Module 2: SDLC and Security Principles

CYBR 515: Software Security
Summer 2024
Mohammad Jamil Ahmad, Ph.D.

West Virginia University
JOHN CHAMBERS COLLEGE OF BUSINESS AND ECONOMICS

Table of Contents

How are software designed?

- The need of "getting something done" is probably the main reason we have software applications.
- In the world of business, these are known as business requirements.
  - Track customers, sales, stock, data.
  - The origin of a software, is the business/ human requirement.
- How can we communicate this need to a team of software developers?

The problem with communicating business requirements

- If you give 10 people an image of something (cat, dog, a planet), will they have the same description?
- There has to be a systematic way, set of rule, and unified framework to describe that
  - For example: use “feline” to describe a cat
  - Use action verbs to describe hat the cat is doing
  - Use unified language to explain how a business needs their needs satisfied

The Software Development Life Cycle (SDLC)

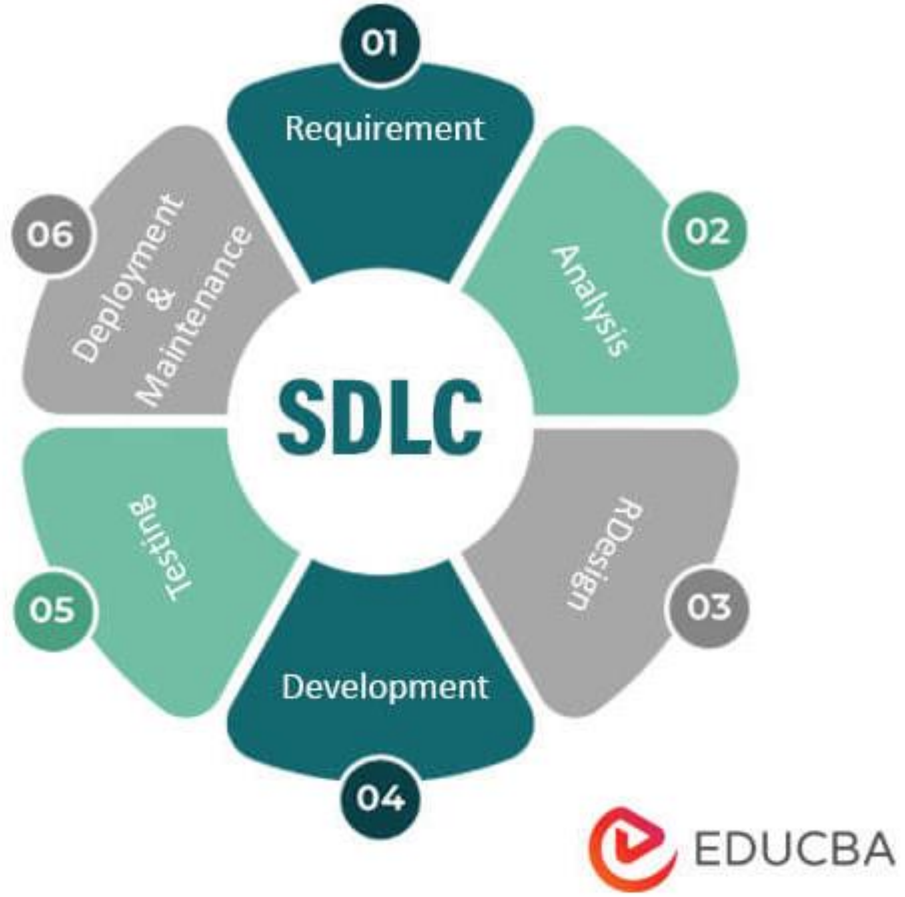
- The Software Development Life Cycle (SDLC) is a structured process used for developing software. It ensures the quality and correctness of the software built. SDLC defines a detailed plan that describes how to develop, maintain, replace, and alter or enhance specific software.
- The life cycle defines a methodology for improving the quality of software and the overall development process.

Why is SDLC important?

1. **Structured Process**: SDLC provides a systematic approach to software development, ensuring that all aspects of the project are addressed methodically.
2. **Improved Quality**: By following predefined phases, the quality of the software is enhanced, reducing the likelihood of errors and defects.
3. **Time Management**: Proper planning and phase-wise execution help in meeting deadlines and managing time efficiently.
4. **Cost Efficiency**: Identifying and fixing issues early in the development process can significantly reduce costs.
5. **Customer Satisfaction**: A well-structured development process ensures that the final product meets the customer's requirements and expectations.

Stages of SDl

1. Planning
2. Requirements Analysis
3. Design
4. Implementation (Coding)
5. Testing
6. Deployment
7. Maintenance (some resource combine this with Deployment)



1. Planning

- **Objective**: Define the project goals, scope, resources, schedule, and budget.
- **Importance**: Lays the foundation for the entire project, ensuring alignment with business objectives.
- **Activities**:
  - Feasibility study
  - Project charter creation
  - Resource allocation
  - Schedule planning
- **Inputs**:
  - Business idea or problem statement
  - Preliminary requirements
- **Outputs**:
  - Project plan
  - Feasibility report
  - Budget and timeline

2. Requirements Analysis

- **Objective**: Gather detailed requirements from stakeholders.
- **Importance**: Ensures all stakeholder needs are understood and documented.
- **Activities**:
  - Stakeholder interviews
  - Requirements workshops
  - Use case analysis
- **Inputs**:
  - Project plan
  - Preliminary requirements
- **Outputs**:
  - Requirements specification document (SRS)
  - Use cases
  - User stories

3. Design

- **Objective**: Architect the system and create detailed design specifications.
- **Importance**: Provides a blueprint for developers to follow during implementation.
- **Activities**:
  - System architecture design
  - Database schema design
  - User interface design
- **Inputs**:
  - Requirements specification document
- **Outputs**:
  - Design specification document
  - UML diagrams (Class, Sequence, Activity diagrams)
  - Prototypes

4. Implementation (Coding)

- **Objective**: Develop the software according to the design specifications.
- **Importance**: Translates design into a working software product.
- **Activities**:
  - Writing code
  - Code reviews
  - Unit testing
- **Inputs**:
  - Design specifications
  - UML diagrams
- **Outputs**:
  - Source code
  - Build scripts
  - Unit test results

5. Testing

- **Objective**: Verify and validate the software to ensure it meets requirements.
- **Importance**: Detects and fixes bugs before deployment, ensuring software quality.
- **Activities**:
  - System testing
  - Integration testing
  - Performance testing
  - User acceptance testing (UAT)
- **Inputs**:
  - Source code
  - Test plans
- **Outputs**:
  - Test reports
  - Bug reports
  - Test case documents

6. Deployment

- **Objective**: Deploy the software to the production environment.
- **Importance**: Makes the software available for use by end-users.
- **Activities**:
  - Release planning
  - Deployment scripts
  - User training
- **Inputs**:
  - Tested software
  - Deployment plan
- **Outputs**:
  - Deployed software
  - Installation guides
  - Training materials

7. Maintenance

- **Objective**: Provide ongoing support and updates to the software.
- **Importance**: Ensures the software remains functional and relevant.
- **Activities**:
  - Bug fixes
  - Updates and patches
  - Performance tuning
- **Inputs**:
  - Deployed software
  - User feedback
  - Bug reports
- **Outputs**:
  - Updated software
  - Maintenance logs
  - Performance reports

Summary of Examples Highlighting What Could Go Wrong in Software Development

- **Malicious Software Security Attacks**
- **Target Data Attack**:
  - **Incident**: Approximately 40 million customers were affected by a data breach.
  - **Impact**: Massive exposure of personal and financial information.
- **Facebook-Cambridge Analytica Scandal (2018)**:
  - **Incident**: Unauthorized access to user data through a third-party app's API.
  - **Impact**: Data of millions of users was exploited for political advertising.
- **Myspace XSS Scripting Attack**:
  - **Incident**: Over one million profiles were infected within 20 hours.
  - **Impact**: Rapid spread of malicious scripts affecting a large user base.

Unintended User Mistakes Due to Poor Software Design

- **NASA's Mars Climate Orbiter**:
  - **Incident**: Mismatch in metric and imperial units between navigation software and ground control data.
  - **Impact**: Incorrect calculations led to the orbiter's failure by approaching Mars at too low an altitude.
- **Ariane 5 Flight 501**:
  - **Incident**: Incompatible software reused from Ariane 4 caused a data conversion error.
  - **Impact**: Catastrophic failure within 40 seconds of liftoff, resulting in the rocket's destruction.
- **Therac-25 Radiation Therapy Machine**:
  - **Incident**: Design flaws and software errors led to race conditions and insufficient testing.
  - **Impact**: Patients received massive overdoses of radiation, causing severe harm and fatalities.

Unintended Programmer Mistakes

- **Heartbleed Bug**:
  - **Incident**: A programming error in the OpenSSL cryptographic software.
  - **Impact**: Allowed attackers to read sensitive data from the memory of thousands of web servers, compromising user privacy.

HOW CAN SECURITY BE INTEGRATED INTO THE SDLC? WE NEED TO UNDERSTAND THE BASIC SECURITY PRINCIPLES FIRST.

Software Security Principles

- These are the fundamental practices, guidelines, and frameworks that can helps us designing, developing, and maintaining secured software systems

1. **Least Privilege**
2. **Defense in Depth**
3. **Fail-safe Defaults**
4. **Input Validation**
5. **Output Encoding**
6. **Secure Data Storage**
7. **Authentication and Authorization**
8. **Session Management**
9. **Secure Communication**
10. **Error Handling**
11. **Secure Dependencies**
12. **Security Testing**
13. **Security Training and Awareness**
14. **Incident Response**
15. **Security by Design**
16. **Secure Development Life Cycle (SDLC)**
17. **Threat Modeling**

1. Least Privilege

- Grant individuals or processes only the minimum access and permissions they need to perform their tasks.
  - This minimizes the risk of unauthorized access or misuse of privileges.
- **Example**: A database administrator should only have access to the databases they need to manage. If they don't require access to financial databases, their permissions should not include these.

2. Defense in Depth

- Implement multiple layers of security controls and safeguards rather than relying on a single security measure.
  - This approach helps mitigate risks and reduces the impact of security breaches.
- **Example**: A web application might use HTTPS to secure data in transit, employ firewalls to block unauthorized access, and implement intrusion detection systems to monitor for suspicious activity.

3. Fail-safe Defaults

- Systems and applications should be configured to operate securely by default.
  - If a configuration or access control is not explicitly defined, it should be denied or disabled.
- **Example**: A file-sharing system should default to private settings for new files, ensuring only the owner can access them unless additional permissions are granted.

4. Input Validation

- Validate and sanitize all user inputs to prevent injection attacks, such as SQL injection and cross-site scripting (XSS).
  - This ensures that data is safe before being processed.
- **Example**: A login form should validate the entered username and password to ensure they meet expected formats and lengths, and sanitize inputs to remove any potentially malicious code.

5. Output Encoding

- Encode data before displaying it to users to prevent attacks like XSS by treating potentially malicious input as data, not executable code.
- **Example**: An application displaying user comments should encode special characters (e.g., <, >, &, ") to prevent them from being interpreted as HTML or JavaScript.

6. Secure Data Storage

- Use strong encryption and access controls to protect sensitive data at rest. Encrypting data in databases, files, and other storage locations ensures it's secure even if accessed improperly.
- **Example**: Customer information stored in a database should be encrypted using AES-256 encryption, and access to this data should be restricted based on roles.

7. Authentication and Authorization

- Implement robust authentication mechanisms to verify user identities and authorization controls to ensure users can only access permitted resources and functionality.
- **Example**: A banking app uses multi-factor authentication (MFA) for login and role-based access controls to ensure users can only view their own accounts and not others'.

8. Session Management

- Properly manage and secure user sessions to prevent session fixation, session hijacking, and other session-related attacks.
- **Example**: A web application should generate a new session ID after a successful login and use secure cookies with the HttpOnly and Secure flags.

9. Secure Communication

- Use secure protocols like HTTPS to protect data transmitted between clients and servers.
- **Example**: An online shopping site uses HTTPS to encrypt all data exchanged between the user's browser and the server, protecting credit card information during transactions.

10. Error Handling

- Implement appropriate error handling and reporting mechanisms to provide minimal information to users while logging detailed error messages for administrators.
- **Example**: A web application should display a generic error message ("An error occurred, please try again later") to users while logging detailed technical information about the error for developers.

11. Secure Dependencies

- Regularly update and patch all software components and libraries to address known vulnerabilities.
  - Ensure third-party dependencies are secure and up to date.
- **Example**: A developer should regularly check for updates to libraries like OpenSSL and apply patches to fix known security vulnerabilities such as Heartbleed.

12. Security Testing

- Conduct thorough security testing, including code reviews, penetration testing, and vulnerability scanning, to identify and remediate security issues in the software.
- **Example**: A company conducts regular penetration tests on its web application to identify and fix security vulnerabilities before attackers can exploit them.

13. Security Training and Awareness

- Promote security awareness among developers, users, and administrators and provide training to ensure they understand and follow security best practices.
- **Example**: Regular security training sessions for developers on the latest secure coding practices and awareness programs for employees on recognizing phishing attacks.

14. Incident Response

- Develop a plan to effectively respond to security breaches and minimize their impact.
- **Example**: A company has an incident response team that follows a detailed plan to contain, investigate, and remediate security incidents such as data breaches.

15. Security by Design

- Consider security from the initial stages of software design and architecture rather than attempting to bolt security onto a finished product.
- **Example**: During the design phase of a new application, developers include threat modeling and security requirements to ensure the system is secure from the start.

16. Secure Development Life Cycle (SDLC)

- Implement a secure software development life cycle incorporating security activities at each development phase, from requirements to deployment.
- **Example**: In the SDLC for a mobile banking app, each phase includes security reviews, from planning and requirements gathering to coding, testing, and deployment.

17. Threat Modeling

- Identify and assess potential threats and vulnerabilities specific to your application, allowing proactive security measures.
- **Example**: During the design phase of a new e-commerce platform, developers create threat models to identify and mitigate risks like SQL injection, XSS, and data breaches.

FRAMEWORKS TO BE AWARE OF AND FAMILIAR WITH

1. Open Web Application Security Project (OWASP)
2. The National Institute of Standards and Technology (NITS) Secure Software Development Framework
3. MITRE frameworks
4. Common SDLC models
5. Microsoft's Security Development Lifecycle
6. Confidentiality, integrity, and availability (C-I-A)

1. Open Web Application Security Project (OWASP)

- The Open Web Application Security Project (OWASP) is a nonprofit, community-driven, foundation dedicated to improving the security of software.
  – OWASP provides free and open resources, such as tools, documentation, forums, and projects, to help organizations build secure software.
- **Key Resources**:
  1. **OWASP Top Ten**: A list of the top ten most critical web application security risks.
  2. **OWASP ZAP (Zed Attack Proxy)**: A tool for finding vulnerabilities in web applications.
  3. **OWASP ASVS (Application Security Verification Standard)**: A framework for testing the security of web applications.

OWASP top 10

- **Description**: The OWASP Top 10 is a list of the top 10 most critical web application security risks.
- **Purpose**: Serves as a guideline for developers and organizations to prioritize their security efforts.
- **Importance**: Helps in identifying and mitigating the most prevalent and critical security vulnerabilities.

OWASP top 10 resources

- A must watch and read resources about OWASP top 10
  - https://www.synopsys.com/glossary/what-is-owasp-top-10.html)
  - https://www.cloudflare.com/learning/security/threats/owasp-top-10/
  - https://www.cyberdb.co/real-world-examples-for-owasp-top-10-vulnerabilities/

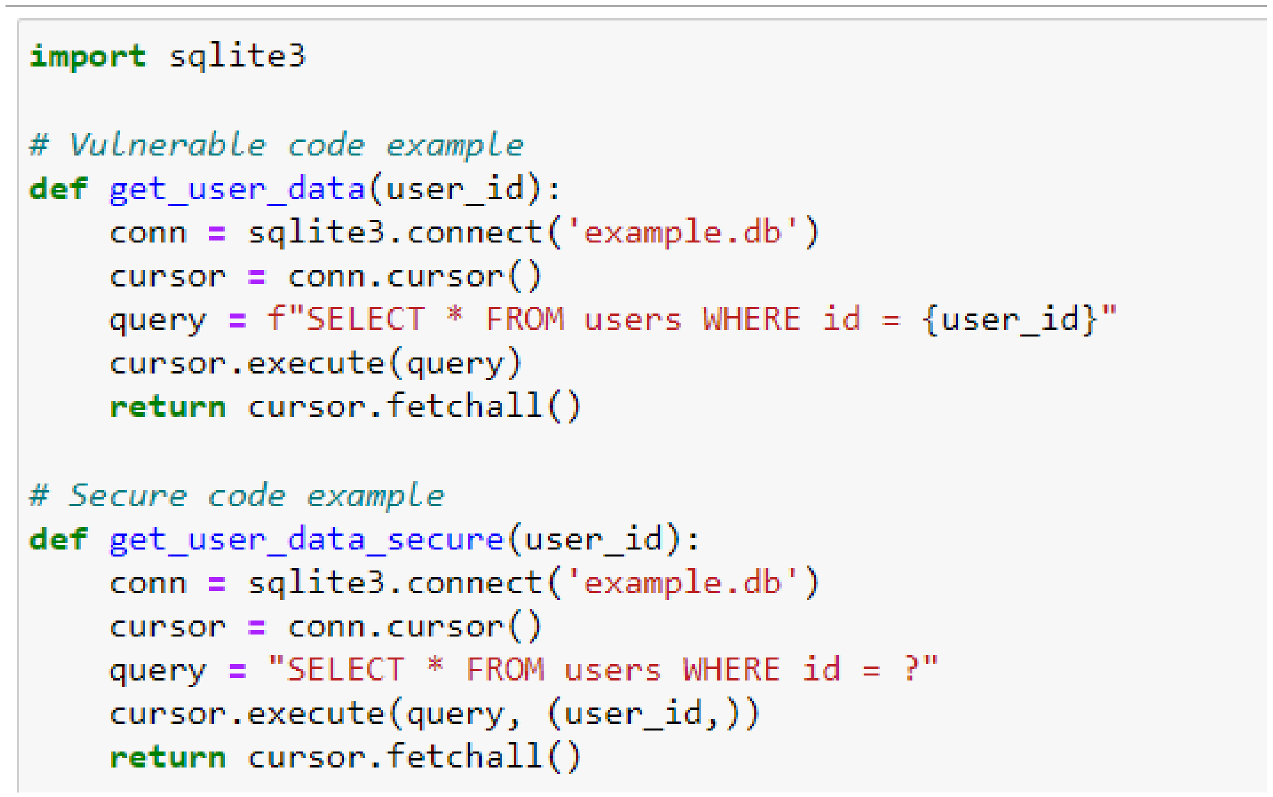
OWASP top 10 list

1. Injection
2. Broken Authentication
3. Sensitive Data Exposure
4. XML External Entities (XXE)
5. Broken Access Control
6. Security Misconfiguration
7. Cross-Site Scripting (XSS)
8. Insecure Deserialization
9. Using Components with Known Vulnerabilities
10. Insufficient Logging & Monitoring

OWASP top 10 - Injection

- **Description**: Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query.
- **Impact**: Attackers can execute arbitrary commands or access data without proper authorization.
- **How to Secure**:
  - Use parameterized queries.
  - Validate and sanitize inputs.

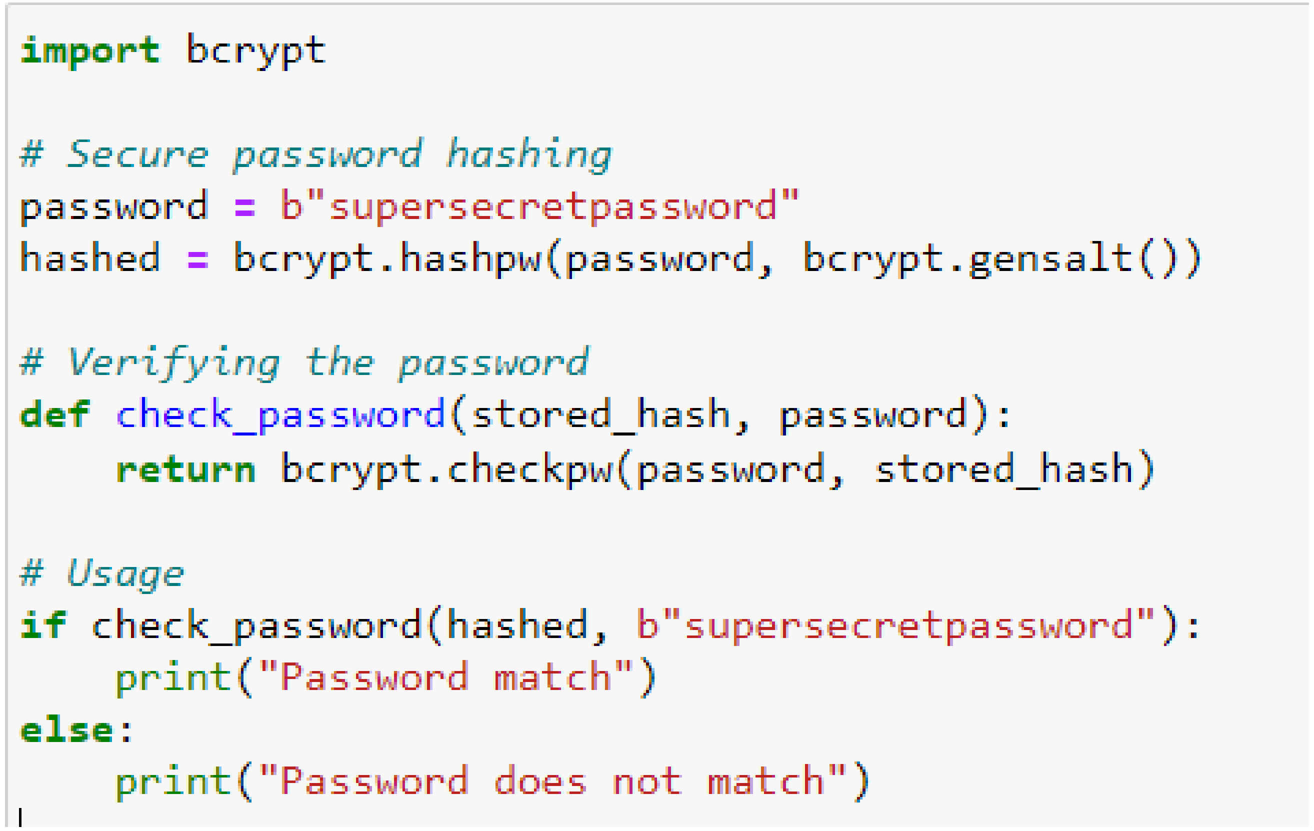
OWASP top 10 – Injection example


```
import sqlite3

# Vulnerable code example
def get_user_data(user_id):
    conn = sqlite3.connect('example.db')
    cursor = conn.cursor()
    query = f"SELECT * FROM users WHERE id = {user_id}"
    cursor.execute(query)
    return cursor.fetchall()

# Secure code example
def get_user_data_secure(user_id):
    conn = sqlite3.connect('example.db')
    cursor = conn.cursor()
    query = "SELECT * FROM users WHERE id = ?"
    cursor.execute(query, (user_id,))
    return cursor.fetchall()
```

OWASP top 10 - Broken Authentication

- **Description**: Broken authentication vulnerabilities allow attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities.
- **Impact**: Unauthorized access to user accounts and data.
- **How to Secure**:
  - Implement multi-factor authentication (MFA).
  - Use secure password storage techniques.

OWASP top 10 - Broken Authentication example

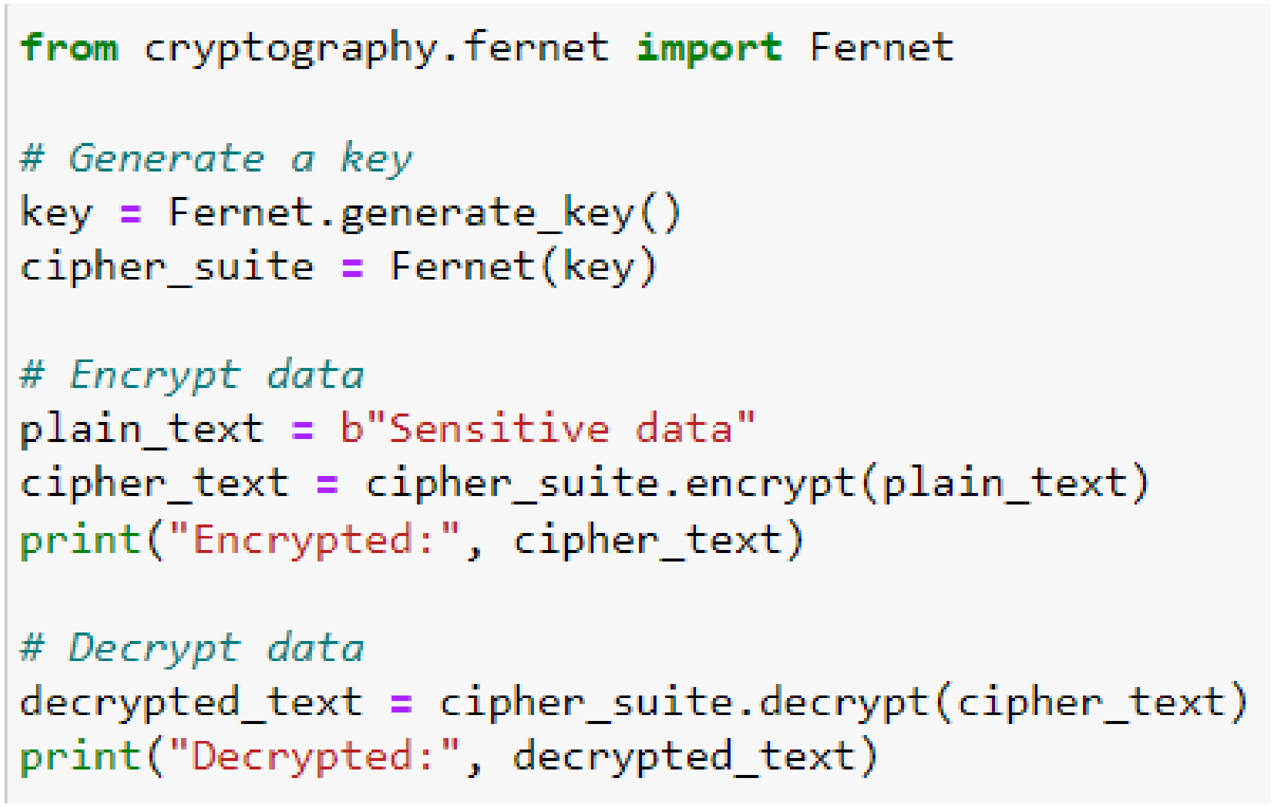


```python
import bcrypt

# Secure password hashing
password = b"supersecretpassword"
hashed = bcrypt.hashpw(password, bcrypt.gensalt())

# Verifying the password
def check_password(stored_hash, password):
    return bcrypt.checkpw(password, stored_hash)

# Usage
if check_password(hashed, b"supersecretpassword"):
    print("Password match")
else:
    print("Password does not match")
```

OWASP top 10 - Sensitive Data Exposure

- **Description**: Sensitive data exposure vulnerabilities occur when sensitive data is not properly protected.
- **Impact**: Unauthorized access to sensitive information, such as credit card data, personal information, etc.
- **How to Secure**:
  - Use encryption for data at rest and in transit.
  - Implement strong access controls.

OWASP top 10 - Sensitive Data Exposure example

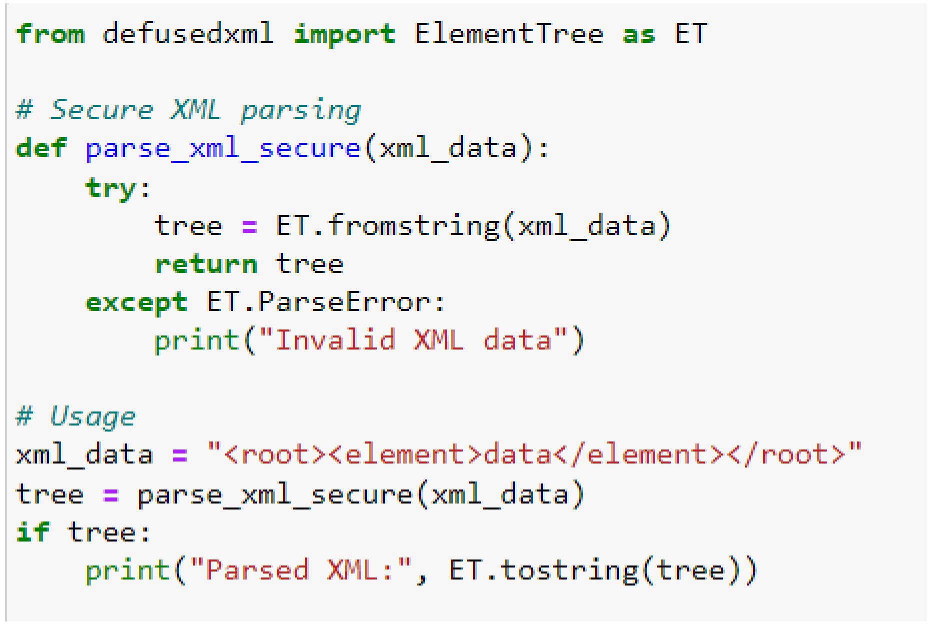


```python
from cryptography.fernet import Fernet

# Generate a key
key = Fernet.generate_key()
cipher_suite = Fernet(key)

# Encrypt data
plain_text = b"Sensitive data"
cipher_text = cipher_suite.encrypt(plain_text)
print("Encrypted:", cipher_text)

# Decrypt data
decrypted_text = cipher_suite.decrypt(cipher_text)
print("Decrypted:", decrypted_text)
```

OWASP top 10 - XML External Entities (XXE)

- **Description**: XXE vulnerabilities occur when XML input containing a reference to an external entity is processed by a weakly configured XML parser.
- **Impact**: Attackers can access internal files and services.
- **How to Secure**:
  - Disable DTDs (Document Type Definitions) in XML parsers.
  - Use less powerful data formats like JSON.

OWASP top 10 - XML External Entities (XXE) example

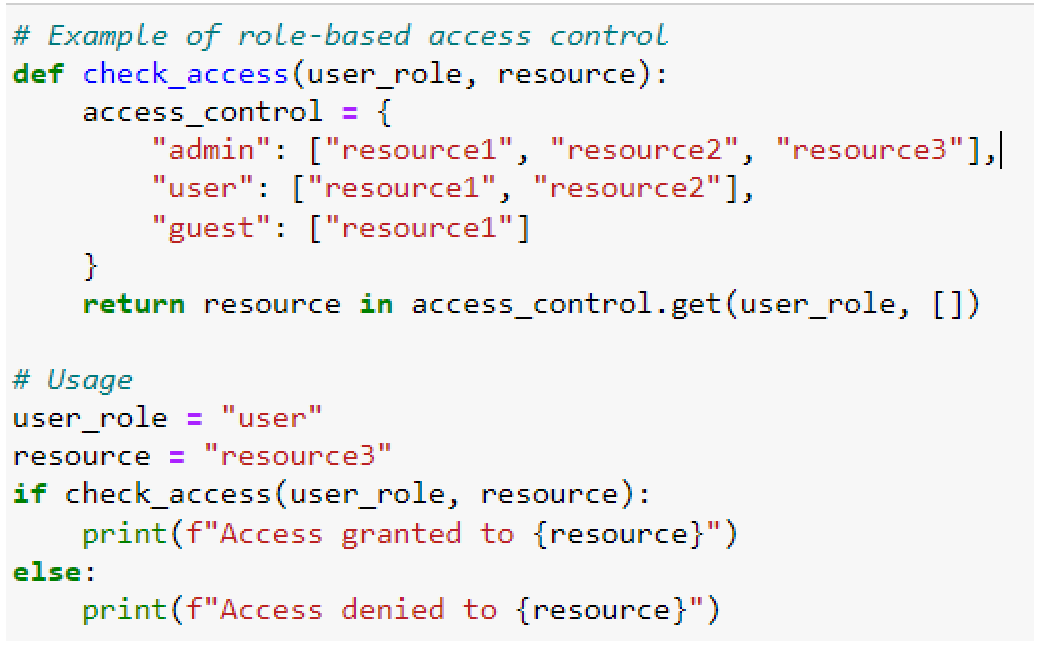


```python
from defusedxml import ElementTree as ET

# Secure XML parsing
def parse_xml_secure(xml_data):
    try:
        tree = ET.fromstring(xml_data)
        return tree
    except ET.ParseError:
        print("Invalid XML data")

# Usage
xml_data = "<root><element>data</element></root>"
tree = parse_xml_secure(xml_data)
if tree:
    print("Parsed XML:", ET.tostring(tree))
```

OWASP top 10 - Broken Access Control

- **Description**: Broken access control vulnerabilities occur when restrictions on authenticated users are not properly enforced.
- **Impact**: Attackers can access unauthorized functionality or data.
- **How to Secure**:
  - Implement proper access controls and regularly review them.
  - Use role-based access control (RBAC).

OWASP top 10 - Broken Access Control example

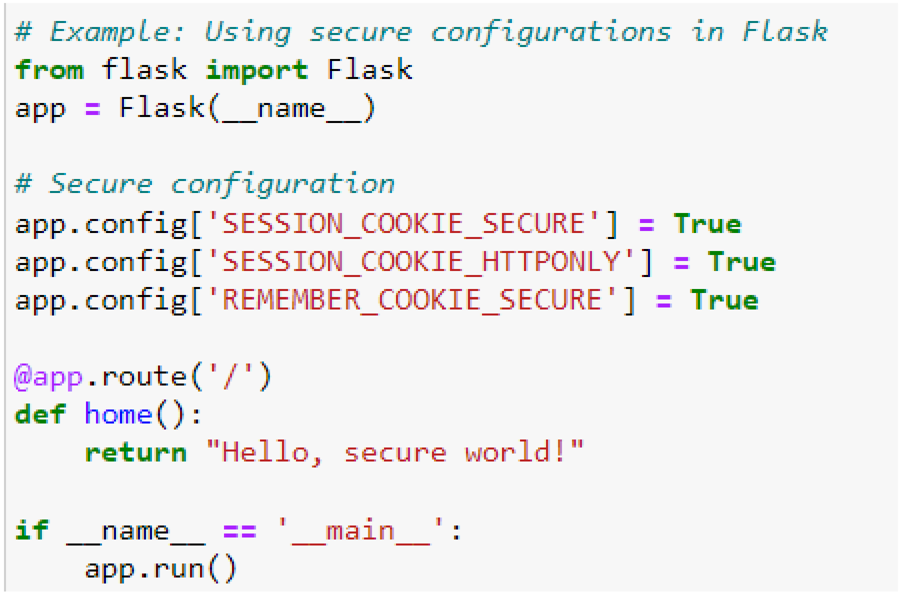


```python
# Example of role-based access control
def check_access(user_role, resource):
    access_control = {
        "admin": ["resource1", "resource2", "resource3"],
        "user": ["resource1", "resource2"],
        "guest": ["resource1"]
    }
    return resource in access_control.get(user_role, [])

# Usage
user_role = "user"
resource = "resource3"
if check_access(user_role, resource):
    print(f"Access granted to {resource}")
else:
    print(f"Access denied to {resource}")
```

OWASP top 10 - Security Misconfiguration

- **Description**: Security misconfiguration vulnerabilities occur when security settings are defined, implemented, and maintained improperly.
- **Impact**: Attackers can exploit misconfigured settings to gain unauthorized access.
- **How to Secure**:
  - Implement a repeatable hardening process.
  - Regularly audit and update configurations.

OWASP top 10 - Security Misconfiguration example

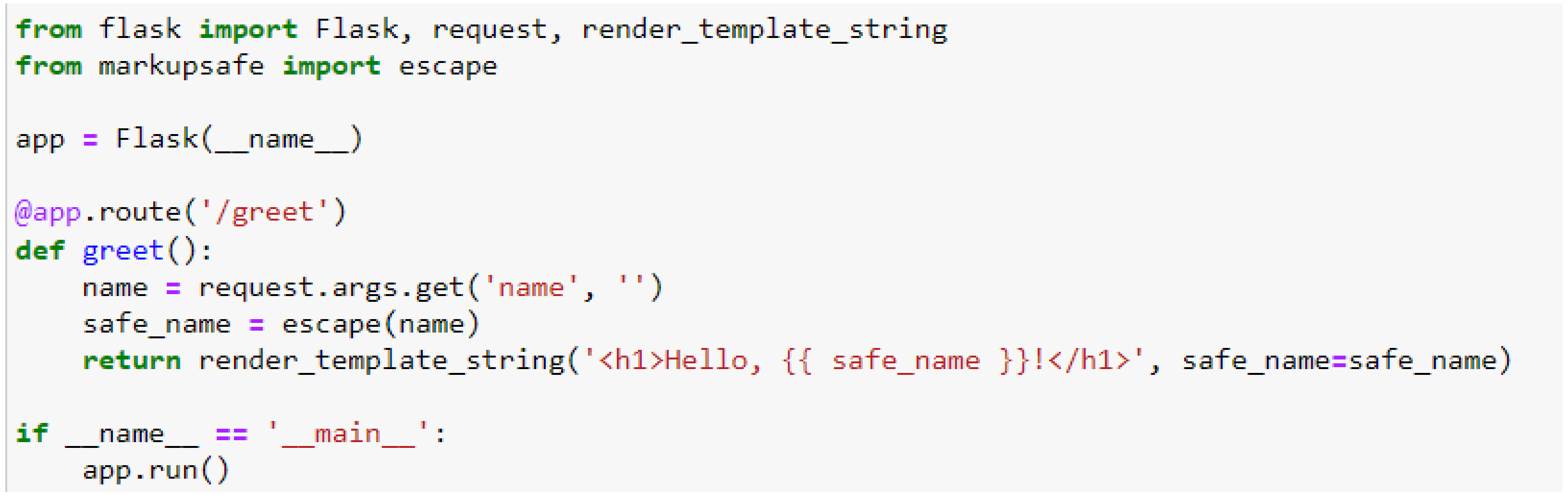


```python
# Example: Using secure configurations in Flask
from flask import Flask
app = Flask(__name__)

# Secure configuration
app.config['SESSION_COOKIE_SECURE'] = True
app.config['SESSION_COOKIE_HTTPONLY'] = True
app.config['REMEMBER_COOKIE_SECURE'] = True

@app.route('/')
def home():
    return "Hello, secure world!"

if __name__ == '__main__':
    app.run()
```

OWASP top 10 - Cross-Site Scripting (XSS)

- **Description**: XSS vulnerabilities occur when untrusted data is included in web pages without proper validation or escaping.
- **Impact**: Attackers can execute scripts in the user's browser, hijack user sessions, or redirect users to malicious sites.
- **How to Secure**:
  - Validate and sanitize input data.
  - Encode output data.

OWASP top 10 - Cross-Site Scripting (XSS) example

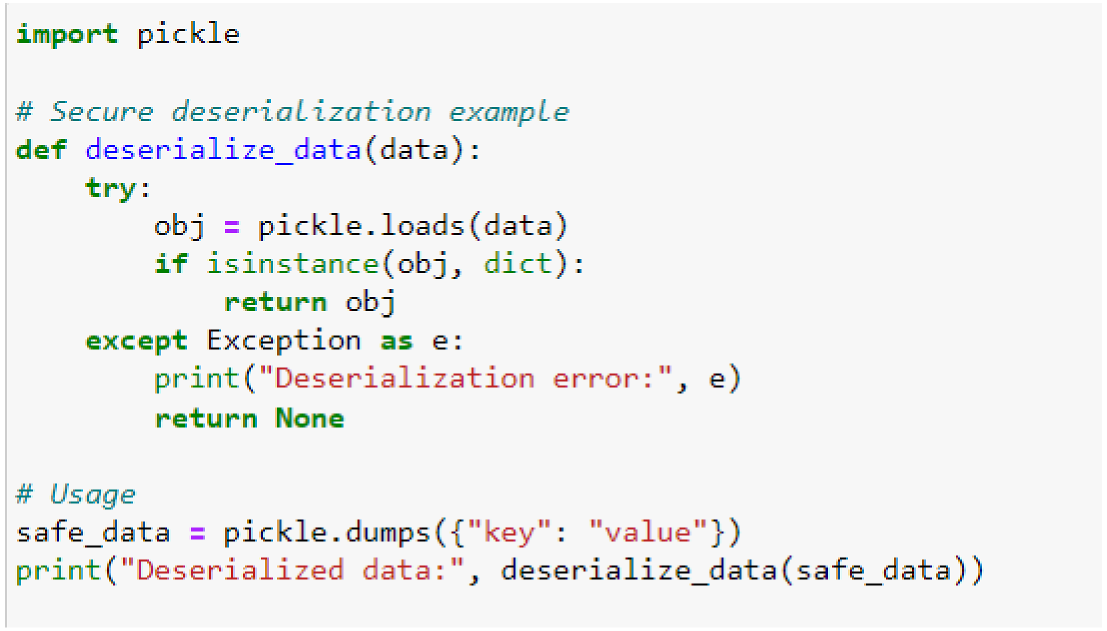


```python
from flask import Flask, request, render_template_string
from markupsafe import escape

app = Flask(__name__)

@app.route('/greet')
def greet():
    name = request.args.get('name', '')
    safe_name = escape(name)
    return render_template_string('<h1>Hello, {{ safe_name }}!</h1>', safe_name=safe_name)

if __name__ == '__main__':
    app.run()
```

OWASP top 10 - Insecure Deserialization

- **Description**: Insecure deserialization vulnerabilities occur when untrusted data is used to abuse the logic of an application, inflict a denial of service (DoS) attack, or execute arbitrary code.
- **Impact**: Attackers can execute arbitrary code or perform unauthorized actions.
- **How to Secure**:
  - Avoid using serialized objects from untrusted sources.
  - Use serialization formats that support integrity checks.

OWASP top 10 - Insecure Deserialization example

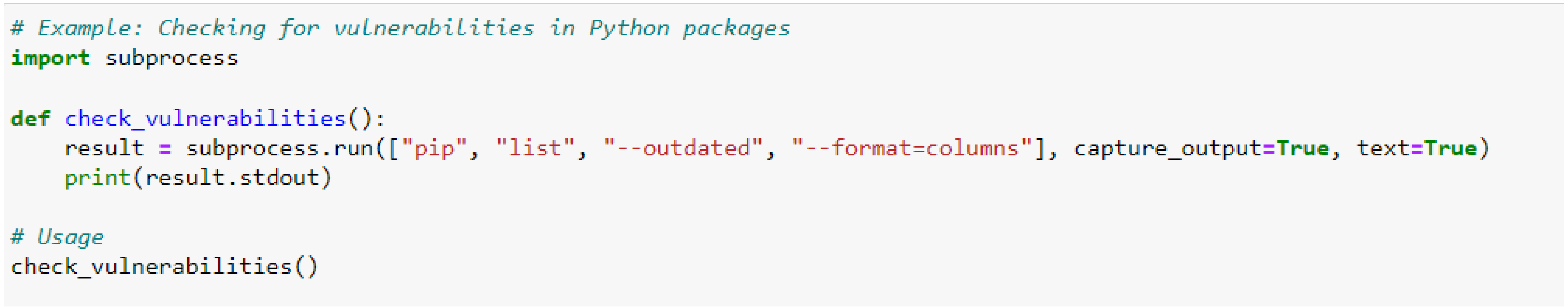


```python
import pickle

# Secure deserialization example
def deserialize_data(data):
    try:
        obj = pickle.loads(data)
        if isinstance(obj, dict):
            return obj
    except Exception as e:
        print("Deserialization error:", e)
        return None

# Usage
safe_data = pickle.dumps({"key": "value"})
print("Deserialized data:", deserialize_data(safe_data))
```

OWASP top 10 - Using Components with Known Vulnerabilities

- **Description**: Using components with known vulnerabilities can expose applications to attacks.
- **Impact**: Attackers can exploit vulnerabilities in third-party components to compromise the application.
- **How to Secure**:
  - Regularly update and patch components.
  - Use tools to identify vulnerable components.

OWASP top 10 - Using Components with Known Vulnerabilities example

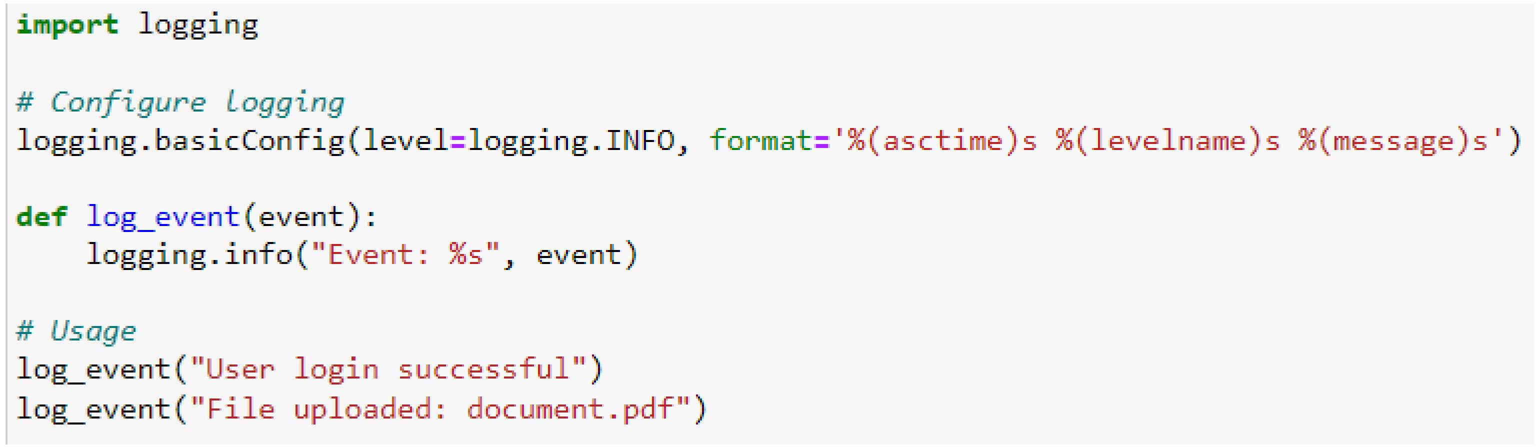


```python
# Example: Checking for vulnerabilities in Python packages
import subprocess

def check_vulnerabilities():
    result = subprocess.run(["pip", "list", "--outdated", "--format=columns"], capture_output=True, text=True)
    print(result.stdout)

# Usage
check_vulnerabilities()
```

OWASP top 10 - Insufficient Logging & Monitoring

- **Description**: Insufficient logging and monitoring can delay the detection of security breaches.
- **Impact**: Attackers can exploit the lack of monitoring to continue their attacks undetected.
- **How to Secure**:
  - Implement comprehensive logging and monitoring.
  - Regularly review and analyze logs.

OWASP top 10 - Insufficient Logging & Monitoring example



```python
import logging

# Configure logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s %(levelname)s %(message)s')

def log_event(event):
    logging.info("Event: %s", event)

# Usage
log_event("User login successful")
log_event("File uploaded: document.pdf")
```

OWASP Projects

- OWASP sponsors and supports various open-source projects related to web application security.
- **Scope**: These projects cover a wide range of security topics, including secure coding, vulnerability scanning, and penetration testing.
- **Objective**: To provide free, high-quality tools and resources that help developers and security professionals improve the security of their web applications.
- **Categories**:
  - Tools and utilities
  - Documentation and guides
  - Educational and training resources

OWASP Projects- Zed Attack Proxy (ZAP)

- ZAP is an open-source tool used for finding vulnerabilities in web applications.
- **Purpose**: Ideal for both beginners and professionals to perform security testing on web applications.
- **Key Features**:
  - Automated scanners
  - Passive and active scanning
  - Manual testing tools
  - Integration with other tools
- Tutorial: https://www.youtube.com/watch?v=3u7aKXXCCKA

OWASP Projects- OWASP Dependency-Check

- A Software Composition Analysis (SCA) tool that identifies vulnerable components in project dependencies.
- **Purpose**: Helps in identifying and mitigating risks associated with third-party libraries.
- **Key Features**:
  - Supports multiple languages and dependency management systems
  - Generates detailed reports
  - Integrates with CI/CD pipelines
  - Tutorials: https://www.youtube.com/watch?v=DF22sTpcE6w https://www.youtube.com/watch?v=X47ZkdYnGZI

OWASP – Web Application Security Testing

- WASP offers guidelines, tools, and resources for testing the security of web applications, helping organizations identify and address vulnerabilities.
- **OWASP Web Security Testing Guide (WSTG):** Although not a tool in the software sense, WSTG is a comprehensive guide for testing the security of web applications and services. It provides a standardized approach to web application security testing.
- **Features**: Includes detailed instructions and best practices for testing various security aspects of web applications, including authentication, authorization, session management, and data validation.

NIST’s Secure Software Development Framework

- The **National Institute of Standards and Technology** (**NIST**) released various guidelines and frameworks for secure software development.
- One of the critical resources NIST provides is the NIST **Secure Software Development Framework** (**SSDF**), designed to help organizations enhance the security of their software development processes.

Secure Software Development Guidelines

- Example: Using the NIST Special Publication 800-53 as a guide for security controls.
- Tool: JIRA - Can be customized to track and manage security requirements and tasks throughout the development lifecycle.

NIST Secure Software Development Principles

- **Description**: The framework promotes fundamental security principles and best practices that should be integrated into every phase of the software development life cycle (SDLC).
  - These principles include secure coding, security testing, threat modeling, and secure software architecture.
- **Purpose**: To embed security considerations into the DNA of the development process, making security a primary concern rather than an afterthought.

NIST’s Security Standards and References

- **Description**: NIST’s SSDF references various security standards, guidelines, and resources that can help organizations implement secure software development practices effectively.
  - This includes NIST Special Publications, industry standards, and other authoritative sources.
- **Purpose**: To provide a robust foundation of trusted and proven security practices that organizations can rely on to enhance their software security posture.

NIST’s Security Standards and References

- **Example**: Adopting the CIS Controls framework for implementing a robust cybersecurity posture.
- **Tool**: **NIST Cybersecurity Framework (CSF) Tool** - A tool that helps organizations align their security practices with NIST standards.

NIST’s Security Risk Management:

- **Description**: The framework emphasizes the importance of risk management throughout the software development process.
  - It encourages organizations to identify, assess, and mitigate security risks associated with their software projects.
- **Purpose**: To proactively manage and reduce security risks, ensuring that potential vulnerabilities are addressed before they can be exploited.

NIST’s Security Risk Management

- **Example**: Conducting regular threat modeling sessions using the STRIDE methodology to identify and mitigate risks.
- **Tool**: **Microsoft Threat Modeling Tool** - Helps create and analyze threat models for applications.

NIST’s Integration with Existing Processes

- **Description**: NIST’s SSDF is designed to be flexible and adaptable.
  - Organizations can integrate its guidance and recommendations into their existing software development processes, including Agile and DevOps methodologies.
- **Purpose**: To facilitate the adoption of secure software development practices without disrupting established workflows, making it easier for organizations to improve their security posture.

NIST’s Integration with Existing Processes

- **Example**: Integrating security checkpoints into an Agile development workflow using Scrum.
- **Tool**: **GitLab CI/CD** - Can be configured to include security tests as part of the continuous integration and delivery pipeline.

NIST’s Security Training and Awareness

- The framework underscores the significance of training and raising awareness among developers, testers, and other stakeholders regarding secure coding practices and the implications of security vulnerabilities.
- **Purpose**: To ensure that all individuals involved in the software development process understand the importance of security and are equipped with the knowledge and skills to implement secure practices.

NIST’s Security Metrics and Measurement

- NIST’s SSDF encourages organizations to define and track security metrics to assess the effectiveness of their secure software development initiatives. Metrics can help identify areas for improvement and evaluate the impact of security measures.
- **Purpose**: To provide a quantitative basis for evaluating and improving the security of software development processes, leading to more effective and measurable security practices.

Secure SDLC

- NIST’s SSDF promotes incorporating security activities into the entire SDLC, from initial planning and requirements to post-deployment maintenance and monitoring.
- **Purpose**: To ensure that security is a continuous consideration throughout the entire lifecycle of software development, resulting in more secure and resilient software applications.

Continuous Improvement

- The framework emphasizes continuous improvement by learning from security incidents, security assessments, and feedback from the development process. It encourages organizations to adapt and refine their practices over time.
- **Purpose**: To foster a culture of ongoing improvement in software security practices, enabling organizations to stay ahead of emerging threats and vulnerabilities.

MITRE Frameworks

- MITRE Corporation is a not-for-profit organization that operates **Federally Funded Research and Development Centers** (**FFRDCs**) in the United States.
- One of MITRE's key contributions to software security is the development and maintenance of the **Common Weakness Enumeration** (**CWE**) and **Common Vulnerabilities and Exposures** (**CVE**) standards

Common Vulnerabilities and Exposures (CVE)

- **CVE**: Focuses on documenting specific instances of vulnerabilities in software products.
  - Each CVE entry describes a particular vulnerability and provides an identifier, description, and references to more detailed information.
- Deals with individual vulnerabilities that have been identified in specific software products. It is a database of known vulnerabilities with unique identifiers (e.g., CVE-2023-12345).
- Contains entries that include the CVE ID, a brief description of the vulnerability, and references to additional details such as advisories, patches, and reports.
- Used by security professionals to identify and track specific vulnerabilities in software products. It helps in vulnerability management, patch management, and incident response.

CVE example

- **CVE-2023-12345**: This might describe a specific buffer overflow vulnerability found in version 2.3.4 of a software product. It would include a brief description, an identifier, and links to detailed reports and patches.

Common Weakness Enumeration (CWE)

- Focuses on the classification and categorization of types of software weaknesses that can lead to vulnerabilities.
  - CWE provides a hierarchical list of software weaknesses, detailing the nature of the weakness and its potential impact.
- Deals with the broader classes of software weaknesses and coding errors that could potentially lead to vulnerabilities. It provides a taxonomy of these weaknesses to help developers understand and prevent them (e.g., CWE-79 for Cross-Site Scripting).
- Contains entries that describe the nature of weaknesses, examples of how they can be introduced, potential consequences, and mitigation strategies.
- Used by developers, security researchers, and educators to understand common types of software weaknesses. It is used in secure coding practices, software assurance programs, and training.

CWE exmaple

- **CWE-120**: This describes the general class of "Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')."
- It explains what a buffer overflow is, how it can be introduced into software, and ways to prevent it.

MITRE ATT&CK Framework

- The MITRE Adversarial Tactics, Techniques, and Common Knowledge (ATT&CK) framework is a knowledge base that describes the actions and behaviors of cyber adversaries. It provides a comprehensive view of adversaries' various tactics and techniques to achieve their objectives.
  - The ATT&CK framework catalogs the tactics and techniques used by adversaries to infiltrate networks, maintain persistence, escalate privileges, and achieve their objectives.
  - It includes detailed descriptions of each tactic and technique, as well as the specific methods adversaries use to carry them out.
  - The framework is continually updated with information gathered from real-world observations and cybersecurity research.

MITRE ATT&CK Framework use case examples

- **Threat Hunting**: Security teams use the ATT&CK framework to proactively search for indicators of compromise (IOCs) and adversary behaviors within their networks.
- **Incident Response**: Incident responders leverage the framework to understand the tactics and techniques used in an ongoing attack and to develop effective mitigation strategies.
- **Security Posture Assessment**: Organizations assess their defenses against the tactics and techniques described in the ATT&CK framework to identify gaps and weaknesses.
- **Red Teaming and Penetration Testing**: Red teams use the ATT&CK framework to simulate adversary behaviors and test the effectiveness of security controls.
- **Security Operations Center (SOC) Automation**: The framework is integrated into SOC tools and platforms to automate the detection and response to known adversary techniques.

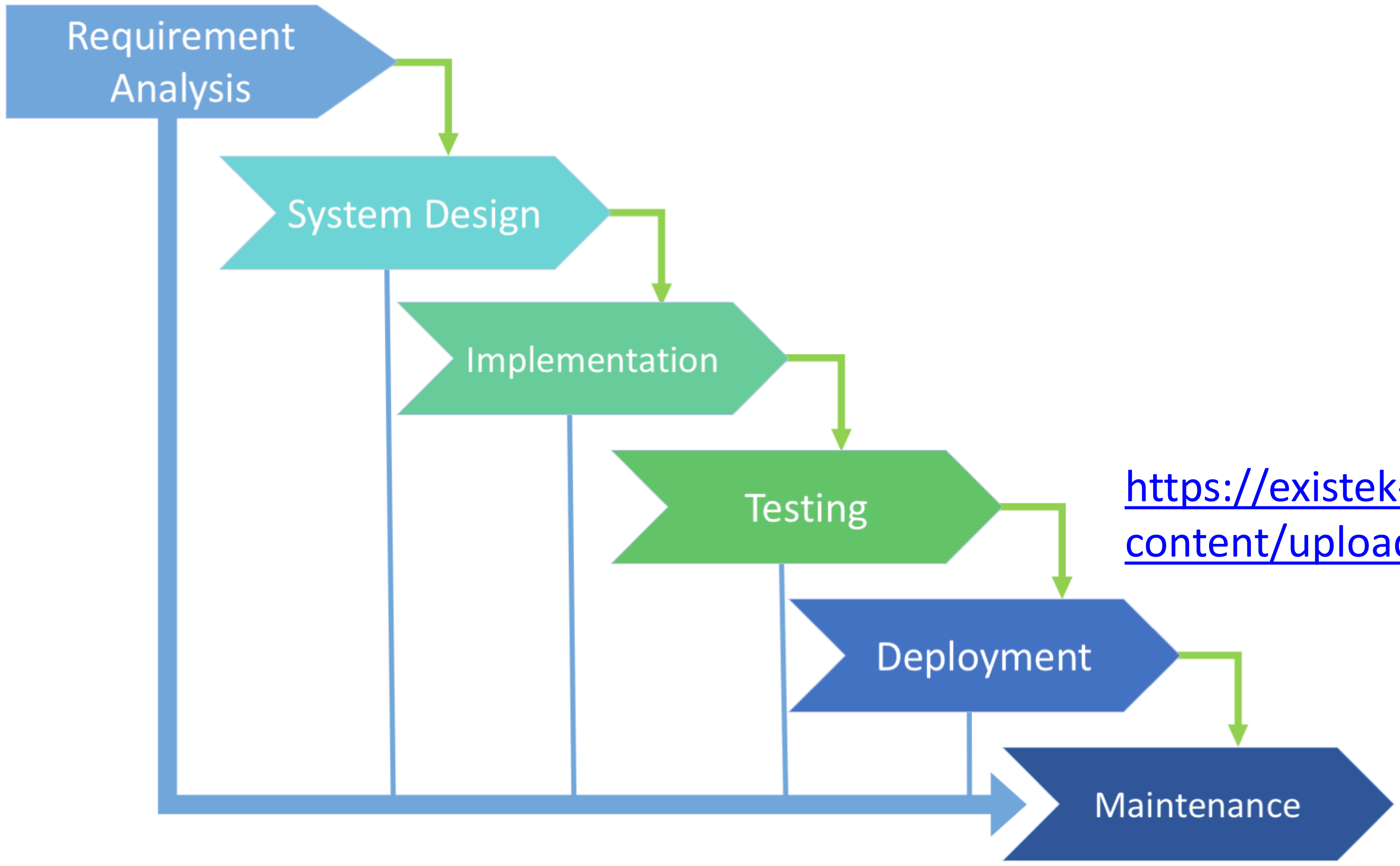
MITRE ATT&CK Framework: key components

- **Tactics**: The "why" of an adversary's behavior, representing the goal they are trying to achieve. Examples include Initial Access, Execution, Persistence, Privilege Escalation, Defense Evasion, Credential Access, Discovery, Lateral Movement, Collection, Command and Control, Exfiltration, and Impact.
- **Techniques**: The "how" of an adversary's behavior, detailing the specific methods used to achieve a tactic. Each technique includes information on the procedure, detection, mitigation, and examples of real-world usage.
- **Sub-Techniques**: Further granularity within techniques, providing more specific details about how an adversary performs a particular technique.
- **Procedures**: Real-world instances of how adversaries have implemented tactics and techniques, including examples from cyber threat intelligence reports.

Most common SDLC models

- Although SDLC is a standardized method to develop Software, the implementation however differs depending on the needs of the organization.
- **Waterfall Model**
- **Agile Model**
- **Scrum**
- **Kanban**
- **Iterative Model**
- **Spiral Model**
- **DevOps**
- **V-Model (Validation and Verification Model)**
- **Rapid Application Development (RAD)**
- **Big Bang Model**

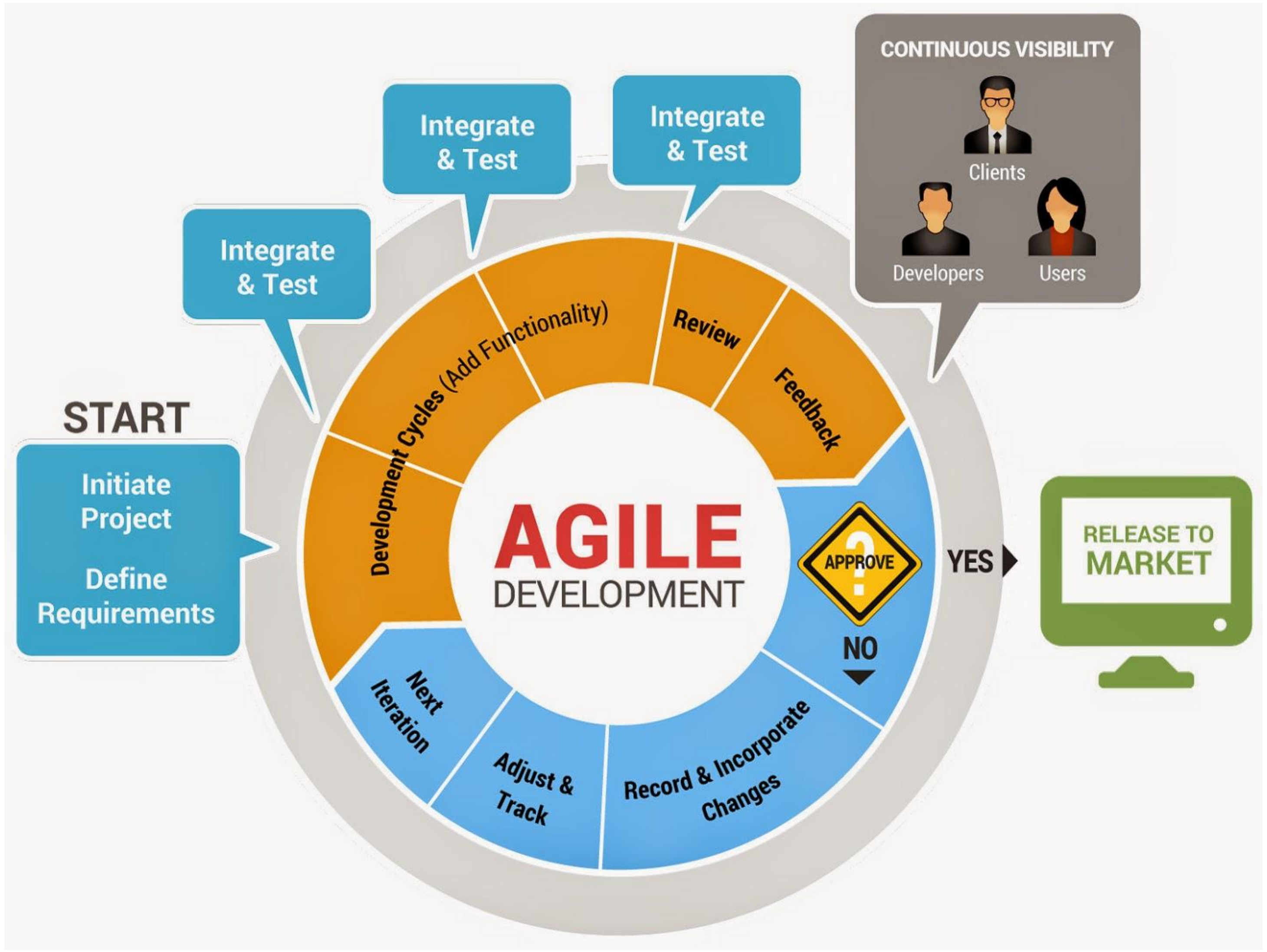
Waterfall SDLC



https://existek-838c.kxcdn.com/wp-content/uploads/2017/08/Itterative-SFDC-Model.png

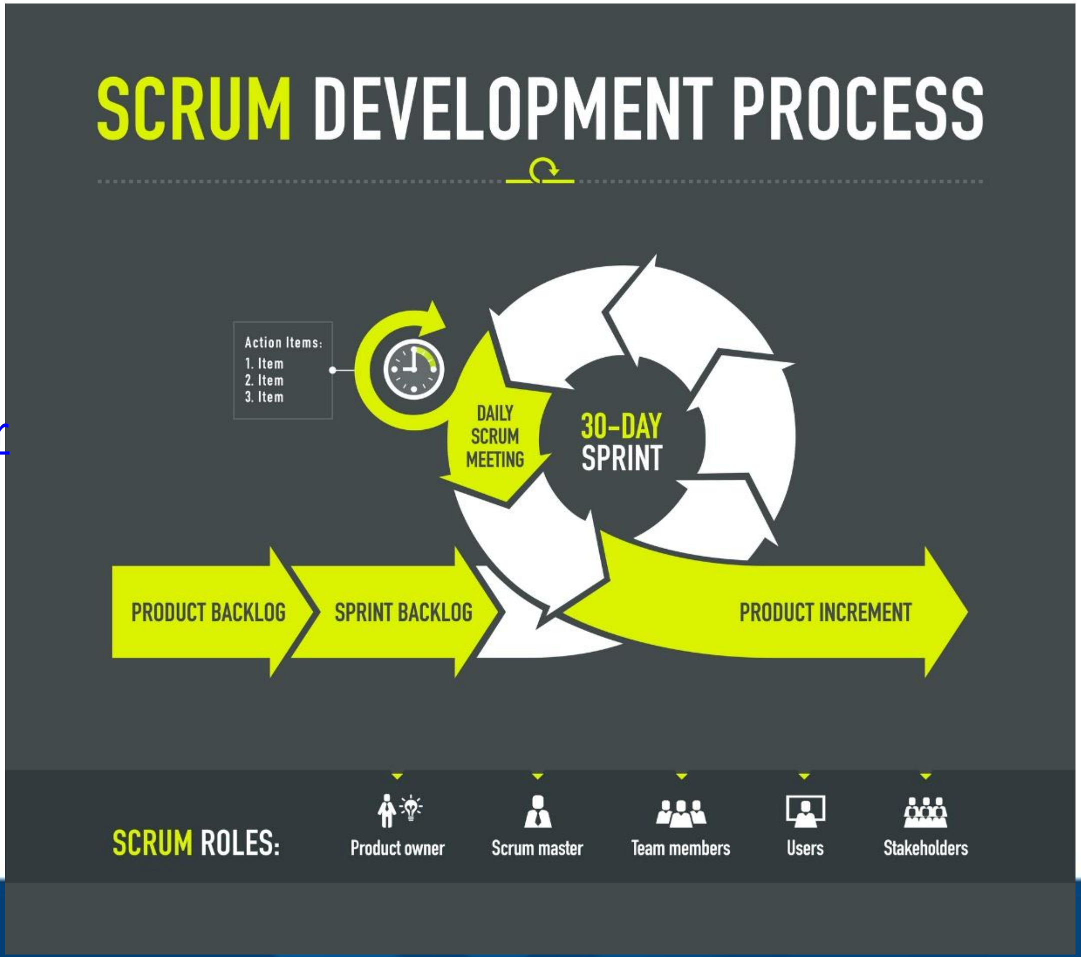
Agile SDLC

- https://4.bp.blogspot.com/-YqfVxrkO-Jk/VBiLJG7kY3I/AAAAAAAABv8/3L_341MB0Z0/s1600/Agile-sdlc.jpg

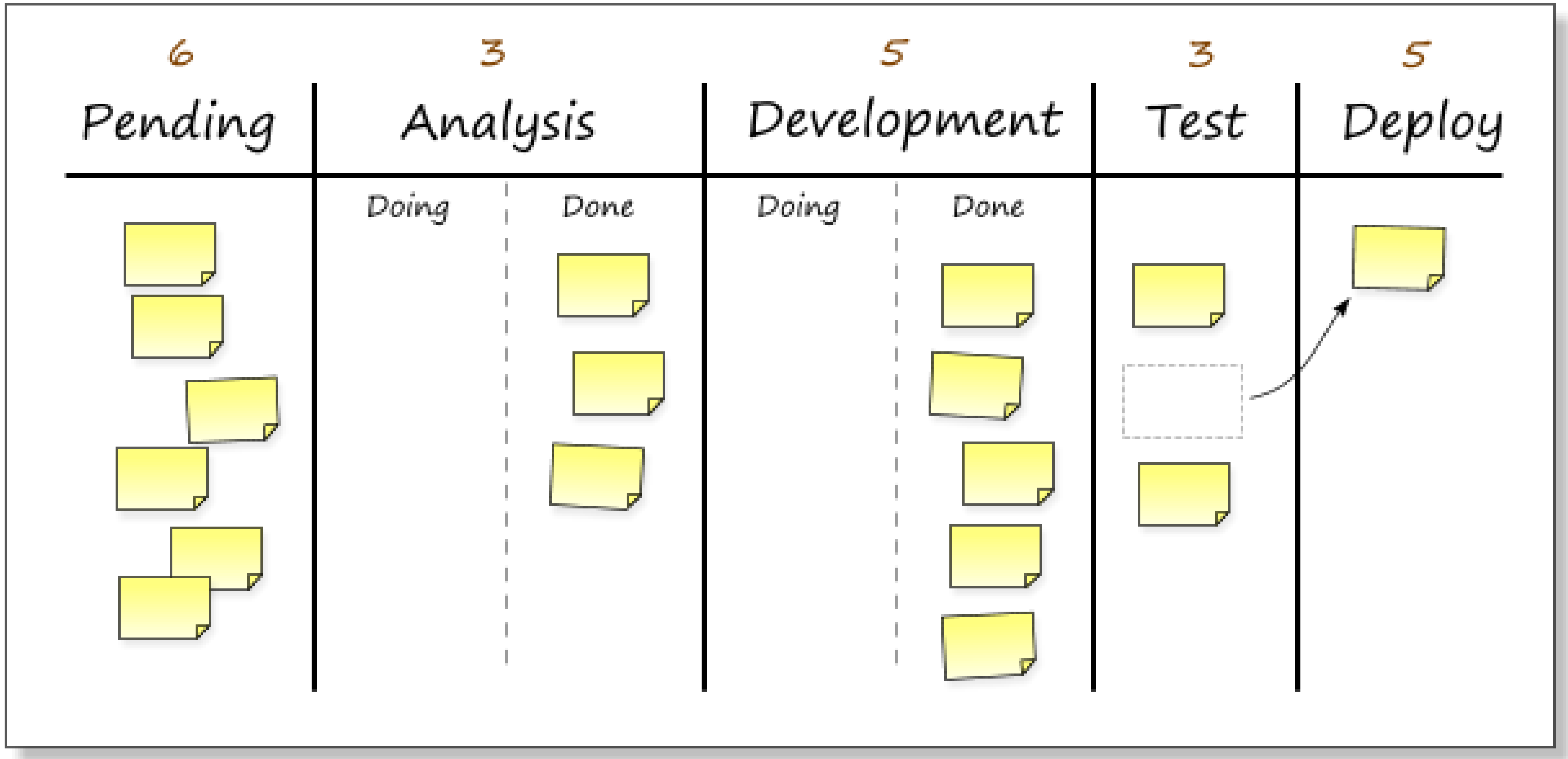


SCRUM SDLC

http://projektwelten.projectplant.de/wp-content/uploads/2016/07/ProjectPlant_ScrumMethode_Ueberblick.jpg



KANBAN SDLC



https://kanbanblog.com/explained/image/kanban-board-2.png

ITTERATIVE SDLC

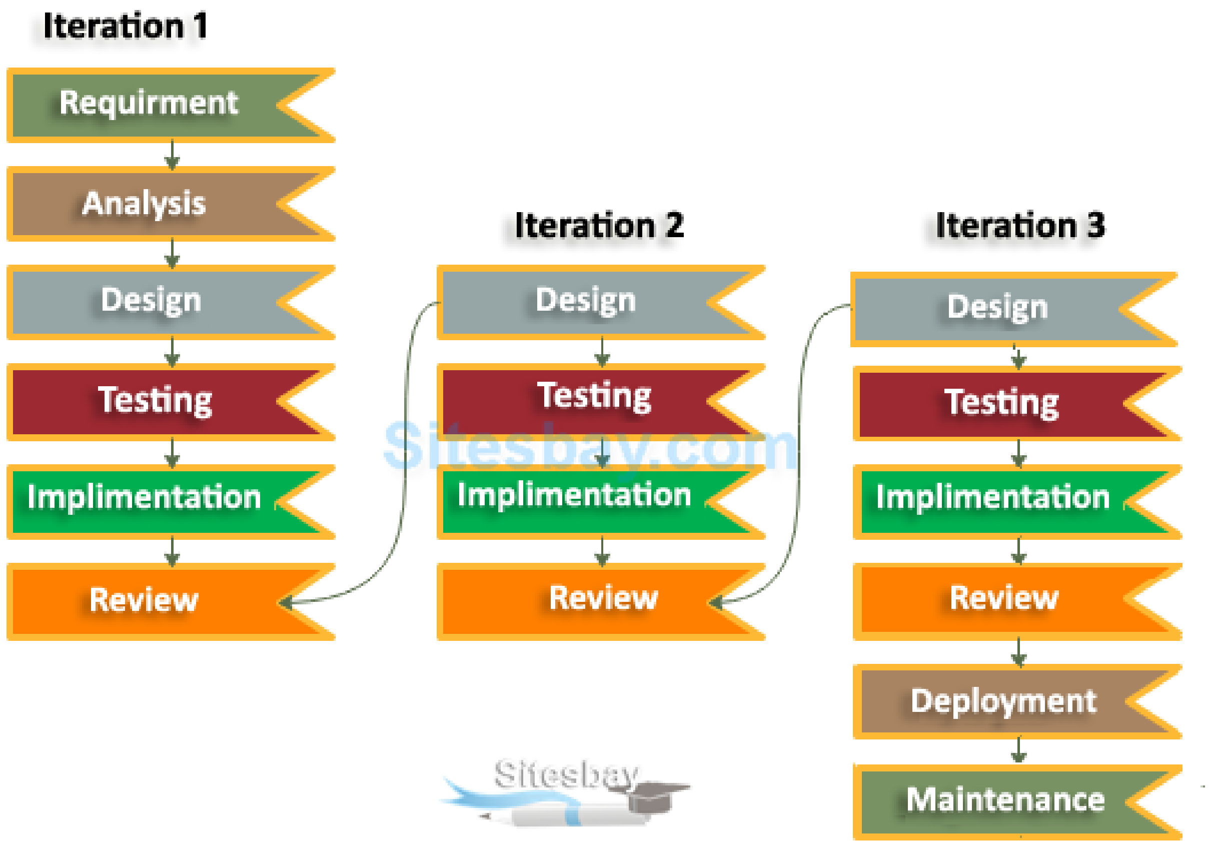


Fig: Iterative Model

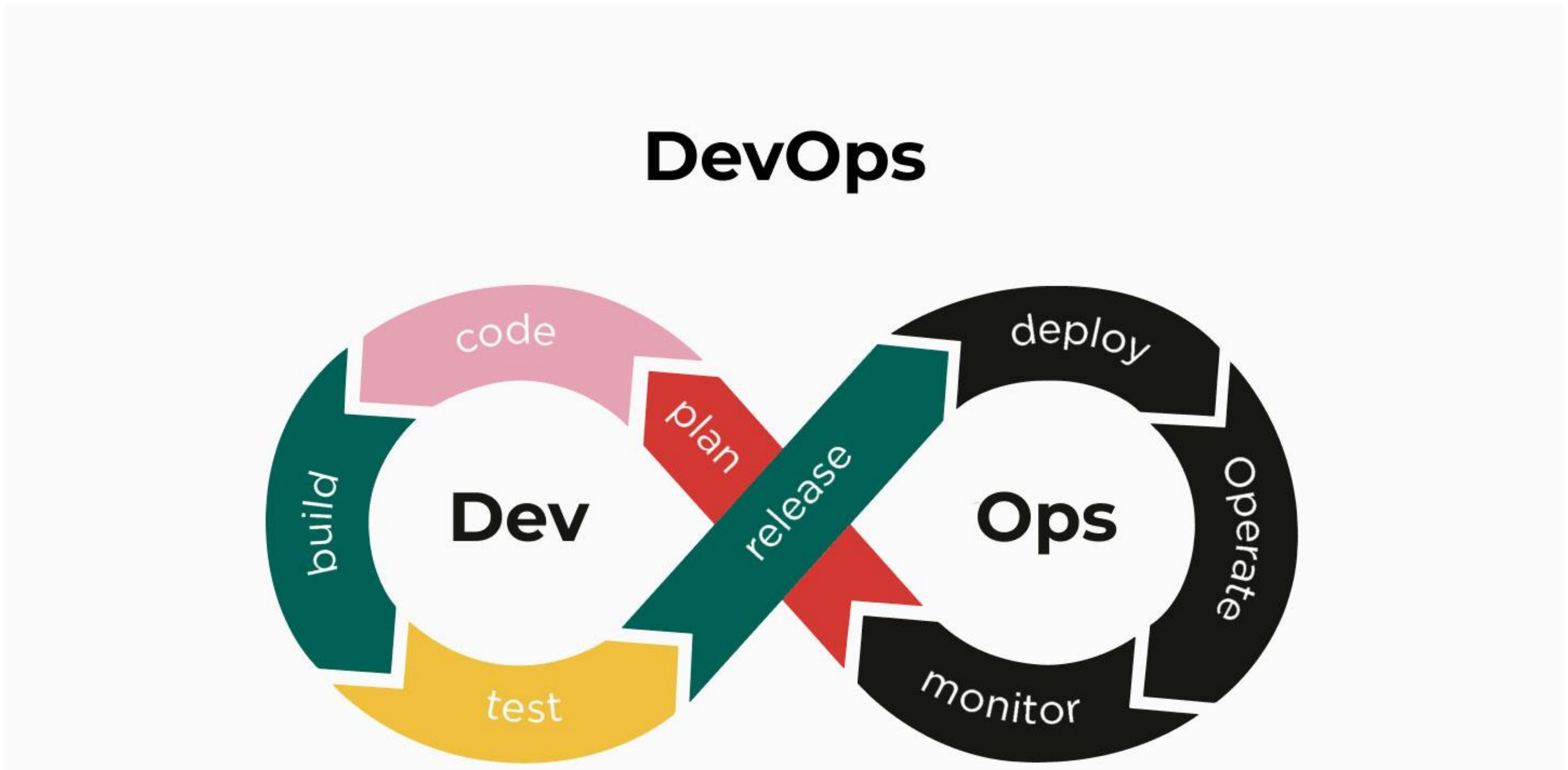
https://external-content.duckduckgo.com/iu/?u=https%3A%2F%2Fwww.sitesbay.com%2Fsoftware-engineering%2Fimages%2Fiterative-model-in-sdlc.png&f=1&nofb=1&ipt=66e1e14e4987c5ec8ff767ccb13b050adf569f58580bb6f62af4b03ac21e1709&ipo=images

SPIRAL SDLC

Insert your desired text here.
Insert your desired text here.
Insert your desired text here.
Placeholder
Insert your desired text here.
Insert your desired text here.
Insert your desired text here.
Placeholder
Insert your desired text here.
Insert your desired text here.
Insert your desired text here.
Placeholder
Insert your desired text here.
Insert your desired text here.
Insert your desired text here.
Placeholder

https://external-content.duckduckgo.com/iu/?u=https%3A%2F%2Fcdn.slidemodel.com%2Fwp-content%2Fuploads%2F20539-01-spiral-diagram-concept-for-powerpoint-16x9-9.jpg&f=1&nofb=1&ipt=02c7185255f9c825972775f0f22fcfa6877f8271ce44838e61dfb033d9c73864&ipo=images

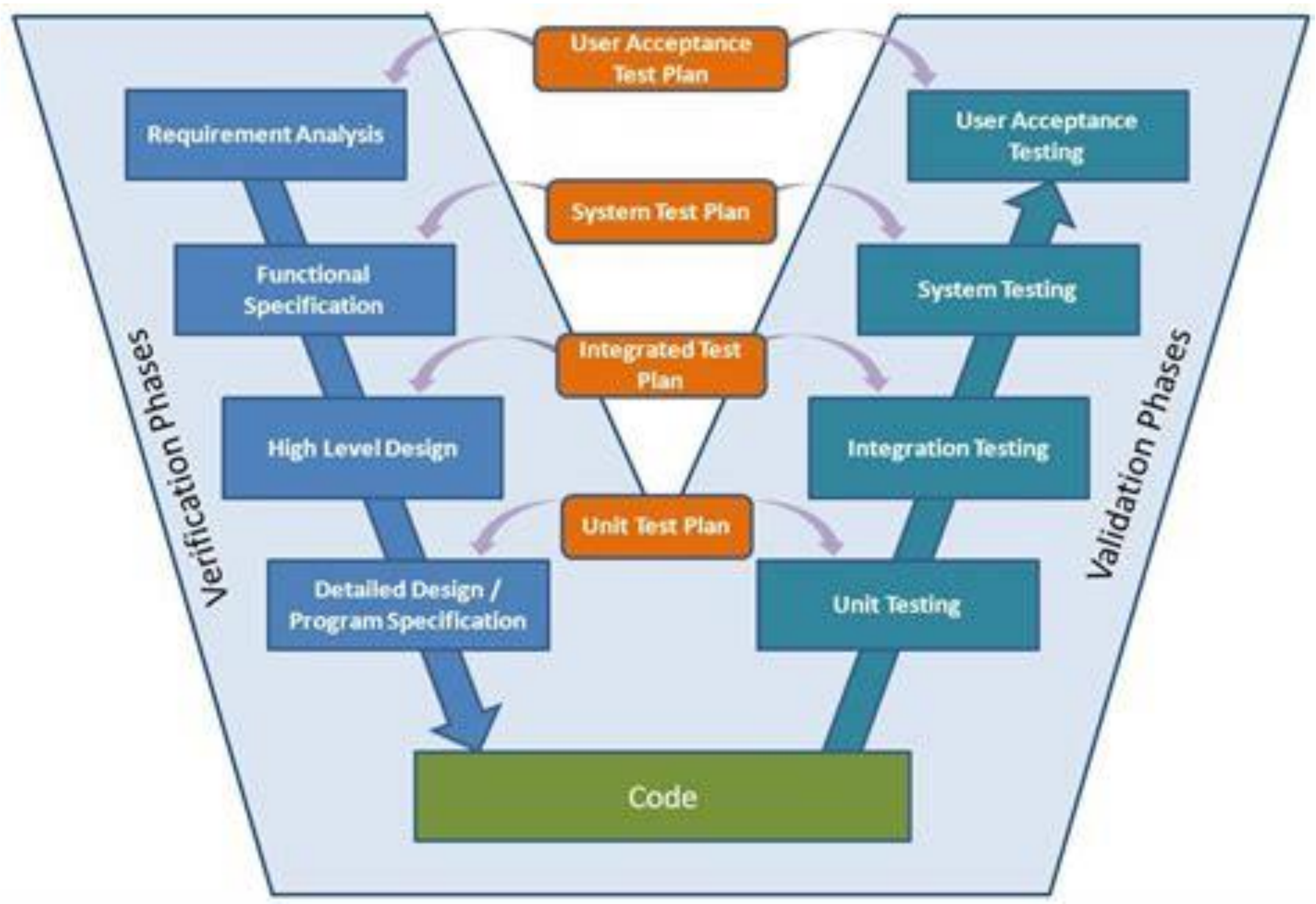
DevOPS SDLC



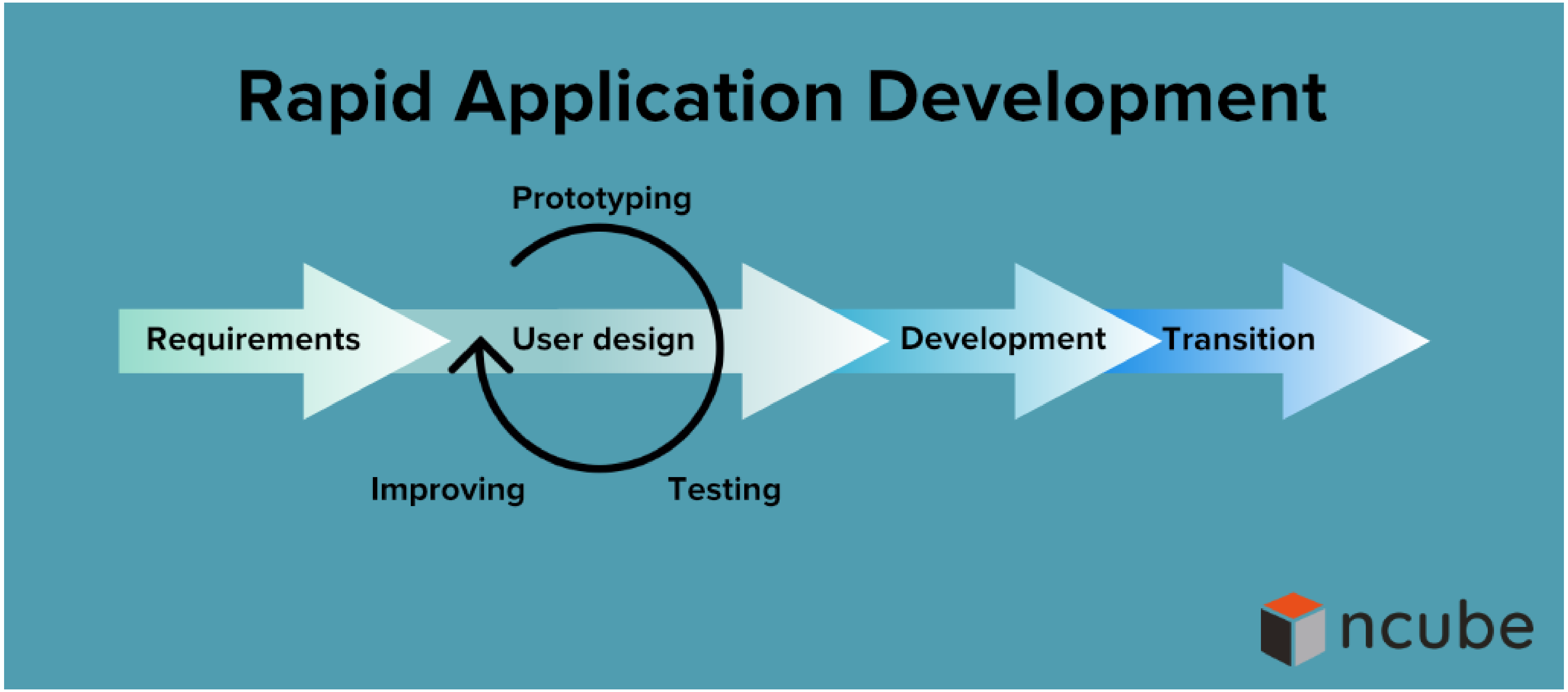
https://external-content.duckduckgo.com/iu/?u=https%3A%2F%2Fthemindstudios.com%2Fblog%2Fcontent%2Fimages%2F2020%2F01%2FDevOps.jpg&f=1&nofb=1&ipt=5ca681d573ac2d4b1a12dfb9c08271a1a77c17f39458e0cb45434c9df9b52e09&ipo=images

V-Model (Validation and Verification Model)

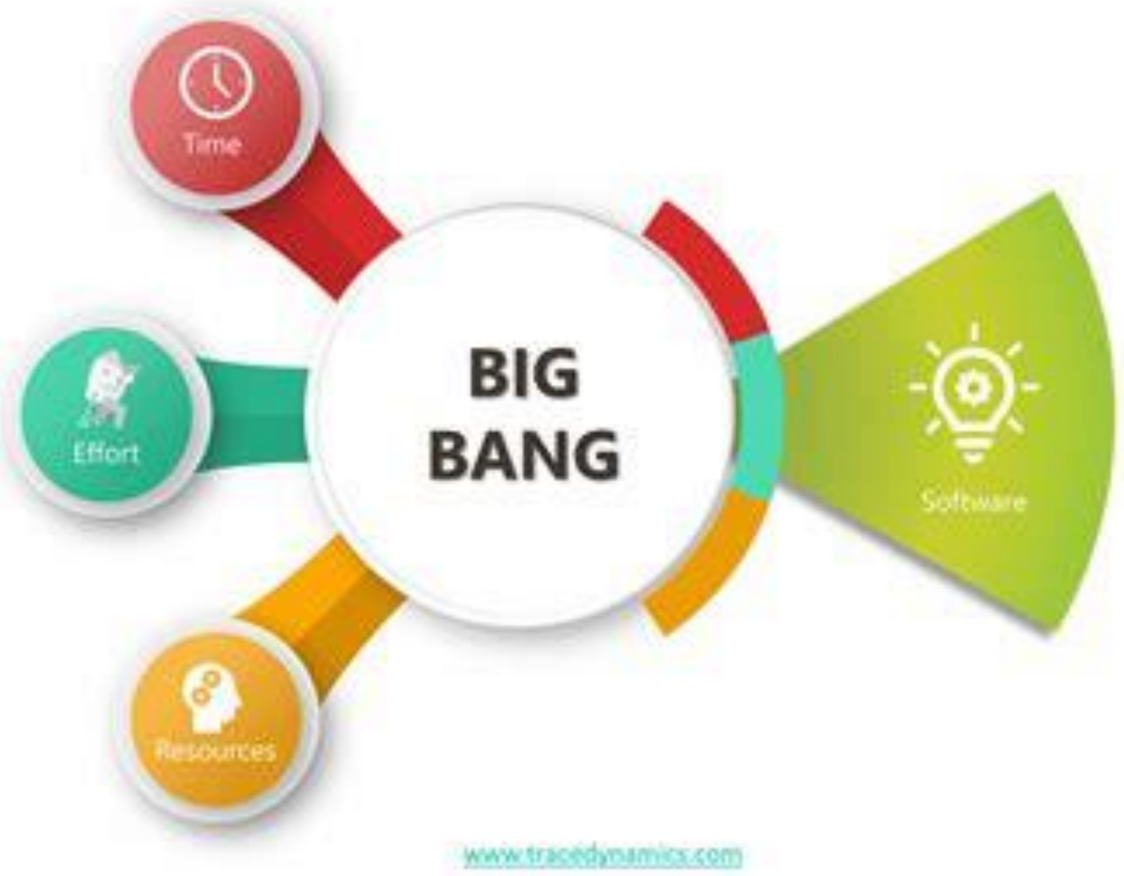
- https://external-content.duckduckgo.com/iu/?u=https%3A%2F%2Ftse2.mm.bing.net%2Fth%3Fid%3DOIP.9hfM8Zz0blYY6viv2AsXHgHaFF%26pid%3DApi&f=1&ipt=a106f6df97b7d0b89c06de8e195cab0ff029cee62175a6ead02faab9a49d3aa7&ipo=images



RAD SDLC


Rapid Application Development
Prototyping
Requirements
User design
Development
Transition
Improving
Testing
ncube

BIG BANG SDLC



SDLC models – Comparison

Aspect	Waterfall Model	Agile Model	Scrum	Kanban	Iterative Model
Flexibility	Low - sequential process, difficult to change	High - iterative and adaptive	High - iterative with regular adjustments	High - continuous flow, dynamic adjustments	Medium - iterative cycles allow adjustments
Customer Involvement	Low - limited to initial and final phases	High - continuous collaboration and feedback	High - regular involvement in sprints	High - ongoing feedback through board updates	Medium - feedback at the end of each iteration
Risk Management	Low - risks addressed late in process	High - continuous risk assessment and mitigation	High - risks managed within sprints	Medium - risks managed as work progresses	Medium - risks addressed per iteration
Delivery Speed	Slow - follows a strict sequence	Fast - delivers usable increments regularly	Fast - delivers usable increments per sprint	Continuous - work is delivered as completed	Medium - delivers usable parts per iteration
Best Suited For	Stable, well-defined requirements	Evolving or unclear requirements	Projects needing regular delivery of features	Continuous delivery and dynamic management	Projects needing ongoing refinement and feedback

SDLC models – Comparison

Aspect	Spiral Model	DevOps	V-Model	RAD	Big Bang Model
Flexibility	High - iterative with risk-based approach	High - integrates continuous delivery practices	Low - follows strict verification phases	High - focuses on rapid prototyping	Very High - minimal planning
Customer Involvement	High - involved in each cycle and risk assessment	High - continuous collaboration and feedback	Low - limited to initial and final phases	High - ongoing feedback in rapid cycles	Low - limited involvement
Risk Management	Very High - central to the model	High - continuous monitoring and mitigation	Medium - testing linked to each phase	Medium - iterative feedback reduces risks	Very Low - risks often unaddressed
Delivery Speed	Medium - iterative but thorough	Very Fast - continuous integration and delivery	Slow - follows a strict sequence	Very Fast - rapid cycles and feedback	Unpredictable - no structured process
Best Suited For	Large, complex projects with high risks	Projects needing rapid, reliable deployments	Stable, well-understood requirements	Fast time-to-market projects	Small, experimental projects

Microsoft’s Security Development Lifecycle (SDL)

- A major issues with the “old” traditional SDLC was the absence, or the lack of attention, of security aspect.
- Security was not the major goal when designing software
  - Developers wants to get things done ASAP.

Traditional SDLC vs Microsoft's SDL

Aspect	Traditional SDLC	Microsoft SDL
Focus	Functionality and performance	Security throughout all phases
Security Integration	Security often addressed in testing phase	Security integrated into every phase
Risk Management	Managed within phases, sometimes reactive	Proactive, continuous risk management
Training	Not typically included	Mandatory security training for all personnel
Verification	Functional and performance testing	Comprehensive security testing (static, dynamic, fuzz)
Threat Modeling	Rarely included	Integral part of the design phase
Post-Release	Maintenance and bug fixes	Incident response and continuous monitoring
Tools and Practices	General development tools	Use of approved security tools and best practices

SDLC vs. MS SDL: Deeper comparison

Aspect	**Traditional SDLC**	**Microsoft SDL**
Primary Focus	Functionality, performance, meeting user requirements	Security integrated throughout the development process
Phases	Requirements, Design, Implementation, Testing, Deployment, Maintenance	Training, Requirements, Design, Implementation, Verification, Release, Response
Security Integration	Primarily during testing phase	Integrated into every phase
Risk Management	Managed within specific phases, often reactive	Proactive and continuous risk management
Training	Not typically included	Mandatory security training for all personnel
Requirements Phase	Focus on functional requirements	Security requirements defined alongside functional requirements
Design Phase	Emphasis on system and software architecture	Includes threat modeling and secure design principles
Implementation Phase	Coding and development of the software	Enforces secure coding practices and use of approved tools
Verification Phase	Functional and performance testing	Comprehensive security testing (static analysis, dynamic analysis, fuzz testing)

Aspect	Traditional SDLC	Microsoft's SDL
Release Phase	Deployment and transition to maintenance	Final security review and incident response planning
Response Phase	Maintenance and bug fixes	Continuous monitoring and response to security incidents
Threat Modeling	Rarely included	Integral part of the design phase
Security Testing	Basic functional and performance testing	Extensive security testing (static, dynamic, fuzz)
Incident Response	Limited, reactive maintenance	Proactive incident response and continuous monitoring
Use of Tools	General development tools	Use of approved security tools and best practices
Continuous Improvement	Focus on functional updates	Emphasizes learning from incidents and improving security practices
Flexibility	Sequential, changes difficult after phase completion	Iterative, allows for changes and continuous improvement
Documentation	Comprehensive documentation at each phase	Comprehensive documentation with security focus
Customer Involvement	Mainly during requirements and final delivery	Continuous collaboration and feedback
Development Speed	Slower, due to sequential phases	Can be faster, especially in identifying and mitigating security issues early
Best Suited For	Projects with stable, well-defined requirements	Projects with high security needs, complex environments