

# **Unit 1: Introduction to Secure Software Development**

# Software Architecture vs Software Design

- Though both terms sometimes are often used interchangeably, the rough distinction of architecture versus design can be summarized as follows:

Software Architecture	Software Design
is involved with the higher level of description structures and interactions in a system.	is all about the organization of parts or components of the system and the subsystems involved in making the system.
is concerned with those questions that entail decision making about the skeleton of the system,	The problems here are typically closer to the code or modules in question
involving not only its functional but also its organizational, technical, business, and quality attributes.	
is about the design of the entire system	is mostly about the details, typically at the implementation level of the various subsystems and components that make up those subsystems.

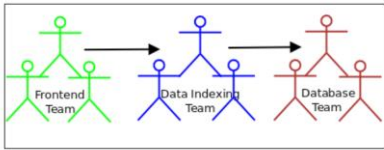
# Aspects of Software Architecture

In both the formal IEEE definition and the rather informal definition given earlier, we find some common, recurring themes. It is important to understand them in order to take our discussion on software architecture further:

- **System:** A system is a collection of components organized in specific ways to achieve a specific functionality. A software system is a collection of such software components. A system can often be subgrouped into subsystems.
- **Structure:** A structure is a set of elements that are grouped or organized together according to a guiding rule or principle. The elements can be software or hardware systems. A software architecture can exhibit various levels of structures depending on the observer's context.
- **Environment:** The context or circumstances in which a software system is built, which has a direct influence on its architecture. Such contexts can be technical, business, professional, operational, and so on.
- **Stakeholder:** Anyone, a person or groups of persons, who has an interest or concern in the system and its success. Examples of stakeholders are the architect, development team, customer, project manager, marketing team, and others.

(Pillai, 2017: 4)

# Characteristics of Software Architecture

An architecture	Explanations
defines a structure	Use class diagram in order to represent the relationships between the subsystems
picks a core set of elements	does not set out to document everything about every component of the system
captures early design decisions	Early design decisions need to be arrived at after careful analysis of the requirements and matching them with the constraints – such as organizational, technical, people, and time constraints.
manages stakeholder requirements	However, it is not possible to address each stakeholder requirement to its fullest due to an often contradictory nature of such requirements. An architecture also provides a common language among the stakeholders, which allows them to communicate efficiently via expressing these constraints, and helping the architect zero-in towards an architecture that best captures these requirements and their trade-offs.
influences the organizational structure	<div>The following diagram shows the mapping to the team structure which would be building this application:</div> 
is influenced by its environment	An environment imposes outside constraints or limits within which an architecture must function (often called architecture in context)
documents the system	The simplest way to document an architecture is to create diagrams for the different aspects of the system and organizational architecture such as Component Architecture, Deployment Architecture, Communication Architecture, and the Team or Enterprise Architecture. Other data that can be captured early include the system requirements, constraints, early design decisions, and rationale for those decisions.
often conforms to a pattern	Examples of such patterns are Client-Server, Pipes and Filters, Data-based architectures, and others.

# Importance of Software Architecture

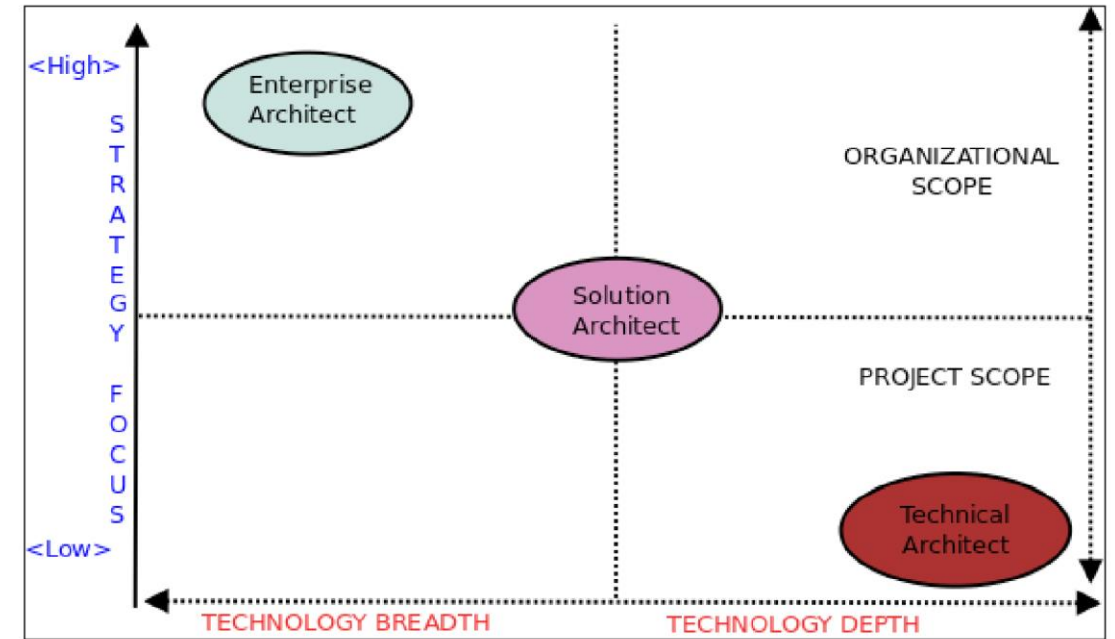
- Every system has an architecture, whether it is documented or not
- Documenting an architecture makes it formal, allowing it to be shared among stakeholders, making change management and iterative development possible
- All the other benefits and characteristics of Software Architecture are ready to be taken advantage of when you have a formal architecture defined and documented
- You may be still able to work and build a functional system without a formal architecture, but it would not produce a system which is extensible and modifiable, and would most likely produce a system with a set of quality attributes quite far away from the original requirements

"Enterprise Architecture is a conceptual blueprint that defines the structure and behavior of an organization. It determines how the organization's structure, processes, personnel and flow of information is aligned to its core goals to efficiently achieve its current and future objectives."

"A system architecture is the fundamental organization of a system, represented by its structural and behavioral views. The structure is determined by the components of the system and the behavior by the relationships between them and their interaction with external systems."

(Pillai, 2017: 16)

The following diagram depicts the different focus areas and scopes of the different architect roles that we've discussed so far:



Scope and focus of various architect roles in a software organization

(Pillai, 2017: 17)

# Architectural quality attributes

Modifiability	<p>Modifiability can be defined as the ease with which changes (local, non-local, global) can be made to a system, and the flexibility at which the system adjusts to the changes.</p> <ul style="list-style-type: none"><li>• Difficulty: The ease with which changes can be made to a system</li><li>• Cost: In terms of time and resources required to make the changes</li><li>• Risks: Any risk associated with making changes to the system</li></ul>
Testability	<p>Testability refers to how much a software system is amenable to demonstrating its faults through testing. Testability can also be thought of as how much a software system hides its faults from end users and system integration tests – the more testable a system is, the less it is able to hide its faults.</p>
Scalability	<p>Scalability of a system is its capacity to accommodate increasing workload on demand while keeping its performance within acceptable limits.</p> <p>Horizontal &amp; Vertical Scalability</p>
Performance	<p>"Performance of a computer system is the amount of work accomplished by a system using a given unit of computing resource. Higher the work/unit ratio, higher the performance."</p> <p>Linked to vertical scalability</p>
Availability	<p>Availability refers to the property of readiness of a software system to carry out its operations when the need arises.</p> <p>Linked to reliability of a system</p>
Security	<p>Security, in the software domain, can be defined as the degree of ability of a system to avoid damage to its data and logic from unauthenticated access, while continuing to provide services to other systems and roles that are properly authenticated.</p>
Deployability	<p>Deployability is the degree of ease with which software can be taken from the development to the production environment. It is more of a function of the technical environment, module structures, and programming runtime/languages used in building a system, and has nothing to do with the actual logic or code of the system</p>

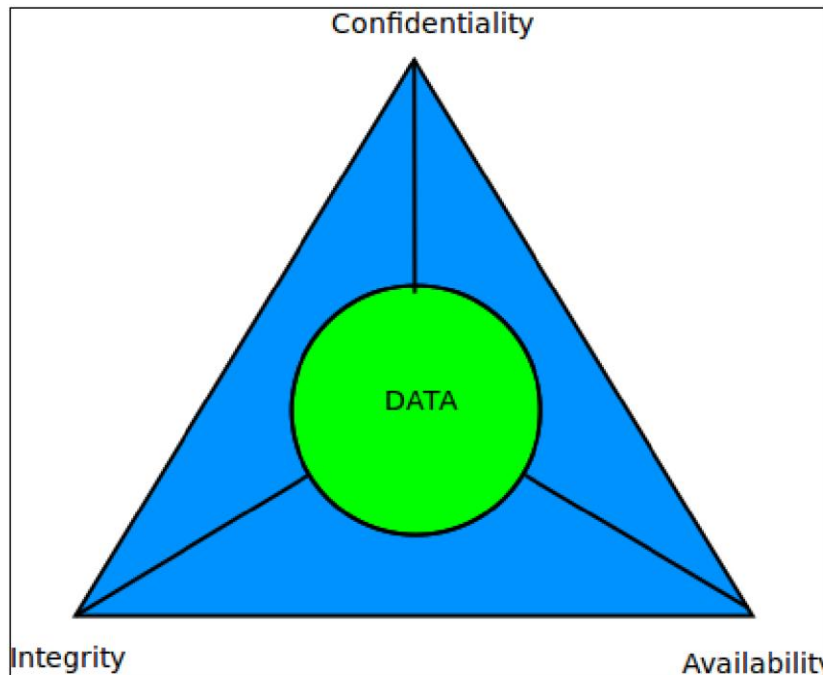


# Information Security Architecture (Pillai, 2017: chapter 6).

- Security is an important aspect of software architecture

*Chapter 6*

The three aspects of Confidentiality, Integrity, and Availability – often called the CIA triad form the corner stones of building an information security architecture for your system.



CIA triad of Information Security Architecture



Aided by:  
Authentication (verifies identity of user)  
Authorisation (gives right to perform a task)  
Non-reputability (Security techniques that guarantee that users involved in a transaction cannot later deny that the transaction happened).



# Secure Coding

- Secure coding is the practice of software development that guards programs against security vulnerabilities, and makes it resistant to malicious attacks right from program design to implementation.
- It is about writing code that is inherently secure as opposed to thinking of security as a layer which is added on later.
- Security requirements should be identified early in the development cycle, and these should be propagated to subsequent stages of development of the system to make sure that compliance is maintained

# The practice or strategies of secure coding include the following main tasks:

1. Definition of areas of interest of the application: Identify important assets in code/data of the application which are critical and needs to be secured.
2. Analysis of software architecture: Analyze the software architecture for obvious security flaws. Secure interaction between components with a view to ensure data confidentiality and integrity. Ensure confidential data is protected via proper authentication and authorization techniques. Ensure availability is built into the architecture from ground up.
3. Review of implementation details: Review the code using secure coding techniques. Ensure peer review is done with a view to finding security holes. Provide feedback to the developer and make sure the required changes are made.
4. Verification of logic and syntax: Review code logic and syntax to ensure there are no obvious loop holes in the implementation. Make sure programming is done keeping with commonly available secure coding guidelines of the programming language/platform.
5. Whitebox/Unit Testing: The developer unit tests his code with security tests apart from tests ensuring functionality. Mock data and/or APIs can be used to virtualize third party data/API required for testing.
6. Blackbox Testing: The application is tested by an experienced QA engineer who looks for security loop holes such as unauthorized access to data, path ways accidentally exposing code and or data, weak passwords or hashes etc. The testing reports are fed back the stakeholders including the architect to make sure the loopholes identified are fixed.

# Common security vulnerabilities

- Overflow errors: These include the popular and often abused buffer overflow errors, and the lesser known but still vulnerable arithmetic or integer overflow errors:
  - Buffer overflow: Buffer overflows are produced by programming errors that allow an application to write past the end or beginning of a buffer. Buffer overflows allow attackers to take control over systems by gaining access to the applications stack or heap memory by carefully crafted attack data.
  - Integer or arithmetic overflow: These errors occur when an arithmetic or mathematical operation on integers produces a result that is too large for the maximum size of the type used to store it.
- Unvalidated/Improperly validated input: A very common security issue with modern web applications, unvalidated input can cause major vulnerabilities, where attackers can trick a program into accepting malicious input such as code data or system commands, which, when executed, can compromise a system. A system that aims to mitigate this type of attack should have filters to check and remove content that is malicious, and only accept data that is reasonable and safe to the system.
- Improper access control: Modern day applications should define separate roles for their classes of users, such as regular users, and those with special privileges, such as superusers or administrators. When an application fails to do this or does it incorrectly, it can expose routes (URLs) or workflows (series of actions specified by specific URLs containing attack vectors), which can either expose sensitive data to attackers, or, in the worst case, allow an attacker to compromise and take control of the system.
- Cryptography issues: Simply ensuring that access control is in place is not enough for hardening and securing a system. Instead, the level and strength of security should be verified and ascertained, otherwise, your system can still be hacked or compromised.
  - HTTP instead of HTTPS
- Insecure authentication
- Use of weak passwords
- Reuse of secure hashes/secret keys
- Weak encryption techniques
- Weak hashing techniques
- Invalid or expired certificates/keys
- Password enabled SSH
- Information leak: Server meta information, Open index pages, Open ports, Race conditions, System clock drifts, Insecure file/folder operations

# Python & Security

Types of security issues with Python (on the console)	Solutions & Advice
Reading input (passwords, global variables are exposed)	prefer <code>raw_input</code> over <code>input</code> For reading passwords, use libraries such as <code>getpass</code> , and also perform validations on the returned data.
Evaluating arbitrary input (The <code>eval</code> function in Python is very powerful, but it is also dangerous)	avoid using <code>eval</code> and its cousin <code>exec</code> If you have to use <code>eval</code> , make it a point to never use it with user input strings, or data read from third-party libraries, or APIs on which you have no control. Use <code>eval</code> only with input sources and return values from functions that you have control of and that you trust.
Overflow Errors	Guard against integer overflows by using exception handlers. Python doesn't suffer from pure buffer overflow errors, as it always checks its containers for read/write access beyond the bounds and throws exceptions. For overridden <code>__len__</code> methods on classes, catch the overflow or <code>TypeError</code> exceptions as required
Serialising Objects	Don't use <code>pickle</code> or <code>cPickle</code> for serialization. Favor other modules such <code>JSON</code> or <code>yaml</code> . If you absolutely have to use <code>pickle</code> / <code>cPickle</code> , use mitigation strategies such as a chroot jail or sandbox to avoid the bad effects of malicious code execution if any.
String formatting	Prefer the newer and safer <code>format</code> method of template strings over the older and unsafe <code>%s</code> interpolation.
Files	When working with files, it is a good idea to use the <code>with</code> context managers to make sure that the file descriptors are closed after the operation.
Handling passwords and sensitive information	When validating sensitive information like passwords, it is a good idea to compare cryptographic hashes rather than comparing the original data in memory
Local data	As much as possible, avoid storing sensitive data local to functions

- Security Issues with web applications such as Flask

- Server Side Template Injection (SSTI): The attack uses weaknesses in the way user input is embedded on the templates. SSTI attacks can be used to figure out internals of a web application, execute shell commands, and even fully compromise the servers. Check chapter 6 for example in Flask
- Server-Side Template Injection – Mitigation
- Denial of Service
- Cross-Site Scripting(XSS)
- Mitigation – DoS and XSS

- ADVICE

- Keep your system up to date: These reports usually go by the name of Common Vulnerabilities and Exposures (CVEs)—and sites such as Mitre (<http://cve.mitre.org>) provide a constant stream of updates.
- Similarly, the Python Open Web Application Security Project (OWASP) project is a free, third-party project aimed at creating a hardened version of Python more resilient to security threats than the standard Cpython. It is part of the larger OWASP initiative. • The Python OWASP project makes available its Python bug-reports, tools, and other artifacts via the website and associated GitHub projects. The main website for this is, and most of the code is available from the GitHub project page at: <https://github.com/ebranca/owasp-pysec/>.

# Secure coding strategies

We are coming towards the end of our discussion on the security aspects of software architecture. It is a good time to summarize the strategies that one should try and impart to a software development team from a security architect's point of view. The following is a table summarizing the top 10 of these.

SL	Strategy	How it helps
1	Validate inputs	Validate inputs from all untrusted data sources. Proper input validation can eliminate a vast majority of software vulnerabilities.
2	Keep it simple	Keep program design as simple as possible. Complex designs increase the chances of security errors being made in their implementation, configuration, and deployment.
3	Principle of least privilege	Every process should execute with the least set of system privileges necessary to complete the work. For example, to read data from <code>/tmp</code> , one doesn't need root permission, but any unprivileged user is fine.
4	Sanitize data	Sanitize data read from and sent to all third-party systems such as databases, command shells, COTs components, third-party middlewares, and so on. This lessens the chances of SQL injection, shell exploit, or other similar attacks.
5	Authorize access	Separate parts of your application by roles that need specific authentication via login or other privileges. Don't mix different parts of applications together in the same code that requires different levels of access. Employ proper routing to make sure that no sensitive data is exposed via unprotected routes.
6	Perform effective QA	Good security testing techniques are effective in identifying and eliminating vulnerabilities. Fuzz testing, penetration testing, and source code audits should be performed as part of the program.
7	Practice defense in layers	Mitigate risks with multiple layers of security. For example, combining secure programming techniques with secure runtime configuration will reduce the chances of any remaining code vulnerabilities being exposed in the runtime environment.
8	Define security requirements	Identify and document the security constraints in the early lifecycle of the system, and keep updating them making sure that any further features down the line keep up with these requirements.
9	Model threats	Use threat modeling to anticipate the threats to which the software will be subjected.
10	Architect and design for security policies	Create and maintain a software architecture that enforces a pattern of consistent security policies across your system and its subsystems.

(Pillai, 2017: 325)

# Notes from Pohl, C. & Hof, H-J. (2015)

- Considering the current attacking landscape, it is clear that developing secure software should be a main concern in all software development projects
- Agile software development methods like **Scrum** are known for reducing the initial planning phases (e.g., sprint 0 in Scrum) and for focusing more on producing running code. Scrum is also known for allowing fast adaption of the emerging software to changes of customer wishes.
- For security, this means that it is likely that there are no detailed security architecture or security implementation instructions from the start of the project. It also means that a lot of design decisions will be made during the runtime of the project.
- Hence, to address security in Scrum, it is necessary to consider security issues throughout the whole software development process.



- Secure Scrum consists of four components:
  - Identification component
  - Implementation component
  - Verification component
  - Definition of Done component

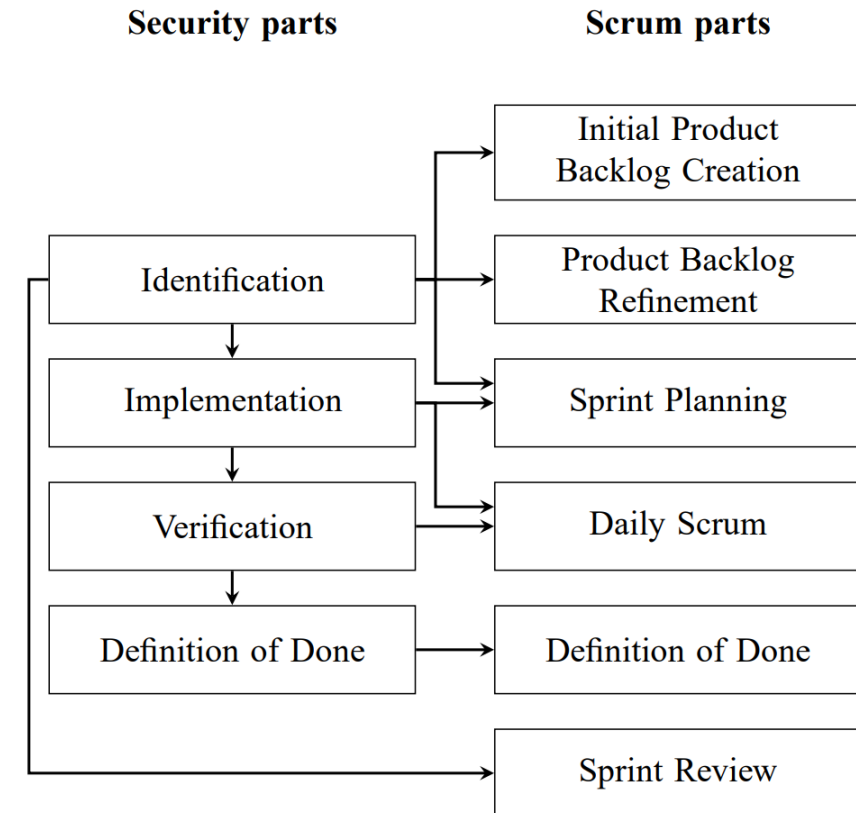


Fig. 1. Integration of Secure Scrum components into standard Scrum

# References:

- Peoples, C. (2021), Lecture Notes, Secure Software Development SSDCS\_PCOM7E, University of Essex Online, delivered November 2021
- Pillai, A.B. (2017) *Software Architecture with Python*. Birmingham, UK. Packt Publishing Ltd.
- Pohl, C. & Hof, H-J. (2015) *Secure Scrum: Development of Secure Software with Scrum*, in Proc. of the 9th International Conference on Emerging Security Information, Systems and Technologies.