Security-aware Software Development Life Cycle (SaSDLC) – Processes and Tools

Asoke K Talukder, Vineet Kumar Maurya, Santhosh Babu G, Jangam Ebenezer, Muni Sekhar V, Jevitha K P, Saurabh Samanta, Alwyn Roshan Pais

Information Security Lab, Department of Computer Engineering,
National Institute of Technology Karnataka, Surathkal
asoke.talukder@geschickten.com, vineet.nitks@gmail.com, santhosh003@gmail.com, ebenezer.jangam@gmail.com, munisek@gmail.com, jevitha@gmail.com, saurabh.samanta@gmail.com, alwyn.pais@gmail.com

***Abstract* —Today an application is secured using in-vitro perimeter security. This is the reason for security being considered as nonfunctional requirement in Software Development Life Cycle (SDLC). In Next Generation Internet (NGI), where all applications will be networked, security needs to be in-vivo; security must be functions within the application. Applications running on any device, be it on a mobile or on a fixed platform – need to be security-aware using Security-aware Software Development Life Cycle (SaSDLC), which is the focus of this paper. We also present a tool called *Suraksha* that comprises of Security Designers' Workbench and Security Testers' Workbench that helps a developer to build Security-aware applications.**

***Keywords* — Security-aware Software, Security-aware Software Development Life Cycle, SaSDL, Secure Software Engineering, Security Designers' Workbench, Security Testers' Workbench.**

I. INTRODUCTION

In 1968 October NATO Science Committee organized a conference on Software Engineering [1]. In last forty years different techniques have been proposed to establish software development as a part of mainstream engineering that includes formal techniques of requirement elicitation, design, construction, testing, deployment, and maintenance of software.

Security for a software system has always been in-vitro and addressed only in the production environment through perimeter security like firewall, proxy, intrusion prevention system, antivirus, and platform security. This was the reason of security being considered as nonfunctional requirement.

In Next Generation Internet (NGI) all applications are networked; and, these applications are accessible by everybody – legitimate users and hackers alike. NGI applications will be even mobile; therefore, an application needs to be security aware so that it can protect itself from security threats. This implies that security must be in-vivo - security needs to be in-built within the application. This is achieved through Security-aware Software Development Life Cycle (SaSDLC) that will be part of Secure Software Engineering (SSE).

In this paper we present the SaSDLC process and a tool named for SaSDLC named *Suraksha. Suraksha* in Sanskrit means safety and security. This tool helps a security designer to elicit security requirement followed by security design, through Security Designers' Workbench and security testing, with secured Web deployment through Security Testers' Workbench.

The organization of the paper is as follows. Section 2 explains the Security Requirement Analysis. Section 3 gives the outline of Security Design and Security patterns. Section 4 describes Safe Programming. Section 5 describes Security Testing. Section 6 describes Security Deployment. Section 7 describes the SaSDLC tool *Suraksha*. Section 8 concludes the paper.

II. SECURITY REQUIREMENT ELICITATION

During requirement elicitation, both functional and non-functional requirement of security needs to be captured. This is done in 8 steps in SaSDLC as following.

Step 1 – Functional Requirements: In this step, Functional requirements of the system are analyzed and captured using UML tools and methodology. Use case diagram is used for specifying the functional requirements of the system.

Step 2 – Identification of Assets: In this step assets are identified and their criticality to the organization is established. Security measures depend on state of mobility -- they are either stationary assets or assets in transit [2]. A brainstorming session is conducted to list all assets; in addition, various existing documents are examined to identify important assets. Assets are then categorized based on their perceived value and impact in case a security attack happens. To evaluate the value, an asset is taken and viewed from different perspectives i.e. organization, user, administrator, and attacker. From

978-1-4244-3474-9/09/$25.00 ©2009 IEEE

these perspectives, each asset is assigned a number indicating the importance from STRIDE and CI5A perspective. STRIDE [3, 4] is used by Microsoft for threat modeling of their systems – threats are identified by exploring the possibilities of Spoofing Identity, Tampering with Data, Repudiation, Information Disclosure, Denial of Service and Elevation of Privilege in the given case. Threats are also identified with respect of CI5A that deals with Confidentiality, Integrity, Availability, Authentication, Authorization, Accounting, and Anonymity. Valuations of each asset are added and the asset with highest sum is ranked as the most valuable asset. An example of asset identification by using S*uraksha* is shown in Figure 1.

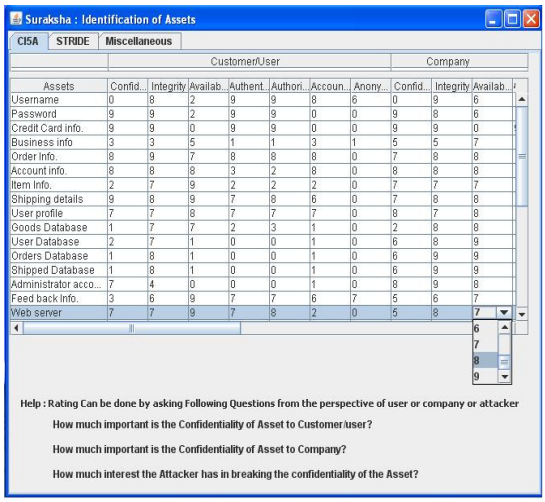

Figure 1*:* Assets identification for an Application (*Suraksha*)

Step 3 – Security Requirements: For each actor in the Use case, one or more misactors are identified; STRIDE with CI5A concepts are applied in connection with each action and assets related to it. Through STRIDE, the possibilities for Spoofing Identity, Tampering with Data, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege are considered; and, through CI5A, Confidentiality, Integrity, Availability, Authentication, Authorization, Accounting, and Anonymity in the given case by misactor are explored. This yields a list of possible abstract threats.

Step 4 – Threat and Attack Tree: Each abstract threat in the Misuse case diagram is considered as a root node and corresponding attack tree [5, 6, 7] is constructed to understand what are the AND and OR relationship in the threat path. Here the user goes through each and every Misuse case [8, 9, 10]; a node in the attack-tree is an actual threat.

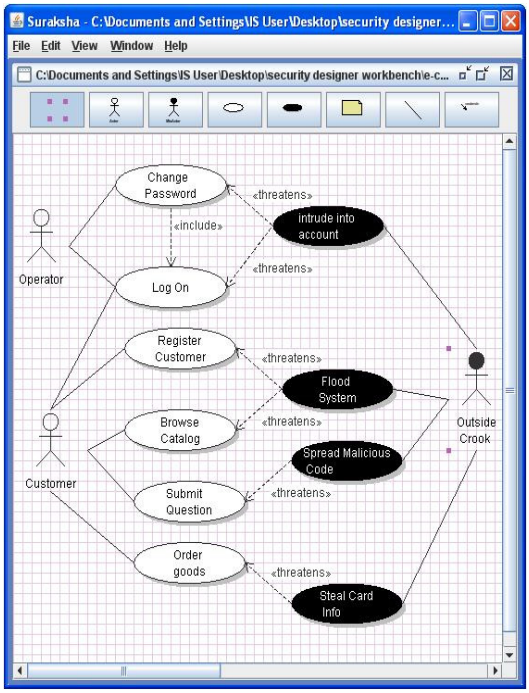

Figure 2: Misuse-Case diagram for an Web-based Application (*Suraksha*)

Step 5 – Rating of Risks: For each attack type, DREAD [4, 11] is used to rate a threat. This is done by assigning values to each node beginning from leaf nodes using simple formula,

Risk DREAD = (D+ R + E +A + D) / 5

Where,

D = Damage Potential,

R = Reproducibility,

E = Exploitability,

A = Affected Users,

D = Discoverability.

The calculation always produces a number between 0 and 10; the higher the number, higher the risk.

Step 6 – Decision on In-vivo Versus In-vitro: After careful examination of each threat rating using DREAD, a threat is ranked as a high risk or moderate risk or low risk threat. These ratings are compared with value of assets as measured in Step 2. All high value assets must be secured. If it is too expensive to secure an asset in-vivo, compared to the cost of the asset, those threats need not be secured in-vitro – a candidate for in-vitro security could be Denial-of-Service attack.

Step 7 – Nonfunctional to Functional Requirement: All these threats that are decided to be protected in-vivo through countermeasures now become candidate for functional requirement for a Security-aware application. In other words, all these

countermeasures that need to be included in-vivo will move into the software as functional requirements.

Step 8 – Iterate: In the last step all misuse cases go through above 7 steps. The security design might force revisit of these 7 steps. Also, there might be necessity for some refinements.

III. Security Design with Security Patterns

Joseph Yoder and Jeffrey Barcalow [12] were first to adapt seven security design patterns for information security. It is easy to document what the system is required to do. They are:

1) *Single Access Point*: Providing a security module and a way to log into the system. This pattern suggests that keep only one way to enter into the system.
2) *Check Point*: Organizing security checks and their repercussions. Authentication and authorization are two basic entity of this pattern.
3) *Roles*: Organizing users with similar security privileges.
4) *Session*: Localizing global information in a multi-user environment.
5) *Full View with Errors*: Provide a full view to users, showing exceptions when needed.
6) *Limited View*: Allowing users to only see what they have access to.
7) *Secure Access Layer*: Integrating application security with low-level security.

To manage the security challenges in NGI, we need to look at many more design patterns. However following patterns must be included in any security system [2].

8) *Least Privilege*: Privilege state should be shortest lived state.
9) *Journaling*: Keep a complete record of usage of resource.
10) *Exit Gracefully*: Designing systems to fail in a secure manner.

If there are patterns but no misuse-case identified in the requirement phase, the iteration starts from Step 2 of requirement analysis. At the end of the design, the attack surface is analyzed. If the attack surface area is high, above process is repeated until the attack surface is reduced to the minimum level.

IV. Safe Programming

A system can be made secured only against known threats. However, a security-aware application must be able to handle unknown threats as well. Therefore, during construction of the application, safe programming techniques must be adopted. At this state, the application must use all proven security algorithms and protocols for data security starting from data on transit to database security. It is advisable to use a tested libraries and framework where security is already in-vivo. Also, exceptions must be handled properly to channel all unknown exceptions.

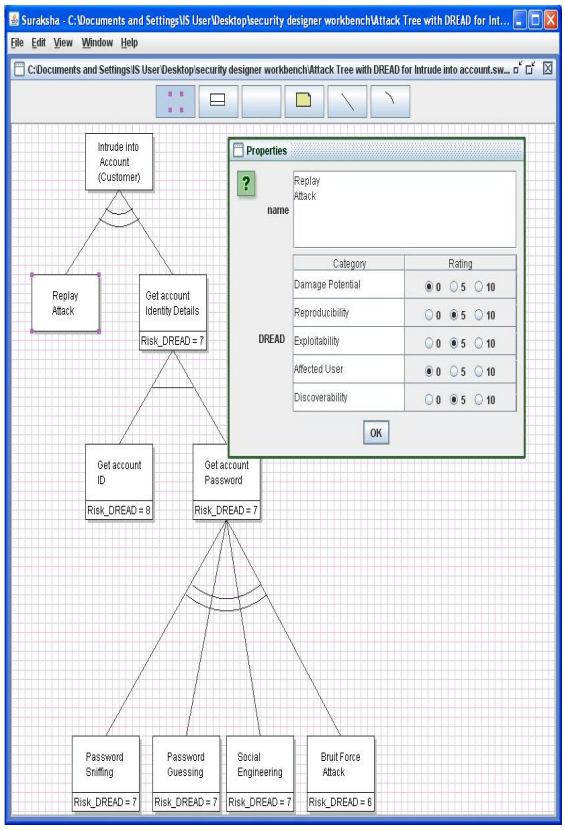

Figure 3*:* Attack tree with DREAD Rating for Intrude into Customer Account in Web-based Application (*Suraksha*)

Application must not trust any input coming from external sources. Every data coming from external sources must be validated and verified for Type, Length, and Value. Here applications are advised to use Artificial Hygiene [2] techniques to ensure that all input from external sources are hygienic to the application.

Also during coding, proper care must be taken for separation of concerns. A security function always has tendency to become anti-pattern at a later point in time. Therefore, one security function must not embed another security function – each security function must be coded separately. Secure Coding practice is required to ensure that secure coding techniques are adopted.

V. Security Testing

One of the most important phases in the secure software development is the testing of the applications and the hosted environment for potential security bugs that might be exploited by the hackers.

There will be whitebox and blackbox tests for all functions. Also, there will be negative and nonfunctional tests. For critical function, fault injection and Fuzz testing is also advised. Tests must include all security tests be it for the in-vivo security functions or in-vitro security functions.

Testing must include penetration test and ethical hacking. The testing starts from the *network perimeter*, proceeds to *platform-level testing* which basically tests the operating system, then to the *middleware* which essentially checks for the servers used for application deployment, and then *application-level* testing which tests the application, database for vulnerabilities.

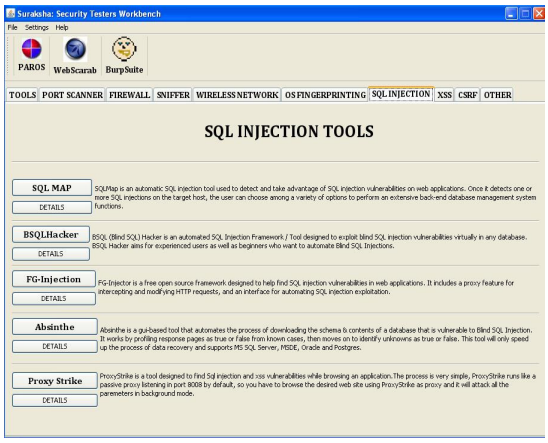

Figure 4: Security Testers' Workbench

The penetration tests done before the deployment of an application ensures that the application is secured against the attack vectors implemented in the tools and the existing vulnerabilities. It may be worth noting a point here is that the effectiveness of penetration test depends on the power of the tool; however, for ethical hacking it is the skill of the ethical hacker. Figure 4 shows the GUI of security testers' workbench which helps in performing penetration testing.

VI. Secured Deployment

In traditional Software Development Life Cycle (SDLC), deployment is considered nonfunctional requirement. However, in SaSDLC, production environment is part of functional requirement. During DREAD rating some security function has been set-aside to be addressed in-vitro through perimeter security. Therefore, it is important that these threats are looked into as functional requirement.

The deployment security deals with the security of an application at runtime. This is addressed using firewalls, proxies, antivirus, IDS, IPS etc. The concept of Artificial Hygiene is implemented hare through *application-level firewalls*, which intercepts the input to the application, and tests them for any malicious data and filters them. This ensures that the application is guarded against the real-time attacks in the production environment.

VII. Suraksha – Tool for SaSDLC

We have taken the above philosophy of SaSDLC and developed a tool called *Suraksha* (http://isea.nitk.ac.in/suraksha/) that helps Security Requirement Analysis through Security Designers' workbench. *Suraksha* offers the Asset evaluation (Figure 1), standard Use-case analysis and to depict it graphically. *Suraksha* provide a simple and efficient GUI to draw Misuse case diagram as shown in Figure 2. User can easily add actor node, misactor node, use case node, misuse case node and can easily draw various relationship between them like extend, mitigate, threaten etc by selecting suitable item from the panel. Use case and Misuse case are combined to define the system. To co-represent Use cases and Misuse cases together, Use case is black in white and Misuse case is shown in an inverted format – white in black. *Suraksha* also offers GUI for the user to document the textual representation of Misuse cases. Sindre and Opdahl focused on templates for Misuse cases in [13]. *Suraksha* uses the Misuse case template suggested by them.

Suraksha provides GUI for attack tree analysis as depicted in Figure 3. For each abstract threat mentioned in Misuse case, detailed information about the threat can be obtained by drawing an attack tree corresponding to the threat. User can draw an attack tree easily using this tool starting with abstract threat as root node. Various paths possible to achieve the goal (root node) are explored. User can draw all possibilities by creating children to a node and connect these children using AND or OR component. AND component is represented by straight line and OR component is represented using double line arc. User needs to select the required items from panel and can place the items in required position. To facilitate the designer, there are some standards threat models available in the library and can be used by the user. These threat models help to identify various attacks and their relationship. In real system these threats need to be mitigated. Also, the impact of these threats needs to be measured.

To measure the impact of each threat, DREAD technique is used. When a node in an attack tree is selected and right clicked, there is provision for the user to enter suitable values for Damage Potential, Reproducibility, Exploitability, Affected Users and Discoverability.

Suraksha offers Testers' Workbench to perform a structured security testing. Open source and

proprietary tools are used [14, 15] for security testing of application and the production environment. The following are the tools available for performing various tests in *Suraksha*:

- *Network discovery/ Port Scanning:* Nmap, Netcat, Hping, Scapy, NBTScan.
- *Network Protocols Testing:* Yersinia supports testing the network level protocols like Spanning tree protocol (STP), Cisco Discovery protocol (CDP), Dynamic Trunking protocol (DTP).
- *Firewalls Testing:* Netcat, HPing
- *Network Sniffing:* Wireshark, TCPDump/WinDump, Ettercap, Dsniff
- *Wireless network tools:* Kismet, Aircrack, KisMAC
- *VPN Testing Tool:* Ike-Scan
- *OS Fingerprinting:* Nmap, Hping, P0f, Xprobe.
- *System auditing:* Nemesis combined with Fragrouter provides a good system audit.
- *Web server Testing:* Nikto, Wikto.
- *Application Fingerprinting:* Nmap, THC-Amap.
- *Web Application Testing:* The web applications are vulnerable to different types of attacks depending on various scenarios, top of them being Injection attacks like SQL Injection, Cross-site Scripting and Cross-site request forgery. There are exclusive tools available for testing each of the above mentioned attacks:
- *SQL Injection Testing:* Absinthe, SQLMap, BSQL Hacker, SQL Ninja, SQLIer, SQL Power Injector, Sara, FG-Injector, Paros Proxy, SPIKE Proxy, Burp Suite.
- *Cross-site scripting:* VulnDetector, Paros Proxy, Web scarab.
- *Cross-site Request forgery:* Web scarab, CSRFTester.

VIII. CONCLUSION

This paper focuses on Security-aware Software Development Life Cycle and a tool to facilitate such activity. This Open Source Workbench *Suraksha* tool is developed at the *Information Security Lab*, *Department of Computer Engineering, National Institute of Technology Karnataka, Surathkal*. The workbench allows a security professional to design security-aware applications starting at security requirements through misuse case, followed by identification of threats through attack tree. Once the attack paths are known, the tool allows the user to rate different threats and to prioritize them. This results in a list of possible attacks. These attacks along with their corresponding countermeasures are included as part of functional requirement. This is then used to design the secure system through security patterns. During construction, the safe coding techniques need to be used so that the application is capable of protecting itself against unknown threats. Finally the application must be tested against various threats. At the last step the application is deployed in a secured environment.

REFERENCES

[1] P. Naur and B. Randell, (Eds.). Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO (1969) 231pp.

[2] Asoke K Talukder and Manish Chaitanya, *Architecting Secure Software Systems*, Auerbach Publications, 2008.

[3] Shawn Hernan, Scott Lambert, Tomasz Ostwald, Adam Shostack, "Uncover Security Design Flaws using The STRIDE Approach" msdn.microsoft.com, Nov. 2006. [Online]. Available: http://msdn.microsoft.com/en-us/magazine/cc163519.aspx. [Accessed: July 21, 2008].

[4] F. Swiderski and W. Snyder, *Threat Modeling*. Washington: Microsoft Press, 2004.

[5] Bruce Schneir, "Modeling Security Threats", December 1999. [Online]. Available: http://www.schneier.com/paper-attacktrees-ddj-ft.html [Accessed: Aug. 17, 2008].

[6] Fredrik Moberg, "Security Analysis of an Information System using an Attack tree-based Methodology," Master's Thesis, Chalmers University of Technology, Goteborg, Sweden, 2000.

[7] Mamadou H. Diallo, Jose Romero-Mariona, Susan Elliott Sim and Debra J. Richardson, "A Comparative Evaluation of Three Approaches to Specifying Security Requirements," presented at 12th Working Conference on Requirements Engineering: Foundation for Software Quality, Luxembourg, 2006.

[8] Guttorm Sindre and Andreas L Opdahl, "Capturing Security Requirements by Misuse Cases," in Proc. *14th Norwegian Informatics Conference (NIK'2001)*,Troms, Norway, Nov. 2001.

[9] G. Sindre and A.L. Opdahl, "Eliciting Security Requirements by Misuse Cases," in Proc. *37th Conf. Techniques of Object-Oriented Languages and Systems, TOOLS Pacific 2000*, 2000, pp. 120–131.

[10] G. Sindre and A.L. Opdahl, "Eliciting security requirements with misuse cases," *Requirements Engineering*, Vol. 10, No. 1, pp. 34-44, Jan.2005.

[11] J.D. Meier, et al., "Improving Web Application Security: Threats and Countermeasures," *msdn.microsoft.com*, June 2003.[Online].Available:www.msdn.microsoft.com/en-us/library/aa302419.aspx [Accessed: July.29, 2008].

[12] Joseph W. Yoder and Jeffrey Barcalow, "Architectural Patterns for Enabling Application Security," in Proc.*4th Conference on Patterns Languages of Programs (PLoP '97)* Monticello, Illinois, Sept.1997.

[13] Guttorm Sindre and Andreas L. Opdahl, "Templates for Misuse Case Description," in Proceedings *of the 7 th International Workshop on Requirements Engineering, Foundation for Software Quality (REFSQ'2001),* Interlaken, Switzerland, June 2001.

[14] Security tools [Online]. Available: www.insecure.org

[15] Clayton, R. (2007) 'Hacking tools guidance finally appear, http://www.lightbluetouchpaper.org/2007/12/31/hacking-tool-guidance-finally-appears/