security policies because of encapsulation of data and methods. Java also has the idea of “ sandboxing” especially with the use of applets [11, 15, 32, 7, 31].

Lewis, DePasquale and Chase [19] do not addres secure programming directly (see Table 1). While introducing a new Java method System.out.printf(), which most of the reviewed books do, they do not condone its use but do not really say why. They indicate that Java does array bounds checking with the array operator [ ] and indicate an ArrayoutofBoundsException will be thrown. They discuss the testing and code review of java classes in a chapter.

Bell and Parr [2] do discuss security issues (see Table 1). There is a section discussing that Java does not use pointers, has strong typing, and encapsulated data and methods, all of which present a strong security model. They indicate that while Java checks arrays bounds and subscripts, there is potential for a hacker to declare an array and use it to gain entry. There is a detailed chapter on testing where they treat “blackhat” and “whitehat” testing.

Deitel and Deitel [4] discuss Java’s sandbox security model when they present applet programming (see Table 1). They further discuss security measures when looking at network programming and mention Java’s Security API. They briefly discuss input validadation when dealing with regular expression coding. They present Java’s use of the printf() method but do not raise any securrity issues. They write about “arithmetic overflow” but not in a security context.

Johnson [12] argues that programmers should write well formed code (see Table 1). He also discusses the use the printf() method. He raises a security issue briefly when discussing visibility modifies of data and methods, writing: “Why is visibility important? It is a matter of security and integrity” [p. 553].

Reges and Stepp [27] discuss security and refers to intrusions by hackers [p. 35]. They discuss security problems with arrays (see Table 1) and present a buffer (array) overflow example and relate that these are “still [the]most common sources of computer security problems” [p. 383]. They discuss the “off-by-one” bug in sizing arrays and buffers.

Gaddis [8] writes about security issues (see Table 1) dealing with aggregate classes because they present problems similar to the pointer problems of C/C++. In his discussion of arrays, he present a discussion of how Java does array bounds checking, but raises a concern over “off-by-one” errors.

Liang [20] writes comprehensively about Java fundamentals but does not explicitly address security issues (see Table 1). He indirectly deals with security issues by focusing on well formed code. He discusses the printf() method and talks of runtime errors. He cautions about the “off-by-one” error when dealing with arrays. He discusses visibility modifiers and their use to prevent outside manipulation (the security conscious person would speak of “exploitation”) of data and methods. He indirectly discusses security issues in terms of thrown exceptions, including integer overflow.

Text	Lewis, DePasquale & Chase	Bell & Parr	Deitel & Deitel	Johnson	Gaddis	Reges & Stepp	Liang
General Remarks	0	1	1	1	1	1	1
Buffer(Array)Overrun	1	1	0	0	0	1	0
FormatString	0	0	0	0	0	0	0
IntegerOverflow	0	0	0	0	0	1	0
SQLInterjection	0	0	0	0	0	0	0
CommandInterjection	0	0	0	0	0	0	0
ErrorHandling	0	0	0	0	0	0	0
Testing	1	1	0	0	0	0	0

Table 1. Summarizing Java™ textbooks and security

C++ Textbooks face some different issues with regard to security. Textbooks suitable for a first course seldom deal with security directly, and often do not highlight problems which may occur. The most prevalent security issues are involved with pointers and array boundaries, old C-style strings, and functions with variable numbers of parameters.

Dale and Weems [3] (see Table 2) use the ANSI string class, and only add C-strings at the very end of the text. Students following the examples in the early part of the text wouldn't even know that these strings exist. In general consequences of bad practice are not discussed, but good practice is emphasized. All of the texts include C legacy issues in a relatively late chapter, explaining their use, but not overly emphasizing problems.

Most introductory texts feature examples from the DOS or unix shell prompt. They only indicate Data Validation as desirable, but seldom explore in detail the issues with improper data.

Exception handling is not included as part of Dale and Weems [3], but is a chapter in Liang [21], Savitch [28], and Deitel and Deitel [5] (see Table 2). These chapters are invariably late in the text, and are often skipped in introductory courses.

The three texts above all mention that C++ does not check array bounds, but do not emphasize the consequence of neglecting to check for boundaries. They suggest good practices, but not the consequence of bad practices. Deitel and Deitel [5] extensively note possible problems with failing to check.

Integer overflow (see Table 2) isn't a real problem in C++, as it doesn't occur. However the variables, if signed, become negative. None of the texts mention this behavior.

Command injection (see Table 2) is accepted as part of C, but neither the topic nor its consequences are generally explained in C++ textbooks.

The stdio.h functions are generally no longer included in C++ textbooks, so the inherent problems with scanf and printf having variable number of parameters aren't mentioned. Deitel and Deitel [5] have chapters on legacy topics, as does Liang [21] and Savitch [28]. Liang [21] doesn't inform students about the newer string class, and Dale and Weems [3] concentrate almost exclusively on it.. In general, the textbooks never emphasize the ability of programs to include legacy headers and features which may cause problems. Even when included, the texts merely explain how they are used.

Text	Dale & Weems	Savitch	Liang	Deitel & Deitel
General	1	1	1	1
BufferOverrun	1	1	1	1
FormatString	0	0	0	0
IntegerOverflow	0	0	0	0
SQLInterjection	0	0	0	0
CommandInterjection	0	0	0	0
ErrorHandling	0	1	1	1

Table 2. C++ textbooks and security

CONCLUSIONS AND RECOMMENDATIONS

A possible flaws model (buffer overrun, format string, integer overflow, SQL injection, command line injection, and exception handling) was constructed and discussed. Several selected introductory textbooks about Java and C++ programming concepts were reviewed and evaluated according to the model of possible flaws. Although very little about security issues was directly expressed, the sense of all of the texts is that if the student writes well formed code, security problems are minimized. But generally security is simply not discussed to any great extent. A recommendation is that authors add a "security note"(many already have all kinds of cautionary notes and tips) to various syntactic discussions. Further, the idea of testing and code review ought to be emphasized.

REFERENCES

1. Beaver, K. (2004). *Hacking for dummies*. Indianapolis, IN: Wiley Publishing.
2. Bell, D. & Parr, M. (2006). *Java™ for students, (5th Ed.)*. Harlow, England: Pearson Education Limited.
3. Dale, N. & Weems, C. (2005). *Programming in C++, (3rd Ed.)*. Boston, MA: Jones and Bartlett.
4. Deitel, P. J. & Deitel, H. M. (2007). *Java™: How to program, (7th Ed.)*. Upper Saddle River, NJ: Pearson Prentice Hall.
5. Deitel, H. M. & Deitel, P. J. (2001). C++: How to program, (3rd Ed.). Upper Saddle River, NJ: Pearson Prentice Hall.
6. Erickson, J. (2003). *Hacking: The art of exploitation*. San Francisco: No Starch Press.
7. Farley, J. & Crawford, W. (with Malani, P., Norman, J. G., & Gehtland, J.). (2006). *Java™ enterprise in a nutshell, (3rd Ed.)*. Sebastopol, CA: O'Reilly Media.
8. Gaddis, T. (2008). *Starting out with Java™: Early objects, (3rd Ed.)*. Boston, MA: Pearson Addison Wesley.
9. Graff, M. G. & van Wyk, K. R. (2003). *Secure coding*. Sebastopol, CA: O'Reilly Media.
10. Greene, S. S. (2006). *Security policies and procedures: Principles and practices*. Upper Saddle River, NJ: Pearson Prentice Hall.
11. Hartman, B., Flinn, D. J., Beznosov, K., & Kawamoto, S. (2003). *Mastering web services security*. Indianapolis, IN: Wiley.
12. Hoffman, L. J., Lawson-Jenkins, K., & Blum, J. (2006, July). Trust beyond security: An expanded trust model. *Communications of the ACM 49*(7), 95-101.
13. Howard, M., LeBlanc, D., & Viega, J. (2005). *19 deadly sins of software security: Programming flaws and how to fix them*. New York: McGraw-Hill/Osborne.
14. Johnson, R. A. (2007). *An introduction to Java™ programming and object-oriented application development*. Boston, MA: Thomson Course Technology.
15. Kalinovsky, A. (2004). *Covert Java™: Techniques for decompiling, patching, and reverse engineering*. Indianapolis, IN: Sams Publishing.
16. Kim, M. & Ahn, J. (2006, Fall). Comparison of trust sources of an online market-maker in the e-marketplace: Buyer's and seller's perspectives. *Journal of Computer Information Systems XLVII*(1), 84-94.
17. Kruck, G. P. & Kruck, S. E. (2006, Fall). Spoofing – a look at an evolving threat. *Journal of Computer Information Systems XLVII*(1), 95-100.
18. Kuperman, B. A., Brodley, C. E., Ozdoganoglu, H., Vijaykumar, T. N., & Jalote, A. (2005, November). Detection and prevention of stack buffer overflow attacks. *Communications of the ACM 48*(11), 51-56.
19. Lewis, J., DePasquale, P. J. & Chase, J. (2008). *Java™ foundations: Introduction to program design and data structures*. Boston, MA: Pearson Addison Wesley.
20. Liang, Y. D. (2007). *Introduction to Java™ programming: Comprehensive version, (6th Ed.)*. Upper saddle River, NJ: Pearson Prentice Hall.
21. Liang, Y. D. (2007). *Introduction to C++ programming: Comprehensive version.* Upper saddle River, NJ: Pearson Prentice Hall.
22. Lindner, F. (2006, June). Software security is software reliability. *Communications of the ACM 49*(6), 57-61.
23. Neumann, P. G. (2006, July). Risks relating to system compositions. *Communications of the ACM 49*(7), 128.
24. Norman, D. A. (1988). *The design of everyday things*. New York: Currency Doubleday.
25. Norretranders, T. (1991/1998). *The user illusion: Cutting consciousness down to size*. (J. Sydenham, Trans.). New York: Viking.
26. Pfleeger, C. P. (1997). *Security in computing, (2nd Ed.)*. Upper Saddle River, NJ: Prentice Hall.
27. Reges, S. & Stepp, M. (2008). *Building Java™ programs*. Boston, MA: Pearson Addison Wesley.
28. Savitch, W. (2006). *Problem solving with C++: The object of programming*, (6th Ed.). Boston, MA: Pearson Addison Wesley.

29. Schneier, B. (2000, December). Semantic network attacks. *Communications of the ACM 43*(12), 168.
30. Skovira, R. J. (2007). Framing the corporate security problem: The ecology of security. *Issues in Informing Science and Information Technology*. (Accepted for publication).
31. Stiegler, M., Karp, A. H., Yee, K. Close, T., & Miller, M. S. (2006, September). Polaris: Virus-safe computing for Windows XP. *Communications of the ACM 49*(9), 83-88.
32. Thorsteinson, P. & Ganesh, G. G. A. (2004). *.NET security and cryptography*. Upper Saddle River, NJ: Prentice Hall PTR.