Software Reengineering Based Security Teaching

Sam Chung
Computing & Software Systems
Institute of Technology
University of Washington
Tacoma, WA 98402
chungsa@uw.edu

Barbara Endicott-Popovsky
Center for Information Assurance and Cyber Security
University of Washington
Seattle, WA 98195
endicott@uw.edu

ABSTRACT

The problem of developing secure code is well known to high-tech sector companies. Some, like Microsoft, have found it necessary to establish ongoing security training for their developers to make up for the absence of college-level, secure coding curriculum. This research takes a unique, software reengineering-based, thread approach. Curriculum modules are built around a concept such as input data validation, encapsulation, errors, etc. A software engineering case study is developed for each module that will produce code the traditional way, without regard to security, then re-engineer the code to transform it to include security concepts. Going through the cases in this manner, will give attending faculty, not only specific labs they can implement in their own courses, but also an understanding of how to transform their own existing assignments to incorporate secure coding practices.

Keywords: Secure Coding, 4+1 Views, 5W1H Based Re-Documnetation Technique, Reverse Software Engineering, Forward Software Engineering

1. INTRODUCTION

The problem and importance of developing secure code is notorious in the high-tech sector. Mike Howard, Principal Security Program Manager at Microsoft, notes that due to ever evolving code and the constant vigilance of hackers, "security is a never ending battle" and stresses the importance of providing ongoing training in secure programming[1]. For high-tech companies like Microsoft that employ large populations of developers, testers, engineers, and program managers, the costs of ongoing security training are high, but the cost of not implementing such training is always higher.

At the University of Washington's Center for Information Assurance and Cybersecurity (CIAC), an NSA/DHS Center of Academic Excellence in Information Assurance Education and Research, the authors believe that the absence of college level secure coding curriculum, along with instructor unfamiliarity with security, are the major obstacles preventing more schools from teaching this important set of skills to their students.

Consequently, the CIAC is proposing an approach to educating and training faculty members who are not familiar with security concepts and who must teach many non-traditional students who were not exposed to security concepts from the beginning of their CS/IS education and across the curriculum. A pilot program is proposed that will train seven faculty members per year for two years who will reach over 1200 computer science students in the Puget Sound area. Success of this program, as determined by internal and external evaluation, will allow broadening this reach, making this work available to more faculty in-, and outside, the Pacific Northwest.

2. PREVIOUS WORKS

It is estimated that 90 percent of reported security incidents result from exploits against defects in the design or code of commonly used software [7]. Furthermore, the threats from exploited software vulnerabilities are growing, along with the number of known vulnerabilities. Carnegie Mellon University's Computer Emergency Response Team (CMU-CERT) provides a yearly report on the number of software vulnerabilities which has grown steadily over the past decade and hit an all-time high of 8,064 in 2006 dropping slightly in 2007 to 7,236 [3], a 42-fold increase from 1995 when they began keeping these statistics. By improving the education of computer scientists, there should be a significant reduction in the number of software vulnerabilities produced in their code.

There have been three approaches to instruction in secure coding [13, 20]: the single-course approach, the track

[1] Howard, Michael. "Lessons Learned from Five Years of Building More Secure Software". MSDN Magazine. November 2007. http://msdn.microsoft.com/en-us/magazine/cc163310.aspx

approach, and the thread approach. Table 1 shows comparisons of the three approaches.

Table 1: Comparisons of the three approaches to undergraduate instruction in secure coding

Criteria	Single-Course	Track	Thread
The depth of learning	Low	High	Medium
The breadth of learning	Low	High	Medium
The demand of resources	Medium, a security faculty member is demanded.	High, multiple faculty members are demanded.	Low, current faculty members are retrained.
The change of CS curriculum	Medium, only one elective course needs to be added.	Yes, multiple new courses need to be added.	Low, the security concepts are integrated into current courses.
Across the CS core curriculum	Very Low	Medium	High
Across the CS elective curriculum	Low	Medium	High
Transferring across institutions	Medium	Low	High

Recently, the thread approach, which integrates security concepts into existing Computer Science (CS) and Information Systems (IS) curriculum, has been recognized as effective, without causing severe expense of depth and breadth of a complete curriculum change for resource-limited institutions. This approach has a strong advantage in that a complete thread can be transferred more easily across both core and elective curricula, as well as different kinds of institutions. There is no need for introduction of complete new courses and the internal curriculum review process that may slow implementation. Several successful attempts at the thread approach have been reported [16, 2, 17, 18].

3. PROPOSED SOLUTIONS

In this project, the authors propose to develop/implement a secure software reengineering-based thread approach. By this is meant the development of software engineering case studies that produce code the traditional way, without regard to security measures, then demonstrating how these same case studies can be transformed to include security across the life cycle, resulting in secure code. Going through several cases in this manner will give instructors taking the authors' workshops, not only specific labs they can implement in their own courses, but also an understanding of how they can transform their own existing assignments to incorporate secure coding practices. While some instructors may wish to adopt the cases introduced in the workshop for their own courses, it is recognized that many are committed to exercises they have developed in years past. Therefore efforts will be directed toward assisting them in converting their existing lab assignments to ones that reflect secure coding practices.

The development of the project's teaching modules takes into consideration life cycle concepts from the approach to software assurance proposed by the Department of Homeland Security (DHS) National Cyber Security Division [6]. DHS stresses a multi-faceted approach, which includes the following components:

- People - Education and training for developers and users.
- Processes - Practical guidelines and best practices for the development of secure software.
- Technology - Tools for evaluating software vulnerabilities and quality.

People –Non-Traditional Students and Instructors Unfamiliar with Security

The approach reflects the unique student bodies the authors are choosing to address. The University of Washington at Tacoma Computer Science department caters to transfer students from two-year programs and returning adults interested in software engineering as a career change. The University of Washington Seattle campus Information School caters to a similar student body. Faculty on both campuses who are teaching these students will be engaged in this project, as well as faculty from two-year institutions that produce the students who transfer to both schools, in an effort to build a common understanding of secure coding practices across a four-year program, regardless from where students come.

Schools that offer computer security subjects may offer a course or two in secure coding principles [7, 16], typically targeted toward upper division students after they have completed a number of programming courses and have developed insecure programming habits. The justification for teaching secure programming to more seasoned students is that novice programmers couldn't understand the concepts underlying defensive coding practices. Yet these same students may be more difficult to penetrate, having ingrained bad programming habits already.
An alternative is to present secure coding principles in the beginning courses, before student form habits, and continue reinforcing these concepts in more advanced courses. The advantage of this approach is that introducing secure programming concepts to new programmers from the

beginning of their programming careers avoids the problem of having to retrain them later after having developed bad programming habits. Secure coding can be introduced to beginning students if coding examples stress the consequences of insecure input and explain the dangers of using insecure function calls [18]. The introductory concepts don't have to be sophisticated to be effective.

Between the two approaches–teaching secure programming later versus earlier in a student's education–the authors believe that security should be taught from the beginning. While there is not a substantial amount of data to support this belief [16, 2], the literature on performance and human errors reinforces this view. While ideal, this approach cannot be applied to 4-year universities who accept many transfer students from local community colleges and/or master-level students who already have earned CS degrees without learning security and information assurance through their degree programs. The approach presented in this paper addresses these kinds of students.

A survey of the security education literature and a review of the current courses in the authors' respective schools and in the region reveal that there is no standard or consistent approach described for teaching non-traditional students who transfer from local community colleges to a four-year college or who apply for a professional masters program accepting students who are non-CS majors. These are the students the respective schools in the pilot study propose to reach. The University of Washington (UW) Seattle and Tacoma campuses accept many transfer students from 35 Community and Technical Colleges (CTC)[2]. In particular, UW Tacoma was founded as an urban university that supports the 2+2 model for transfer students (two years of community college and two years of a four-year college program) and also offers a professional MS degree program for non-CS majors, most of whom have *not* been exposed to computer security or information assurance in their programs. The authors believe this unique body of students, as yet undescribed in the literature, can be reached and that there is a great need for education in secure coding proposed to be provided.

Process – Transforming Insecure Lab Assignments to Secure Ones

Without inventing new secure coding electives, secure coding concepts and practicum can be injected into existing core or elective courses through programming assignments. In order to minimize the changes to existing computer science curriculum, injection of secure coding concepts into existing lab assignment as a means of modernizing/transforming requirements is recommended. Many references to secure coding projects are currently available [1, 5, 9, 10, 11, 12, 14, 19]; however, all of them focus on either specific secure coding techniques or forward software development from given security requirements to secure applications. To be effective at convincing faculty to adopt secure coding practices, the authors wish to demonstrate how simple it is to transform existing assignments into secure coding assignments, thus minimizing the changes needed for their teaching materials. The methodology proposed will be the subject of publications discussing the need to effectively deal with natural instructor resistance to change that adopting secure coding practices creates.

The approach to transforming existing curriculum is similar to software reengineering which is "the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form [4]." Software reengineering consists of two processes: reverse software engineering and forward software engineering. Through reverse engineering, students gain a sufficiently high-level of understanding of how to discover the weaknesses of an application's code through security attack testing procedures such as white/black hat testing, penetration testing, etc. The insecure legacy application is then "modernized" to become a secure target application by answering the discovered security weaknesses and implementing new security requirements.

In the forward software engineering process, secure coding topics are covered, regardless of the programming language: validation of all inputs, insuring files are handled correctly, insuring variables are well defined with the proper types and that memory is allocated and collected depending on the language used, cross-site script vulnerabilities, SQL injection which affects most modern databases, script vulnerabilities in multiple common scripting languages are all introduced. Figure 1 (following the reference section) shows the conceptual description of software reengineering for lab assignments.

By stressing the security implications of incorrect programming techniques as these subjects are introduced, students can be exposed to secure programming at the same time they are first learning the material. It is not more difficult to design programs, exercises and labs that stress security as opposed to using other types of examples. In fact, one can argue that the security purpose behind correct handling of inputs, memory and files adds an extra element of motivation since the consequences of incorrect programming can be dire. Examples from real programs will be used to illustrate both the correct and incorrect ways to write programs. Since students taking these courses are assumed to already have a programming basis, more advanced concepts can be addressed dealing with dependable distributed system design issues, among others.

[2] Washington State Board for Community and Technical Colleges (SBCTC), http://www.sbctc.ctc.edu/ (Accessed on 2/14/08)

Technology – Visualizing, Specifying, Constructing, and Documenting Lab Assignments

Teaching modules developed through this grant will not only benefit the institutions directly involved in creating these teaching materials, but will be disseminated to faculty in other institutions. Activities fall under the Institutional Faculty Development component of the Capacity Building portion of the Scholarship for Services (SFS) Program announcement which requires dissemination to outside faculty. One of the main motivations for developing these secure code modules is to assist as many faculty members as possible with teaching secure coding to their students.

In order to address a larger potential population of students and CS educators, in alignment with different institutional interests, the workshop curriculum will focus on effective understanding of the software reengineering process and lab assignments by using visual diagrams in Unified Modeling Language (UML) [8] during the software reengineering process. UML is a general-purpose modeling language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. In addition to the handouts of lab assignments with source code, UML diagrams are employed to enhance clear understanding on the legacy and the target systems.

Since UML has been widely adopted in CS courses to help students to understand object-oriented concepts, it can be used without having serious difficulties in order to discuss the visualization, specification, construction, and documentation of how a legacy system has been made more secured in the target system. For example, a class diagram of a target system shows how design of the target system has been improved to handle files correctly. A use case diagram of the target system shows how security requirements are described and managed. A state chart diagram for a form in the target system shows how all inputs are validated by events.

4. CASE STUDY

The Information Technology and Systems (ITS) program at the University of Washington (UW) Tacoma offers a programming course for junior students called TINFO 300. Most of the students in TINFO 300 have transferred from local community colleges. Since the students already have studied an introductory level programming course before taking TINFO 300, they study object-oriented analysis and design in this course. One of the homework assignments is to develop a Windows GUI (Graphical User Interface) application in the C# .NET object-oriented programming language. In Figure 2, a user invests $100 every month for 1 year earning a 1% yearly interest rate. The future value of the investment is calculated when the 'Calculate' button is pressed. The source code of this example comes from Joel Murach's C# 2008[3].

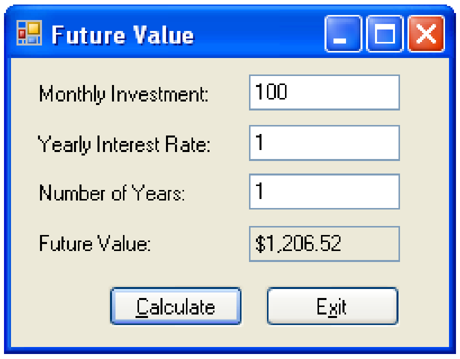
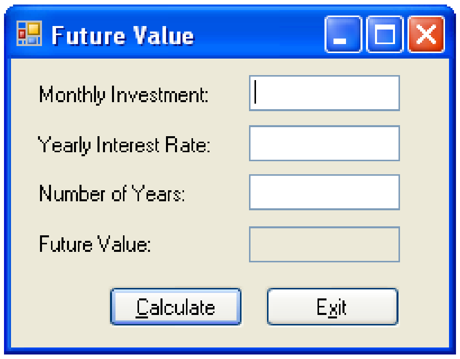
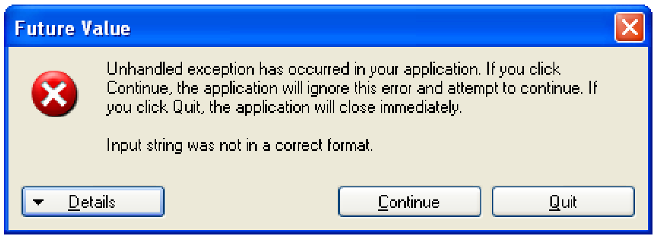

Future Value
Monthly Investment: 100
Yearly Interest Rate: 1
Number of Years: 1
Future Value: $1,206.52

Figure 2: A desktop application to calculate a monthly investment (an insecure legacy system)


Future Value
Monthly Investment:
Yearly Interest Rate:
Number of Years:
Future Value:


Future Value

Unhandled exception has occurred in your application. If you click Continue, the application will ignore this error and attempt to continue. If you click Quit, the application will close immediately.

Input string was not in a correct format.

Figure 3: A desktop application is executed without any input data (an insecure legacy system).

The homework consists of 4 problems: The first problem is to implement the desktop application without input validation. Although input data validation is one of the most important and simple countermeasures against external threats, this technique has not been emphasized in traditional programming courses. And then, test cases using invalid data are tested. For example, Figure 3 shows that the implemented future value calculation system is vulnerable to an invalid input data threat. Without entering any data, an attacker can crash the system.

[3] Joel Murach. Murach's C# 2008. Mike Murach & Associates, Inc. (978-1-890774-46-2)

The second problem is to analyze the insecure legacy system by using both class and sequence diagrams. The class of 'Form' in Figure 4 shows that the class does not have any input data validation methods at all. Also, the sequence diagram in Figure 5 (after the reference section) describes that the '*CalculateFutureValue( )*' method is invoked without having any filtering for invalid input data.

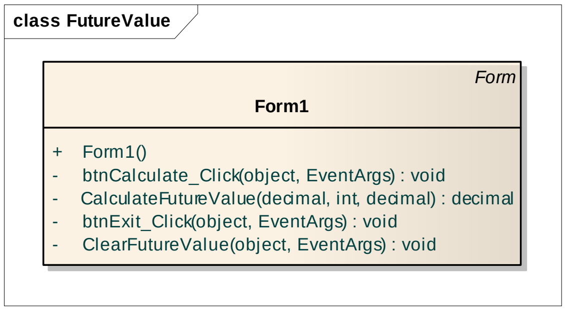

class FutureValue
Form
Form1
+ Form1()
- btnCalculate_Click(object, EventArgs) : void
- CalculateFutureValue(decimal, int, decimal) : decimal
- btnExit_Click(object, EventArgs) : void
- ClearFutureValue(object, EventArgs) : void

Figure 4: A class invoking the '*CalculateFutureValue( )*' without input data validation (an insecure legacy system)

The third problem is to design a secure target system in which the invalid input data vulnerability is counter-measured by data validation methods. Figure 6 shows that there are input validation methods such as '*IsValidData( )*', '*IsPresent( )*', etc. These methods are invoked before the user presses the 'Calculate' button, which will call the '*CalculateFutureValue( )*' method. This sequence diagram in Figure 7 (after the reference section) describes that the '*CalculateFutureValue( )*' method is invoked after input data validation.

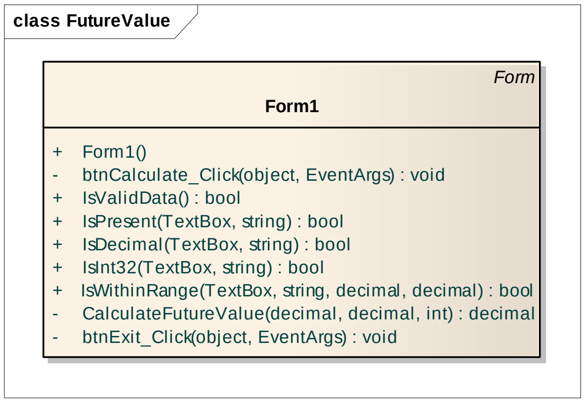
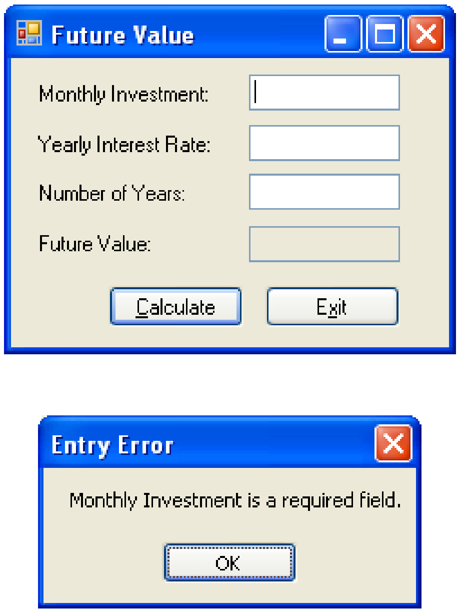

class FutureValue
Form
Form1
+ Form1()
- btnCalculate_Click(object, EventArgs) : void
+ IsValidData() : bool
+ IsPresent(TextBox, string) : bool
+ IsDecimal(TextBox, string) : bool
+ IsInt32(TextBox, string) : bool
+ IsWithinRange(TextBox, string, decimal, decimal) : bool
- CalculateFutureValue(decimal, decimal, int) : decimal
- btnExit_Click(object, EventArgs) : void

Figure 6: A class invoking the '*CalculateFutureValue( )*' with input data validation (a secure target system)

The last problem is to demonstrate that the target system is secure. Students in TINFO 300 implement the input data validation methods and test the newly implemented secure target system against the invalid input data threats. Figure 8 shows that the null input data threat is detected and a warning message is displayed.



Monthly Investment is a required field.

Figure 8: A desktop application is executed with input data (a secure target system).

5. DISCUSSIONS

The software re-engineering approach described above is unique to this secure coding research and designed to empower faculty taking workshops to transform their own curriculum, In addition, it is the intention of this research to provide curriculum threads that bridge community colleges and traditional four-year schools. While much attention has been given to the value of teaching secure coding principles from the beginning of a CS curriculum, little has been described about how this might occur in the case of transfer students coming from community colleges and returning adults who have already formed habits to develop insecure code.

It is the purpose of this research to optimize the approach described in this paper through a series of faculty workshops for which robust evaluation will be performed. Future work will involve describing these results to a broader academic community to encourage adoption of practices that will make it relatively easy to convert existing programming curriculum to reflect secure coding practices.

ACKNOWLEDGMENT

This research has been supported by the NSF (National Science Foundation) DUE (Division of Undergraduate

Education) Federal Cyber Service: Scholarship for Service (SFS) under Grant No. #0912109.

References

[1] Anderson, R., (2008). Security Engineering: A Guide to Building Dependable Distributed Systems, (2nd ed.). New York: John Wiley & Sons.

[2] Bishop, M. and B. J. Orvis. (2006). *A Clinic to Teach Good Programming Practices*. Proceedings of the 10th Colloquium for Information Systems Security Education, University of Maryland: Adelphi, MD, pp. 168-174.

[3] CERT Coordination Center (CERT/CC). (2008). *Vulnerability Remediation Statistics*. Available at: http://www.cert.org/stats/vulnerability_remediation.html, (Access on 11/11/2008)

[4] Chikofsky, E. J. and H.J. Cross II. *Reverse Engineering and Design Recovery: A Taxonomy*, IEEE Software, Vol. 7, No. 1, January 1990, pp. 13-17.

[5] Chess, B., and J. West. (2007). Secure Programming with Static Analysis. New York: Addison-Wesley Professional.

[6] DHS (Department of Homeland Security) National Cyber Security Division. (2005). *Build Security In: Setting a Higher Standard for Software Assurance*. Available at: https://buildsecurityin.us-cert.gov/daisy/bsi/mission.html (Accessed on 11/11/2008)

[7] Endicott-Popovsky, B.E., Frincke, D., V.M. Popovsky. (2005). *Secure Code: The Capstone Class in an IA Track.* Proceedings of the 9th Colloquium for Information Systems Security Education, Georgia Institute of Technology: Atlanta, GA, pp.100-108.

[8] Martin Fowler. (2004). UML Distilled: A Brief Guide to the Standard Object Modeling Language, (3rd ed.). New York: Addison-Wesley Object Technology Series.

[9] Graff, M. G., and K.R. van Wyk. (2003). Secure Coding Principles and Practices. San Francisco: O'Reilly.

[10] Howard, M and D. LeBlanc. (2002). Writing Secure Code, (2nd ed.). Seattle: Microsoft Press.

[11] Howard, M., LeBlanc, D., and J. Viega. (2005). 19 Deadly Sins of Software Security. New York: McGraw-Hill/Osborne.

[12] Microsoft (2003). *Improving Web Application Security: Threats and Countermeasures*. Microsoft.
Available at: http://www.microsoft.com/downloads/details.aspx?FamilyID=E9C4BFAA-AF88-4AA5-88D4-ODEA898C31B9&displaylang=en

[13] Perrone, L. F., Aburdene, M., and X. Meng. (2005). *Approaches to Undergraduate Instruction in Computer Security*, Proceedings of the American Society for Engineering Education Annual Conference & Exposition: Portland, OR.

[14] Seacord, R. (2005). Secure Coding in C and C++. New York: Addison-Wesley.

[15] Swiderski, F. and W. Snyder. (2004). Threat Modeling. Seattle: Microsoft Press.

[16] Taylor, B. and S. Azadegan. (2006). *Threading Secure Coding Principles and Risk Analysis into the Undergraduate Computer Science and Information Systems Curriculum*. Proceedings of the 3rd Annual Information Security Curriculum Development, Kennesaw, GA.

[17] Taylor, B. and S. Azadegan. (2007). *Using Security Checklist and Scorecards in CS Curriculum*. Proceedings of the 10th Colloquium for Information Systems Security Education, Boston University: Boston, MA.

[18] Taylor, B. and S. Azadegan. (2008). *Moving Beyond Security Tracks: Integrating Security in CS0 And CS1*. Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education, Portland, OR, p. 320-324.

[19] Viega, H., and G. McGraw. (2002). Building Secure Software: How to Avoid Security Problems the Right Way. New York: Addison-Wesley.

[20] Whitman, M. E. and H.J. Mattord. (2004). *Designing and Teaching Information Security Curriculum*. Proceedings of the 1st Annual Information Security Curriculum Development, Kennesaw, GA.

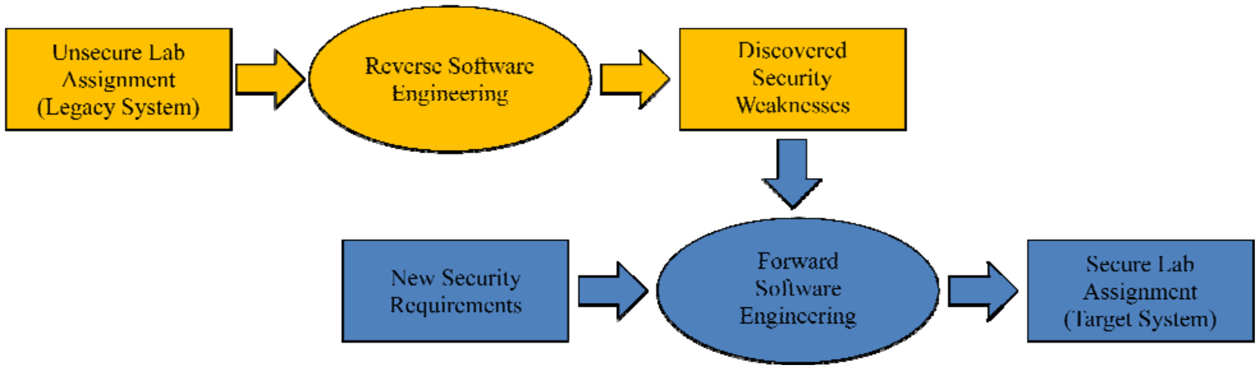

Unsecure Lab Assignment (Legacy System)
Reverse Software Engineering
Discovered Security Weaknesses
New Security Requirements
Forward Software Engineering
Secure Lab Assignment (Target System)

Figure 1: The conceptual description of software reengineering for lab assignments

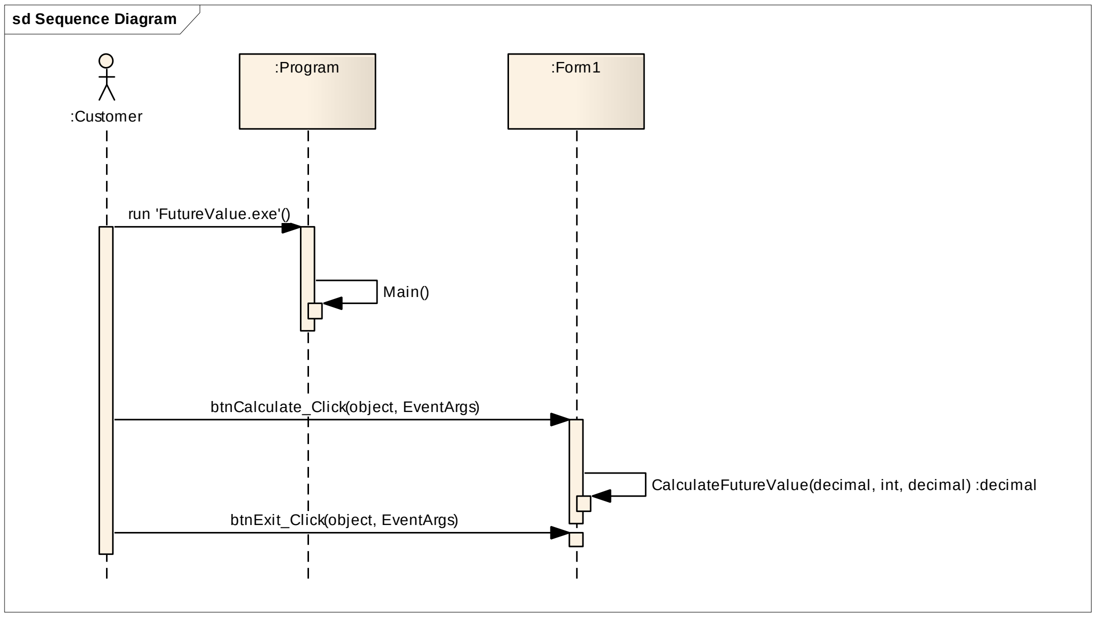

sd Sequence Diagram
:Customer
:Program
:Form1
run 'FutureValue.exe'()
Main()
btnCalculate_Click(object, EventArgs)
CalculateFutureValue(decimal, int, decimal) :decimal
btnExit_Click(object, EventArgs)

Figure 5: A sequence diagram showing that the '*CalculateFutureValue( )*' method was invoked without having any input data validation checking (an insecure legacy system).

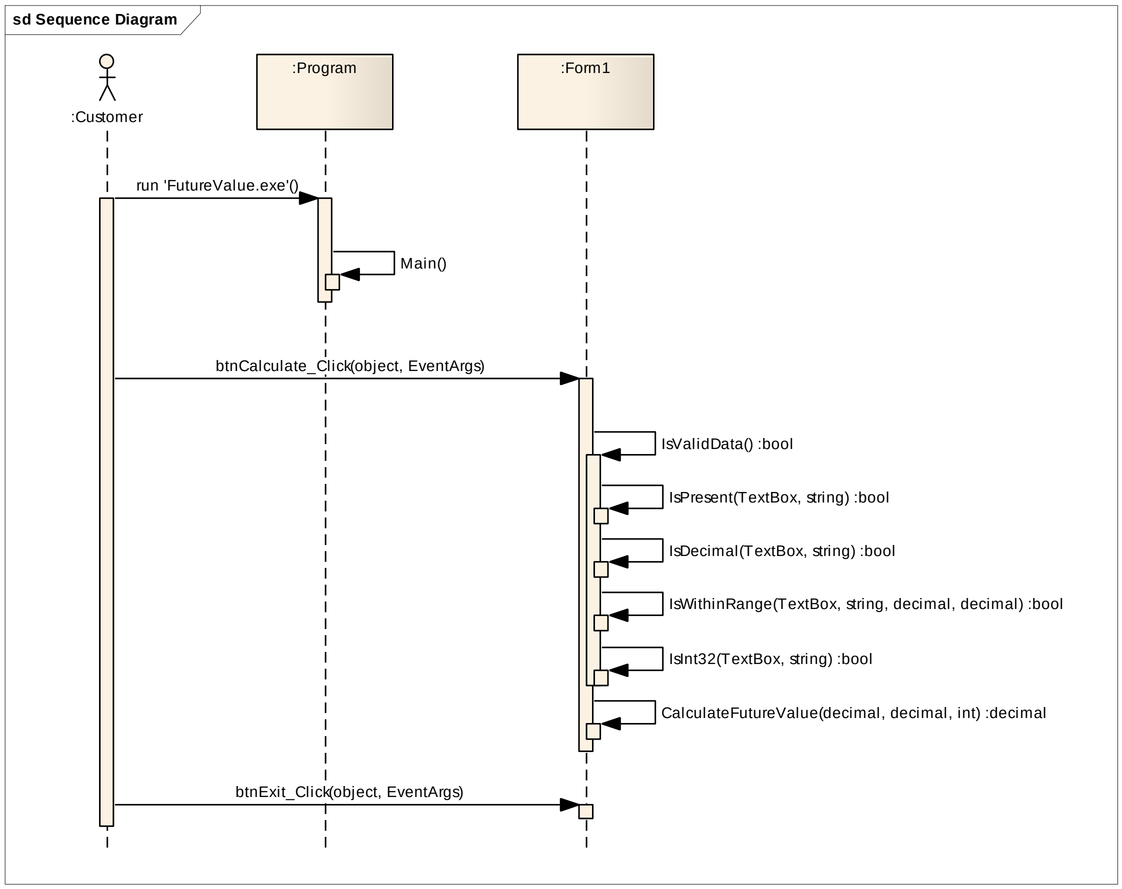

sd Sequence Diagram
:Customer
:Program
:Form1
run 'FutureValue.exe'()
Main()
btnCalculate_Click(object, EventArgs)
IsValidData() :bool
IsPresent(TextBox, string) :bool
IsDecimal(TextBox, string) :bool
IsWithinRange(TextBox, string, decimal, decimal) :bool
IsInt32(TextBox, string) :bool
CalculateFutureValue(decimal, decimal, int) :decimal
btnExit_Click(object, EventArgs)

Figure 7: A sequence diagram showing that the '*CalculateFutureValue( )*' method was invoked after having input data validation checking (a secure target system).