High-Leverage Techniques for Software Security

Idongesit Mkpong-Ruffin and Dr. David Umphress
Auburn University

Software security addresses the degree to which software can be exploited or misused. Software development approaches tend to polarize security efforts as being reactive or proactive; a blend of both approaches is needed in practice. Three categories of tools provide such a blend: threat modeling, risk analysis, and security assessment and testing. These tools provide leverage as they are currently in use as quality assurance methods and can be modified with relatively little effort to address security.

Software security deals with the ability to protect software and its underlying systems from being exploited by unauthorized users and from mishaps from authorized users. This includes safeguarding against direct attacks [1], detecting and preventing indirect attacks that take advantage of software defects, removing defects, and preserving the intrinsic value (such as the inherent intellectual property) of the system.

Approaches to ensure that software systems are secure fall along a continuum ranging from reactive to proactive techniques. Reactive proponents tend to favor a post-facto approach – applying security measures to software after it has been developed. This is based on the opinion that it is not possible to protect against every single conceivable threat. This philosophy is articulated as *if security is the absence of risk, then we will never get a system that is both secure and useful. There is a need to balance risk and control* [2]. Reactive approaches hinge on three assumptions. First, systems that are exhaustively assessed for security become hard to use and the cost in time and effort to produce and use such systems defies the economics of software development. Second, risk handling should be deferred until a problem actually occurs. Third, tools to handle problems need to be in place to handle issues when they arise [2].

The proactive approach, on the other hand, requires that security measures be implemented during the software life cycle as part of the development process, so that fielded systems require little or no patching. This is based on the premise that if software is built with security in mind then vulnerabilities will be addressed early in the development cycle. To do so, proponents of proactive security measures require that architectures and designs that actively promote security be created and that risk management be applied throughout the development process. Put bluntly, this is the philosophy of *building things right, designing for security, analyzing security over the whole life cycle, and coding securely* [3].

In practice, both approaches are necessary to create secure software. Since systems are often a composite of unreliable parts, putting those parts together to ensure security is an engineering problem, and as such, software development should be approached with the attitude that secure and effective solutions can be created even when some of the materials used are flawed [2].

“The purpose of threat modeling is to examine the security of a software component from the perspective of what types of attack are likely to take place on it.”

Recent media attention on the frequency of security patches that are required by popular software packages may lead developers to think that software security is an overwhelmingly complex issue. While security is difficult, it is not impossible. There are three specific high-leverage techniques that can be used, such as threat modeling, risk analysis, and software assessment and security testing. Each of these is well within the reach of the skills of most developers.

Threat Modeling

The purpose of threat modeling is to examine the security of a software component from the perspective of what types of attack are likely to take place on it. Traditionally, the practice of using graphical software representations, such as Unified Modeling Language (known as UML), focuses on describing what is expected of a system, not what is unexpected. Threat modeling augments this approach by requiring developers to consider ways a component might be misused based on past history. It helps developers think beyond standard features and consider negative or unexpected events. Arising from threat modeling are misuse and abuse cases which address abnormal behavior. Extending use-cases to include *misuse*-cases that depict side-by-side what behavior should be supported and what should be prevented has been proposed [4]. Threat modeling helps developers view the software component from the perspective of an attacker, thus bringing security to the forefront during all phases of development.

Threat modeling begins with a catalog of agents and attacks that have been carried out on other systems. There are vulnerability databases such as the National Vulnerability Database (NVD) [5] and Open Source Vulnerability Database (OSVDB) [6] that incorporate publicly available vulnerability resources and would make a good starting point in documenting these agents and attacks. At its most fundamental, the catalog consists of well-known vulnerabilities, such as problems arising from boundary conditions, intersystem communication, system assumptions, etc. Other vulnerabilities are added as they are observed.

The threat model specific to a software component – from which use and misuse cases are constructed – is constructed in the following three steps [7]:

1. A behavior model of intended functions is defined based on the functional requirements.
2. Based on the behavior model, decisions on potential misuse/abuse or anomalies of the intended functions that would violate any of the security goals are made.
3. Mitigations for misuse or anomalies in the threat model are specified.

The steps are frequently carried out iteratively, with the derived security models built with different levels of detail.

Tools and methodologies that are helpful in threat modeling are attack trees [8], attack nets [9], and attack pattern matching [10]. These methodologies approach threat identification at different levels of abstraction. Attack trees diagram the steps an attacker would take in completing his/her objective [8]. Attack trees tend to be most effective in smaller applications. An attack net, an extension of an attack tree, is a Petri net used to represent complex security threats with variation within attack patterns [9]. Attack nets are not meant to model the actual behavior of a system/component when an attack happens but are used to organize the development of probable attack scenarios. Concurrency and attack progress are modeled as tokens, and intermediate and final objectives are modeled as places, while commands or inputs are modeled as transitions [9]. Attack pattern matching processes such as Security Analysis for Existing Threats [10] and Microsoft's Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege (STRIDE) [11] approach, are designed to match attack patterns to system designs. These processes are used at the design level to provide a higher level of abstraction so that the evolving system design can incorporate security measures to the process.

Risk Analysis

Risk is the probability that an event with a negative impact will occur. It is determined by factors such as the ease of executing an attack, the attacker's motivation and resources, a system's existing vulnerabilities, and the cost or impact in a particular business context [1]. Risk analysis deals with the way threats are described. It takes into account the impacts, the probable consequences, and the probability and frequency of occurrence of each threat [1, 12]. It helps in determining ways to alleviate identified risks by providing selection criteria for safeguards and other means of preventing or lessening identified risks to a level that is considered acceptable.

The activities that are done during risk analysis are determined by the security requirements received during requirements and design phase analysis. Common risk analysis activities include risk identification, assessment, characterization, communication, mitigation, and risk-specific policy definition [12]. Techniques such as asset valuation, quantitative risk analysis, and qualitative risk analysis are used in risk analysis to gather required information so that security-relevant design specifications can be created [13]. The information gained from risk analysis is used while choosing tools and mechanisms for the security design process. The value of assets and the cost of attacks are compared with the costs of tools and mechanisms in order to ensure the chosen tools and mechanisms are appropriate and proportionate to the risk to which the application or system is exposed.

"Quantitative risk analysis is used to identify the key risk elements and the value associated with identified risk."

Asset valuation is the process used to determine the worth of an asset. It examines information not only of the hardware and software pertinent to the system but also personnel and other physical assets. The value of the asset is made up of its inherent value and the short and long term impacts and consequences of its compromise [14]. This aids in justifying proposed mitigation to stakeholder, legal and other regulatory requirements [1, 12].

Quantitative risk analysis is used to identify the key risk elements and the value associated with identified risk. This information allows for the estimation of potential loss and the ability to analyze potential threats. Quantitative risk analysis is used to compute what is known as the Annual Loss Expectancy (ALE):

ALE = Single Loss Expectancy (SLE) x Annualized Rate of Occurrence (ARO)

where,

ARO is the frequency of threat per year, SLE is the asset value x the exposure factor (EF), and EF is the percentage of asset loss caused by the potential threat.

The values garnered from the quantitative analysis can then be ranked and decisions can then be made based upon the information.

Lastly, qualitative risk analysis is used to discover the threats and vulnerabilities that apply to different identified scenarios. Safeguards and countermeasures that reduce or prevent the probability and effect of occurrence are identified, based on the threats and vulnerabilities found [12]. Most qualitative risk analysis methodologies assign weight and values to categories such as probability of occurrence, impact, exposure, and cost. Higher values are assigned to categories that are assumed to be of greater importance, thereby reflecting their impact in the overall priority score.

The information gathered during risk analysis is then used in the rest of the development cycle. Fault-injection methods and security tests are driven by the vulnerabilities and risks discovered and annotated during risk analysis.

Software Assessment and Security Testing

Testing traditionally involves exercising an application to see if it works as it should. In contrast, security testing entails identifying and removing vulnerabilities that could result in security violations. It also validates the effectiveness of security measures that are in place [15].

Most of the testing methodologies used fall into one of two categories: black-box or white-box testing. Black-box tests are those whose data are derived from the specified functional requirements in which attention is not given to the final program structure [16, 17]. Commonly used black-box testing approaches for software security are penetration, functional, risk-based, and unit testing.

White-box tests are those tests and assessment activities where the structure and flow of the software under review are visible to the tester. Testing plans are made based on the details of the software implementation and test cases are based on the program structure [15, 16, 17]. Commonly used white-box assessment approaches that can assess security are source code analysis and profiling.

The method by which security assessment and testing is carried out depends on the perspective of the tester relative to the software component. Test cases that are constructed based on functional requirements without regard to specific knowledge about software internals are known as black-box tests; test cases that take advantage of internal structure are known as white-box tests. Often, the information gathered during risk analysis is used to develop white-box and black-box test cases. In particular, flaws identified during risk analysis can be purposely added to a software component to forcibly change the program state and demonstrate the effects of a successfully exploited vulnerability. This approach, known as fault injection, allows for absolute worst-case prediction [18]. It gives an insight into predictive measures such as mean-time-to-hazard, minimum-time-to-hazard, and mean-time-to-failure; all of which quantify risk.

Three approaches are commonly taken to test the security of a component in a black-box fashion. Risk-based testing demonstrates that security functionalities work as intended [19]. Penetration testing examines the ease with which a component can be infiltrated. Unit security testing assumes that adversaries will take a two-stage approach to attack: First, they get access to the software, then second, control the software after access. As such, the assumptions that developers make about the environment and incorporate into the components should be checked at the unit testing level. Attack trees have been used by many as a method for identifying and modeling security threats, especially those that involve many stages for implementation [20].

Two high-leverage white-box techniques for assessing and validating security are source code analysis and profiling. Static analysis tools are used to look at the text of a program while it is not executing so that it can discover vulnerabilities within the program. A fixed set of patterns or rules are used as basis for scanning the source code. For example, many vulnerabilities are known to come from reusable library functions such as stropy() and stat (); so, a static analyzer could scan the programs to see if they contain any calls to those functions. The result of the source code analysis aids in the development of test cases and gives a good perspective of the security posture of the application. White-box testing should be used to verify that the potential vulnerabilities uncovered by the static analysis tool will not lead to security violations [21].

Profiling tools enable the tester to observe the performance of an application while it is running. This provides insight into where performance bottlenecks may be occurring. It also enables the tester to see and understand the sequence of function calls and the time spent in different areas of the software, and thereby brings it to the open areas of vulnerability that are not apparent when using static code analyzers [12].

"Software security demands a balance of reactive and proactive measures, and it requires that more time be spent in determining the risks that can or will affect the system."

Although security aspects of software should be tested, it is also important to understand that security is not just a function that can be checked off but is an emergent property of the application. In other words, this would be analogous to saying that *being dry* is an emergent property of being inside a tent during a rainstorm. The tent only keeps a person dry if the poles are made stable, vertical, and able to support the weight of the wet fabric; the tent also must have waterproof fabric (without any holes) and be large enough to protect all those who want to remain dry. Lastly, everyone must stay under the tent the whole time it is raining. So, although having poles and fabric is important, it would not be enough to say *the tent has poles and fabric, thus it keeps one dry!* [22].

Conclusion

To develop software systems with security as an emergent feature entails that the high leveraged techniques discussed be incorporated into the whole software development life cycle. Threat modeling that drives risk analysis begins with the garnering of requirements and use cases. Risks generated from the threat modeling activities act as a barometer for design, development of tests, and development of rules for software code assessment and as one of the benchmarks for testing.

Software security demands a balance of reactive and proactive measures, and it requires that more time be spent in determining the risks that can or will affect the system. Software systems have to be designed from a high enough level of abstraction with security of the system as an emergent feature of the system in question. The processes utilized to create secure systems need more refinement so that the ubiquity of software is not hampered by inherent insecurity due to poor design.◆

References

1. Verdon, Denis, and Gary McGraw. "Risk Analysis in Software Design." IEEE Security and Privacy 2.4 (2004).
2. Cheswick, B, Paul Kocher, G. McGraw, and A. Rubin. "Bacon Ice Cream: The Best Mix of Proactive and Reactive Security?" IEEE Security and Privacy 1.4 (2003).
3. McGraw, Gary. "Building Secure Software: Better Than Protecting Bad Software." IEEE Software 5.7 (2002).
4. G. Sindre, and A.L. Opdahl. Templates for Misuse Case Description. Proc. of the Seventh International Workshop on Requirements Engineering, Foundation for Software Quality (REFSQ 2001), 4-5 June 2001, Switzerland.
5. United States. Department of Homeland Security (DHS). National Vulnerability Database 7 Dec. 2006 <http://nvd.nist.gov/>.
6. OSVDB. Open Source Vulnerability Database. 8 Dec. 2006 <www.osvdb.org>.
7. Dianxiang Xu, and Kendall Nygard. "A Threat-Driven Approach to Modeling and Verifying Secure Software." Proc. of the 20th IEEE/ACM International Conference on Automated Software Engineering ASE, Nov. 2005, Long Beach, CA. New York: ACM Press, 2005.
8. Schneier, B. "Attack Trees: Modeling Security Threats." Dr. Dobb's Journal Dec. 1999.
9. McDermott, J.P. "Attack Net Penetration Testing." Proc. of the 2000 Workshop on New Security Paradigm,

Sept. 2000. Ballycotton, County Cork, Ireland. New York: ACM Press, 2000.

10. Gegick, M., and L. Williams. "Matching Attack Patterns to Security Vulnerabilities in Software-Intensive System Designs." Proc. of the 2005 Workshop on Software Engineering For Secure Systems; Building Trustworthy Applications, 15-16 May 2005, St. Louis, MS. New York: ACM Press, 2005 <http://doi.acm.org/10.1145/1083200.1083211>.
11. Hernan, Shawn, Scott Lambert, Tomasz Ostwald, and Adam Shostack. "Threat Modeling – Uncover Security Design Flaws Using The STRIDE Approach." MSDN Magazine Nov. 2006.
12. Steel, Christopher, Ramesh Nagappan, and Ray Lai. Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management. Prentice Hall, 2005.
13. McGraw, Gary. Software Security: Building Security In. Addison-Wesley Professional, 2006.
14. United States. Department of Commerce. An Introduction to Computer Security – The NIST Handbook. NIST, 1995.
15. Pan, Jiantao. "Software Testing – 18-849b Dependable Embedded Systems." Carnegie Mellon University, 1999 <www.ece.cmu.edu/~koop man/des_s99/sw_testing>.
16. Howard, Michael, and David C. LeBlanc. Writing Secure Code. 2nd ed. Redmond, WA: Microsoft Press, 2002.
17. Hetzel, William C. The Complete Guide to Software Testing. 2nd ed. Wellesley, MA: QED Information Sciences, 1988.
18. Voas, Jeffrey M., and Gary McGraw. Software Fault Injection: Inoculating Programs Against Errors. New York, NY: John Wiley & Sons, 1998.
19. Michael, C.C., and Will Radosevich. "Risk-Based and Functional Security Testing." DHS. BuildSecurityIn Portal <https://buildsecurityin.us-cert.gov/portal/article/bestpractices/security_testing/overview.xml#Risk-Based-Testing>.
20. Schneier, B. Secrets and Lies: Digital Security in a Networked World. New York: John Wiley & Sons, 2000.
21. Janardhanudu, Girish. "White Box Testing." DHS. BuildSecurityIn <https://buildsecurityin.us-cert.gov/portal/article/bestpractices/white_box_testing/overview.xml>.
22. Hope, Paco, Gary McGraw, and Annie I. Anto'n. "Misuse and Abuse Cases: Getting Past the Positive." IEEE Security and Privacy 2.3 (2003).

About the Authors

Idongesit Mkpong-Ruffin is a computer science and software engineering doctorate student at Auburn University. She has a Bachelor of Science in computer information Science from Freed-Hardeman University, a Master of Business Administration from Tennessee State University and a Master of Science in computer information science from Troy University, Montgomery campus.

Department of Computer Science and Software Engineering
Auburn University
107 Dunstan Hall
Auburn, AL 36849-5347
Phone: (334) 844-7001
E-mail: mkponio@auburn.edu

David A. Umphress, Ph.D., is an associate professor of computer science and software engineering at Auburn University. He has worked over the past 25 years in various software development capacities in both industry and academia. He is also an Air Force reservist, currently serving as a researcher for the College of Aerospace Doctrine, Research and Education, Maxwell AFB, Alabama. Umphress is an Institute of Electrical and Electronics Engineers certified software development professional.

Department of Computer Science and Software Engineering
Auburn University
107 Dunstan Hall
Auburn, AL 36849-5347
Phone: (334) 844-6335
E-mail: umphrda@eng.auburn.edu