2006-2373: INTEGRATING SECURE DEVELOPMENT PRACTICES INTO A SOFTWARE ENGINEERING COURSE

James Walden, Northern Kentucky University

Dr. James Walden received his Ph.D. from Carnegie Mellon University in 1997. He worked at Intel Corporation as a software engineer, with a focus on security sensitive applications, before becoming a Visiting Professor of Computer Science and Engineering at the University of Toledo in 2003. He is a member of the computer science faculty at Northern Kentucky University.

Dr. Walden has taught software engineering and computer security to both undergraduate and graduate students. His research interests focus on both of those subjects and particularly their intersection: software security, the science and engineering of designing, implementing, and testing secure software systems.

Rose Shumba, Indiana University of Pennsylvania

Dr. Rose Shumba received her Ph.D. from Birmingham University in 1995. She is a professor in the department of computer science at Indiana University of Pennsylvania, where she teaches courses in both computer security and software engineering.

Integrating Secure Development Practices into a Software Engineering Course

Abstract

Many security incidents arise from flaws in the code or design of software systems. CERT reported over 5000 software vulnerabilities in 2005. These vulnerabilities are the result of inadequate consideration of security during the development process. However, typical software engineering courses and textbooks do not address security issues. In response to this problem, software engineering courses with an integrated coverage of security have been introduced at two universities. Information security has been integrated into every phase of the software life-cycle. Teams in both courses developed web application software, requiring them to address common web application security issues such as access control and injection flaws. Students have come out of the courses with a better appreciation of the need for software security and a basic understanding of how to develop secure software. However, finding the time required to cover software security effectively remains a considerable challenge, especially as both institutions only offer a single semester of software engineering.

Introduction

Application software has become highly interconnected as the Internet and wireless networking have grown in importance. While security flaws were previously exposed only to users sitting in front of the computer, the Internet allows attackers from around the world to exploit security vulnerabilities in networked applications. Even embedded systems like cell phones are vulnerable to remote attacks.[1] This increased exposure to attack has greatly increased the importance of software security.

CERT reported over 5000 software vulnerabilities in 2005.[2] These flaws result from inadequate consideration of security during requirements analysis, design, implementation, and testing of software systems. This lack of consideration is often the result of security being viewed as an add-on feature. This viewpoint typically leads to the "penetrate-and-patch" methodology, where security issues are dealt with by issuing a patch after the software product has been released.

The scale of this problem results from the fact that many developers aren't aware of the importance of security or don't know how to build secure applications. Typical software engineering courses and textbooks pay little attention to security issues. In order to significantly reduce the number of vulnerabilities, security must be taught as part of the foundation of the development process in the software engineering curriculum.

Software engineering courses with an integrated coverage of security have been introduced at two universities. Security issues have been integrated into every phase of the software life-cycle from requirements through testing. Both approaches use a threat model to document and drive security concerns throughout the development process. Students analyze the risk of each threat documented in the threat model, then use the evaluations to design appropriate security measures such as access control and encryption. Implementation is guided by checklists and verified with code reviews. Finally, students test their systems against the threats they've analyzed to verify their software's security properties.

Course Goals

As the software industry begins to address the issue of security, companies need new employees who understand the basics of software security. However, few university graduates have the education necessary to design, develop, or test secure software. Many graduates have not taken any security coursework, while those who have taken security classes have typically focused on topics like access control or cryptography instead of software security.[3]

The main purpose of a software engineering class at our institutions is to teach students how to work in teams to develop a secure software project from specification through delivery. The software engineering class has prerequisite classes in programming, but has no security prerequisites. Therefore all security concepts necessary for our secure software development processes must be introduced in the software engineering class.

As a single semester doesn't offer the necessary time to broadly cover information security in addition to software engineering, the class focuses on security topics that fit directly into the software development lifecycle. For example, the course teaches students how to securely use cryptographic APIs in their projects but doesn't delve deeply into the mathematical details of cryptology.

In order to teach students how to develop secure software, the course focused on the following goals:

1. Students should be aware of common types of security vulnerabilities.
2. Students should understand how to document security requirements and design.
3. Students should be able to apply secure design principles to the creation of secure software.
4. Students should know how to verify that the security mechanisms implemented in their system enforce the system's security requirements.

Due to the time limitations of a single semester software engineering course, these goals must be achieved primarily through adapting existing features of the software engineering class instead of adding new topics. One important adaptation was the choice of a web application project. Web applications require that students deal with fundamental problems of security, including authentication, access control, and input and output validation. It's important to give these security features the same importance as functional features when evaluating student projects. Other adaptations included the addition of misuse cases to document security requirements, leveraging student experience with use cases, and the application of web testing tools to test security in addition to functionality.

Course Concepts

Security is fundamentally different from the functional qualities of software, as security focuses on restricting capabilities instead of providing features. The techniques for analyzing, designing, and testing functional features of software often are not effective for addressing the security level of software. This section details adaptations and additions we made to the software development process in our software engineering courses in order to help students learn how to produce secure applications.

Secure Development Process

The development process begins with team organization and planning. Teams of 8-9 students are assembled. Roles and responsibilities within the team are assigned including that of Information Systems Security Officers (ISSOs.) A team organization report with assigned roles and responsibilities and skills within the group is produced. Typical projects assigned over the past few semesters include a health management system, an energy monitoring system, and a help desk application. The requirements specification for the project is produced by an external client.

The ISSOs set out a security policy, describing the types of data accessible to each group of users, for the systems development process. The process shares both iterative and incremental characteristics. The main functionality increments of the system are identified, then development is organized into a series of fast increments for specification, development and certification. During requirements specification for each increment, the ISSOs evaluate the functionality of each increment to determine what security controls to be implemented. They also collaborate with the user to assist in defining security requirements for their operations.

Increments are small in relation to entire systems, and developed fast enough to permit rapid response to user feedback and changing requirements. After completion of each increment, ISSOs evaluate its functionality to determine what security controls have been implemented, evaluate the system controls and determine whether or not the controls provide proper level of security, and ensure that the increment meet the audit and integrity review requirements of the user. The identification of vulnerabilities and the selection and implementation of safeguards continue as the system progresses.

All security considerations are documented in the standard systems development lifecycle documents. By making security an integral part of the development of each increment, it is ensured that the security implications or goals are considered at a manageable scale and resolved in a systematic manner. Awareness of security concerns is increased by involving all the individuals responsible for the development of the application in the process of identifying vulnerabilities and developing safeguards, from system security officers down to the final users of the system.

Misuse Cases

Many software engineering classes apply use cases for the purposes of eliciting and documenting functional requirements of software systems. As use cases typically describe some function that the system will perform for its users, use cases are not as effective for documenting non-functional requirements, especially security requirements which focus on actions that the system should not perform.

Misuse cases[4] are the inverse of use cases, documenting actions that the system should not allow. For example, a misuse case named Replay Token describes a malicious user reading another user's session cookies and reusing them in order to impersonate that user. Relationships between use cases and misuse cases include the "includes" and "extends" relationships, along with two new relationships: "threatens" and "mitigates." Misuse cases "threaten" use cases. The Replay Token misuse case threatens the Logon use case, in which a user is authenticated to the application. A new Encrypted Logon use case could mitigate the Obtain Password misuse case. Misuse cases can include other misuse cases. For example, an Obtain Password misuse case may include a Sniff Network use case, in which the attacker reads network packets containing security critical data,

that would also be used by the Replay Token misuse case.

As students don't have the security experience to construct good misuse cases in the first iteration of the project, they are given a set of misuse cases describing some of the most important vulnerabilities, such as SQL injection.[5] In later iterations, students develop their own misuse cases using their new security experience to address other vulnerabilities such as session management flaws (cookies that can be re-used by other users or altered to grant administrative privileges) and cross-site scripting.[6]

Threat Modeling

A threat is an attacker with the means and motivation to attack the application. Threat modeling[7] is a systematic method of assessing and documenting the security risks presented by the various threats to the application. While constructing the threat model, students examine the application from an attacker's point of view, documenting the threats presented by potential adversaries. Class lectures explain common threats, giving an overview of typical attacker motivations for attacking their application and illustrating common attack patterns that adversaries use to exploit security vulnerabilities in web applications.

The threat model is the primary document driving the secure development process. It documents both entry points where the attacker can inject malicious input and assets that the attacker would want to affect. Entry points include both well known sources of input, such as form parameters and cookies, and less obvious sources, including configuration files and the web server through which the web application runs. Almost all systems will have assets such as user accounts and disk space, but many systems will have additional assets that an attacker may want to access such as credit card numbers.

This information aids the information systems security officers in defining and updating application security policies, shows developers where security mechanisms are needed, and indicates to testers where to focus quality assurance efforts. If a threat exists and is not prevented by a security mechanism, either because no mechanism exists or because the mechanism fails to prevent this particular threat, then the application has a security vulnerability.

The process of threat modeling followed by the class consists of the following steps:

1. Enumerate assets of interest to attackers.
2. Analyze entry points.
3. Identify and document threats.
4. Evaluate risk presented by each threat.

In the first iteration of the process, student teams are given an initial set of threats to address in their application, including assets targeted by the attacker and entry points that will be used. These threats are the only ones involved in evaluating the security of the student's application in the first iteration. After the initial iterations, instructors explain general patterns of attack to students and common attacker goals and students are responsible for expanding their threat model to account for new threats. The threat model may also be updated when new threats are discovered during testing.

Secure Design Principles

Secure design principles guide project security design decisions. Saltzer and Schroeder[8] describe eight design principles for creation of security mechanisms. The principles explained in class included:

1. Economy of Mechanism: keep the design as simple as possible.
2. Fail-safe defaults: all actions are forbidden by default and only enabled when explicitly permitted.
3. Complete Mediation: check every access to objects for authorization.
4. Open Design: security should not depend solely on the secrecy of the design.
5. Separation of Privilege: include multiple layers of security (e.g., an ATM card *and* a PIN.)
6. Least Privilege: programs should only have the privileges required to perform their function.
7. Least Common Mechanism: minimize the amount of mechanism shared between users.
8. Psychological Acceptability: security mechanisms must not greatly increase the difficulty of using the software.

While most design principles are readily understood, a few principles such as complete mediation may require additional explanation. Most filesystems violate this principle by checking access control only when a file is opened, thus allowing a process with an open file to read or write it even if the user changes the access control list to forbid such actions. However, web applications must follow the principle of complete mediation by sending a cookie or similar session token with every request due to the stateless nature of HTTP.

Understanding how to apply these principles, particularly understanding how to make design decisions when two principles conflict, requires seeing examples of design choices in deployed software[9] and experience applying the principles in the class project. An iterative approach to project design is particularly helpful, as it allows students to create and test multiple designs of security mechanisms. Students can alter their design decisions based on the previous iteration's experience, perhaps deciding that sacrificing some economy of mechanism in favor of separation of privilege will improve application security or realizing that complex password requirements violate the principle of psychological acceptability, causing users to forget their passwords unless they write them down.

Programming Language

Student teams are free to choose a language for project development. The choice of the programming language strongly impacts the development process for creating a secure software project, as different languages expose the developer to different security risks. For example, buffer overflows are a common type of security vulnerability in C and C++, but aren't a major problem in languages that provide automatic garbage collection like Java or Python. The choice of language also influences which security libraries and tools are available or important to the development process. For

example, in C or C++, it's important to develop code with secure string libraries and to use a tool like Purify[10] to check for improper memory accesses.

However, it is important to stress to students that use of a particular language does not guarantee security. While use of a language with garbage collection protects developers against most buffer overflows, most languages don't prevent developers from writing common web application vulnerabilities, such as SQL injection or cross-site scripting. It's also worth noting that with care and substantial effort secure applications can be written in a weakly typed language like C.

Most student teams chose Java or Python, though some teams chose other languages, such as PHP. Python is more strongly typed than Java in that it has no automatic conversions between numeric types; however, Java performs most type checks at compile time while Python performs most type checks at runtime. Both languages offer unit test and web development frameworks, and most web application testing tools can deployed without regard to the language in which the application is written.

Code Reviews

Code reviews verify the functionality and security of completed increments. Code review sessions last about one hour on average, but the length of a session will vary depending on the complexity of increment. In preparation for a code review session, four formal roles are assigned: the Moderator, Reader, Recorder, and Author. The Moderator, who must be a strong technical contributor, leads the inspection process. He or she paces the meeting, coaches other team members, deals with scheduling a meeting place and disseminating materials before the meeting, and follows up on any rework. The Reader takes the team through the code by paraphrasing its operation. A Recorder notes each error on a standard form, freeing other team members to focus on thinking deeply about the code. The Author's role is to understand the errors found and to illuminate unclear areas.

Before a code review session, students are expected to work individually in trying to understand the program. They document the meaning of every vital statement in the code. Understanding and reading a program is an essential skill for all programmers. Reading competence is most obviously relevant to program maintenance, for which the programmer must gain sufficient understanding of the code to design modifications that extend, adapt, or correct it, while retaining its logical, functional, and stylistic integrity. Effective code verification and code reviews require students to understand and analyze the code under scrutiny.

A set of security developer guidelines and checklists[11] are used in the code review. Checklists help the students with the list of items to be checked or remembered. Students are also given language-specific guidelines[12] for secure programming practices.

Testing

Testing in most software engineering courses focuses on verifying that each application feature works as specified and validating that the application performs the tasks desired by its customers. Security testing is a different problem. Most security vulnerabilities are not flaws in the code for security features; instead, vulnerabilities result from unexpected misuse of functional features of the application. Any application input may expose a SQL injection vulnerability.

Security testing focuses on checking for negatives, such as the absence of SQL injection flaws, instead of checking for positives. While a tester can verify a positive relatively easily, demonstrat-

ing a negative is a more difficult problem. A set of tests covering the application's features can demonstrate that they work as expected, but a set of tests that do not exploit a security flaw only shows that no flaws exist that could be exposed by those tests. While no set of tests can prove that an application is free of security defects, it is possible to focus our testing efforts on the highest risk parts of the application using the threat model developed during the design phase.

Students applied two strategies for testing security. The first strategy was to test security mechanisms such as authentication and access control using standard functional testing techniques. Functional testing can verify that authorized users are able to authenticate and obtain the appropriate level of access to the system. The second strategy was risk-based security testing based, using the threat model to ensure that the highest risk parts of the application were tested with the appropriate types of input. Following this strategy, the tester checks whether it's possible to spoof another user's identity, tamper with data that should not be accessible with their account, or deny service to other users. The threat model is used to determine entry points and assets to target. Individual test cases can be developed from misuse cases and prioritized according to their risk as documented in the threat model.

Functional testing consists of unit tests and acceptance testing performed with a web browser. Security testing also uses a web browser for penetration testing[13] in the first iteration. Testers can alter several types of web application input through the browser interface, including paths and form parameters submitted as part of the URL, and can view other types, such as cookies and hidden fields. Typical input alterations include inserting new inputs, removing expected inputs, altering data to be of the wrong type or size, and the insertion of special characters, including HTML, Javascript, SQL, and URL-encoded characters. The browser also offers the possibility of disabling client-side components of the web application like Javascript.

In later iterations, students use tools to allow them to alter every aspect of web input, including POST parameters submitted in the body of an HTTP request and cookies, which many web developers expect to be inviolable because they cannot be altered through the browser interface. Web proxy tools like Paros[14] allow testers to intercept and modify requests between the browser and server, while the perl language offers a virtual browser module, `WWW::Mechanize`, which allows students to automate form submission tests with less than a dozen lines of code. Students also expand the scope of their testing to take in the web and database servers to check for vulnerabilities such as direct database access or the possibility of negotiating an unencrypted connection with the web server.

Students need to be cautioned to avoid gaining a false sense of security from penetration testing. A penetration test is only as good as the team conducting it, and a penetration test can only demonstrate the presence of security flaws. It cannot demonstrate the absence of security flaws in the application. Penetration testing is only one part of the software verification process, which also includes code inspections and formal methods. Code inspections are discussed above. Formal methods are mentioned in class, but not applied to the project due to time limitations.

Challenges

The primary challenges in integrating security into a software engineering class are a lack of student security experience entering the course, the paucity of textbooks covering secure software development, and the difficulty of fitting additional topics into an already packed course. Neither of our institutions require a prerequisite computer security class, and a one semester software en-

gineering course does not offer sufficient time to provide students with a broad introduction to information security. The solution is to cover only those security topics necessary for developing the class project can be covered.

Many security mechanisms, such as cryptography, must be treated as black boxes due to the time limitations. The inner workings of such mechanisms can be learned in a separate computer security or cryptography class. However, the remaining task of teaching students to use cryptography securely is nontrivial. More vulnerabilities arise from improper use of cryptography than from poor cryptography algorithms.[15]

Popular software engineering textbooks[16,17] cover security in a handful of pages or less, while most software security books written for professionals[18,19] focus on a single part of the lifecycle, typically implementation, without addressing information security throughout the entire software development lifecycle. However, two new professional books addressing secure development processes have been announced recently.[20,21] Instructors wishing to add security to their software engineering classes will need to supplement their textbooks with a variety of books, papers, and online sources.

Assessment

Students enter the class with a preference for building core functionality features that they can see working, leaving security as an afterthought to be addressed in an ad hoc manner during development. However, by the end of class, student course evaluations made it clear that students appreciated the importance of integrating security concerns into the software development life-cycle, with one student writing, "Every system needs to be secure. Almost every software controlled system faces threats from potential adversaries." Another student noted, "Most security activities traditionally left to implementers are actually best handled by the software architects and designers."

However, one student made the comment, "The secure development approach is not mature enough to be realistically used for any project." A number of concerns about integrating into the development process were noted in student evaluations:

- Time pressure on the project completion.
- Unavailability of a good secure software engineering textbook. Most reading materials were journal articles or whitepapers.
- The output from the static analysis tools can be too verbose and hard to understand.

Students showed the greatest appreciation for applied aspects of the secure development process, such as the threat model and checklists. One student noted that "case studies on the application of the process to a real client project would be useful."

Future Work

There are many additional security topics and techniques that could be integrated into the curriculum given sufficient time. Within the constraints of a one semester software engineering class, we would like to apply more security tools to the development process. Code reviews could be enhanced through the assistance of better static analysis tools with more instructor assistance. There

are also a wide variety of web application security scanners, which offer features beyond the proxy and custom perl tools used in previous versions of the course.

To address the issue of time constraints, we are creating a graduate class in secure software engineering. This class has both software engineering and information security prerequisites, allowing us to build upon a foundation of both security and software development skills. Graduate students also typically have a stronger mathematical background than undergraduates, opening the possibility of covering the application of formal methods to the secure development process.

Conclusions

In this paper, we discussed the integration of information security in software engineering classes at two universities. Changes to the course structure and development process were described, including new topics added to the course and existing topics adapted to deal with security issues. One essential change was the choice of a web application project to require students to address security issues common in complex networked applications. Course evaluations indicated an increased awareness and knowledge of software security, but time constraints and the immaturity of the field of software security were noted as concerns.

References

[1] John Oates, "Virus attacks mobiles via Bluetooth," The Register, `http://www.theregister.co.uk/2004/06/15/symbian_virus/`, June 15, 2004.

[2] CERT, Cyber Security Bulletin 2005 Summary, `http://www.us-cert.gov/cas/bulletins/SB2005.html`, Dec. 29, 2005.

[3] S. Lipner, "The Trustworthy Computing Security Development Lifecycle", 20th Annual Computer Security Applications Conference, `http://www.acsac.org/2004/dist.html`, Dec. 2004.

[4] G. Sindre and A.L. Opdahl, "Eliciting Security Requirements by Misuse Cases," Proceedings of the TOOLS Pacific Conference, pp. 120-131, Nov. 20-23, 2000.

[5] K. Spett, "SQL Injection," `http://www.spidynamics.com/whitepapers/WhitepaperSQLInjection.pdf`, 2002.

[6] G. Zuchlinski, "The Anatomy of Cross Site Scripting," `http://www.cgisecurity.com/lib/xss_anatomy.pdf`, 2003.

[7] F. Swiderski and W. Snyder, *Threat Modeling*, Microsoft Press, 2004.

[8] J. Saltzer and M. Schroeder, "The Protection of Information in Computing Systems," Proceedings of the IEEE 63(9), pp. 1278-1308, Sep. 1975.

[9] N. Provos, M. Friedl, and P. Honeyman, "Preventing Privilege Escalation," 12th USENIX Security Symposium, Washington, DC, Aug. 2003.

[10] R. Hastings and B. Joyce, "Purify: Fast Detection of Memory Leaks and Access Errors," Proceedings of the Winter Usenix Conference, pp. 1-12, Jan. 1992.

[11] D. Gilliam, T. Wolfe, J. Sherif, and M. Bishop, "Software Security Checklist for the Software Life Cycle," Proceedings of the 12th IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2003) pp. 243-248, June 2003.

[12] OWASP, *A Guide for Building Secure Web Applications and Web Services*, `http://www.owasp.org/documentation/guide.html`, 2.0 Black Hat Edition, July 27, 2005.

[13] B. Arkin, S. Stender, and G. McGraw, "Software Penetration Testing," IEEE Security and Privacy 3(1), pp. 84-87, Jan. 2005.

[14] Paros Proxy Tool, `http://www.parosproxy.org/`.

[15] R. Anderson, "Why Cryptosystems Fail," Proceedings of the 1st ACM Conference on Computer and Communications Security (ACMCCS 1993) pp. 215-227, 1993.

[16] S. Pfleeger, *Software Engineering: Theory and Practice, 2nd edition*, Prentice Hall, 2001.

[17] I. Sommerville, *Software Engineering, 7th Edition*, Addison-Wesley, 2004.

[18] J. Viega and G. McGraw, *Building Secure Software*, Addison-Wesley, 2001.

[19] M. Howard and D. LeBlanc, *Writing Secure Code, 2nd edition*, Microsoft Press, 2002.

[20] G. McGraw, *Software Security: Build Security In*, Addison-Wesley, 2006.

[21] M. Howard and S. Lipner, *The Security Development Lifecycle*, Microsoft Press, 2006.