Practical Methods for Software Security Education

J. Todd McDonald[1], Stuart H. Kurkowski[1], Richard A. Raines[1], Robert W. Bennington[1]
Department of Electrical and Computer Engineering
Center for Cyberspace Research
Air Force Institute of Technology
Wright Patterson AFB, OH 45433-7765
{jmcdonal, skurkows, rraines}@afit.edu, robert.bennington@wpafb.af.mil

Abstract

The Department of Defense (DoD) has vested interest in protection of application software critical to national security. As part of a holistic approach to train leaders in diverse aspects of cyber security, the Air Force Institute of Technology integrates secure software engineering principles throughout its computer science and cyber operations curricula. We present here a practical approach to teaching principles of secure software design as manifested through historical course offerings related to secure software development and actual student exercises that reinforce software security principles.

Keywords: Secure software, software engineering, curriculum development, cyber security education

1. Introduction.

In December 2001, the U.S. Deputy Under Secretary of Defense (Science and Technology) launched the Software Protection Initiative (SPI) with the goal of improving the protection of application software critical to national security. Based on one of the major SPI goals to train and educate the DoD community, the Air Force Institute of Technology (AFIT) partnered to incorporate a wide variety of secure software engineering principles into its educational curriculum. AFIT offers advanced academic degree programs in Computer Science (GCS), Computer Engineering (GCE), Cyber Operations (GCO), and Electrical Engineering (GE) to a wide variety of DoD and civilian students. Within the context of such degree programs, we outline in this paper the implementation of specific curriculum content designed to train current and future leaders in the art and science of secure software design. We discuss the layout of course material, exercises, and appropriate labs that allow hands-on learning of specific course objectives related to software security. We offer our approach to teaching secure coding principles to the greater academic community as one example of successful hands-on integration within an overall security-centric academic curriculum.

2. Educational Context.

AFIT currently offers a wide range of elective options for graduate students to fulfill their requirement for specific degree programs (GCS, GCE, GE, GCO). The *software engineering* sequence is one such option that highlights both theoretical and practical aspects of software development. The necessity of integrating robust programming practices and specific security-related techniques into software engineering curriculum cannot be understated [1]. Within the software engineering sequence itself and within various courses related to cyber operations, secure

[1] The views expressed in this article are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U.S. Government

software principles find their greatest prominence. We discuss in this paper specifically the content and design of one of our required software-engineering-sequence courses for our graduate students: *CSCE-526 Secure Software Design and Development*.

2.1. Curriculum Goals

Computer and network software is developed at a rapid pace, most often without the application of software security principles. Additionally, software often contains sensitive information that "needs" to be protected from reverse engineering. For many AFIT students, follow-on assignments or possible future career moves will place them in acquisition or technology related jobs within the DoD where software systems security plays a central role. Based on the DoD-centric view of the world held by a majority of our students, we see the need to provide curriculum where they may learn sound security principles that should be incorporated into the software development process.

In *CSCE-526*, we specifically look at common exploits that uncover fundamental security flaws in modern software applications. Because of its relevance to critical technologies, we also address software exploitation based on (malicious) reverse engineering and tampering attacks, addressing how current tools may implement various anti-tamper techniques. To provide a basis to understand current security trends and vulnerability assessment techniques, we address topics such as software security principles, security analysis techniques, buffer overruns, access controls, race conditions, input validation, network software security, and software protection/anti-tamper technologies. The ultimate goal is for students to understand the threats to software security, visualize how hackers exploit poorly written software, and practice how to actually implement countermeasures while realizing associated limitations.

2.2. Academic Foundations

CSCE-526 currently synthesizes elements from computer networking, operating systems, computer architecture, cryptography, and computer security. Several texts provide possible background material on secure coding and software security in general. Though we in no way can capture all possible texts which are applicable for such a diverse topic range, we do mention here the historical texts we have found to be beneficial for both academic and practical exploration.

Because of our emphasis on hands-on learning, Howard *et al.*'s text on the *19 Deadly Sins* text [2] has shown to be useful for mapping vulnerability analysis with laboratory exercises. The book provides reasonably-sized synopses of major software vulnerabilities along with appropriate references for further study. The "sins" themselves provide a manageable method for outlining specific laboratory exercises as well.

In historical offerings, Viega and McGraw's *Building Secure Software* [3] has served as an alternate text for outlining lab material across the course as well as Howard and LeBlanc's *Writing Secure Code* [4]. For supplemental texts, we have also found a wealth of practical examples for lecture material in *The Art of Software Security Assessment* [5], *Secure Programming w/ Static Analysis* [6], *Reversing: Secrets of Reverse Engineering* [7], and *Exploiting Software: How to Break Code* [8].

In addition to course text books which provide both practical code examples and a framework for organizing lab material, students in the *CSCE-526* course explore current literature and writing on exploits via a research project and a term paper. Students are expected to research one of the course topics and provide both in-class presentations and a final report that summarizes their research efforts. The research project focuses on applying real-world or work-related experience to one of the course topics, performing a demonstration of tools related to secure coding (static analysis tools, bug finders, code analyzers, obfuscation tools, reverse engineering tools, disassemblers, decompilers, etc.), or reviewing a single article or paper from a conference proceeding, book section, journal, technical

magazine, or security related website. The term paper itself is broader in scope and requires the student to explore a wide variety of literature on a related topic to software security, incorporate independent thought and analysis, and provide a research quality write-up of their findings.

3. Practical Learning Context.

As Figure 1 depicts, the misguided attempt of "little Bobby Tables" to change his grades at school can be *best* understood by someone who has actually attempted to perform a SQL injection attack (which evidently include Bobby's mother). In order to reinforce material from the course texts and provide a real framework to understand the concepts included in course lecture material, we organize *CSCE-526* lab work in a synergistic manner to achieve this goal. Our general approach takes one piece of code (a C program) that has a growing set of functionality. With each lab assignment, students either begin with or implement a particular piece of code, normally with an inherent weakness or "lack" of security. Students then attempt to apply a corrective action to the piece of code in order to address the security vulnerability that is demonstrated. The laboratory exercised map closely with the course text, and in this case we show how the *19 Sins* text may be easily integrated. After learning and applying specific secure coding practices, students then get to experiment with anti-tampering products such as obfuscators and digital rights management (DRM) tools. We discuss next the actual lab exercises.

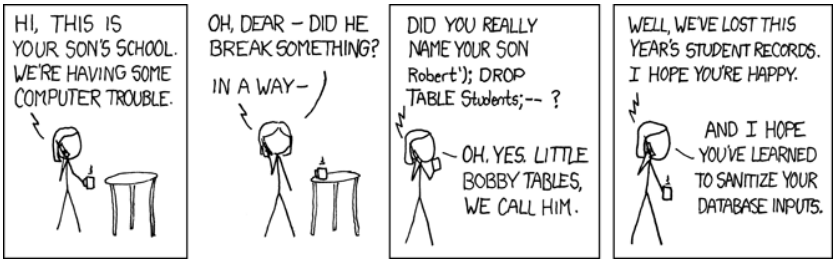

Figure 1: The Saga of Little Bobby Tables[2]

3.1. Laboratory Projects

With laboratory projects at the graduate level, there arises a tension between existing student skills in specific programming languages or environments and the desired course learning objectives. In the case of software security, the practical application of possible techniques could literally require skills and understanding in a wide variety of applicable languages and frameworks: C++, Java, C, C#, SQL, PHP, HTML, CGI, ASP, JavaScript, Perl, Python, JSP, Ruby, AJAX, XSLT, XML, and Visual Basic (only to name a few?). Of course, real-world exploits are typically complex and require expert knowledge in the *specific* software target environment.

For sake of time in the course, we level the playing field for students in accomplishing the *programming-related* lab work by limiting their required knowledge to that of C (which as a language enables some of the most horrific software exploitation possibilities) and the ability to use a Linux/UNIX development environment via standard GNU C[3] and Cygwin[4]. The Cygwin environment also provides a seamless ability for students with predominantly Windows-based

[2] http://blog.wired.com/monkeybites/2007/10/jokes-for-nerds.html

[3] http://gcc.gnu.org/

[4] http://www.cygwin.com/

computer access to work at home and in a lab-facility environment supported by the school itself. For exploits that would require a non-C environment (a SQL injection attack for example), the lab material provides specific non-programming related work that illustrates the exploits described in lecture and course text books. In some cases, we use specific tools that are provided for the students in a network-isolated laboratory for class-specific use (the use of some of these tools would alert standard network-monitoring tools or anti-virus software). Table 1 summarizes the lab work in the course and the correlation with the *19 Sins* text.

Table 1: Lab Summary

	Sin Covered	**Goals**
1	Simple Password Program	1) Write a program: prompt for password, read, compare with a hard-coded version
2	Buffer Overruns Format String Problems Integer Overflows	1) Intentionally cause the buffer to overflow; 2) Modify the program to remove the buffer overflow; 3) Intentionally cause a format string problem; 4) Remove the format vulnerability 5) Intentionally create an integer overflow problem by declaring an integer variable in your program, initializing the value of the integer to the max allowable for the compiler you are using, and then incrementing the integer value until an overflow occurs. Document what you observe. 6) Fix the overflow
3	*SQL Injection* Command Injection	1) Watch a video demonstrating web-based SQL inject vulnerability; 2) Incorporate provided C code that has a command inject weakness, run, and observe with specific inputs; 3) Fix the weakness
4	Failing to Handle Errors *Cross-Site Scripting*	1) Watch a video demonstrating cross-site attacks; 2) Incorporate C code that fails to handle a dynamic memory allocation error and then intentionally cause an error, observing what you see 3) Change the code to handle the error properly
5	Failing to Protect Network Traffic Use of Magic URLs	1) Use the *Cain and Abel* tool and demonstrate how you can crack a POP password (the students are required to sign up for a free email account or use their existing POP3 server) 2) Find a web site that used a hidden form field to hide the price of an item: print the source code out and explain it
6	*Improper Use of SSL and TLS* Use of Weak Password-Based	1) Given a URL, access the web site and report the error message encountered related to the web site's certificate 2) Document and use *Cain and Abel* to search for personal information and passwords on a local computer
7	Failing to Store/Protect Data Information Leakage	1) Use *OllyDbg* to open a classmate's executable program: find the memory location of the breakpoint used to do the password check and the hard coded password; 2) Use *nmap* to scan a particular host computer and correctly identify the web server being used
8	Improper File Access *Trusting DNS*	1) Modify your current C program so that it prompts the user for a file name in the current directory, then after reading the file, displays the contents; 2) Create a file in another directory and document results of attempting to open it from the current directory; 3) Fix the code so that only files in the current directory are readable 4) Write up a summary on a famous DNS spoofing incident and how it could have been prevented
9	Race Conditions *Unauthenticated Key Exchange*	1) Run a shell program that demonstrates a race condition and observe its behavior; 2) Demonstrate how a malicious user can link another file and observe your results 3) Write up a summary of a case where unauthenticated key exchange resulted in an exploit
10	Cryptographically Strong RNG *Poor Usability*	1) Write a sequence of programs that uses different types of random number generation, including entropy-based and pseudo-random generators: document observations; 2) Write up a summary that provides recommendations for specific Air Force computer systems that would have greater security if usability were considered

We begin the labs with a very simple student-written C program that prompts the user for a password, reads the password, compares the password to a hard-coded password in the program, then displays the results of the compare to the user (indicating if the password entered was correct or not). In the original version, we give no mention of security. Subsequent labs provide specific C code fragments to incorporate into this program, each of which embodies a vulnerability of interest. For the programming-specific examples, the students learn to cause or observe the specific

vulnerability, provide a remedy in code for the vulnerability, and document their learning experience. For the non-programming-specific cases (where C does not fully provide a good example space), students typically either use a tool of interest (*nmap*[5], *Cain and Abel*[6]) to demonstrate a vulnerability or they find specific instances of the vulnerability online and report on their observations.

3.2. Tool Experimentation

To provide overall context for software protection techniques in later labs, students use a variety of specific protection tools on their fully coded C program and then use debuggers and disassemblers such as *OllyDebug*[7] and *IDAPro*[8] to observe effects of various tools. In historical offerings, students have experimented with Arxan's *EnforcIT*[9], Aladdin's *HASP HL*[10] (a hardware-based protection), and Sofpro's *PC GUARD*[11]. In particular, they are given the same task of locating another classmate's hardcoded password when specific protection tools are used. They write-up the results of their observations and provide some analytical discussion on the type of protection mechanism that is used. We provide the tools in a laboratory environment to facilitate licensing and usage restrictions. As part of the project basis in the course, we also encourage students to evaluate and report on open-source or academic obfuscation tools such as University of Arizona's SandMark[12].

4. Summary

We present here an approach to teaching secure coding principles as part of an overall software engineering sequence curriculum. We believe that our approach to hands-on application and analytical understanding of the applicable theory provides one (good) example of practical software security education. We believe the exercises themselves may prove beneficial to educators desiring to integrate practical learning techniques into their own course material in the future.

5. Bibliography

[1] A. Yasinsac and J. T. McDonald, "Foundations for Security Aware Software Development Education," in *Proceedings of the Hawaii International Conference on System Sciences (HICSS'05)*, January 4-7, 2006.

[2] M. Howard, D. LeBlanc, and J. Viega, *19 Deadly Sins of Software Security*, McGraw Hill/Osborne, 2005.

[3] J. Viega and G. McGraw, *Building Secure Software: How to Avoid Security Problems the Right Way*, Addison-Wesley Professional Computing Series, 2001.

[4] M. Howard and D. LeBlanc, *Writing Secure Code, Second Edition*, Microsoft Press, 2002.

[5] M. Dowd, J. McDonald, and J. Schuh, *The Art of Software Security Assessment*, Pearson Education, 2007.

[6] B. Chess and J. West, *Secure Programming w/ Static Analysis*, Pearson Education, 2007.

[7] E. Eilam, *Reversing: Secrets of Reverse Engineering*, Wiley, 2005.

[8] G. Hoglund and G. McGraw, *Exploiting Software: How to Break Code*, Addison-Wesley Professional, 2004.

[5] http://nmap.org/

[6] http://www.oxid.it/cain.html

[7] http://www.ollydbg.de/

[8] http://www.hex-rays.com/idapro/

[9] http://www.arxan.com/anti-tamper/EnforcITPlatform.php

[10] http://www.aladdin.com/hasp/hasphl.aspx

[11] http://www.sofpro.com/pcgw32.htm

[12] http://sandmark.cs.arizona.edu/