Proceedings of the 10th Colloquium for Information Systems Security Education
University of Maryland, University College
Adelphi, MD June 5-8, 2006

Bottom-Up meets Top-Down: A New Paradigm for Software Engineering Instruction

Wm. Arthur Conklin, *Center for Infrastructure Assurance and Security, The University of Texas at San Antonio*

Abstract – College curricula for computer programming has been developed from a bottom up, primitive to system-level approach. Although efficient from a task centric viewpoint, this methodology leaves crucial learning tasks until after behavioral habits are reinforced through several courses of instruction. These habits are inadequate to meet the needs of current programming standards. The current learning process deemphasizes important issues such as security and testability, sacrificing them in the name of time. This paper outlines a new approach to provide a more comprehensive, systems engineering based education approach in an attempt to correct these deficiencies in the current instructional methodology.

Index terms – Software Engineering, Computer Programming Curriculum

I. Introduction

There has been significant interest in the development of secure software, programs that by definition do not suffer from the multitude of software vulnerabilities that are common in today's programs [1]. One avenue of attack is in the addition of secure coding classes into computer science and information systems curriculums. This paper approaches the problem from a different direction, from a systems point of view and holistic fashion. Current curriculums place security classes after basic coding classes. Rather than adding security instruction after coding habits have been formed, the concept of incorporating security related material from day one is examined. This information is targeted to the software engineering or computer programming instructor who is not a security specialist.

One method of addressing change in an organization is to answer four questions concerning the desired change. The four questions are; where are we now?, where do we want to be?, how will we get there? and how will we measure progress? Understanding the answers to these questions assists in the change process. These four questions will be used to guide the reader through an examination of an alternative approach that will transform computer programming classes into a software engineering based curriculum. This will require a change in paradigm for all parties involved, students and instructors.

Paradigms have been defined as models, examples or patterns that guide thoughts. According to Kuhn, paradigms are essential for the advancement of science [2]. In addition to paradigms, Kuhn suggests the role of revolution or upheaval as being occasionally needed for advancement. The current paradigm for computer programming instruction is one that begins with simple concepts or primitives and then slowly advances in complexity. This is a classic example of a bottom-up approach. Over a period of several courses, layers of complexity are added until a complete system-level solution is achieved in senior level courses.

A new model for software engineering instruction is proposed to address shortcoming of current instructional practices. The objective of this new approach is to present a complex subject in a fashion that highlights a wide array of issues. By including a wide range of critical issues at each level of instruction, a more complete picture of the total solution is presented to the student. Rather than starting with limited, partial views of a total solution that are centered upon a specific aspect of programming, this solution covers several aspects simultaneously.

Software engineering includes a wide range of concepts. Two of these are important with respect to the topic under investigation. The first is a concept referred to as primitives. This concept is that of basic coding functions learned in programming classes. The basics of I/O, data structures, control structures, and program flow, etc. are referred collectively as primitives. The second concept is one referred to as system issues. System issues are context related issues such as testability, maintainability, security and other higher level functionality. System-level issues are addressed using primitives, but also include elements of program design and structure.

Mixing system-level issues, such as requirements, security, testability and maintainability with each level of instruction presents a more comprehensive picture to the student. Students will by nature focus on what is graded for each assignment. A focus on individual elements,

Center for Infrastructure Assurance and Security, The University of Texas at San Antonio.

ISBN 1-933510-98-6/$15.00

data structures, control structures, objects, etc. over a series of courses places an over-emphasis on primitive level aspects versus system-level aspects of a solution. The end objective of a computer program is to act as a solution to what is typically a complex problem.

The habits formed from initial programming classes can persist for a long time. Having students focus repeatedly on issues of syntax, and primitive details of data structures, control structures, etc. forms habits associated with this level of concern. Higher level issues, testability, requirements, security and maintainability may be covered late in coursework, but never to the degree to form strong work habits. Future attempts to change programmer behavior will continually run against initial habits formed early in their educational experience.

For clarity, the remainder of this paper will address security issues as a system-level issue neglected by the current curriculum methodology. The principles being presented for security also apply to other critical system-level issues that plague today's software. The concept of spreading security across a curriculum is not a new concept and this paper is advocating specific actions to employ this security throughout concept to this specific domain [3, 4].

Domain specific knowledge associated with employing security principles in software is in abundance. Threat analysis, common software errors, industry standards and best practices are well publicized and known [5-8]. In spite of the knowledge, significant security issues still prevail in software today. This paper proposes that the primary problem lies not in knowledge, but the application of knowledge. This application of knowledge is driven by habit as much as it is driven by the knowledge itself.

II. Current Methodology

The first question; "where are we now," addresses the current state of a process. In this example, we are referring to the current state of computer programming instruction and the outcomes because of the current methodology. This question is used as a framing foundation, not in a critical fashion. Based on the information from this question, the gap between "is" and "desired" states can be determined.

Instruction in computer programming is an element of both computer science and information systems curricula. The emphasis may be slightly different between the two disciplines, yet both have remarkably similar patterns. The primary pattern employed in both curricula is a bottom-up approach. In this approach, students are taught a variety of primitive items, from I/O to data structures to control structures and a myriad of other programmatic elements. Algorithmic structure, in which a variety of algorithms are presented, including linked lists, remote procedure calls, sorting algorithms, tree searching and the like is also presented to the student, one concept at a time. These items are typically presented in isolation and later combined in small quantities through various specific programming assignments.

This method of instruction is easy and clear for both instructor and student. By isolating the issues, it is easier to present, easier to grade and easier for students to assimilate and learn concepts. This tried and true methodology has been used in a variety of curricula, including computer programming, and many engineering disciplines. This methodology works because there exists a need to understand these primitive items before constructs combining them can be understood and developed. These basic building blocks are key to learning how programs are designed and constructed. However, just understanding the primitive functions from which a program is crafted is only part of the story.

Interviews with recent graduates have led to some interesting findings [9]. Although they state they are aware of system-level concerns in programming, such as specific elements associated with security, they are viewed as a firm level issue as opposed to an individual responsibility. The former students stated that they were constrained by the system within which they were employed. The system within which they used their programming skills was not borne of a vacuum, but was developed by previous programmers and management. These two guiding forces, senior programmers and management are typically products of the same educational system that creates the current generation of programmers.

The current curriculum in computer programming can thus be said to have two products, the programmers and the system they work within. The strengths of this methodology are reinforced throughout the system that creates modern software programs. However, just as the strengths become ingrained, so do any weaknesses and an examination of the security issues associated with current software suggests that there are problems with our programming approach.

One potential weakness of the current system is the delaying of system-level issues until late in the curriculum cycle, well after initial programming habits are established and reinforced. Managing a project to specific requirements and programming for just requirements is a more difficult task than the typical classroom assignment. For instance, in a class involving validation of screen controls for specific content, the student's project is graded based on whether or not a set

of requirements are met. Does the zip code field only allow numerical entries of the correct size and format? Is the entry in an email box of the correct format? These are typical concerns and easily graded. What about security requirements? Does the student screen allow cross site scripting attacks, or will it allow command or SQL injections? Will a student constructed program be susceptible to buffer overflows or race condition attacks?

These are all valid concerns in the business of software engineering. They are products typically of system-level and design issues, not just coding errors. Although students may be warned of them, and have these issues covered in a senior level course when their skill level is sufficient to understand them, the years of habits associated with previous programming assignments will be the primary influence of their behavior. In a student's first job, with a typical work process that is also by-product of the same education methodology, one driven by coding primitives ability, the student is rewarded again for coding behavior vice system-level behaviors.

III. Desired End-State

The question; "where do we want to be," addresses the desired end-state of a process. In this example, the desired end-state is a computer programmer with a good understanding of primitive functions and the system-level context within which they need to be employed to produce a secure solution. This "and" condition is not a trade-off or a balance, but a more comprehensive goal. As will be proposed later, this increase in ability can be achieved with minor increases in time and efforts over the course of an entire curriculum.

The desired end-state of a software engineering based curriculum is to produce students with the necessary skills to engineer the software solutions needed in the current environment. Numerous specific characteristics are desired in a graduate of a software engineering program [3]. Of particular interest in light of the subject of this article are the concepts of coding primitives and system-level issues. There are numerous other characteristics, including ethics, business knowledge, etc., but for the purposes of this paper only coding primitives and system-level issues will be addressed.

Coding primitives encompasses the body of knowledge associated with language, syntax and algorithmic issues. This is the base knowledge necessary to create the elements of a computer program. There are some specific security habits associated with this level of knowledge, i.e. replacing gets() with fgets() in c programs [10, 11]. A firm grasp of algorithms and data structures, i.e. linked list, binary tree traversal, etc., the building blocks of computer programs, is needed to construct software solutions. This knowledge base has both language independent and language dependent concepts. Syntax is the most common language dependent construct, but objects and levels of operating system access can also be language dependent. Algorithms and many data structures are language independent. These topics are typically well covered in most curricula.

The system-level issues desired from software engineers include items such as maintainability, security and testability issues. These can be viewed as contextual issues that typically result from design criteria and requirements. Having software engineers capable of addressing these context issues is not unlike expecting college students to be able to properly write a paper or report. Just as a history professor should take off points from a poorly composed, poorly written paper, so too should a computer programming instructor reduce a grade for a programming project that has security issues or other system-level issues. Increasing levels of understanding and accountability of system issues should be incorporated into curricula from the earliest stages. Beginning programs should have clear documentation, then design elements and security slowly introduced. The objective of system-level instruction is the creation of good system-level behavioral habits. These habits take time to create and reinforce, hence this level of thinking must occur throughout the curriculum, not just at the end.

The understanding of primitives and the context within which they need to be employed has been successfully achieved in other engineering curriculums. Electrical engineering students learn the basics of electricity, of fields and waves, of power laws and transmission principles and learn the complete system aspect before they enter practice. They are trained to understand the context of their designs and the roles that the primitive elements play in creating the solution. This is achieved partly through system engineering classes and emphasis in senior design classes. These elements assist students in developing skills necessary for proper comprehensive designs, covering both primitive elements and system-level constructs.

In addition to system-level instruction near the end of the curriculum, other elements are built in throughout the entire curriculum. Concepts of risk analysis, failure mode effects analysis and reliability and other system-level effects are essential elements of an engineering curriculum and infused throughout the material during the entire education process. The demonstration of an infusion of system-level thinking alongside primitive manipulation skills in other engineering curricula clearly shows that this option should be available for software engineering curricula.

The last aspect of the desired end-state is one of compatibility with the system in which software engineers act. Software engineers produce software within a process involving other programmers, designers, architects, testers and other team members. It is important that the new additions to this team are capable of contributing appropriately in the team process.

IV. Proposed Methodology

The question; "how do we get there," addresses the path between the initial state and the desired end-state of a process. In this example, the path is the set of proposed changes necessary to achieve the desired end state, specifically what are the curriculum changes and when do they occur. As stated earlier, one of the key issues to be addressed is the formation of a habit forming behavioral environment. This environment requires time and repetition to achieve habit formation.

Repetition by nature is a time consuming process. One method of learning is through experiential repetitive activities. A series of homework exercises and class projects over the course of years leaves a lasting impression upon a student. Currently, this repetitive experience is based around primitives and algorithms, with system-level issues only entering at the end of the total experience. This provides a lot of opportunity for issues associated with items such as data structures, I/O, and other primitives, but significantly limits learning opportunities for system-level constructs as security.

The proposed methodology is aimed directly at the heart of this issue; increase repetitive opportunity for system-level issues. By including a series of system related instructional objectives, beginning with early lessons and repeating them over time will increase learning and retention of these valuable skills. Early programming assignments should have simple constructs, such as good documentation. With primitives such as 'Try – Catch – Finally' blocks comes the opportunity to address error trapping and handling. With function inputs comes the opportunity to address input validation, and buffer overflows. Rather than save security functionality for the security course at the end of the curriculum, intersperse it throughout the programming courses.

Including context based system-level thinking and learning in a primitives-based class challenges students on different, yet complimentary levels. Bloom's taxonomy of learning characterizes several types of learning. Bloom believed that education should focus on 'mastery' of subjects and the promotion of higher forms of thinking, rather than a utilitarian approach to simply transferring facts [12]. The primitives-based material is primarily a knowledge type learning task. This task is centered around rote memorization or recall of information. Systems level material is learned using a combination of comprehension and application learning styles. Comprehension based learning involves understanding the meaning and interpretation of instructions and problems within a work context. Application based learning involves applying material in novel situations in the work place.

The following are a set of examples to illustrate the desired combination of primitives and system-level concepts in practice.

Example 1: Pass by value, pass by reference and input validation. When the concept of passing by value vs. passing by reference is covered, this is an ideal time to illustrate the effects of bad parameters on a function. Demonstrate how when a function is expecting one specific type of input, and it gets a different type of input, that the results can be catastrophic. This is the ideal place to introduce the students to the concept of input validation and its role in ensuring secure code. Although one could form a whole lesson on input validation, the idea is to introduce the concept with the primitive concept of "pass by" and then to reinforce the lesson on later assignments. Once input validation has been introduced, all future assignments should have some aspect of this concept included to assist in repetitive learning.

Example 2: Try-Catch-Finally and error trapping. When the concept of program blocks with 'Try – Catch – Finally' is covered, this is an ideal time to discuss error trapping and handling. Again, the objective is to gently introduce the student to the system concept and over the course of future assignments reinforce and refine the system-level concept. The first assignment might have a simple error catch, whereas future assignments can catch and parse multiple errors and require a more comprehensive solution to the system-level issue.

With each of these examples, the pattern is to introduce the concept in simplest form first. Then, with each successive example, build upon the already introduced system examples while introducing the new set of primitives. As the student's repertoire of primitives knowledge increases, so does the student's practice of building system-level solutions. Over time, this increasing level of complexity and repetitive activity will increase retention of these concepts.

Software engineers that are products of this new process will need to become productive members of software engineering teams in the workplace. This proposed methodology will produce a product that has the same characteristics as the current process, but also enhancements that make a student a stronger member of

the team. The commonalities with the current methodology make the student fit with other members of the team. The additional material from a systems viewpoint enable the student to contribute in additional positive fashion that will further desired progress towards secure code and improved processes.

V. Challenges

The major challenge associated with any curriculum change is where and how to fit new requirements into an already packed time schedule. Without adding more time, how can more be asked in terms of content delivery? It is clear that this paper proposes adding additional material into a whole series of courses. The addition of material, when accomplished via simultaneous presentation is not a directly linear function with respect to presentation time. Using system-level material as contextual examples and including it into homework and projects does increase the complexity of the projects for both the student and the instructor. The increase is not significant in overall course presentation length.

The increase in material complexity for the student will result in more time consuming assignments. But this extra time is rewarded through better learning. By engaging more than a single learning style and through the coordination of basic knowledge and system-level knowledge, the level of student comprehension is increased dramatically. The sustained use of this practice across an entire curriculum engages the power of behavioral habits to create a lasting learning experience.

The additional workload on an instructor can be significant. Most learning materials, books, presentations and supporting materials are focused on the primitives-based material and rote learning methods. The additional material needed to present context based examples that emphasize system-level concepts as security require the instructor to prepare and present them. This can be a significant amount of work. Changing lesson plans to include the examples, changing homework and laboratory assignments to include system-level concepts in grading is also significant amounts of work. This work does benefit from the ability to reuse the work. Once it has been developed and presented, it can be reused from one class to the next.

VI. Conclusions

Using concepts from Bloom and practical knowledge from other engineering disciplines, the path to a more comprehensive software engineering education is clear. Rather than concentrate on the bottom and work up the hierarchy of complexity, or attempt a top down explanation; it is proposed that that both directions be employed simultaneously. Bottom up to provide the base level knowledge needed to understand the basics of the material. Top down to understand the correct context to employ the basics.

This paper is targeted to non-security specialists. There has been literature on the topic of infusing security across a computer science curriculum [3, 4, 13]. The majority of instructors of computer programming and software engineering courses are not security specialists and the concept of infusing security across and through a curriculum may be new to them. In essence, this proposed methodology is just a replication of the concept of spreading security throughout a CS curriculum into the software engineering domain.

In closing it is worth noting that the current industry methodology is to use software engineering processes such as CMM [14] to manage process improvements associated with software engineering. This is not contrary to the objectives of the proposed methodology. CMM requires knowledgeable input from the participants in the process. Although some instructors already follow this proposed methodology, and work processes such as CMM exist to further develop software engineers into more skilled and more experienced developers, the evidence of current software products is clear. We have a problem with developing secure software, and if the current system isn't addressing the problem, then we need to look at ways to enhance it to achieve a different outcome. This proposed solution is just such an enhancement, one that is already being done by some, is compatible with the existing software engineering community and is achievable by all.

VII. References

[1] S. E. Institute, "Build Security In," vol. 2006: Strategic Initiatives Branch of the National Cyber Security Division (NCSD) of the Department of Homeland Security (DHS) https://buildsecurityin.us-cert.gov/portal/, 2006.

[2] T. S. Kuhn, *The Structure of Scientific Revolutions*, 3rd edition ed: University Of Chicago Press, 1996.

[3] C. E. Irvine, S.-K. Chin, and D. Frincke, "Integrating Security into the Curriculum," *IEEE Computer*, pp. pp. 25-30, 1998.

[4] G. White and G. Nordstrom, "Security Across the Curriculum: Using Computer Security to Teach Computer Science Principles," presented at Proc. 19th National Information Systems Security Conference, Baltimore, Md., 1996.

[5] M. Howard and D. C. LeBlanc, *Writing Secure Code*, Second Edition ed: Microsoft Press, 2002.
[6] F. Swiderski and W. Snyder, *Threat Modeling*: Microsoft Press, 2004.
[7] M. Howard, D. LeBlanc, and J. Viega, *19 Deadly Sins of Software Security*: McGraw-Hill Osborne Media, 2005.
[8] G. McGraw, *Software Security: Building Security In*: Addison-Wesley Professional, 2006.
[9] Students, "Personal Communication: Discussion on secure coding practices with graduate students and former students," A. Conklin, Ed., Fall semester 2005.
[10] M. Ranum, "Security bug? My programming language made me do it!" in *ACM Queue*, vol. 2, 2004.
[11] J. Viega and G. McGraw, *Building Secure Software: How to Avoid Security Problems the Right Way*. Boston, MA: Addison-Wesley Professional, 2001.
[12] B. S. Bloom, *Taxonomy of Educational Objectives, Handbook I: The Cognitive Domain*. New York: David McKay Co Inc., 1956.
[13] E. Crowley, "Information System Security Curricula Development," presented at CITC4 03, Lafayette, Indiana, USA., 2003.
[14] M. C. Paulk, B. Curtis, M. B. Chrissis, and C. V. Weber, "CMM Version 1.1," *IEEE Software*, vol. 10, pp. 18-27, 1993.