4.2 2 Knowledge Area: Software Security

The Software Security knowledge area focuses on the development and use of software that reliably preserves the security properties of the information and systems it protects. The security of a system, and of the data it stores and manages, depends in large part on the security of its software. The security of software depends on how well the requirements match the needs that the software is to address, how well the software is designed, implemented, tested, and deployed and maintained. The documentation is critical for everyone to understand these considerations, and ethical considerations arise throughout the creation, deployment, use, and retirement of software.

The Software Security knowledge area addresses these security issues. The knowledge units within this knowledge area are comprised of fundamental principles and practices.

4.2.1 Knowledge Units and Topics

The following table lists the principles essentials, knowledge units, and topics of the Software Security knowledge area. These knowledge units have been validated by the Software Security Working Group using the Open Web Application Security Project (OWASP) Top 10 and the IEEE "Avoiding the Top 10 Software Security Design Flaws."

SOFTWARE SECURITY

Essentials

- Fundamental design principles including least privilege, open design, and abstraction,
- Security requirements and their role in design,
- Implementation issues,
- Static and dynamic testing,
- Configuring and patching, and
- Ethics, especially in development, testing and vulnerability disclosure.

Knowledge Units	Topics	Description/Curricular Guidance
Fundamental Principles [*See also* *Component Security KA* *for related content*]		This knowledge unit introduces the principles that underlie both design and implementation. The first five are restrictiveness principles, the next three are simplicity principles, and the rest are methodology principles.
	Least privilege	Software should be given only those privileges that it needs to complete its task.
	Fail-safe defaults	The initial state should be to deny access unless access is explicitly required. Then, unless software is given explicit access to an object, it should be denied access to that object and the protection state of the system should remain unchanged.
	Complete mediation	Software should validate every access to objects to ensure that the access is allowed.
	Separation	Software should not grant access to a resource, or take a security-relevant action, based on a single condition.

	Minimize trust	Software should check all inputs and the results of all security-relevant actions.
	Economy of mechanism	Security features of software should be as simple as possible.
	Minimize common mechanism	The sharing of resources should be reduced as much as possible.
	Least astonishment	Security features of software, and security mechanisms it implements, should be designed so that their operation is as logical and simple as possible.
	Open design	Security of software, and of what that software provides, should not depend on the secrecy of its design or implementation.
	Layering	Organize software in layers so that modules at a given layer interact only with modules in the layers immediately above and below it. This allows you to test the software one layer at a time, using either top-down or bottom-up techniques, and reduces the access points, enforcing the principle of separation.
	Abstraction	Hide the internals of each layer, making only the interfaces available; this enables you to change how a layer carries out its tasks without affecting components at other layers.
	Modularity	Design and implement the software as a collection of co-operating components (modules); indeed, each module interface is an abstraction.
	Complete linkage	Tie software security design and implementation to the security specifications for that software.
	Design for iteration	Plan the design in such a way that it can be changed, if needed. This minimizes the effects with respect to the security of changing the design if the specifications do not match an environment that the software is used in.
	Design	This knowledge unit describes techniques for including security considerations throughout the design of software.
	Derivation of security requirements	Beginning with business, mission, or other objectives, determine what security requirements are necessary to succeed. These may also be derived, or changed, as the software evolves.
	Specification of security requirements	Translate the security requirements into a form that can be used (formal specification, informal specifications, specifications for testing).
	Software development lifecycle/Security development lifecycle	Include the following examples: waterfall model, agile development and security.

	Programming languages and type-safe languages	Discuss the problems that programming languages introduce, what type-safety does, and why it is important.
	Implementation	This knowledge unit describes techniques for including security considerations throughout the implementation of software.
	Validating input and checking its representation	For this topic: ● Check bounds of buffers and values of integers to be sure they are in range, and ● Check inputs to make sure they are what is expected and will be processed/ interpreted correctly.
	Using APIs correctly	For this topic: ● Ensure parameters and environments are validated and controlled so that the API enforces the security policy properly, and ● Check the results of using the API for problems.
	Using security features	For this topic: ● Use cryptographic randomness, and ● Properly restrict process privileges.
	Checking time and state relationships	For this topic: ● Check that the file acted upon is the one for which the relevant attributes are checked, and ● Check that processes run.
	Handling exceptions and errors properly	For this topic: ● Block or queue signals during signal processing, if necessary, and ● Determine what information should be given to the user, balancing usability with any need to hide some information, and how and to whom to report that information.
	Programming robustly	This topic is sometimes called secure or defensive programming. Curricular content should include: ● Only deallocate allocated memory, ● Initialize variables before use, and ● Don't rely on undefined behavior.
	Encapsulating structures and modules	This topic includes classes and other instantiations. Example: isolating processes.
	Taking environment into account	Example: don't put sensitive information in the source code.
Analysis and Testing *[See also Component Security KA for related content]*		This knowledge unit introduces testing considerations for validating that the software meets stated (and unstated) security requirements and specifications. Unstated requirements include those related to robustness in general.
	Static and dynamic analysis	This topic describes the different methods for each of these, includes how static and dynamic analysis work together, and the limits and benefits of each, as well as how to perform these types of analyses on very large software systems.

	Unit testing	This topic describes how to test component parts of the software, like modules.
	Integration testing	This topic describes how to test the software components as they are integrated
	Software testing	This topic describes how to test the software as a whole, and place unit and integration testing in a proper framework.
	Deployment and Maintenance	This knowledge unit discusses security considerations in the use of software, and in its deployment, maintenance, and removal.
	Configuring	This topic covers how to set up the software system to make it function correctly.
	Patching and the vulnerability lifecycle	This topic includes managing vulnerability reports, fixing the vulnerabilities, testing the patch and patch distribution.
	Checking environment	This topic covers ensuring the environment matches the assumptions made in the software, and if not, how to handle the conflict
	DevOps	This topic combines development and operation, and the automation and monitoring of both.
	Decommissioning/Retiring	This topic describes what happens when the software is removed, and how to remove it without causing security problems.
	Documentation	This knowledge unit describes how to introduce and include information about security considerations in configuration, use, and other aspects of using the software and maintaining it (including modifying it when needed).
	Installation documents	This topic includes installation and configuration documentation.
	User guides and manuals	This topic includes tutorials and cheat sheets (brief guides); these should emphasize any potential security problems the users can cause.
	Assurance documentation	This topic focuses on how correctness was established, and what *correctness* means
	Security documentation	This topic focuses on potential security problems, how to avoid them, and if they occur, what the effects might be and how to deal with
Ethics *[See also Organizational Security KA,and Societal Security KA for related content.]*		This knowledge unit introduces ethical considerations in all of the above areas, so students will be able to reason about the consequences of security-related choices and effects.

	Ethical issues in software development	This topic covers code reuse (licensing), professional responsibility, codes of ethics such as the ACM/IEEE-CS Software Engineering Code of Ethics and Professional Practice.
	Social aspects of software development	This topic covers considerations of the effects of software under development, both when the software works properly and the consequences of poor or non-secure programming practices.
	Legal aspects of software development	This topic discusses the liability aspects of software, regulations; also compliance and issues related to it.
	Vulnerability disclosure	This topic covers how to disclose, to whom to disclose, and when to disclose ("responsible disclosure").
	What, when and why to test	This topic describes the ethical implications of testing, especially including corner cases.

4.2.2 Essentials and Learning Outcomes

Students are required to demonstrate proficiency in each of the essential concepts through achievement of the learning outcomes. Typically, the learning outcomes lie within the *understanding* and *applying* levels in the Bloom's Revised Taxonomy (http://ccecc.acm.org/assessment/blooms).

Essentials	Learning outcomes
Fundamental Design Principles; Least Privilege, Open Design, and Abstraction	
	Discuss the implications of relying on open design or the secrecy of design for security.
	List the three principles of security.
	Describe why each principle is important to security.
	Identify the needed design principle.
Security requirements and the roles they play in design	
	Explain why security requirements are important.
	Identify common attack vectors.
	Describe the importance of writing secure and robust programs.
	Describe the concept of privacy including personally identifiable information.
Implementation issues	
	Explain why input validation and data sanitization are necessary.
	Explain the difference between pseudorandom numbers and random numbers.
	Differentiate between secure coding and patching and explain the advantage of using secure coding techniques.
	Describe a buffer overflow and why it is a potential security problem.
Static, dynamic analysis	
	Explain the difference between static and dynamic analysis.
	Discuss a problem that static analysis cannot reveal.
	Discuss a problem that dynamic analysis cannot reveal.

Configuring, patching	
	Discuss the need to update software to fix security vulnerabilities.
	Explain the need to test software after an update but before the patch is distributed.
	Explain the importance of correctly configuring software.
Ethics, especially in development, testing, and vulnerability disclosure	
	Explain the concept that because you can do it, it doesn't mean you should do it.
	Discuss the ethical issues in disclosing vulnerabilities.
	Discuss the ethics of thorough testing, especially corner cases.
	Identify the ethical effects and impacts of design decisions.