

Course Book



**SECURE SOFTWARE
DEVELOPMENT**

DLMCSEEDS001_E

iu

INTERNATIONAL
UNIVERSITY OF
APPLIED SCIENCES

LEARNING OBJECTIVES

Today there is almost no field that can exist without information technology (IT). Without modern technology, we cannot even conceive of the internet, business, entertainment, and our current modes of social interaction. Software makes all of this possible. Our social environments are changing under the pressure of emerging technologies, which, on the one hand, bring benefits and comfort to our lives; on the other hand, the use of novel technologies exposes us to new risks and challenges.

This accelerated usage of technologies raises a couple of key questions: “How can we operate in a world where our online security is constantly under attack?” and “How can we meet users’ expectation for quality and reliability?” The **Secure Software Development** course book will explore the root causes of software vulnerabilities and how they can be addressed at the software development stage. Among the main software security facets discussed within this course book are software vulnerabilities and their impact on computer security; the problems of identifying the different types of software vulnerabilities, such as software design errors, defects, and incorrect configurations; and understanding the importance of preventative measures to avoid damage caused by software vulnerabilities.

In this course book, you will understand how to manage security during the software development process and learn the basics of DevSecOps. You will also discover how and why it is important to recognize security and privacy early in the development process. Finally, this course book covers the basics regarding software bugs, including how to find them through testing and how to prevent them in the overall software supply chain.

UNIT 1

SECURITY BY DESIGN

STUDY GOALS

On completion of this unit, you will be able to ...

- understand the “shifting left” approach.
- understand the basics of infrastructure as code.
- identify and manage security issues during software development.

1. SECURITY BY DESIGN

Introduction

Recent threat analysis reports show a significant increase in cybersecurity threats, for example, a growing number of online attacks and malicious activities (Radware, n.d.). These reports indicate that most security incidents have a connection to one or more software vulnerabilities: security-related bugs that can be used by intruders (World Economic Forum, 2023). “Security by design” is a complex software development approach that considers security at every level of the software life cycle and builds it into the system, starting with a solid architectural design.

Security architecture design solutions are based on well-known security strategies, tactics, and patterns defined as reusable methodologies. The “shifting left” approach is popular within security-by-design practices to improve the quality and reliability of the software by finding and fixing defects before they become costly and time-consuming to fix later. Shifting left allows one to consider activities for testing or security aspects that are “further left” on the time axis (i.e., in an earlier phase of the software development). The approach is referred to as “infrastructure as code” (IaC) and can help to decrease the problems with misconfigurations in the infrastructure on which the software runs. With **infrastructure as code**, the definition of the environment becomes part of the source code so that the software source code can be tested along with the infrastructure (Dotson, 2019, p. 79).

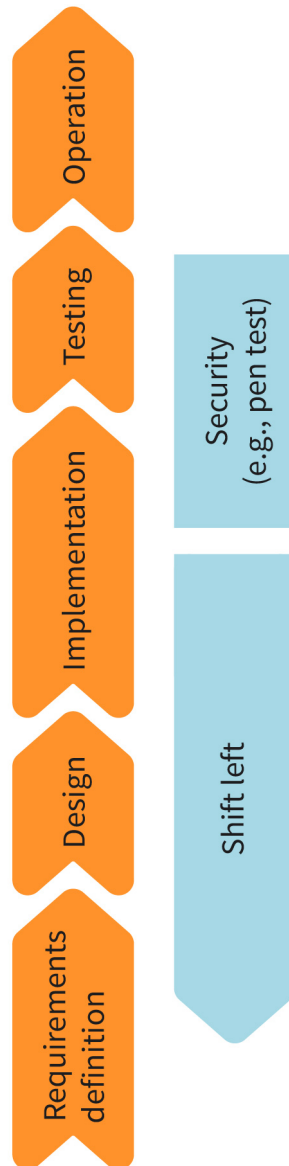
To mitigate the risks, it is necessary to exclude the possibility of vulnerabilities occurring at the very beginning of the process via software development. Considering security early is **extremely** important for software projects because it is **much** more difficult and time-consuming to add security into the system later. Patching vulnerabilities later can be a difficult and expensive process that will never be as effective as designing systems that are secure from the beginning.

1.1 IT Support and Testing Using the “Shifting Left” Methodology

“Shifting left” is a term that describes the practice of moving software testing or security activities to earlier stages of the software development life cycle (SDLC). The SDLC describes activities or steps of a software process, as shown in Figure 1 (Sommerville, 2016, p. 45). The idea is to improve the quality and reliability of the software by finding and fixing defects as early as possible before they become more costly and time-consuming to fix later. In a worst-case scenario, security is not even considered during development and vulnerabilities are discovered through successful attacks. If security is considered, **but only later during a software project**, there may be a penetration test just before the planned launch; the penetration test may then find several vulnerabilities, meaning

that the planned launch of the software is delayed (Sawano et al., 2019). Figure 1 indicates that security should be shifted left in the software development process. Security should already be involved from the “requirements definition” phase shown below.

Figure 1: The Shifting Left Approach



Source: Petra Beenken (2024), based on Sommerville (2016, p. 47).

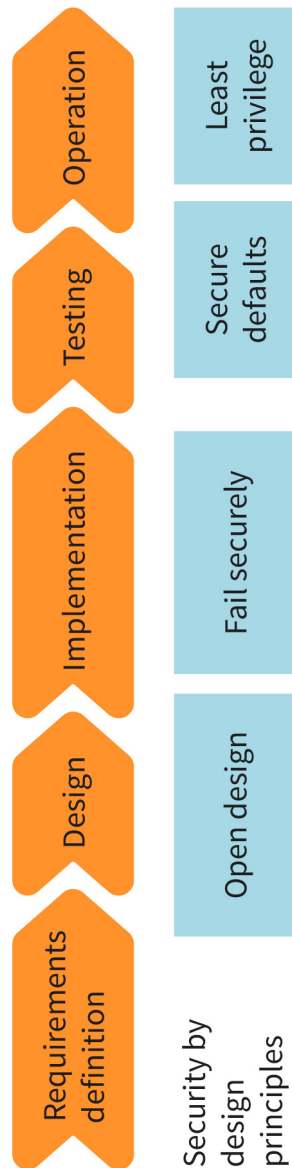
Integrating software testing into early stages of the development process, from design to deployment, can help identify and fix issues early before they become more difficult and costly to fix (Microsoft, 2022).

In April 2023, the Cybersecurity and Infrastructure Security Agency (CISA) published a guide entitled “Shifting the balance of cybersecurity risk: Security-by-design and default principles,” in which they defined the following:

Products that are secure-by-design are those where the security of the customers is a core business goal, not just a technical feature. Secure-by-design products start with that goal before development starts. Secure-by-default products are those that are secure to use ‘out of the box’ with little to no configuration changes necessary and security features available without additional cost. (CISA, 2023)

There are three fundamental security-by-design principles that are applied to achieve the basic goals of security: integrity, confidentiality, and availability. Some common principles are also shown in Figure 2 and described in more detail below (Stallings & Brown, 2018, p. 40).

Figure 2: Security-by-Design Principles



Source: Petra Beenken (2024).

Least Privilege

“Least privilege” means that users should receive only the minimum privileges required to do their job, limited to essential access to resources. For example, a user with guest privileges shouldn’t have the possibility to edit the website content (Stallings & Brown, 2018, p. 41).

Open Design

“Open design” means that the security mechanism (e.g., an encryption algorithm) should be public. It is the opposite of the “security by obscurity” approach. The National Institute for Standards and Technology (NIST) provides programs for standardization and demonstrates how the open design should work. Everyone can approve the security of the code and a standard can be published (Stallings & Brown, 2018, p. 41).

Secure Defaults

Establishing “secure defaults” means that software should be secure by default and the security features should be set to a high level by default. For example, requirements for passwords (e.g., how long and complicated they should be, how often they should be changed).

Fail Securely

The “fail securely” design principle means that software should stop working (fail) in a secure way: It shouldn’t give the user or intruder the possibility of resource abuse. An example of a fail-secure approach is to encrypt sensitive data stored on a client system so that, if the systems crash or fail, none of the data can be exposed. Even if a system failure on the client happens, there would at least be no sensitive data in plain text exposed (Sommerville, 2016, p. 397).

1.2 Infrastructure as Code

Infrastructure as code (IaC) has become very popular due to the DevOps approach, which has adopted it as the basic method of an effective and modern approach to infrastructure management. It is a method that increases the speed of implementation while reducing costs and the risk of errors.

The infrastructure in the IaC model is considered in the same way as developed software. Its entire configuration is kept in the code repository (e.g., Git, Bitbucket) and is automatically tested for security and stability using continuous integration/continuous deployment (CI/CD) tools (e.g., Jenkins, Gitlab CI/CD). What is especially important is that it can be reused and declaratively configured. Following the IaC approach, we can manage the information technology (IT) infrastructure (e.g., networks, virtual machines, load balancers, or virtual cloud connection topology) using configuration files with the same tools that the DevOps team uses to manage application source code. The model described in the IaC template generates the same environment every time it is run.

In the standard model of the infrastructure provisioning process and its subsequent configuration, there is a combination of manual work and automation through scripts. They can be placed in a code repository for the other IT teams members, but they are not usually reviewed or tested. Such scripts are usually written once and not updated, which can become an issue. In this model, we experience an unstable process based on manual

actions, which results in a new environment that is not repeatable. We are also unable to check whether it is stable and safe. An alternative is to use the benefits of IaC, as defined in the core practices for IaC by Morris (2020, p. 26):

- to automate these processes, which is the basic principle of CI/CD and DevOps, and to continuously test and deliver
- to configure the entire infrastructure at once and define everything as code
- to restore configuration in case of errors and build small pieces that can be independently changed


We use IaC to the same standards as in the code development process, which allows us to discover potentially dangerous errors at an early stage.

IaC helps by considering security aspects because of its ability to change configurations quickly. Software-defined networks (SDN) automate the configuration of software, as well as being controlled by software, so are easier to manage. There are different languages for coding infrastructure components like software, some of which are declarative and domain-specific languages like Terraform. “Declarative” in this context means that the definition of the needed infrastructure is separated from implementation aspects. Some infrastructure tools also support common software programming languages, including Python and Java (Morris, 2020, p. 46).

Figure 3: Example of Creating Infrastructure With Code

```
Example procedural code:
import ,cloud-api-library'
network_segment = CloudApi.find_network('private')
app_server = CloudApi.find_server('test_application_server')
app_server = Cloud.Api.create_server(
  name: 'test_application_server',
  image: 'base_linux',
  cpu: 2,
  ram: '2GB',
  network: network_segment
)

Example declarative code:
virtual_machine:
  name: test_application_server
  source_image: 'base_linux'
  cpu: 2
  ram: 2GB
  network: private_network_segment
```



The diagram consists of a dashed rectangular box on the left labeled 'Network private'. Inside this box is a teal rectangular box labeled 'Server test_application_server'.

Source: Petra Beenken (2024), based on Morris (2020, p. 54).

Figure 3 shows the code for a private network server. The code for creating this server is defined in a procedural language similar to Java and in a declarative language similar to Terraform. Both codes will create a server named “test_application_server” in a network named “private.” This is just the beginning of a code for an infrastructure that could also contain docker images, virtual machines, or several network segments. This shows how easily a complex network can be generated via code (Morris, 2020).

1.3 Advantages of Considering Security Early

Software security is complex in nature; however, we can focus on a limited number of basic principles that can minimize the risk. Every year, many organizations, such as the SANS Institute and the Open Worldwide Application Security Project **Open Worldwide Application Security Project** (OWASP), create lists of top vulnerabilities. The OWASP Top 10 is a well-renowned listing of the most critical risks for web applications (OWASP, 2023a). Furthermore, The MITRE Corporation publishes a list of common weaknesses called the Common Weakness Enumeration (CWE; MITRE, 2023b). The CWE list can be matched to a risk category in the OWASP Top 10, as shown in the table below. Understanding the nature of threats and risks makes software developers less prone to mistakes. By mentioning security early, common vulnerabilities can be prevented. The most common vulnerabilities are as follows:

Open Worldwide Application Security Project is a non-profit organization for software security. The “Worldwide” part of the name also appears as “Web” in some sources.

Table 1: OWASP Top 10 in 2021 With CWEs Mapped

No.	OWASP Top 10 (Risk category)	CWEs mapped (Weakness category)
1	A01:2021 – Broken access control	CWE-22 Path traversal, CWE-23, ... CWE-352 Cross-site request forgery (CSRF), ... CWE-862 Missing authorization, CWE-863, ... In total, 30 CWEs
2	A02:2021 – Cryptographic failures	CWE-261 Weak encoding for password, ... CWE-310 Cryptographic issues, ... In total, 29 CWEs
3	A03:2021 – Injection	CWE-89 SQL injection, ... CWE-20 Improper input validation, ... CWE-79 Cross-site scripting, ... In total, 33 CWEs
4	A04:2021 – Insecure design	CWE-256 Unprotected storage of credentials, ... CWE-257 Storing passwords in a recoverable format, ... In total, 40 CWEs
5	A05:2021 – Security misconfiguration	CWE-260 Password in configuration file, ... CWE-547 Use of hard-coded, security-relevant constants, ... In total, 20 CWEs
6	A06:2021 – Vulnerable and outdated components	CWE-1104 Use of unmaintained third-party components, ... In total, 3 CWEs
7	A07:2021 – Identification and authentication failures	CWE-287 Improper authentication, ... CWE-306 Missing authentication for critical function, ... CWE-798 Use of hard-coded credentials, ... In total, 22 CWEs
8	A08:2021 – Software and data integrity failures	CWE-353 Missing support for integrity checks, ... CWE-494 Download of code without integrity check, ... In total, 10 CWEs

No.	OWASP Top 10 (Risk category)	CWEs mapped (Weakness category)
9	A09:2021 – Security logging and monitoring failures	CWE-532 Insertion of sensitive information into log files, ... In total, 4 CWEs
10	A10:2021 – Server-side request forgery	CWE-918 Server-side request forgery (SSRF)

Source: Petra Beenken (2024), based on OWASP (2023a) and MITRE (2023c).

In the table above, the OWASP Top 10 of 2021 are listed. In the following sections, we will have a look at some of these OWASP risks in more detail.

The risk category titled “A01: Broken access control” is the highest risk since the update of 2021. However, the risk of broken authentication was fifth in the previous version of the OWASP Top 10. The higher rating may depend on the many CWE list entries that correspond to this risk, shown in the right column of the table above. These CWEs are a categorization of weaknesses that can be mapped to an OWASP risk category. Basically, the A01 risk is about missing access control in different variations, which could be caused by a bypass of an access control via a manipulated Uniform Resource Locator (URL), unprotected files, or replay attacks with JavaScript Object Notation (JSON) Web Tokens (JWT). Mitigation of this risk can be achieved by checking the authorization of the client every time. The enforcement of the security-by-design principle “Least privilege” (see Figure 2) is a possible countermeasure or mitigation of the A1 risk category “Broken access control.” Furthermore, appropriate input validation can help to prevent this risk.

Cross-Site Request Forgery (CSRF)

Vulnerability is a weakness in a computer-based system that can be exploited to cause harm or loss (Sommerville, 2016, p. 377).

Cross-site request forgery (CSRF) is a type of web security **vulnerability** that exploits the trust that a web application has in user’s browser (Conklin & Shoemaker, 2022). This attack usually works in combination with a social engineering attack, such as sending a malicious link or email to the user or embedding a hidden form or script on a website that the user visits, because the victim must perform the action consciously. For example, an online bank allows customers to transfer money to other accounts by filling out a form on the bank’s website. An attacker wants to steal money from such an account. The attacker creates a malicious website that contains a hidden form:

```
<form action = https://bank.com/transfer method="POST">
<input type="hidden" name="amount" value="10000">
<input type="hidden" name="to" value="Attacker's account number">
</form>
<script>
document.forms[0].submit();
</script>
```

The attacker sends an email with a link to their malicious website or embeds their website in a frame on another website. The unsuspecting user clicks on the link or visits the website that contains the frame, and the user's browser loads the attacker's website. This website automatically submits the hidden form to the bank's website, using the user's cookies that are stored in their browser.

Cross-Site Scripting (XSS)

Cross-site scripting (XSS) is a type of web security vulnerability that allows an attacker to inject malicious code into a web page or application that is viewed by other users. There are three main types of cross-site scripting attacks: reflected, stored, and Document Object Model (DOM)-based (Deleting Solutions, n.d.).

Cross-site scripting exploits the trust a user has in a website (Conklin & Shoemaker, 2022).

Reflected XSS

In reflected XSS, the attacker sends a malicious link or email to the user, which contains some code that is embedded in the URL or the form data. When the user clicks on the link or submits the form, the code is sent to the web server, which reflects it back to the user's browser without validating it. The browser then executes the code as if it was part of the web page. For example, a website allows users to search for products by entering a keyword in a form.

The URL of the search results page looks like this: `https://example.com/search?keyword=shoes`

The attacker changes a malicious URL like this: `https://example.com/search?keyword=<script>alert('XSS')</script>`

If the user clicks on this link, the web server will return a page that contains the script tag in the keyword parameter, and the browser will display an alert box saying "XSS".

Stored XSS

The attacker stores some code on the web server, which is then displayed to other users who visit the affected web page or application. For example, a website allows users to post comments on articles. The attacker can post a comment that contains some code like this: `` This code creates an image element with an invalid source attribute, which triggers an error event. The on-error attribute specifies a script to run when the error event occurs. If another user views this comment, the browser will try to load the image, fail, and then execute the script that displays an alert box saying "XSS".

DOM-Based XSS

The attacker modifies the structure or content of the web page's DOM using some code that runs in the user's browser. The code can be embedded in the URL, the HyperText Markup Language (HTML), or a script file that is loaded by the web page. The web server is not involved in this type of attack. For example, a website uses JavaScript to display some

content based on the value of a URL parameter. The URL of the web page looks like this: `https://example.com/page?name=User` The JavaScript code looks like this: `document.write('Hello ' + location.href.split('=')[1]);`

This code writes “Hello User” to the web page by taking the value of the name parameter from the URL. The attacker can craft a malicious URL like this: `https://example.com/page?name=<script>alert('XSS')</script>` If the user visits this URL, the JavaScript code will write “Hello `<script>alert('XSS')</script>`” to the web page, and then execute the script that displays an alert box saying “XSS”.

Input validation can be called the first line in software defense. Each software is created to receive, process, and release data. The information given by a user (e.g., via a formula or in the URL) might be malformed. The verifying of external input data should be included in the well-designed software.

Server-Side Request Forgery (SSRF)

Server-side request forgery (SSRF) is a type of web security vulnerability that allows an attacker to cause the server-side application to make requests to an unintended location. For example, an attacker can exploit a web application that allows users to enter a URL and fetches the content of that URL for them, or it can exploit a web application that runs on a cloud platform, such as Amazon Web Services (AWS), Azure, or Google Cloud, and uses metadata endpoints to obtain configuration information or credentials (Google Cloud, n.d.).

Defensive input validation is effective against injection attacks, listed as number three in the OWASP Top 10. “A03: Injection” could refer to, for example, command injection, Structured Query Language (SQL) injection, code injection, or cross-site scripting. OWASP has compiled a “cheat sheet” as a list of recommendations for providing the input validation functionality in software (OWASP, n.d.-a). A clear example of improper input validation is represented in CWE-20 (see the row “A03: Injection” in the table above). The user is asked to give the height and width of the game board with maximum dimension of 100 squares. The code doesn’t permit the user to input large positive integers, but it doesn’t check whether the numbers are negative. The result of this mistake can be “Resource consumption” (CWE-400), “Memory allocation with excessive size value” (CWE-789), or “Integer overflow and wraparound” (CWE-190).

The following source code shows insecure code in the programming language C that reflects the CWE-20 example for insecure input validation and is also an example for the OWASP risk category “A03: Injection”:

```
...
#define MAX_DIM 100
...
/* board dimensions */

int m,n, error;
board_square_t *board;
```

```

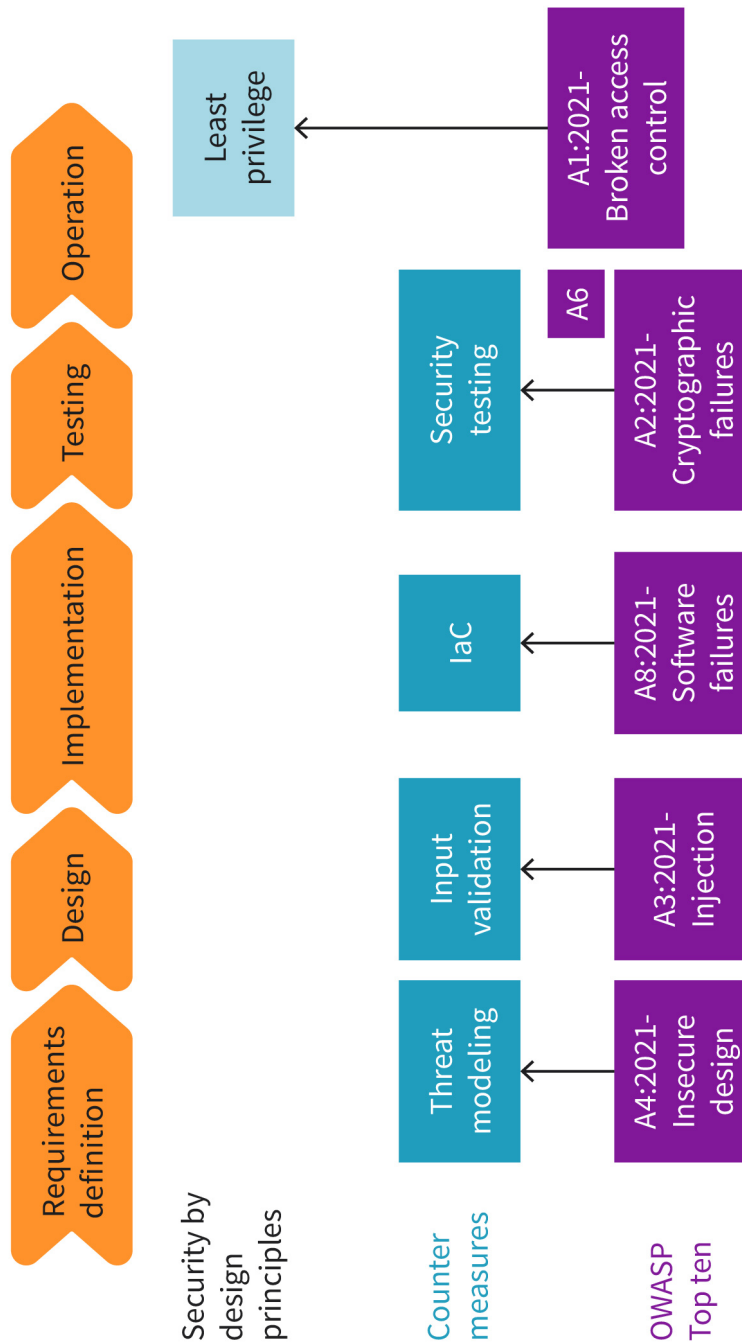
printf("Please specify the board height: \n");
error = scanf("%d", &m);
if ( EOF == error ){die("No integer passed: Die evil hacker!\n");
}
printf("Please specify the board width: \n");
error = scanf("%d", &n);
if ( EOF == error ){die("No integer passed: Die evil hacker!\n");
}
if ( m > MAX_DIM || n > MAX_DIM ) {die("Value too large: Die evil hacker!\n");
}
board = (board_square_t*) malloc( m * n * sizeof(board_square_t));

```

The cases of unanticipated use of services and feature interactions are investigated and recorded by many groups, in particular SANS and OWASP. The CWE description contains examples in different languages and observes examples with a Common Vulnerabilities and Exposures (CVE) number (MITRE, 2023a). Whereas CWE is a categorization for software weaknesses and vulnerabilities, CVE is a list of known vulnerabilities.

The following figure shows that the above weakness of improper input validation (see “A03 Injection” in the table above) can be addressed in the design phase via appropriate input validation. Meanwhile, other risks, like A02 “Cryptographic failures” can be addressed in the testing phase. A security test can find weak cryptography used in code and help find vulnerable and outdated components (A06).

Figure 4: Countermeasures for OWASP Risks



Source: Petra Beenken (2024).

An insecure design, perhaps due to the unprotected storage of passwords (CWE-256), could be prevented through threat modeling. If a software project starts by using threat modeling for important aspects, like password credentials and their proper protection, at the beginning of the project – during either the planning phase or requirements definition phase – these threats would be addressed and dealt with early. This may be achieved, for

example, via a security requirement that forces the hashing of stored passwords. Figure 6 shows this countermeasure against the risk of insecure design (A04 in Table 1) through threat modeling.

Another OWASP risk category mentioned in Figure 6 is A07 “Software failures.” The use of an infrastructure generated via code (IaC) can help mitigate this risk via an appropriate integration and integrity check of used infrastructure components.

All classified vulnerabilities belong to diverse types and are different in nature. Despite all of this, possible threats can be minimized if the software developer understands the current security risks from the very beginning and follows the rules and best practices of SDLC. SDLC provides security activities as measures in each phase. Some of them are mentioned in the figure above. One can see that the risk category OWASP A01 “Broken access control” can be mitigated by mentioning the security-by-design principle “Least privilege,” which requires a minimum set of access rights for a user. Further secure software coding best practices can be found in OWASP (2010).

It is important to be aware of the possible risks and weaknesses to prevent vulnerabilities being exploited, resulting in successful attacks on software products. One of the advantages of considering security early is software launches starting on time, preventing costly bug fixing.



SUMMARY

In this unit, the “shifting left” methodology was introduced. This methodology seeks to consider security aspects earlier in the software development process. Bringing security in right before software launch can lead to expensive and complex bug fixing. If security is mentioned in the design phase, known as “security by design,” weaknesses can be prevented from the outset.

With infrastructure as code (IaC), a whole infrastructure for software projects can be implemented, such as software via code. This brings security benefits because the automation and fast deployment can help to prevent (manual) misconfigurations.

The OWASP Top 10 lists the main risks for software (web) projects. As a developer, it is important to know these kinds of potential weaknesses and how to mitigate the risks.

UNIT 2

PRIVACY BY DESIGN

STUDY GOALS

On completion of this unit, you will be able to ...

- know principles of privacy by design.
- decide which principles relating to the processing of personal data are most important for a project.
- understand how to improve privacy using encryption, differential privacy, and zero-knowledge proofs.

2. PRIVACY BY DESIGN

Introduction

Privacy by design is an essential aspect of security measurements, ensuring that privacy protection measures are built into the design of the systems. Security by design is a philosophy that focuses on protecting a product over its entire lifetime. Software development is a cyclical process that includes several recurrent stages: analysis, design, development, testing, deployment, and maintenance. (Various sources sometimes suggest other software development life cycle [SDLC] phases/stages; these include more detailed sub-categories and the general, but the general idea remains the same). Privacy measurements should be considered and integrated into the product from the first steps of the SDLC process, originating in the stage of analysis and gathering requirements. However, existing tools, such as design patterns or privacy enhancing technologies, are assigned to be used only within design and development stages.

The concept of privacy by design was developed by Ann Cavoukian and widely accepted at the International Conference of Data Protection and Privacy Commissioners in Jerusalem in 2010. The main goal of a privacy-by-design approach is the treatment of privacy as the “default modus operandi.” To achieve this goal, the seven foundational principles of privacy by design were developed (Cavoukian, 2010; Nissen, 2014):

1. **Proactive not reactive; preventative not remedial:** Each product should be designed in such a way that any possible risks are identified from the very beginning of the development process and that threats are minimized and not fixed post-factum.
2. **Privacy as the default setting:** Personal data should be automatically “by-default” protected in any product – default settings should be set to a level that provides maximum privacy.
3. **Privacy embedded into design:** Privacy should be integrated into the core of the product. It can’t be embedded as a separate module or level.
4. **Full functionality – Positive-sum, not zero-sum:** According to this principle, the optimal balance between privacy, functionality, usability, and security should be found.
5. **End-to-end security – Full life cycle protection:** Privacy should be integrated into the product throughout the life cycle. Different mechanisms (e.g., anonymization techniques or data processing based on profiles or default encryption) should be implemented throughout all of the stages.
6. **Visibility and transparency – Keep it open:** The common user of the product should have the possibility to check that declared measures are in accordance with the implemented and used ones.
7. **Respect for user privacy – Keep it user-centric:** The user’s interests and desires should be given top priority. Regardless of whether an organization has another legitimate interest, the user should always be able to manage their private data.

The **General Data Protection Regulation** (GDPR) sets privacy or “data protection by design” as an obligatory requirement to be fulfilled (EU, 2016).

The measures taken within the privacy by design can be divided into two groups: organizational methods and technical methods. Experience shows that technical methods have greater effect and are preferable because they are less susceptible to human error. Article 32 of the GDPR defines technical and organizational measures (EU, 2016). Examples of some privacy-by-design organizational methods include the following:

- measures aiming to create a simple, transparent, and easy-to-keep-up-to-date policy of using private data
- measures designating a person/team responsible for security and privacy with respect to data protection in the organization
- measures restricting access via an organizational directive to personal data (and permit it only to a limited group of people/resources)

The restrictions of access to personal data could also be implemented via a technical measure. Examples of some privacy-by-design technical methods include the following:

- implementing **encryption**
- using access control and authentication
- implementing data minimization techniques, like anonymization or pseudo anonymization in respect to pseudonymization (Bowman et al., 2015, p. 117) to reduce the scope of processed personal data
- implementing privacy-enhancing technologies (PETs), like differential privacy and homomorphic encryption

In the following sub-units, three technical measures will be discussed in more depth: encryption, differential privacy, and zero-knowledge proofs.

2.1 Encryption

One of the most popular technical methods for protection of data within the privacy-by-design policy is encryption, which can be used for data in rest and in transit. All encryption algorithms are based on the two general principles of substitution and transposition. Substitution is the replacement of a character/group of characters with a different character/group of characters. Substitution can be monoalphabetic (the same alphabet is used for the entire message) or polyalphabetic (different alphabets are used for different parts of the message). Transposition is changing the order of characters/groups in the text according to the predefined rules. It can be keyless (a fixed pattern is used for rearranging) or keyed (different patterns based on a key are used). Substitution and transposition are usually used together in product ciphers.

There are two main types of encryptions: symmetric and asymmetric encryption.

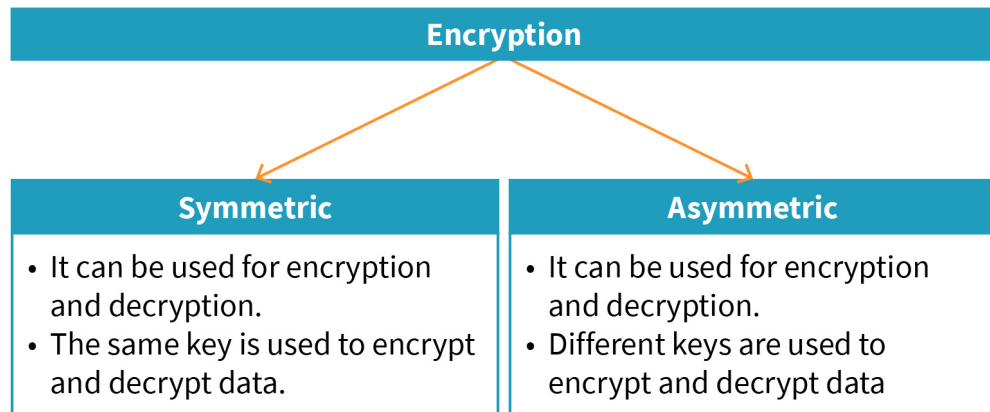
General Data Protection Regulation

is an EU privacy and security law that was passed on May 25, 2018. It describes how personal data should be processed, stored, and used.

Encryption

is the process of using an algorithm to transform plain text into cypher text to ensure that sensitive data remain unreadable to unauthorized users (Rouse, 2023).

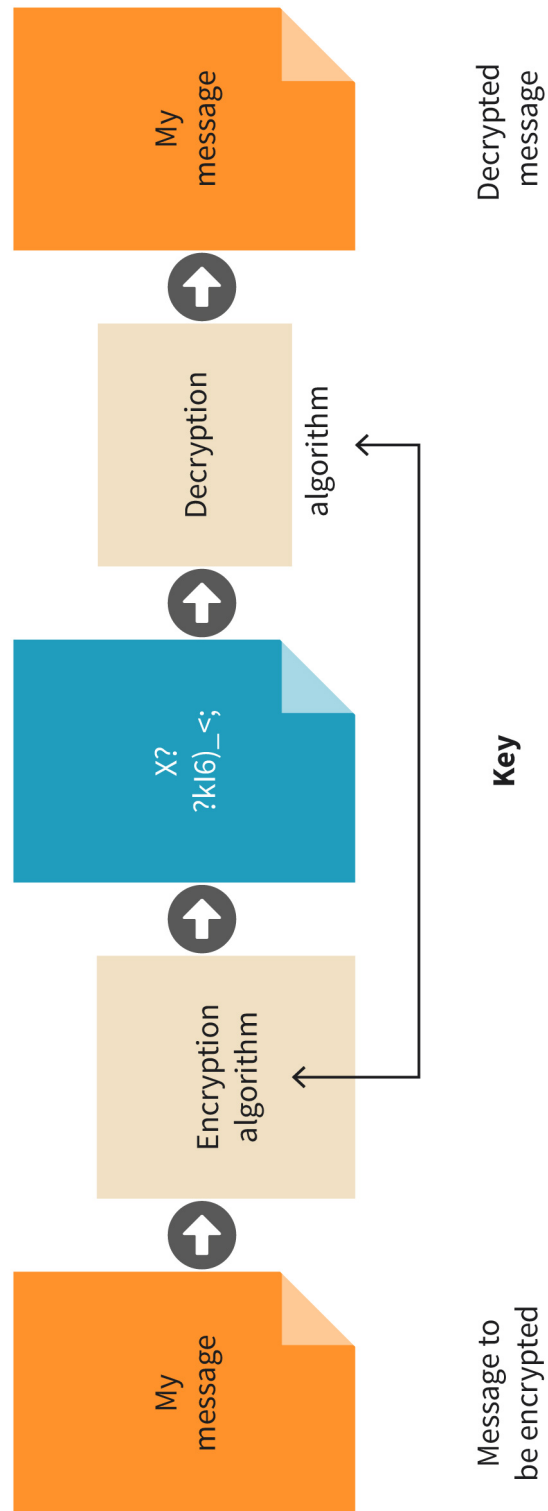
Figure 5: Encryption Types



Source: Vladyslava Volyanska (2024).

Symmetric Encryption

Figure 6: Symmetric Encryption



Source: Vladyslava Volyanska (2024).

In symmetric encryption, a two-way algorithm is used. During decryption, the mathematical procedure is turned back and the same key as for encryption is used.

There are two main algorithms and two main methods (with respective keys) for implementing symmetric encryption: block algorithms (block ciphers) and stream algorithms (stream ciphers).

Block ciphers

Block ciphers encrypt data in fixed-sized blocks, or “chunks.” These blocks are worked out one after another with the same key. Block ciphers are generally resistant to attacks but have an important security gap: During the process of encryption, the system waits for all blocks, which can lead to padding in data or delays. To solve this problem, a process called “feedback” was proposed. Feedback allows the output of the encryption operation to be used as an input for the next one, which increases unpredictability. There are two main types of feedback: cipher feedback (CFB) uses ciphertext from the previous block as an input for the next block; meanwhile, output feedback (OFB) uses the output of the encryption function as an input for the next block, disregarding the ciphertext (Dworkin, 2001).

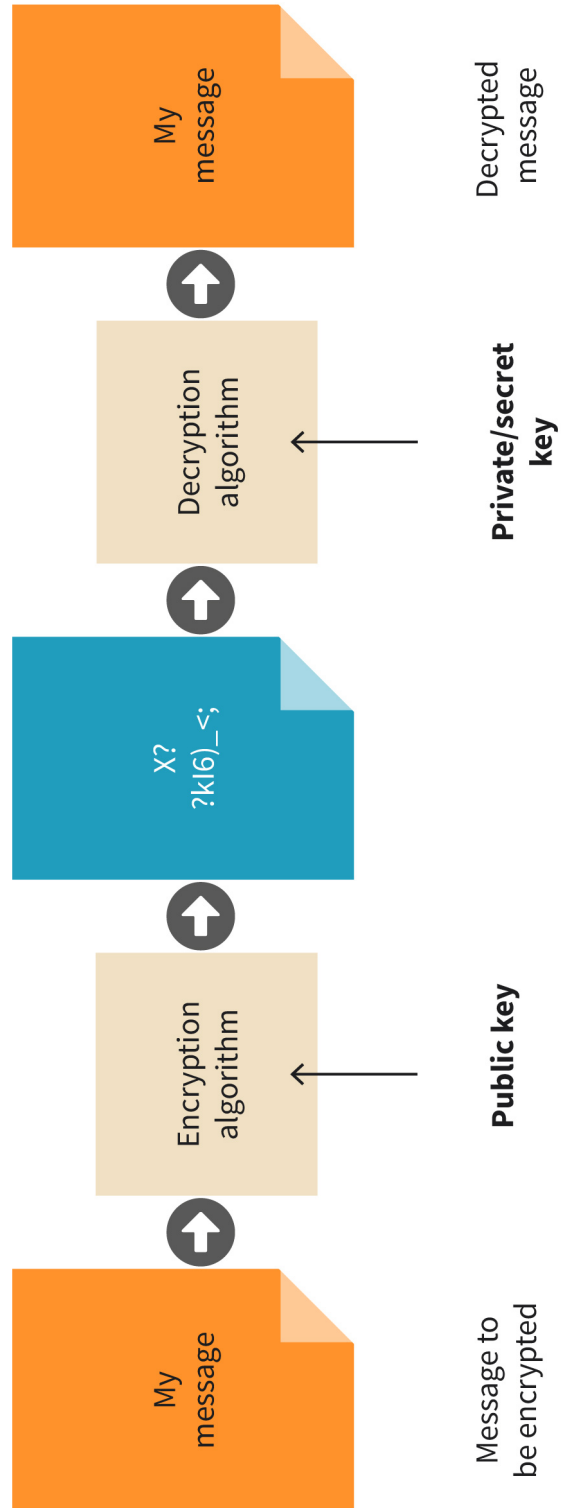
Stream ciphers

Stream ciphers encrypt data one bit or byte at a time using a pseudorandom cipher digit stream (“keystream”). This method is safer because it doesn’t keep unencrypted data in memory; however, it also has some limitations, such as vulnerability to repeated keystream attacks, sensitivity to bit errors or insertions, and the fact that it only provides confidentiality but not integrity or authentication, which means it is necessary to combine data with a hash function or message authentication code (MAC). A MAC, also called an “authentication tag,” is a short piece of information that is used to verify the authenticity and integrity of a message. A MAC is calculated by using a secret key and a message as inputs, and applying a cryptographic algorithm that produces a unique output. The MAC can then be attached to the message and sent to the receiver, who can use the same key and algorithm to verify that the message is original. There are different ways to generate a MAC, such as using block ciphers, hash functions, or specially designed algorithms.

The most typical symmetric encryption examples are block ciphers like the Advanced Encryption Standard (AES), Data Encryption Standard (DES), Rivest Cipher 5 (RC5), and Blowfish, or stream ciphers like RC4, Salsa20, A5/1, and ChaCha20. Encryption is widely used to protect privacy and secure data; for example, RC4 is used in various protocols such as Wired Equivalent Privacy (WEP), Wi-Fi Protected Access (WPA), Secure Sockets Layer (SSL), and Transport Layer Security (TLS); ChaCha20 is used in Internet Protocol Security (IPsec), TLS, and SSL; and Salsa20 is used in Domain Name System (DNS) Curve.

Asymmetric Encryption

Figure 7: Asymmetric Encryption



Source: Vladyslava Volyanska (2024).

Asymmetric encryption uses different keys for encryption and decryption (Wagner, 2022). A pair of keys (public and private) – mathematically related but hard to derive from one another – are generated together. This pair is based on long primary numbers. A public key is freely distributed and used to encrypt the message, while a private key, also called a “secret key,” is kept secret and used to decrypt the message. Asymmetric encryption uses functions that are easy to compute in one direction, but it is difficult to do so in the reverse direction. The most popular examples of asymmetric ciphers are based on the Rivest–Shamir–Adleman (RSA) algorithm, Diffie–Hellman, or a family of elliptic-curve cryptography (ECC) algorithms.

Symmetric and asymmetric encryption have different fields of application based on their characteristics. Symmetric encryption is faster and has significantly shorter keys for the same security strength (for example, 256 vs. 2,048 bit). Meanwhile, asymmetric encryption has other benefits, like the fact it needs fewer key overalls. Key sharing is an issue involved in the symmetric approach because the same key should be known when both encrypting and decrypting the message. But how does one share this key with the other party? Asymmetric encryption doesn’t solve this problem – the public key that is needed to encrypt the message is open and generally available. Because of the above, a hybrid approach is generally used in practice: Asymmetric encryption is used for generating signatures and key establishment, while symmetric algorithms are used to encrypt data. This approach combines the benefits of symmetric and asymmetric encryption; for example, it counteracts the slower processes of asymmetric encryption and doesn’t involve the problem of key exchange in symmetric encryption.

Hashing

This is the technique of converting inputs of different length and types into a fixed-sized string of chars.

Char

a pointer to a character, which can be used to point to a single character or the first character in an array of characters

To further increase security, **hashing** can be used. Hashing is often confused with encryption, but it is not an encryption technique. There are some features that are easy to notice; for example, when we encrypt a paragraph of text, a paragraph of encrypted text is generated; when we encrypt a phrase, we get an encrypted phrase; when we are hashing a paragraph of text, we get a string (e.g., 50 **chars**); and when we are hashing a single phrase using the same hash algorithm, we also get a string of 50 chars. It is important to note here that in using the same algorithm, we get as a result a string of the same length for different length inputs. It is a kind of signature used to verify a file or document. The second important characteristic is that encrypted text can be decrypted back, and the use of hash algorithm is a one-way process. Collisions can occur with some hashing techniques. In situations where the hashing algorithm isn’t complicated enough, the same string can appear as an output for different texts.

Homomorphic Encryption

A third form of encryption is homomorphic encryption. This form is newer than the others but already has a variety of applications, as it provides clear services without providing open users’ data. Operations can be performed on encrypted data without decrypting them. The results of such operations remain encrypted and can be obtained in decrypted data (Bowman et al., 2015, p. 23).

There are two main types of homomorphic encryption:

1. **Partially homomorphic encryption:** This type supports the evaluation of only one type of gate. It means that, for example, two encrypted numbers can be added (through addition, subtraction, multiplication, etc.) without the encrypted digits being known. The operation is performed on encrypted numbers without decrypting them before the operation and then the encrypted sum can be deciphered, wherein the input numbers remain unencrypted. Some homomorphic encryption schemes can evaluate two types of gates but only for a subset of circuits. It means that these schemes can perform the operation for a limited set of operations: This limit comes from the fact that ciphertext generates too much noise in the data.
2. **Fully homomorphic encryption (FHE):** It supports the evaluation of multiple types of gates of unbounded depth and is the strongest form of homomorphic encryption. As a subtype of FHE, we can also distinguish leveled FHE. This form supports evaluation for several types of gates of pre-determined (bounded) depth. The significant issue with FHE is low performance. Currently, two approaches are used to solve this problem: the use of leveled fully homomorphic encryption or the use of the ciphertext packing method, where several plaintexts are written into one ciphertext and at the same time during a single operation. When using FHE, it is highly recommended to limit the number of operations performed using data. The error (or error value) that accumulates as a result of too many performed operations can exceed the value of the secret parameter, which can cause data not to be decrypted correctly.

Table 2: Encryption Techniques and Data-Transforming Techniques

Encryption techniques				
Form	Type	Key	Example	Application area
Symmetric encryption	Symmetric	Single	DES, AES, RC4	WLAN, VPN, VoIP, file encryption
Asymmetric encryption	Asymmetric	Public and private	RSA, ElGamal, ECC	Email, blockchain, HTTPS
Homomorphic encryption	Asymmetric	Public and private	BGV, CKKS	Cloud computing, data protection, machine learning
Data-transforming techniques				
Encoding	Substitution or transposition	Algorithm or codebook	ASCII, Enigma, Morse code	Data transmission, steganography, historical cryptography
Hashing	One-way function	None or secret	SHA-256, MD5	Authentication, password storage, data integrity

Source: Vladyslava Volyanska (2024).

To utilize the implementation of encryption in software development, specialized ready-to-use libraries can be used. These libraries exist for many different programming languages.

2.2 Differential Privacy

We are increasingly using virtual assistants in daily life; for example, we may be woken up by a smartphone at a different time each day of the week depending on how long we need to get to work. This is not a complicated task, but smartphones collect some personal information to work effectively: for example, where we live (including the distance to our workplace), how long we need to get to work (including whether we use public transport or a car), and how intensive the traffic is on a particular day. Additionally, smartphones gather information about our personal morning preferences, such as if we usually buy a morning coffee at the local store on the way to the subway and how long it takes (perhaps there is a queue there today!). All of these data can be obtained in different ways: We can provide it ourselves or the information can be collected automatically.

Collecting, storing, and processing such data is complicated in the context of security and privacy. A method known as **differential privacy** is used, where the data are collected from global users, with the source of each piece of information remaining unknown. The main idea behind this approach is to provide privacy while sharing information. It is achieved by introducing small changes to private data, which doesn't change the statistics of interest. The goal of differential privacy is to increase the accuracy of gathered information while reducing the probability of identifying individuals whose information was used as a source.

The main challenge of differential privacy, which is a theme for ethical and social discussions, is the balance between the security of used private data and the utility of gathered data. But some techniques intentionally used to bring **noise** into datasets, while private algorithms seem to be more reliable. A private algorithm is one that satisfies differential privacy and means that the output of these algorithms cannot reveal any sensitive information from the individuals in a dataset.

Among the techniques used to bring noise to datasets are those listed below.

Laplace Mechanism

It provides noise to the exact result of a query (a function's output) by sampling from the Laplace distribution. It preserves $(\epsilon, 0)$ differential privacy, where (ϵ) is the desired level of privacy. The value of (ϵ) determines the level of privacy protection. The smaller value provides better privacy protection, but at the cost of more noise being used in the data. A larger value of (ϵ) provides weaker privacy protection, but less noise is used in the data. The sensitivity of such a function is the amount of output changes when the input changes by 1 (Dwork & Roth, 2014).

To illustrate the Laplace mechanism, the graph below can serve as a useful guide. The x-axis is used for the true value of the function and y-axis for the noise value returned by the Laplace mechanism. The graph also shows the probable density function of the Laplace distribution, which is symmetric and has a peak at the true value. The graph demonstrates how adding noise from the Laplace distribution can obscure the true value of the function, while preserving some statistical properties.

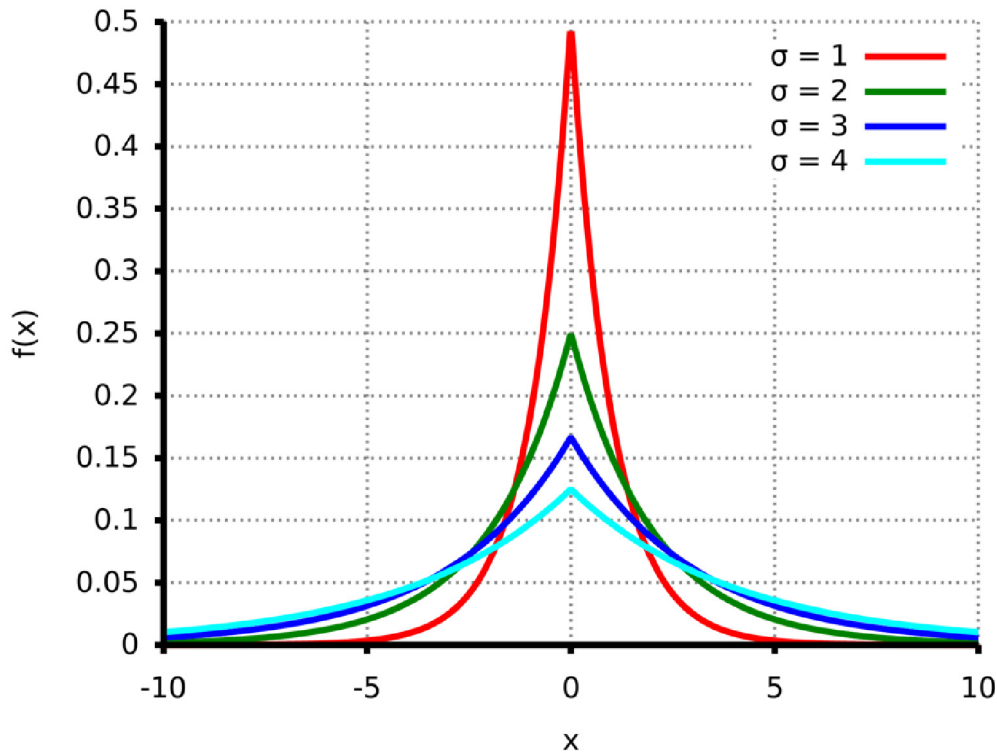
Differential privacy

This is an approach allowing information describing the patterns of groups within datasets to be shared and made public while still protecting the privacy of the individuals whose data are being used (Bowman et al., 2015, p. 118).

Noise

refers to unwanted data items or records. It can cause algorithms to miss out patterns in data or lead to false conclusions.

Figure 8: Laplace Distribution Where $\sigma = 1/\epsilon$ ($\sigma = 4, \epsilon = 0,25$)



Source: StefanPohl (2014). CCO 1.0.

Sub-Sampling Technique

This technique applies a different private algorithm to randomly selected subsets of data. It assumes that the level of privacy protection increases as the subset size decreases. The methods used within this approach are Poisson sampling, Bernoulli sampling, and simple random sampling. This technique can improve the accuracy of estimation, but only when the sensitivity of output doesn't depend on the size of input data. However, this technique shouldn't be used by histograms or statistic counts because the sampling can be strongly decreased (Yang et al. 2023).

Privacy Amplification Using an Iteration Technique

The purpose of this technique is to improve guarantees of privacy. The main idea of iterative aggregation is to update the intermediate solutions iteratively, using only a few data points for each iteration. The practice of not releasing intermediate results is known as "privacy amplification." Privacy amplification is a technique used to increase the level of privacy protection provided by an algorithm by reducing the amount of information that is released about the data. By releasing only a small amount of information at each iteration, the algorithm can provide stronger privacy guarantees than if it were to release all of the information at once.

Privacy-Enhancing Cryptography (PEC)

Privacy-enhancing cryptography (PEC) uses cryptography to enable differential privacy computation, homomorphic encryption, secure multi-party computation, and zero-knowledge proofs. PEC can be used in conjunction with differential privacy to enable data sharing while also protecting privacy; for example, PEC can be used to enable secure multi-party computation without revealing extraneous private information to one another or to third parties. PEC and differential privacy are complementary techniques that can be used to achieve both input and output privacy. PEC can also be used to securely compute a function on multiple datasets without sharing them, while differential privacy can be used to add noise to the output of the function to prevent the leakage of sensitive information; for example, PEC can be used to compute the average salary of employees across different companies without revealing the salaries of individual employees or companies, while differential privacy can be used to add noise to the average salary to prevent inference of outliers or extreme values. By combining PEC and differential privacy, one can achieve both data utility and data privacy in a variety of scenarios.

Differential privacy is used in many different fields that require the use of statistical data analysis along with securing privacy. Some popular areas of application for differential privacy include, for example, genomics, healthcare, the Internet of Things, biomedical analysis, geolocation, and Apple telemetry.

2.3 Zero-Knowledge Proofs/Protocols

Within differential privacy, a zero-knowledge proofs (or protocols; ZKP) method is widely used, which is a type of privacy-enhancing cryptography technique. By using a ZKP method, one part of an interaction (the prover) can prove to the other part (the verifier) that something is true, wherein any additional information wouldn't be revealed except for the fact that an exact statement is true. ZKPs are a type of probabilistic assessment, which means that a fact proved by them is not as certain as clear data. They are used to verify the authenticity of a statement, but because of their probabilistic nature, the probability of the statement being true is not as certain as the data themselves. ZKPs are used in secured supply chains, authentication, private transactions, private identity verification, and to protect data privacy in blockchain (Dilmegani, 2021).

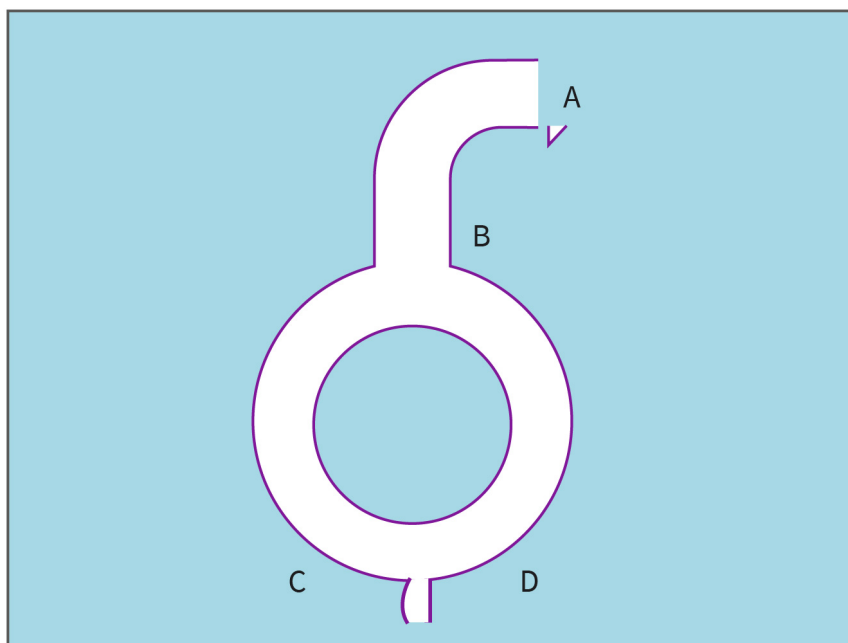
Zero-knowledge proofs belong to interactive cryptography protocols. This means that a protocol requires an interactive input from a verifier, which usually comes in the form of a task. The goal of the prover is to persuade the verifier that they have the solution without revealing the smallest part of the "secret proof" (resulting in zero disclosure). The purpose of the verifier is to ensure that the other part does not lie. This type of protocol has three characteristics (Menezes et al., 1996):

1. **Completeness:** If the statement is true, then the prover will persuade the verifier with predetermined accuracy.
2. **Correctness:** If the statement is false, then any prover will not be able to convince the verifier (negligible probability).
3. **Zero disclosure:** If the statement is true, then any verifier will not learn anything except for the fact that the statement is true.

There are several abstract examples that represent ideas of zero-knowledge proofs, for example, “Ali Baba’s cave” and “two balls and the colorblind friend.”

In the analogy of Ali Baba’s cave (see the figure below), the cave has a ring shape with the entrance on one side and a magic door on the other side that blocks entry from that side. “Peggy” (the name uses the “p” in “prover”) has a secret word (a “key”) that is used to open a magic door in the cave between points C and D in the figure. “Victor” (from “verifier”) needs to know whether Peggy really knows the password because she doesn’t want to reveal any information. Victor and Peggy reach the cave at point A; Victor stays at A and Peggy disappears around the corner out of view. Victor continues to point B and shouts from there: “Peggy, should I go right?” (or “Peggy, should I go left?”). The probability that Peggy knows the password is 50 percent at this point. If we repeat the process k times, the probability will be $\frac{1}{2^k}$; when we repeat it 20 times, it will be about 10^{-6} . It is therefore enough to ascertain whether Peggy knows the password (Quisquater et al., 2007, p. 1990).

Figure 9: Zero Knowledge Cave



Source: Vladyslava Volyanska (2024), based on Quisquater et al. (2007).

“Two balls and the colorblind friend” is another example of zero-knowledge proofs. In this analogy, we have a colorblind friend (“Fred”) and two balls: one red and one green. Fred sees both balls as being identical in color. We need to prove to Fred that they are different colors but, at the same time, we don’t want him to know which one is green and which is red. Fred takes a ball in each hand and hides them behind his back. He can switch the balls between his hands or leave them as they are: The probability of each one being in each hand is 50 percent. Fred moves one hand from behind his back and asks us to guess which ball he has in his hand. Each time, we have a 50 percent chance of guessing correctly. When we repeat this many times, Fred is aware that the balls are different colors, but he will never be able to distinguish between them (Chalkias & Hearn, 2019).



SUMMARY

Privacy should be considered and integrated into the product at the beginning of the software development process, while implemented encryption techniques should also be carefully considered and planned at an early stage.

There are three main encryption techniques: symmetric, asymmetric, and homomorphic encryption. Each have their own field of applications and characteristics of implementation. Encoding and hashing are two other techniques that are closely related to encryption, each of which should be used according to the purpose for which they were created with a knowledge of their limitations.

The use of personal data is important for the efficient functioning of many applications for our daily lives. However, it is important to maintain a balance between utility and data privacy. Collecting, storing, and processing data is complicated in the context of security, and differential privacy techniques are used for these purposes.

Within differential privacy, privacy-enhancing cryptography techniques called zero-knowledge proofs (ZKPs) are used. These methods usually take the form of interactive cryptography protocols, which are a type of task. The main goal of ZKPs is zero disclosure.

UNIT 3

TESTING AND AUDITING

STUDY GOALS

On completion of this unit, you will be able to ...

- define types and methods of testing and auditing.
- understand the challenges of testing and auditing in different software development methodologies.
- effectively apply testing and auditing for a future project.

3. TESTING AND AUDITING

Introduction

Testing and code auditing are two essential aspects of software development that ensure the quality, functionality, and security of software. Testing involves verifying the behavior and performance of software against the specified requirements. Software tests and code audits can be performed at different levels: They can test individual program components (units) or the interaction of several software components and can be done manually or automated. The goal is to create software without bugs before it is delivered to the customer. Therefore, the main purpose of testing and code auditing is finding bugs (Spillner & Linz, 2021, p. 12).

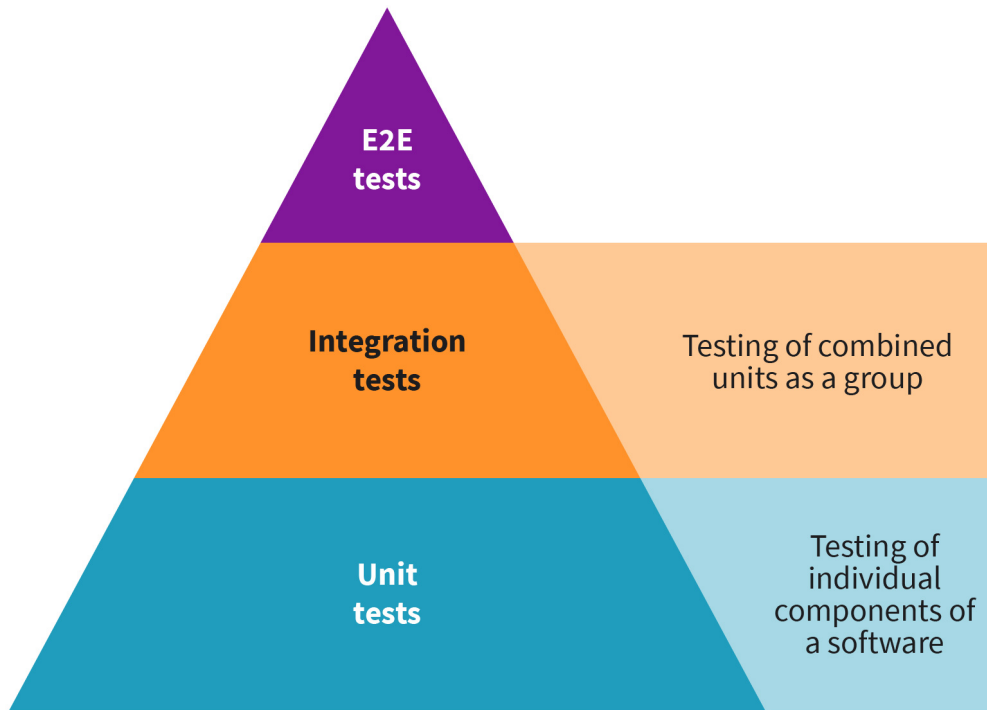
Auditing involves reviewing the software development process and practices to check their compliance with standards, regulations, and best practices. Auditing can be performed by internal or external auditors depending on the purpose and the scope of the audit. In the process of auditing, the software development activities, documents, and deliverables are evaluated. A software audit can have many benefits for the company, such as improving the quality and performance of the software, reducing the number of inactive licenses, ensuring that licenses are up to date, improving business operations and processes, and fulfilling legal and industry requirements. Auditing helps to ensure that the software development process is transparent, efficient, and consistent (Bell, Bryman et al., 2017, p. 155).

Both testing and auditing aim to identify and resolve any risks that may affect the software. Testing and auditing also help to reduce the cost and time of software development by preventing or resolving any problems or failures in the software.

3.1 Unit Testing

The software testing pyramid is a strategy that helps developers and testers create high-quality software in an efficient and effective way. In its classical form, it consists of three main layers: unit tests, integration tests, and end-to-end (E2E) tests. Each layer has a different purpose, scope, and frequency of execution (Singh, 2022).

Figure 10: Testing Pyramid



Source: Vladyslava Volyanska (2024), based on Singh (2022).

Unit tests are at the base of the pyramid. They are small, isolated, and fast tests that check the functionality of individual components or methods of the software. They are usually written by developers using frameworks like JUnit, TestNG, or NUnit. Unit tests should cover as much code as possible and run frequently, ideally with every code change. They provide quick feedback and help identify bugs at an early stage (Adkins et al., 2020, p. 272).

Integration tests are the middle layer of the pyramid. They are larger, more complex, and slower tests that check how different components or services of the software work together. They are usually written by developers or testers. Integration tests should cover the critical interactions and dependencies of the software and run less frequently than unit tests, ideally with every build or release. They provide more confidence and help verify the functionality and performance of the software (Adkins et al., 2020, p. 276).

At the top of the pyramid are E2E tests. They are a type of testing software that simulates the real user experience of an application. They test the entire system or product from start to finish, covering all the possible scenarios and interactions that the user may encounter. E2E tests are important for ensuring the usability and quality of the software, as well as detecting any bugs or errors that may affect user satisfaction (Vocke, 2018).

Other commonly used tests are user interface (UI) or exploratory tests, which are not part of the testing pyramid. Exploratory testing is an approach to software testing that emphasizes the individual tester's creativity and spontaneity. Instead of following pre-defined

test cases, testers dynamically explore the software's functionality. While they may jot down some test ideas prior to execution, the primary focus is on viewing testing as a cognitive process. UI testing focuses on the user interface of an application and verifies that the application's user interface is working as expected and that users can interact with it without any issues.

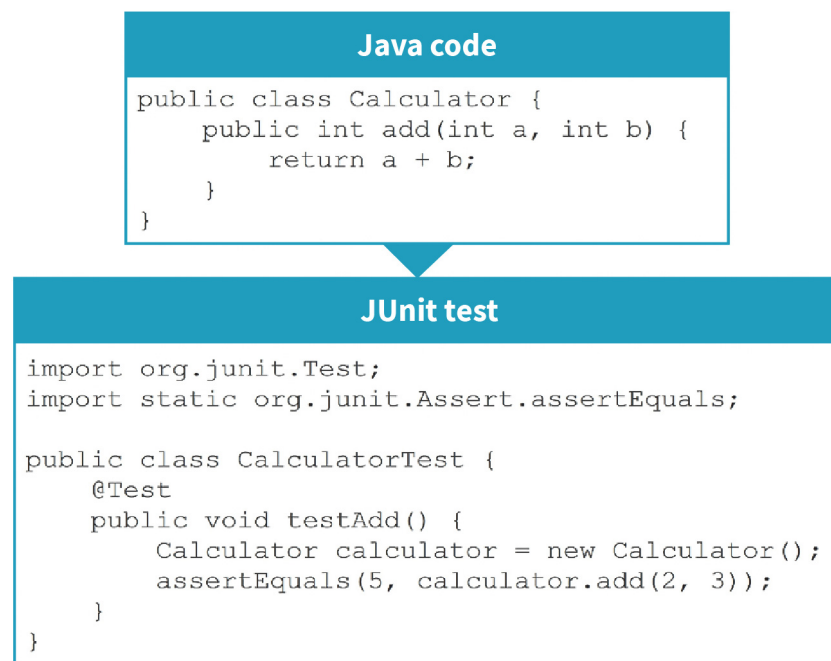
UI or exploratory tests and E2E tests serve different purposes in software testing: UI and exploratory tests are primarily concerned with assessing the software's user interface, while E2E tests aim to evaluate the entire system or product from the perspective of the end user. UI and exploratory tests are usually faster, cheaper, and more isolated than E2E tests, but they do not cover all the possible scenarios and interactions that the user may encounter. E2E tests are usually slower, more expensive, and more complex than UI and exploratory tests, but they provide more confidence and value in validating the usability and quality of the software (Vocke, 2018).

Unit testing is the testing of individual program units (Sommerville, 2016, p. 774).

A **unit test** checks a part (unit) of a software component. Unit tests help to ensure that these small program parts function correctly. The quality and security of the software is improved because errors can be detected early and the source can be accurately identified or limited, namely to the tested unit (Adkins et al., 2020, p. 272).

The following example shows a simple addition as Java code (top image) and a unit test that checks if the function works correctly (bottom image). The JUnit test checks whether the addition of $2 + 3$ equals 5 by using the function `add` of the class "Calculator."

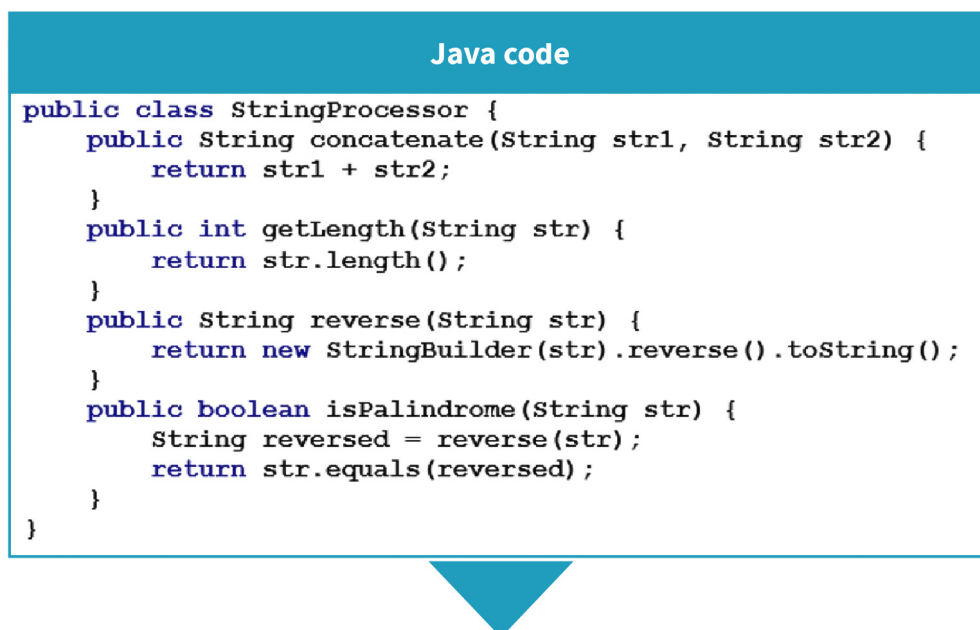
Figure 11: Example of a Java Code and a Corresponding Unit Test



Source: Petra Beenken (2024).

Unit tests are usually automated tests written and run by software developers using frameworks or libraries that provide tools and features to create, run, and report the test results. The outcome of a unit test is binary: either “pass” if the program’s behavior is consistent with the recorded expectations, or “fail” otherwise. Developers will typically write many unit tests (corresponding to a large number of program behaviors of interest) called a “test suite.” By common convention dating back at least to the JUnit family of tools, the color red (as in “getting a red bar”) represents the failure of one or more tests. The color green (“a green bar”) denotes successful execution of “all” unit tests associated with a program. The example in the figure below shows a Java code of a simple String processor, a unit test written for this String processor, and the result of executing it in IntelliJ.

Figure 12: Example of a Java Code, Corresponding JUnit Test, and the Result of Testing in IntelliJ



```
public class StringProcessor {
    public String concatenate(String str1, String str2) {
        return str1 + str2;
    }
    public int getLength(String str) {
        return str.length();
    }
    public String reverse(String str) {
        return new StringBuilder(str).reverse().toString();
    }
    public boolean isPalindrome(String str) {
        String reversed = reverse(str);
        return str.equals(reversed);
    }
}
```

Source: Vladyslava Volyanska (2024).

Unit test

```
public class StringProcessorTest {
    private final StringProcessor stringProcessor = new
StringProcessor();

    @Test
    public void testConcatenate() {
        assertEquals("Hello World!",
stringProcessor.concatenate("Hello ", "World!"));
        assertEquals("We are from Berlin",
stringProcessor.concatenate("We are ", "from Berlin"));
        assertEquals("Let's play a game!",
stringProcessor.concatenate("Let's play ", "a game"));
    }
    @Test
    public void testGetLength() {
        assertEquals(5, stringProcessor.getLength("Hello"));
        assertEquals(0, stringProcessor.getLength(""));
        assertEquals(10, stringProcessor.getLength("Hello, World!"));
    }
    @Test
    public void testReverse() {
        assertEquals("!dlroW ,olleH", stringProcessor.reverse("Hello,
World!"));
        assertEquals("", stringProcessor.reverse(""));
        assertEquals("madaM", stringProcessor.reverse("Madam"));
    }
    @Test
    public void testIsPalindrome() {
        assertTrue(stringProcessor.isPalindrome("madam"));
        assertTrue(stringProcessor.isPalindrome(""));
        assertFalse(stringProcessor.isPalindrome("mau"));
    }
}
}
```

Source: Vladyslava Volyanska (2024).

The result of unit test in IntelliJ

```
Run: StringProcessorTest x
  Tests failed: 2, passed: 2 of 4 tests - 15 ms
  C:\Users\Lucy\.jdk\openjdk-20.0.2\bin\java.exe ...

StringProcessorTest (les_34)
  testConcatenate() 13 ms
  testIsPalindrome() 1 ms
  testReverse() 1 ms
  testGetLength() 1 ms

org.opentest4j.AssertionFailedError:
Expected :Let's play a game!
Actual   :Let's play a game
<Click to see difference>

<6 internal lines>
at les_34.StringProcessorTest.testConcatenate(StringProcessorTest.java:16) <29 internal lines>
at java.base/java.util.ArrayList.forEach(ArrayList.java:1511) <9 internal lines>
at java.base/java.util.ArrayList.forEach(ArrayList.java:1511) <28 internal lines>

org.opentest4j.AssertionFailedError:
Expected :10
Actual   :12
<Click to see difference>

<6 internal lines>
at les_34.StringProcessorTest.testGetLength(StringProcessorTest.java:22) <29 internal lines>
at java.base/java.util.ArrayList.forEach(ArrayList.java:1511) <9 internal lines>
at java.base/java.util.ArrayList.forEach(ArrayList.java:1511) <28 internal lines>

Process finished with exit code -1
```

Source: Vladyslava Volyanska (2024).

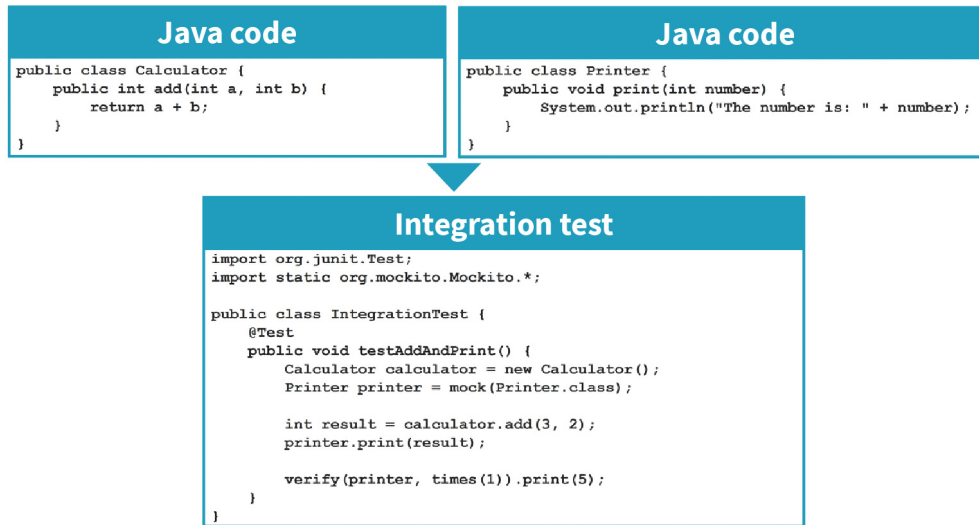
This example shows that only two out of four tests passed. Writing unit tests can also be challenging and prone to mistakes. Some of the common mistakes in writing unit tests include the following:

- **Testing other people's code** means testing code that was written by somebody else, such as external libraries or frameworks, instead of focusing on own code. This can lead to redundant, unnecessary, or unreliable tests. It is better to use mock objects or stubs to simulate the external dependencies in tests.
- **Testing the database connection** means testing functions that deal with databases, such as inserting, updating, or deleting data, instead of testing the logic and behavior of the code. This can lead to slow, fragile, or expensive tests. In such a situation, in-memory databases or fake data should be used to isolate the tests from the database.
- **Writing tests after the fact** means writing tests after the production code, instead of writing them before or along with the code. This can lead to code that is poor-quality and poorly designed, as well as missing bugs and errors. The principles of test-driven development (TDD) should be followed, which is a technique that encourages writing tests first and then writing code that passes the tests.
- **Failing to update the tests after code refactors** means not updating or refactoring the test suite as the production code evolves and changes. This can lead to outdated, redundant, or irrelevant tests that do not match the current state and requirements of the code. The test suite should be kept in sync with the production code and remove any obsolete or duplicated tests.
- **Writing tests are neither unit tests nor acceptance tests** means writing tests that test more than one unit of code but less than the whole system or product from the user's perspective. This can lead to tests that are a burden to maintain and cause people to become frustrated with testing. The testing pyramid strategy should be followed, which consists of three layers: unit tests, integration tests, and UI/exploratory tests. Each layer has a different purpose, scope, and frequency of execution.

Integration Testing

Integration tests are a type of software testing that checks how different components or services of a software system work together. Integration tests are usually larger, more complex, and slower than unit tests, but they provide more confidence and verification of the functionality and performance of the software. Integration tests are typically written and run by software developers or testers using tools or frameworks that allow them to interact with the software components or services through various interfaces, such as application programming interfaces (APIs), web services, and databases (Tran, 2022).

Figure 13: Integration Test of Two Java Classes



Source: Vladyslava Volyanska (2024).

Integration tests are important for ensuring the compatibility and interoperability of the software system, as well as detecting any **bugs** or errors that may arise from the interactions and dependencies between different components or services. Integration tests can also help developers and testers to identify and isolate the root cause and location of the defects, as well as to measure the quality attributes of the software system, such as reliability, scalability, and security.

Bug is a defect resulting from coding errors (Spillner & Linz, 2021).

Integration testing involves different components or modules of a software application being tested as a combined entity (LambdaTest, n.d.). To write effective and maintainable integration tests, developers and testers need to follow some best practices and guidelines (Vocke, 2018):

Define the scope and level of the integration tests

This means it is necessary to decide which components or services are tested together, and how granular or comprehensive the tests should be. Different strategies such as bottom-up, top-down, sandwich, or big-bang can be used. Big-bang testing involves integrating all the components of a system at once and then testing the entire system. Incremental integration testing involves integrating and testing the components of a system incrementally. There are two popular approaches: top-down and bottom-up, while sandwich integration testing is a combination of both top-down and bottom-up approaches. It is important to consider the trade-offs between the cost and benefit of each strategy. See the figure below for a representation of the different types of integration testing.

The appropriate tools and frameworks should be chosen

This means selecting the tools or frameworks that support the chosen strategy, language, platform, and interface for integration testing. There are many options available, such as Postman, SoapUI, Selenium, and JMeter. Different tools or frameworks should be compared and evaluated based on the product needs and preferences.

Write clean and maintainable test code that follows the principles of test-driven development (TDD)

This means writing tests before or alongside the production code.

Use mock objects or stubs to simulate the external dependencies or side effects in the integration tests

This means creating fake objects or methods that mimic the behavior or response of the real objects or methods that are not tested in the integration tests. This can help to isolate tests from the external factors that may affect test results, such as network latency and database availability.

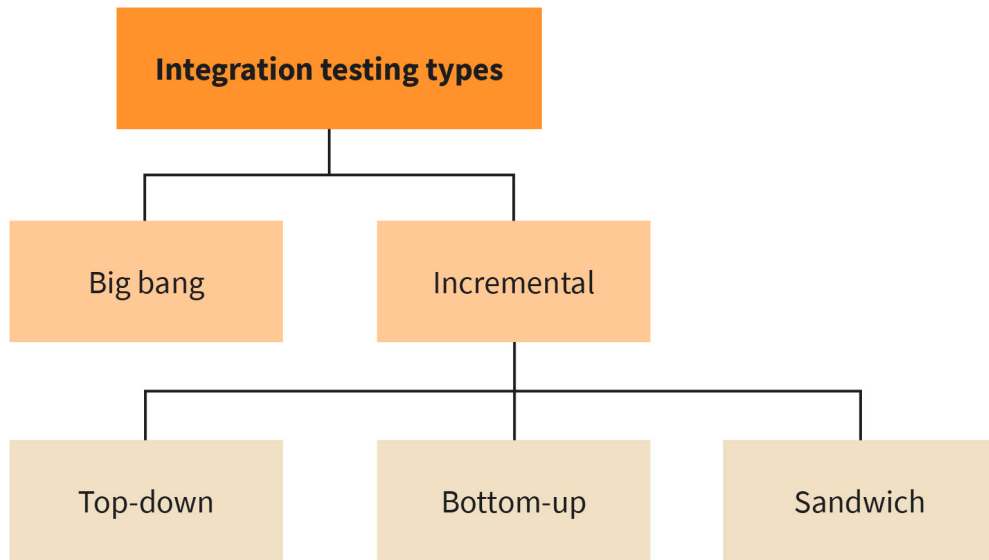
Run integration tests less frequently than unit tests but more frequently than UI or exploratory tests

This means finding a balance between the speed and coverage of integration tests. The integration tests should be run with every build or release of the software system, but not with every code change. The integration tests should be run in a logical order that reflects the dependencies and relationships between different components or services.

Manage test environment and data effectively

This means ensuring that the test environment is realistic and stable and that it resembles the production environment as much as possible. It is necessary to use tools or methods that allow test data to be created, updated, and deleted easily and securely.

Figure 14: Types of Integration Testing



Source: Vladyslava Volyanska (2024).

System and E2E tests are two types of software testing that evaluate the functionality and performance of a complete and fully integrated software solution. They are usually performed after unit and integration tests, which test the individual components or modules of the software. System and E2E tests aim to ensure that the software meets the specified requirements and is suitable for delivery to end users.

System testing is a process where the entirety of the integrated system is tested to ensure it meets specified requirements. This type of testing only begins once all of the individual components have successfully passed integration testing. The goal of integration testing is to identify any discrepancies between the units that have been combined (known as assemblages). The objective of system testing, meanwhile, is to uncover defects within both the “inter-assemblages” and the overall system (Gilmore, 2021).

System testing is a type of testing that is performed on the entire system according to either functional or system requirement specifications (or both). System testing tests both the system’s design and the behavior, as well as the expectations of the customer (ROI4CIO, n.d.). It is also intended to test up to and beyond the bounds defined in the software or hardware requirements specifications. Below are some common approaches for system testing.

Functional Testing

This type of testing verifies that the system meets its functional requirements. It involves testing each function of the system to ensure that it performs as expected.

Performance Testing

This type of testing verifies that the system meets its performance requirements. It involves testing the system under different loads to ensure that it can handle the expected traffic.

Security Testing

This type of testing verifies that the system meets its security requirements. It involves testing the system for vulnerabilities and ensuring that it can withstand attacks.

Nondestructive Testing (NDT)

Nondestructive testing (NDT) is a set of analysis techniques used to evaluate the properties of a material, component, or system without causing damage (Ang, 2022). An example of nondestructive testing in IT is radiographic testing, which uses X-rays or gamma rays to examine the internal structure of an object.

Destructive Testing

Unlike nondestructive testing, destructive testing involves subjecting a specimen to continuous stress until it fails to understand its performance or material behavior under different loads. An example of destructive testing is load testing, which involves subjecting a software application to high loads to determine its breaking point.

Static Testing

Static testing is a type of software testing that involves examining the code and documentation without executing the program. It is used to detect defects in the early stages of the development cycle, for example, during code reviews and inspections.

End-to-End Testing

End-to-end testing (E2E) is a type of software testing that simulates the real user experience of an application. It tests the entire system or product from start to finish, covering all the possible scenarios and interactions that the user may encounter. E2E testing is similar to system testing, but it covers more scenarios and interactions that may not be covered by system testing. E2E testing is important for ensuring the usability and quality of the software, as well as detecting any bugs or errors that may affect user satisfaction. E2E testing can also help developers and testers with writing, rewriting, improving, or optimizing their content.

3.2 Security Testing

Security testing is a type of software testing that assesses the security of a system and identifies its potential vulnerabilities and threats. It is an important part of the software development life cycle (SDLC) and is used to detect security issues in the system to prevent real-world attacks (Pahuja, 2023). Security testing has several advantages, including

- improving the security and quality of the software,
- protecting sensitive data from unauthorized access or disclosure,
- reducing the risk of a security breach and its associated costs,
- enhancing compliance with data protection laws and regulations, and
- increasing productivity and customer satisfaction (Pahuja, 2023).

There are different types of security testing that can be performed on a system, depending on its requirements and objectives. Some common types include vulnerability scanning, penetration testing, static application security testing (SAST), dynamic application security testing (DAST), interactive application security testing (IAST), and security auditing. While each form of security testing employs its own unique strategies and methodologies, their collective goal is to pinpoint system vulnerabilities and weaknesses. They also provide guidance on how to enhance the system's security. Some of the common types include the following:

- **Vulnerability scanning** is a process that employs automated tools to systematically check the system for known security weaknesses. These weaknesses include outdated software, improperly configured settings, or weak passwords.
- **Penetration testing** involves simulating real-world attacks on the system by using various tools and techniques, such as exploiting vulnerabilities, bypassing authentication, or injecting malicious code. Penetration testing can help evaluate the system's resilience and response to attacks, as well as discover unknown or hidden vulnerabilities.
- **Static application security testing (SAST)** involves analyzing the source code or binary code of the system to find security flaws, such as buffer overflows, Structured Query Language (SQL) injection, or cross-site scripting. SAST can help detect issues at an early stage of development.
- **Dynamic application security testing (DAST)** involves testing a system for security issues by sending various inputs and requests, such as Hypertext Transfer Protocol (HTTP) requests, API calls, or user actions. DAST can help verify the functionality and performance of the system under different scenarios and conditions.
- **Interactive application security testing (IAST)** involves combining both static and dynamic analysis techniques to find security issues in the system. IAST can help provide more accurate and comprehensive results by correlating data from both sources.
- **Security auditing** involves reviewing the system's policies, procedures, and documentation to ensure compliance with security standards. Security auditing can help identify gaps and weaknesses in the system's security posture and provide recommendations for improvement.

There are various tools and techniques that can be used for security testing, depending on the type and scope of testing. Some of the common tools and techniques include the following:

- **Web application scanners** are tools that scan web applications for security issues and include the Open Worldwide Application Security Project (OWASP), Zed Attack Proxy (ZAP), Network Mapper (Nmap), Burp Suite, and Acunetix. These tools can perform various tasks, such as crawling web pages, identifying web technologies, sending requests, analyzing responses, and generating reports.
- **Network scanners** scan network devices and services for security issues. Examples include Nmap, Wireshark, Metasploit, and Nessus. These tools can perform various tasks, such as discovering hosts, ports, protocols, services, vulnerabilities, and exploits.
- **Code analysis tools** analyze code for security issues. Examples include SonarQube, Veracode, Fortify, and Checkmarx. These tools can perform various tasks, such as parsing code, detecting errors, finding bugs, or suggesting fixes.
- **Fuzzing tools** test the system's robustness by sending random or malformed inputs to trigger unexpected behavior or errors. Fuzzing tools can be used to test various aspects of the system, such as APIs, protocols, file formats, or user interfaces. Some examples of fuzzing tools are Peach Fuzz, AFL, Radamsa, or JFuzz.
- **Proxy tools** intercept and modify the traffic between the client and the server to manipulate requests or responses. Proxy tools can be used to test various aspects of the system, such as authentication, authorization, encryption, or validation. Some examples of proxy tools are Burp Suite, OWASP, ZAP, Fiddler, and Charles Proxy.

Security testing can be categorized into three types: white box, black box, and grey box. White-box testing involves having access to the internal structure and code of the system. Black-box testing involves having no access to the internal structure and code of the system. Finally, gray-box testing involves having partial access to the internal structure and code of the system. Security testing can also include other techniques, such as vulnerability scanning, code analysis, fuzzing, or proxying. The main goal of security testing is to make sure that the system follows the given requirements and standards for security, and to find and fix any security issues.

Penetration Testing

Penetration testing (or “pentesting”) is a type of black-box testing that uses various tools and techniques to attack the system from an external perspective. Pentesting can help evaluate the system's resilience and response to attacks, as well as discover unknown or hidden vulnerabilities. Pentesting can also be classified into different types, such as network, web application, or mobile application penetration testing.

In some literature, the term “penetration testing” often appears alongside security testing. The difference between security testing and penetration testing is that security testing is a broad term that covers various types of testing to evaluate the security of a system, while pentesting is a specific type of security testing that simulates real-world attacks on the system to find and exploit vulnerabilities.

Security testing and pentesting have different objectives and scopes. Security testing aims to ensure that the system meets the specified requirements and standards for security, as well as to identify and fix any security issues. Pentesting, meanwhile, aims to test the system's limits and boundaries for security, as well as to demonstrate the impact and severity of any security breaches. Security testing and pentesting are both important for

improving the security and quality of the software. However, they are neither mutually exclusive nor interchangeable. They should be performed in conjunction with each other, as well as with other types of software testing, such as functional testing, performance testing, or usability testing (KiwiQA, 2022).

Vulnerability Scanning

Vulnerability scanning is a specific kind of security testing that uses automated tools to scan the system for known vulnerabilities, such as outdated software, misconfigured settings, or weak passwords. Vulnerability scanning can help find the most important and urgent issues that need to be fixed.

The main difference between security testing and vulnerability scanning is that security testing is a more complete and thorough process that covers various aspects of the system's security, while vulnerability scanning is a more focused and narrow process that only covers the system's known vulnerabilities.

Another difference between security testing and vulnerability scanning is that security testing can be performed using different approaches, such as white-box, black-box, or gray-box testing, depending on the level of access and knowledge of the system's internal structure and code. Vulnerability scanning is usually performed using a black-box approach, which means having no access or knowledge of the system's internal structure and code.

A third difference between security testing and vulnerability scanning is that security testing can include different techniques, such as simulating real-world attacks, analyzing code for security flaws, sending random or malformed inputs, or intercepting and modifying traffic. Vulnerability scanning only involves sending predefined inputs and requests to the system and analyzing the responses.

3.3 Security Code Auditing

Security code auditing is a method that examines source code or inspects a program at runtime to find security vulnerabilities, non-compliant licensing, and other programming issues. It is a vital process to ensure the quality, reliability, and security of software products. Some of the things that are checked within the auditing include the following:

- **Input validation/SQL injection** is a technique that uses an SQL query to manipulate or access data in a database. It can lead to data loss, corruption, or disclosure. Security code auditors check if the code properly validates and sanitizes user inputs before passing them to the database.
- **Third-party libraries** are external code components that developers can reuse to save time and effort. However, they can also introduce security risks if they are outdated, incompatible, or malicious. Security code auditors check if the code uses trusted and updated third-party libraries and whether they comply with the licensing requirements.

- **Encryption** is a technique that transforms data into an unreadable form to protect it from unauthorized access or modification. It is essential for sensitive data such as passwords, credit card numbers, or personal information. Security code auditors check if the code uses strong and appropriate encryption algorithms, keys, and protocols (Snyk, n.d.-a).

Some of the tools that can be used for security code auditing are listed below:

- **Static application security testing (SAST)** is a technique that parses source code to find issues such as syntax errors, coding standards violations, security vulnerabilities, or logical flaws. Some examples of SAST tools are PVS-Studio, pylint, OWASP, or ZAP (OWASP, 2021).
- **Dynamic application security testing (DAST)** is a technique that runs your application and evaluates your application security using techniques such as application penetration testing, fuzzing, or scanning. Some examples of DAST tools are Nmap, Burp Suite, OWASP, or ZAP (OWASP, 2021).
- **Software composition analysis (SCA)** is a technique that looks for vulnerabilities in open-source direct and indirect dependencies that your code uses. Some examples of SCA tools are Snyk, Dependabot, or OWASP Dependency-Check (NIST, 2023b).

The difference between SAST and DAST is that the former scans the application code at rest to discover faulty code that poses a security threat, while DAST tests the running application and has no access to its source code. SAST is a white-box testing method that can find issues early in the development cycle, while DAST is a black-box testing method that can find issues that are only visible in the production environment (Phadke, 2016).

Table 3: SAST vs. DAST – Comparison

SAST	DAST
White-box testing	Black-box testing
Examines the code	Examines an application as it's running
Finds software flaws and weaknesses	Finds vulnerabilities that an attacker could exploit
Should be performed early and often against all files containing source code	Should be performed on a running application in an environment similar to production

Source: Vladyslava Volyanska (2024).

SAST and DAST both have their own advantages and disadvantages. SAST is more effective in finding vulnerabilities early in the SDLC, while DAST is more effective in finding vulnerabilities later in the SDLC.

Figure 15: SAST vs. DAST Representation



Source: Vladyslava Volyanska (2024).

SAST and DAST should be used together to achieve a comprehensive security testing strategy (GitLab, n.d.):

- **coverage:** SAST can cover the entire code base and find issues that are not reachable by DAST, such as dead code, unused variables, or hidden backdoors. DAST can cover the entire application functionality and find issues that are not detectable by SAST, such as configuration errors, network issues, or runtime errors.
- **accuracy:** SAST can produce a lot of false positives, meaning that it can report issues that are not actually exploitable or relevant. DAST can produce a lot of false negatives, meaning that it can miss issues that may be critical.
- **speed:** SAST can be faster than DAST, as it does not require the application to be deployed or run. DAST can be slower than SAST, as it depends on the application performance and network latency.
- **cost:** SAST can be more expensive than DAST, as it requires access to the source code and specialized tools. DAST can be cheaper than SAST, as it does not require access to the source code and can use generic tools (Schmitt, 2023).

Figure 16: Software Development Phases



Source: Vladyslava Volyanska (2024).

 **SUMMARY**

The testing pyramid is a useful guide to balance your testing efforts and resources across different types of tests. By following the testing pyramid, faster feedback, lower maintenance costs, higher test coverage, and better software quality can be achieved. The testing pyramid con-

sists of three layers: unit tests, integration tests, and end-to-end (E2E) tests. Each layer has a different purpose, scope, and frequency of execution.

Unit tests are small, isolated, and fast tests that check the functionality of individual components or methods of the software. Integration tests, in the middle layer of the pyramid, are larger, more complex, and slower tests that check how different components or services of the software work together. E2E tests, at the top of the pyramid, are the largest, most expensive, and slowest tests. UI and exploratory tests are not part of the testing pyramid, but they can be seen as an extension or a complement to it.

Security testing is the process of verifying that an application or system meets the security requirements and expectations. It can help prevent unauthorized access, data breaches, or malicious attacks. Security testing can be performed at different stages of the software development life cycle, such as design, development, testing, deployment, or maintenance.

Security code auditing is a process that analyzes source code or examines a program at runtime to uncover security vulnerabilities, non-compliant licensing, and other programming issues. It can help ensure the quality, reliability, and security of software products. SAST is a white-box testing method that parses source code to find issues such as syntax errors, coding standards violations, security vulnerabilities, and logical flaws. DAST is a black-box testing method that runs applications and evaluates application security using techniques such as application penetration testing, fuzzing, or scanning. SCA looks for vulnerabilities in open-source direct and indirect dependencies that code uses. It can help manage third-party libraries and comply with the licensing requirements.

UNIT 4

SOFTWARE SUPPLY CHAIN SECURITY

STUDY GOALS

On completion of this unit, you will be able to ...

- understand the difference between package and container security.
- apply best practices and tools basing on a solid foundation of package security to own projects.
- learn the key aspects of container security.
- estimate the role of testing in the process of software development.

4. SOFTWARE SUPPLY CHAIN SECURITY

Introduction

Today, we witness numerous malicious attacks, most of which are immediately connected to exploited vulnerabilities in the software. Many security vulnerabilities have been disclosed in popular package managers.

In software development, packages and containers are related concepts that simplify the process of application development. Packages are units that contain code and its dependencies (libraries, tools, settings). Containers are units of software that comprise the code and its dependencies to make it easier to run the applications in different environments. Containers are based on common standards to make them compatible across different platforms. There are different types of containers that allow you to code within the container providing different coding environments, enabling you to use different tools and frameworks for development, testing, debugging, and deploying an application.

Containers introduce new security changes that require special solutions. Package and container security is the process that ensures that containers are safe against, for example, unauthorized access or malicious activities. It covers the entire container life cycle, from the development and distribution stages to execution and monitoring. Package and container security use security tools and policies that can protect containers at each stage, as well as best practices and standards that can enhance container security posture. By implementing package and container security, we can benefit from the advantages of containers without compromising security.

There are a lot of firms and container platforms that offer a variety of products, including Docker Enterprise, Red Hat OpenShift, Docker Swarm, Linux Containers (LXC), Container Linux, Portainer, and Apache Mesos.

4.1 Package Security

The term “software supply chain security” is directly related to the software development life cycle (SDLC) and involves activities and practices intended to secure software creation and deployment at all stages of the process. The increase of code reuse and popularization of cloud native technologies has made them more vulnerable to attacks. Even one program defect can be exploited by an attacker to fast-track them into the system, thereby allowing access to the entire supply chain. A supply chain attack is a type of cyberattack that compromises the code of the software’s delivery, threatening and targeting the entire process of software development and distribution.

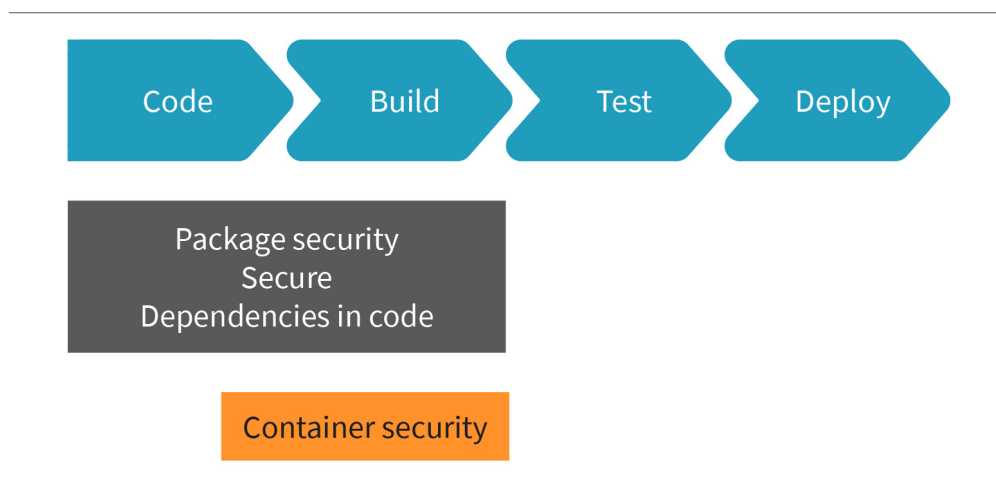
One of the most well-publicized examples of this type of attack is the SolarWinds hack (U.S. Government Accountability Office, 2021). Within this attack, a malicious code known as “Sunburst” or “Solorigate” was inserted into a software update for Orion, a network

management tool from SolarWings. This code involved modifying source code files during the building process and allowed the hackers to insert a backdoor into a legitimate file named `SolarWinds.Orion.Core.BusinessLayer.dll`. The compromised update was distributed to many SolarWings customers, allowing hackers to access the infected networks and steal the data.

Another example of a software supply chain attack is the NotPetya ransomware. The ransomware code was inserted into a software update and automatically installed. This code encrypted users' files and demanded payment for the encryption (Trellix, n.d.).

Software supply chain attacks are difficult to detect and prevent and undermine the users' trust in software. Therefore, it is very important to follow best practices for securing not only the software itself but also the software development process, including distribution with secure code reviewing, encryption, digital signing, patching, and backup verifications.

Figure 17: Software Supply Chain Security



Source: Petra Beenken (2024).

In the field of software development, package security, container security, and continuous integration/continuous deployment (CI/CD) are three critical areas that intersect to ensure the overall security and efficiency of the development process:

1. **Package security** involves ensuring the integrity of software packages that are used in the development process. It includes checking for vulnerabilities in the packages, updating packages to their latest secure versions, and managing package dependencies effectively.
2. **Container security** focuses on securing the application environment. It involves scanning containers for vulnerabilities, configuring them properly to minimize attack surfaces, and continuously monitoring them for any suspicious activities.
3. **CI/CD** is a development practice that involves automating the integration and deployment of code. It allows for code to be integrated, tested, and deployed more frequently, which helps catch and fix issues early. In terms of security, CI/CD pipelines

can be configured to include security checks at various stages, such as static code analysis, dynamic testing, and automatic scanning for vulnerabilities in the code or containers.

When these three areas meet, we have a robust system where code is developed and deployed in a secure manner. Packages used in the code are checked for vulnerabilities, the containers running the application are secured, and the entire process is automated and includes security checks at various stages. This results in a secure, efficient, and reliable software development life cycle (SDLC).

Package Security

Package
a collection of related classes, interfaces, and other components that are grouped and work together to achieve a specific functionality or goal
(Law Insider, n.d.)

Package security is an essential aspect of software development, especially for open source and cloud computing. Packages are reusable code units that can be imported and used by other projects. They provide functionality, convenience, and efficiency, but at the same time, they also pose potential risks, such as introducing vulnerabilities, compromising privacy, or breaking compatibility. For this reason, it is important for developers to understand how to secure their packages and dependencies, as well as how to use packages safely and responsibly.

Package security is the practice of ensuring that the packages or dependencies that are used in software projects are free from known vulnerabilities and have proper authentication. A vulnerability is a weakness or flaw in a package that can be exploited by an attacker to cause harm, such as data loss, service outage, unauthorized access, or code execution. Authentication is the process of verifying the identity and origin of a package, ensuring that it has not been tampered with or replaced by a malicious one.

Package security matters because packages are widely used and trusted by developers, but they can also introduce risks and challenges. According to a report by Snyk, a security platform for developers, 80 percent of the code in a typical application comes from packages, and 85 percent of the packages have at least one vulnerability (Snyk, n.d.-c). Moreover, packages often depend on other packages, creating complex dependency trees that can be hard to manage and update. A single vulnerable or compromised package can affect multiple projects and users, potentially causing widespread damage and reputation loss.

One of the best practices for package security is to audit package dependencies for security issues and fix them as soon as possible. A security audit serves as a rigorous evaluation of package dependencies, specifically aimed at identifying potential security vulnerabilities. These vulnerabilities represent potential points of exploitation for malicious entities, posing a significant risk. By conducting security audits, we can proactively safeguard users of the package by detecting and rectifying known vulnerabilities in dependencies. This preemptive measure is crucial in preventing detrimental outcomes such as data breaches, service disruptions, unauthorized access, and other potential threats.

There are different tools and platforms that can help to perform security audits for package dependencies:

- **npm audit** scans npm package dependencies for security issues and provides recommendations on how to fix them (npm, 2023).
- **dotnet list package --vulnerable** lists any known vulnerabilities in NuGet package dependencies within the projects and solutions (Microsoft, 2023).
- **Snyk** monitors the package dependencies for security issues and provides fixes and alerts (Snyk, n.d.-b).

Another best practice for package security is to use authentication and encryption to verify the integrity and origin of packages. Authentication is the process of verifying the identity and origin of a package, ensuring that it has not been tampered with or replaced by a malicious one. Encryption is the process of transforming the data in a package into an unreadable form, preventing unauthorized access or modification.

There are different methods and tools that can help to use authentication and encryption in packages, including those listed below.

NuGet.config files and V2 plug-in credential providers

These files store and provide credentials for consuming packages from authenticated feeds. NuGet.config files and V2 plug-in credential providers are tools that help mitigate the threat of unauthorized access to package feeds by providing a secure way to store and provide credentials for consuming packages from authenticated feeds.

When NuGet needs to authenticate with a feed, it first looks for credentials in these config files. This allows you to securely store your credentials and use them when needed. V2 plug-in credential providers are plugins that can communicate with NuGet to retrieve credentials. When NuGet needs credentials to authenticate with a feed, it can use these V2 plug-in credential providers. This provides an additional layer of security as the plugins can be designed to retrieve credentials in a secure manner. By using these tools, we can ensure that package feeds are only accessible to authorized users, thereby mitigating the threat of unauthorized access. It's important to note that these tools should be used in conjunction with other security practices for the best results (Microsoft, 2021).

Public-key encryption and Virtru

Public-key encryption, also known as “asymmetric encryption,” is a method that is widely used in many applications, including email encryption, to ensure the confidentiality and integrity of data. Virtru, a data protection platform, utilizes this method of encryption to secure data. When we encrypt an email or a file using Virtru, a unique message key is generated. This message key is then encrypted with a public key. The corresponding private key needed to decrypt the message key is securely stored and managed by Virtru. In addition to this, Virtru offers several key management solutions to ensure control, confidentiality, and compliance. One of these solutions is the Virtru Customer Key Server (CKS), which allows users to host their own encryption keys. This adds an additional layer of protection and control over data. Furthermore, Virtru uses authentication mechanisms to protect against unauthorized access. For instance, the Virtru Access Control Manager facilitates and authenticates key exchanges but cannot access data at any time. There are sev-

eral tools that offer similar functionality to Virtru; for example, Proton Mail, Swisscows, Lockbin, Skizzle Email , Yambuu Mail, Paubox Egnyte, Avanan Cloud and Email Security (Virtru, n.d.).

Advanced Encryption Standard (AES) encryption and Apache Commons Crypto library

These tools support encryption for remote procedure call (RPC) connections between packages and Spark clusters. Advanced Encryption Standard (AES) is a widely used symmetric encryption algorithm. It's known for its speed and security, and it is used in many different applications, from securing web traffic to encrypting sensitive data. The Apache Commons Crypto library is a cryptographic library optimized with AES-NI (Advanced Encryption Standard New Instructions). It provides application programming interfaces (APIs) at both the cipher and stream levels. Developers can use it to implement high-performance AES encryption/decryption with minimum coding and effort. When it comes to securing RPC connections between packages and Spark clusters, AES encryption and the Apache Commons Crypto library can be used to encrypt the data being transmitted. This ensures that even if the data are intercepted during transmission, they cannot be read without the correct decryption key (Apache Commons, 2023). Below is an example of the Apache Commons Crypto library being used to implement AES encryption:

```
import org.apache.commons.crypto.cipher.CryptoCipher;
import org.apache.commons.crypto.utils.Utils;

import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.SecretKeySpec;
import java.nio.charset.StandardCharsets;
import java.security.SecureRandom;
import java.util.Properties;

public class AesExample {
    public static void main(String[] args) throws Exception {
        final SecretKeySpec key = new SecretKeySpec(getUTF8Bytes
("1234567890123456"),"AES");
        final IvParameterSpec iv = new IvParameterSpec(getUTF8Bytes
("1234567890123456"));
        Properties properties = new Properties();
        properties.setProperty("org.apache.commons.crypto.cipher
.aes.enabled", "true");
        final String transform = "AES/CBC/PKCS5Padding";
        final CryptoCipher encipher = Utils.getCipherInstance
(transform, properties);

        byte[] input = getUTF8Bytes("hello world!");
        byte[] output = new byte[32];

        // Initializes the cipher with ENCRYPT_MODE, key and iv.
        encipher.init(Cipher.ENCRYPT_MODE, key, iv);
        // Continues a multiple-part encryption/decryption operation
```

```

for byte array.
    encipher.update(input, 0, input.length, output, 0);
}

private static byte[] getUTF8Bytes(String input) {
    return input.getBytes(StandardCharsets.UTF_8);
}
}

```

A further best practice for package security is to apply access control and authorization mechanisms to packages and methods. Access control and authorization essentially refer to the same concept: They dictate the procedures and decisions involved in identifying, recording, and managing the entities (such as users, devices, or processes) that are permitted access, as well as the targets (like packages, methods, or data) that they can access. Put simply, they establish the rules of what is permissible and what is not.

There are different models and methods that can help to apply access control and authorization to packages and methods:

- **Mandatory access control (MAC)** allows only the owner and the custodian to manage the access controls and uses security models such as Biba and Bell-LaPadula to enforce confidentiality and integrity (Hoffman, n.d.).
- **Spring Security** provides a framework for authentication and authorization in Java applications and supports various mechanisms such as roles, permissions, expressions, filters, and annotations (Spring, n.d.).
- **The Open Worldwide Application Security Project (OWASP) Access Control Cheat-sheet** provides a comprehensive guide on how to design, implement, and test effective access control for web applications (OWASP, n.d.-a).

Package security is a critical aspect of software development. The best practices to ensure the security of packages include the following:

- **Audit dependencies:** You should regularly audit your dependencies using a package manager to identify and fix any vulnerabilities.
- **Avoid publishing secrets:** Be careful not to publish secrets such as API keys or passwords to the package registry.
- **Enforce LockFile:** Use package lock files for deterministic installations across different environments and enforce dependency expectations across team collaboration.
- **Use a JavaScript linter:** A linter can help catch common mistakes in your code before they become security vulnerabilities.
- **Add Subresource Integrity (SRI) checking:** SRI checking can help ensure that the resources your application loads haven't been tampered with.
- **Validate user input:** Always validate user input to prevent attacks such as a Structured Query Language (SQL) injection or cross-site scripting.
- **Escape or encode user input:** Escaping or encoding user input can prevent injection attacks.
- **Use a cross-site request forgery (CSRF) token:** CSRF tokens can help prevent cross-site request forgery attacks.

Another best practice for package security is to test and monitor packages for security performance and compliance. Security testing is the process of identifying and resolving vulnerabilities, flaws, or weaknesses in the packages that could compromise their functionality, reliability, or security. Security monitoring is the process of collecting and analyzing data from packages to detect and respond to security incidents, threats, or anomalies. Compliance is the state of adhering to the regulatory requirements and industry standards that apply to packages, such as the General Data Protection Regulation (GDPR), Payment Card Industry Data Security Standard (PCI DSS), and the Health Insurance Portability and Accountability Act (HIPAA).

There are different tools and techniques that can help to test and monitor packages for security performance and compliance, as detailed below.

GitHub Actions

GitHub Actions is a tool that allows for the automation of workflows for security testing and monitoring in GitHub repositories. This includes tasks such as vulnerability scanning, generating audit logs, and enforcing policies. Even if the CI/CD pipeline is managed by another tool, GitHub Actions can automate several common security and compliance tasks, for example, the auditing of repository access (Holleran, 2021). The `org-audit-action` can be set up to produce a `.csv` and a `.json` file, which detail for each repository the users that have access, the level of permission those users have, the users' login information, and full names, as well as the (optional) Security Assertion Markup Language (SAML) identity of the user. In terms of enforcing security policies, GitHub Actions offers several useful security features by default.

Dependabot alerts notify repository owners of vulnerabilities in their open-source dependencies and automatically open pull requests to update them. The dependency graph contains license information for open-source packages. Moreover, GitHub code scanning will alert users when they have written insecure code. When it comes to strengthening security, best practices for using GitHub Actions include using secrets for sensitive values, avoiding the use of structured data as a secret, registering all secrets used within workflows, and auditing how secrets are handled. Lastly, it's recommended to use minimally scoped credentials. Ensure that the `GITHUB_TOKEN` is configured with the least privileges necessary to run jobs (GitHub, n.d.).

Bright Security

Bright Security is a developer-centric dynamic application security testing (DAST) solution. It provides a platform for security testing and monitoring in cloud environments, such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP). It allows for scanning to start during the unit testing phase and supports various tools and frameworks, such as OWASP and Zed Attack Proxy (ZAP), Network Mapper (Nmap), Selenium, in conducting two separate validations to ensure accurate findings every time, reducing the noise from false positives. Each vulnerability found by Bright Security includes remediation guidelines and resources.

Bright Security can be integrated into a CI/CD pipeline, allowing it to scan every pull request and build or merge via a command-line user interface (CUI). It provides a platform for security testing and monitoring in cloud environments, such as AWS, Azure, and GCP, as well as supporting various tools and frameworks, such as OWASP ZAP, Nmap, and Selenium (Moradov, 2022).

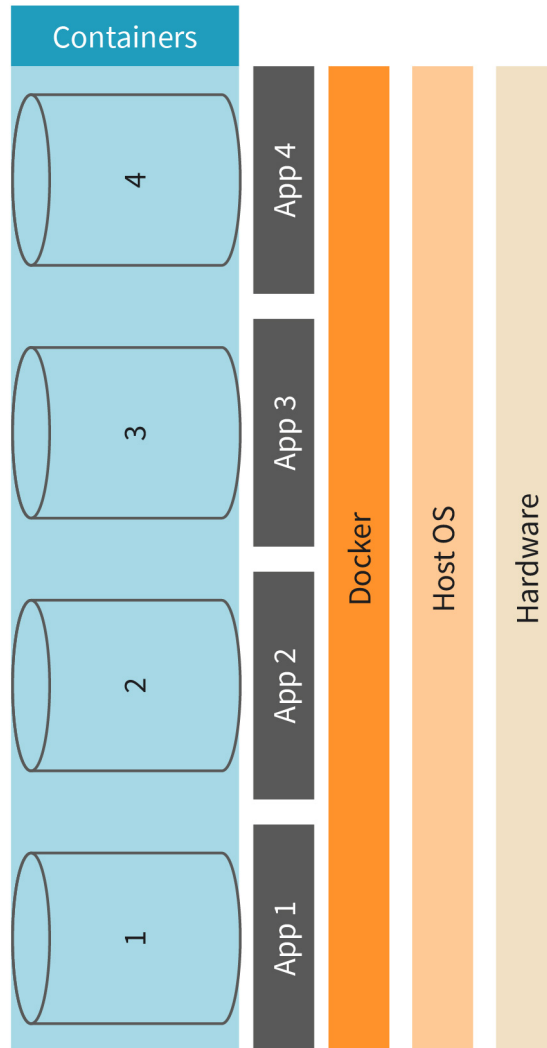
UpGuard

UpGuard offers a solution for security metrics and compliance management, allowing for the measurement and improvement of security posture, risk level, and compliance status across packages and systems. UpGuard is an integrated risk platform that combines third-party security assessments, security assessment questionnaires, and threat intelligence features to provide a complete and comprehensive view of the risk surface. It helps monitor attack surface, prevent data breaches, discover leaked credentials, and protect customer data. The operation of UpGuard is comprehensive: It continuously monitors over a million businesses to proactively identify security exposures, using its proprietary security ratings to achieve this. In addition, UpGuard offers globally leading data leak detection capabilities throughout a company's supply chain (Tunggal, 2023).

4.2 Container Security

Containerization is a technology that allows applications to run in isolated environments, called "containers," on a shared operating system.

Figure 18: Containerization



Source: Vladyslava Volyanska (2024), based on Docker (n.d.).

Container is a package of software and its dependencies, such as code, system tools, settings, and libraries, that can run reliably on any operating system and infrastructure (Puzas, 2023b).

Containers are lightweight, portable, and scalable, which makes them ideal for deploying applications in cloud environments. However, containerization also introduces new security challenges that require careful consideration and mitigation.

Container images are the files that contain the application code and dependencies that are used to create and run containers. Container registries are the repositories that store and distribute container images. Securing the container images and registries is essential to ensure the integrity and confidentiality of the applications and prevent unauthorized access or tampering. Container images and registries face several threats, including unauthorized access, image tampering, outdated or vulnerable components, and insecure communication. Unauthorized access to container images and registries can lead to a vari-

ety of security issues, including data theft, tampering, and even the execution of malicious code. This can occur if access controls are not properly implemented or if credentials are compromised.

Image Tampering

If an attacker gains write access to a container registry, they could potentially put malicious code into an image and replace the original inside the registry (De Oliveira & Fiser, 2023). This can lead to the execution of malicious code when the tampered image is run.

Outdated or Vulnerable Components

Container images consist of multiple software layers, each of which might have vulnerabilities. Developers often upload new containers (or new versions of containers) to the registry, which can introduce new vulnerabilities. Additionally, some open-source images might contain malicious commands that can cause vulnerabilities when deployed (De Oliveira & Fiser, 2023).

Insecure Communication

The communication between registries and container runtime is another potential point of vulnerability. If this communication is not properly secured, it could be intercepted by an attacker, who could then gain access to sensitive information or even alter the contents of the images being pulled from the registry.

Some of the main steps to secure the container images and registries are as follows:

- **Use trusted sources and verified images:** Use only container images from reputable sources and verify their signatures and checksums before downloading or running them. Avoid using images from unknown or untrusted sources or with outdated or vulnerable components.
- **Scan and patch the images regularly:** Use tools such as Docker Scan, Clair, or Anchore to scan the container images for vulnerabilities and apply patches or updates as soon as possible. Keep track of the image versions and dependencies and remove any unused or obsolete images.
- **Encrypt and sign the images:** Use encryption tools such as Docker Content Trust, Notary, or Cosign to encrypt and sign the container images and ensure their authenticity and integrity. Only allow encrypted and signed images to be pulled or pushed from the registries.
- **Secure access to the registries:** Use authentication and authorization mechanisms such as tokens, certificates, or role-based access control (RBAC) to restrict access to the container registries. Only allow authorized users or services to view, pull, push, or delete images from the registries. Use encryption tools such as Transport Layer Security (TLS) or Hypertext Transfer Protocol Secure (HTTPS) to protect communication between the registries and clients.

- **Monitor and audit the registries:** Use tools such as Docker Registry API, Harbor, or Quay to monitor and audit the activities and events in the container registries. Detect and report any suspicious or anomalous behavior such as unauthorized access, image tampering, or deletion (Snyk, n.d.-a).

Container runtime is the software that manages the creation and execution of containers on a host machine. The container host is the operating system that runs the container runtime and provides the resources for the containers. Hardening the container runtime and host is important to prevent unauthorized access to the containers and the host.

Securing the container runtime and host involves several key steps:

1. **Use a minimal and secure host operating system (OS):** The host OS plays a crucial role in container security. A minimal OS designed specifically for running containers, such as CoreOS, RancherOS, or Ubuntu Core, can reduce the attack surface and improve security. These OS are lightweight, contain only the necessary packages and services, and come with built-in security features. If you choose to use a general-purpose OS, ensure it is updated and patched regularly.
2. **Configure the container runtime securely:** The container runtime should be configured to limit the privileges and resources of the containers and isolate them from each other and from the host. This can be achieved by leveraging security features such as namespaces (which provide isolation), cgroups (which limit resource usage), and security modules like seccomp, SELinux, AppArmor, or Capabilities (which restrict actions available to processes). Regular checks should be performed to ensure the security configuration of the container runtime is robust.
3. **Use secure container orchestration tools:** Container orchestration tools like Kubernetes, Docker Swarm, or Mesos help manage and deploy containers across multiple hosts. These tools should be configured to use encryption, authentication, authorization, and logging mechanisms to secure communication and access between the hosts and the containers. Monitoring tools can be used to detect any abnormal or malicious activity in the container clusters (Souppaya et al., 2017).

Container network and storage are the components that enable the communication and data exchange between the containers and the external systems. Container network and storage security are essential to protect the confidentiality, integrity, and availability of the data and prevent unauthorized access or interception.

Some of the main steps to implement network and storage security for containers are as follows:

- **Use secure network protocols and encryption:** Securing network protocols and encryption is a critical aspect of container security. The intention is to ensure that data transmitted between containers and external systems is protected from unauthorized access, tampering, and eavesdropping. To achieve this, the network protocols that support encryption, authentication, and authorization can be used. These include TLS, HTTPS, Secure Shell (SSH), or a virtual private network (VPN). These protocols provide a secure communication channel by encrypting the data in transit. In addition to using secure protocols, the network traffic between the containers and the external systems

can also be encrypted. This can be achieved by using network security solutions that provide features such as automatic encryption, identity-based traffic control, and intrusion detection. Finally, managing the certificates and keys for encryption is also crucial. This involves generating, distributing, rotating, and revoking certificates and keys as needed. It's important to use a secure and reliable system for certificate management to ensure the integrity of the encryption process.

- **Use network policies and firewalls:** The intention of using network policies and firewalls is to control the network traffic between the containers and the external systems, thereby enhancing the security of your container environment. Network policies and firewalls allow you to define and enforce rules for network traffic. For instance, you can specify which ports, protocols, and Internet Protocol (IP) addresses are allowed to communicate with the containers. This helps to prevent unauthorized access and reduces the attack surface. To implement these measures, tools that provide network policy and firewall capabilities are used. For example, Kubernetes Network Policy and Docker Network allow you to define fine-grained network policies for containerized applications. Similarly, iptables is a powerful tool for setting up firewalls and managing network traffic on Linux systems.
- **Use secure storage options and encryption:** Use storage options that support encryption, access control, and backup, such as encrypted volumes, persistent volumes, or cloud storage services. Encrypt the data at rest and in transit using tools such as dm-crypt, Linux Unified Key Setup (LUKS), or AWS Key Management Service (KMS). Use tools such as Kubernetes Storage Class, Docker Volume, or channel state information (CSI) to manage the storage options for containers.
- **Container monitoring and auditing:** These are the processes of collecting and analyzing the data and logs related to the performance, behavior, and security of the containers and host. The intention of container monitoring and auditing is to maintain the health, performance, and security of the container environment. This involves collecting and analyzing data and logs related to the performance, behavior, and security of the containers and the host. Monitoring allows you to track key metrics such as central processing unit (CPU) usage, memory consumption, network traffic, and disk I/O (input/output), which can help to understand the performance of containers and identify any potential issues. It can also help ensure that containers are running efficiently and that resources are being used effectively. Auditing, on the other hand, focuses on recording and reviewing activities within the container environment. This includes tracking user actions, changes to configurations, and security events. Auditing is crucial for detecting anomalies or threats that could indicate a security breach or compliance violation.

Monitoring and auditing container activities and vulnerabilities are crucial aspects of maintaining a secure containerized environment. Container-native monitoring and logging provide visibility into the performance and behavior of containers. They allow you to collect metrics (e.g., CPU usage, memory usage, disk I/O, network traffic) and logs (processes, events, errors, alerts) from containers and hosts. These data can help to identify performance bottlenecks, troubleshoot issues, and detect anomalies or security threats. Container security scanning and auditing are critical for identifying vulnerabilities, misconfigurations, or compliance issues in containers and hosts. These tools can help identify risks and deviations from best practices or standards for container security. Regularly scanning containers and hosts can help catch potential security issues early before they can be exploited. Behavioral analysis and anomaly detection tools can help detect abnor-

mal or malicious activity in containers and hosts. These tools monitor actions or changes in containers and hosts (e.g., file modifications, network connections, process executions, system calls) that could indicate a security threat. By detecting these anomalies, we can respond to potential security incidents more quickly.

Security tools and frameworks are the software and systems that provide security features and functions for containers and the host. Security tools and frameworks are important to enhance the security posture and capabilities of the containers and the host.

Containerized environments face different potential threats that need to be addressed to ensure security. For example, container images can contain vulnerabilities that can be exploited by attackers. These vulnerabilities can be present in the image's code or its dependencies. An attacker can exploit these vulnerabilities to execute malicious code, gain unauthorized access, or cause a denial of service. Misconfigurations can lead to a variety of security issues, including unauthorized access, data leakage, and loss of control over the containers. This can occur if security controls are not properly configured or if default settings are not changed. Container images can contain embedded malware, which can be activated when the container is run. This malware can then perform malicious activities, such as data theft, system disruption, or further propagation within the network. Sensitive information, such as passwords or API keys, can sometimes be embedded in container images in clear text. If an attacker gains access to these secrets, they can potentially gain unauthorized access to systems or data. Neglecting basic authentication and authorization for orchestrators such as Kubernetes and Docker could potentially give attackers unrestricted access to container deployment. This is evident from recent attacks on companies like Tesla, Shopify, and Aviva, which were based on unsecured Kubernetes consoles. Using container images that have vulnerabilities can lead to a host of security problems, ranging from the installation of malware to the creation of kernel panics and access to the host kernel (Gamage, 2019).

Securing a containerized environment involves addressing several potential threats and implementing appropriate measures, such as use of container security platforms and solutions. Comprehensive security platforms and solutions (e.g., Aqua, Prisma Cloud, or NeuVector) offer integrated security features for containers and the host. These platforms provide security across the entire container life cycle, from development to deployment to runtime, and across the container stack, from the host OS to the container runtime to the container image to the container orchestration. Additionally, container security standards and best practices like the Center for Internet Security (CIS) Benchmarks, National Institute of Standards and Technology (NIST) SP 800-190, or OWASP Top 10 define security requirements and guidelines for containers. The host can help maintain a consistent and effective security level for containers and the host and ensure compliance with regulatory and industry standards for container security. The usage of container security testing and verification tools, like Inspec, Gauntlt, or Behavior-Driven Development (BDD) Security automate and validate security tests for containers and the host. They can be integrated with development and deployment pipelines to ensure continuous security testing and verification.

4.3 Programming Language Consideration

The choice of development languages is a complex process. It is difficult to give a universal formula for selecting the best/most suitable programming language, as it is always subjective.

The choice of programming language should result from

- the needs and plans of the project,
- technologies used,
- the current and future usability of the programming language,
- the availability of the libraries and frameworks,
- the syntax complexity of the language, and
- the versatility of the language.

It is also possible to consider programming languages that are popular, which is an obvious element of the technology stack in most software houses.

Important to consider in programming language is the decision between languages from two groups according to the type of translation: compilation or interpretation. The modules provided in a compiled language are executed directly by the processor, which allows you to use its computing power to execute algorithms. Programs written in an interpreted language are easily adapted to the needs of specific users, without code recompilation.

To ensure the efficiency of programs, we should use a compiled language. The fragments that the user can modify should be written in the interpreted language. We can also use both languages at the same time, enabling us to reconcile both approaches. The code that is not translated to machine code will always run slower than binary code. The code that is not provided as source code will always be less readable than source code. A popular approach is to use several programming languages (Binstock, 2013).

We should also pay attention to when and how high-level code is translated into machine code (compilation/interpretation) but also whether the code relies on a runtime environment to provide certain services during execution (managed/unmanaged code).

Managed code is a type of computer program code that will only execute under the management of a Common Language Runtime (CLR) virtual machine, resulting in benefits such as automatic memory management and garbage collection, type safety, and platform independence (LiquiSearch, n.d.). Examples of languages that produce managed code include C# and Java.

Unmanaged code, meanwhile, is directly executed by the operating system. It is compiled into machine code and runs directly on the computer's hardware, making it more efficient than managed code. However, it lacks the benefits provided by a runtime environment, such as automatic memory management and garbage collection. An example of a lan-

language that produces unmanaged code is C++. In summary, while managed code can increase productivity and safety due to its automatic memory management and type safety features, unmanaged code can provide better performance and efficiency.

Therefore, most common programming languages can be divided into two groups.

Table 4: Choice of Programming Language

Managed code	Unmanaged code
<ul style="list-style-type: none"> • C#.NET, Java, Python, JavaScript, Ruby, Visual Basic (.NET) • Advantages: type safety, automated garbage collection, memory management, platform independence. C# and Visual Basic (.NET) are not fully platform-independent. • Disadvantages: may be less performant than unmanaged code 	<ul style="list-style-type: none"> • C, C++, Swift, Go, Rust • Advantages: typically offers higher performance • Disadvantages: can lead to memory leaks, buffer overruns, and pointer overrides

Source: Vladyslava Volyanska (2024).

It's important to note that the choice of programming language should depend on the specific requirements and goals of the project. For instance, if performance is a critical factor, then unmanaged code like C++ might be more suitable. However, if you're developing a large-scale application where memory management and type safety are more important, then managed code like Java or C# could be a better choice. Rust is gaining popularity as a system programming language that combines the performance of C and C++ with additional safety features to prevent common programming errors like null pointer dereferencing and buffer overflow.

SUMMARY

Software supply chain security is the process of making sure that software components are safe and reliable from their creation to use. It involves protecting software from harmful attacks, unauthorized changes, weaknesses, and errors that could affect their function, performance, or safety. Software supply chain security covers different aspects of software creation, delivery, and use.

The secure development life cycle (SDLC) focuses on integrating security into the development process. It is a set of practices that aims to include security in every step of software creation, from requirements to testing. It helps to find and reduce security risks, improve code quality, lower costs, and increase customer satisfaction.

Software supply chain security is a complicated and ever-changing challenge that requires cooperation and coordination among various stakeholders, such as software developers, vendors, customers, and regula-

tors. Software supply chain security also requires constant monitoring and improvement to keep up with the changing threats and technologies. Software supply chain security is important for building trust and confidence in the software ecosystem.

UNIT 5

COMMON CODING ANTI-PRACTICES

STUDY GOALS

On completion of this unit, you will be able to ...

- understand common coding anti-practices.
- describe different types of vulnerabilities and bugs.
- analyze the sources and severity of bugs.

5. COMMON CODING ANTI-PRACTICES

Introduction

In today's rapidly evolving digital landscape, coding has emerged as an indispensable skill for a diverse array of professionals. However, it's not without its fair share of challenges. Missteps in coding, often referred to as "coding anti-practices," can lead to a multitude of issues, including poor-quality code, inefficiency, security vulnerabilities, and frustration for both developers and users.

This unit delves into these problematic practices that are unfortunately all too common in the coding world. We will explore what these anti-practices are, exploring their nature, why they pose problems, and how they can be avoided. By gaining a deeper understanding of these anti-practices, we can learn to navigate the intricate landscape of coding more effectively. This knowledge empowers us to produce code that is not just efficient and secure but also enjoyable to create and use.

By understanding these anti-practices, we can learn to navigate the complex landscape of coding more effectively. This knowledge allows to produce code that is not only efficient and secure but also enjoyable to create and use. Furthermore, by shedding light on these anti-practices, we hope to foster a coding environment that encourages best practices, promotes security, and enhances the overall quality of code. This, in turn, can lead to more robust applications, satisfied users, and a more fulfilling coding experience.

5.1 Classes of Bugs

The case study below is the best way to demonstrate how applying common coding anti-practices can lead to a serious problem. The problem described here isn't a real-world example, but rather a possible case based on information from [isocpp/CppCoreGuidelines](#) (GitHub, n.d.; ISO C++ Standards Committee, n.d.) and [secure C++ coding practices](#) (Stack Overflow, n.d.).

Case Study: Improving the Memory Management of a C++ Game Engine

Background

The game engine is a custom-built software system that uses C++ as the main programming language. It is created for different platforms, including Windows, Linux, Android, iOS. It has a modular architecture that allows developers to create and modify different components, such as rendering, physics, audio, input, and networking. The game engine also uses a scene-based approach, where each scene represents a logical unit of the game world, such as a level, menu, or cutscene.

Problem

The game engine has serious problems: frequent memory leaks, fragmentation, and allocation failures. Several sources were identified, including

- improper use of dynamic memory allocation and deallocation, such as using new and delete operators without proper error handling, memory alignment or ownership management. This can lead to several issues such as memory leaks, dangling pointers, and segmentation faults. Memory leaks happen when memory is allocated but not properly deallocated, which leads to a gradual increase in the memory footprint of the application. Dangling pointers refer to pointers that point to memory that has been deallocated or unallocated. Using such pointers can lead to undefined behavior or crashes. Segmentation faults occur when an application tries to access memory that it doesn't have permission to access.
- lack of consistent and clear coding conventions and guidelines for memory management, such as when to use smart pointers, containers, allocators, or custom memory pools. This can lead to inconsistent code that is hard to read, understand, and maintain. It can also lead to bugs due to misunderstandings about how memory should be managed in different parts of the codebase. For example, if it's not clear when to use smart pointers versus raw pointers, developers might use the wrong type of pointer in a given context, leading to issues like memory leaks or dangling pointers.
- inefficient and redundant memory usage, such as allocating more memory than needed, copying large objects unnecessarily or not releasing unused memory resources. This can lead to an increased memory footprint, which can be a problem in memory-constrained environments. It can also slow down the application due to unnecessary memory allocations and deallocations. In extreme cases, it could even lead to out-of-memory errors.
- insufficient or outdated tools and techniques for memory profiling, debugging, and optimization, such as using standard library functions, third-party libraries, or platform-specific tools that do not provide enough information or control over the memory behavior of the game engine. Without proper tools and techniques, it can be very difficult to identify and fix memory-related issues. This can lead to increased time spent debugging and optimizing the application, and it can also increase the risk of shipping code with memory-related bugs or performance issues.

Solution

It was decided to implement a comprehensive solution to improve the memory management:

- We reviewed and refactored the existing code base to eliminate or reduce the sources of memory problems (replacing raw pointers with smart pointers, using containers and allocators from the standard library, avoiding unnecessary copies or allocations, and releasing unused memory resources).

- We established a set of coding best practices for memory management, documenting the memory requirements and behavior of each component, module, and scene, using consistent naming conventions and coding styles for memory-related variables and functions, and applying design patterns and principles that promote low coupling and high cohesion among the components and modules of the game engine.
- We adopted and integrated new tools and techniques for memory profiling, debugging, and optimization, using specialized libraries or frameworks that provide more detailed and accurate information about the memory usage, leaks, fragmentation, and allocation failures, such as Valgrind, AddressSanitizer, Visual Studio Memory Profiler, or Unreal Engine Memory Profiler.
- We tested and evaluated the results of the solution by measuring the performance, stability, and user experience of the games before and after applying the solution, using various metrics and indicators (memory consumption, allocation time, frame rate, loading time, crash rate, or user feedback).

Results

Significant improvements in the memory management of the game engine were achieved:

- The frequency and severity of memory leaks, fragmentation, and allocation failures decreased by more than 80 percent, which reduced the number of crashes, bugs, and glitches in the games.
- The average memory consumption decreased by more than 30 percent, which increased the performance and responsiveness of the games.
- The average allocation time decreased by more than 50 percent, which reduced the loading time and latency of the games.

This is just a theoretical imitation of a possible situation, but it provides us with the possibility to track and understand some examples of common coding anti-practices and their consequences, like improper use of dynamic memory allocation, lack of consistent and clear coding convention and guidelines for memory management, inefficient and redundant memory usage, and insufficient tools, as well as techniques for memory profiling, debugging, and optimization.

Common Vulnerabilities and Exposures

Vulnerabilities are deficiencies or weak points in a computer-based system that could potentially be taken advantage of, leading to damage or loss (Sommerville, 2016, p. 377).

The Common **Vulnerabilities** and Exposures (CVE) system is maintained by the United States National Cybersecurity Federally Funded Research and Development Center (FFRDC), which is operated by the MITRE Corporation (MITRE, 2023c). The system was officially launched for the public in September 1999. CVE identifiers are assigned by a CVE Numbering Authority (CNA); various CNAs assign CVE numbers for their own products (e.g., Microsoft, Oracle, HP, Red Hat). A third-party coordinator (such as a computer emergency response team [CERT] Coordination Center) may assign CVE numbers for products that are not covered by other CNAs (MITRE, 2023d). The goal of CVE is dissemination of information about the latest vulnerabilities and their severity, as well as patches or mitigations.

There are several tools known as “CVE-compatible,” such as the widely recognized Common Vulnerability Scoring System (CVSS), which adds a score to each vulnerability based on factors like attack vectors, complexity, privileges request, scope, integrity, and availability. CVSS and CVE both utilize IDs to pinpoint specific vulnerabilities. In general, four compatible and mutually complementary standards are used to describe a vulnerability; along with the above-mentioned CVE and CVSS, these are the Common Weakness Enumeration (CWE) and the Common Platform Enumeration (CPE; FIRST, n.d.; MITRE, 2013). CWE provides a classification and identification based on factors such as abstraction level, class, consequences, recommendation, and attack frequency. Meanwhile, CPE provides a standardized method for identifying and documenting software applications, operating systems, and hardware components.

Table 5: The Main Standards for Description and Assessment of Vulnerability

Standard	Description
CVE (Common Vulnerabilities and Exposures)	It assigns a unique ID to each disclosed vulnerability.
CVSS (Common Vulnerability Scoring System)	It adds a score to each vulnerability based on factors like attack vectors, complexity, privileges required, scope, integrity, and availability.
CWE (Common Weakness Enumeration)	It provides a classification and identification based on factors such as abstraction level, class, consequences, recommendation, and attack frequency.
CPE (Common Platform Enumeration)	It provides a standardized method for identifying and documenting software applications, operating systems, and hardware components.

Source: Vladyslava Volyanska (2024).

For example, let’s consider a hypothetical vulnerability in a software application to illustrate how these four standards work together:

1. **CVE:** When a new vulnerability is discovered in the software, the MITRE Corporation assigns it a unique ID; for example, CVE-2023-1234.
2. **CVSS:** It evaluates this vulnerability based on various factors like attack vectors, complexity, privileges required, scope, integrity, and availability. It might determine that the vulnerability has a score of 7.5 out of 10, indicating that it’s a high-severity issue.
3. **CWE:** It provides a classification for vulnerability based on factors like abstraction level, class, consequences, recommendation, and attack frequency. For instance, it might classify this vulnerability as “CWE-89: Improper Neutralization of Special Elements used in an SQL Command (‘SQL Injection’)” (Feiler, 2011).
4. **CPE:** It identifies and documents that this vulnerability exists in a specific version of the software application on a particular operating system or hardware component. For example, it might specify that the vulnerability is present in “Software Application v1.0” running on “Operating System v2.0.”

In this way, these four standards provide a comprehensive description and assessment of the vulnerability from its identification to its classification and scoring, as well as specifying where exactly it exists.

There are also many other, more-specialized, and product-oriented vulnerabilities and exposures databases, catalog systems, and sites that contain information about the security gaps in concrete product (for example, in an application, programming language, or operating system). For example, we can find information about vulnerabilities in C++ on Stack Overflow or from GitHub Project (ZeoVan, n.d.). There are many companies that provide information about vulnerabilities, such as Microsoft, Google, and Apple. Each of these companies has a dedicated security page where they publish details about known vulnerabilities in their products.

Bad Coding Practices and Anti-Practices

There are two concepts used to describe bad coding practices: a bad coding practice and an anti-practice. A bad coding practice is a more general term that can refer to any undesirable or unprofessional practice, while an anti-practice is an action that is already known and considered to be ineffective or harmful.

Examples of poor coding practices include hard-coding values and credentials, premature optimization, magic numbers, God objects, copying and pasting code without understanding or testing it, using inconsistent or unclear naming conventions, writing overly complex or nested code, ignoring error handling and logging, or failing to document or comment code. Hard-coding values and credentials in code can lead to security vulnerabilities if the code is exposed or shared. It also makes the code less flexible and harder to maintain. Instead, use configuration files or environment variables to store this information.

Hard-coding values and credentials

Hard-coding values and credentials are security risks because, if the code is exposed or shared, anyone can see and use these values or credentials. This could lead to unauthorized access or misuse of systems. To mitigate this, it is necessary to use configuration files or environment variables to store this information. This allows us to change the values without changing the code and keep sensitive information out of the codebase.

Premature optimization

Premature optimization can lead to unnecessarily complex code that is hard to read and maintain. It can also lead to wasted effort if optimization isn't needed. It's better to first write clear, correct code and then optimize, if necessary, based on performance testing.

Magic numbers

Magic numbers are numerical values with an unclear meaning. They can make the code confusing and hard to maintain because it's not clear what they represent. It is better to replace them with named constants to make the code more readable and maintainable.

God objects

God objects are classes or modules overloaded with dependencies and responsibilities. They can make the code hard to understand, test, and maintain. They also violate the single responsibility principle, which states that each class or module should have one clear responsibility. To mitigate this, we can refactor code to ensure that each class or module only has one responsibility.

Copying or pasting code without understanding or testing it first

Copying and pasting code without understanding or testing it can lead to bugs and security vulnerabilities because we might not fully understand what the code is doing or how it might interact with existing code. It is good practice always to take the time to understand the code we're using, write tests for it, and ensure it works correctly in the specific context.

Inconsistent or unclear naming

Using inconsistent or unclear naming conventions can make the code hard to read and understand. It can also lead to errors if names are misleading or confusing. Follow a consistent naming convention that clearly indicates what each variable, class, function, file, or other code element does.

Writing overly complex or nested code

Writing overly complex or nested code can make the code hard to read, understand, test, and maintain. Complex code is also more prone to errors and bugs. The code should be kept as simple and straightforward as possible.

Ignoring error handling and logging

Ignoring error handling and logging can make it hard to debug issues and can lead to poor user experiences when errors occur.

Failing to document or comment in the code

Failing to document or comment in the code can make it hard to understand what the code is doing. It can also lead to errors if people misinterpret what the code is supposed to do.

Classification of Bugs

Bugs and vulnerabilities are weaknesses in an application. These weaknesses can be due to design or implementation errors, and they can be exploited by attackers to compromise the system or cause harm to users (OWASP, n.d.-b). Nevertheless, despite both being weakness in an application, bugs and vulnerabilities have several differences. Bugs refer more to functional issues that affect how the software operates, while vulnerabilities refer to security issues that could potentially be exploited to cause harm.

Bugs are errors or flaws in a software program that cause it to produce incorrect or unexpected results. They are usually the result of human error in the software's source code. Bugs can cause a software program to behave unexpectedly or crash, but they do not necessarily pose a security risk. They can be fixed by identifying the error in the source code and correcting it.

Vulnerabilities are weaknesses in a software application that can be exploited by attackers to gain unauthorized access or perform unauthorized actions. Vulnerabilities can be caused by various factors, including poor design, lack of input validation, and misconfiguration. Unlike bugs, vulnerabilities pose a security risk as they can be exploited by attackers to compromise the system or data. Mitigation strategies for vulnerabilities often involve patching the software, changing configuration settings, or sometimes redesigning aspects of the system.

Software bugs can be classified in different ways, but perhaps the most used classification is based on bugs' nature and impact, as shown in the table below.

Table 6: Software Bugs Classification by Nature

Bug class	Description	Example
Functional bugs	These bugs are associated with the functionality of a specific software component.	For example, a calculator app that gives incorrect results, like $2+2=5$
Unit level bugs	These bugs are defects that are related to the functionality of a single software unit.	These bugs occur at the smallest testable part of an application, for instance, a function in a code that is supposed to return the square of a number but instead returns the cube.
Integration level bugs	These bugs occur when individual software modules are combined and tested as a group.	For example, two independently working features causing the application to crash when used together
Usability defects	These defects make the software difficult or confusing to use.	For instance, a button that is supposed to submit a form does nothing when clicked.
Performance defects	These defects cause the software to perform poorly, for example, making it run slowly or consuming excessive resources.	For example, a website taking too long to load or an application using excessive memory
Security defects	These defects can be exploited by attackers to gain unauthorized access or perform unauthorized actions.	For instance, an application that stores passwords in plain text
Compatibility defects	These defects cause the software to function incorrectly on certain platforms or configurations.	For example, a mobile app working fine on Android but crashing on iOS
Syntax errors	These are mistakes in the source code that cause it to fail to compile or run correctly.	For instance, forgetting a semicolon at the end of a statement in languages like C++ or Java

Source: Vladyslava Volyanska (2024).

Vulnerabilities can be classified into different types. The three most commonly used methods of vulnerabilities classification are detailed in the table below.

Table 7: Methods of Vulnerabilities Classification

Classification		Description
Based on their causes, effects, or exploitability	OWASP classification	This includes common types such as injections, broken authentications, sensitive data exposure, cross-site scripting, and insecure deserialization.
Proposed by Thamali Madhushani Adhikari and Yan Wu in their 2020 paper presented at the IEEE Symposium	Bugs Framework classification	This framework defines four dimensions of vulnerabilities: behavior, usage, goal, and scope.
The classification of vulnerabilities into three main categories	Input validation	This is the process of checking the data that are provided by the user or another source before using it in the software. It can prevent malicious or malformed data from causing unexpected behavior or damage to the system.
	Memory management	These vulnerabilities occur when a program does not properly manage memory allocations, which can lead to issues like buffer overflows.
	Logic errors	These are bugs in the program's logic that can lead to unintended behavior.

Source: Vladyslava Volyanska (2024).

For instance, the Open Worldwide Application Security Project (OWASP) classifies vulnerabilities based on their causes, effects, or exploitability; for example, some common types are injections, broken authentications, sensitive data exposure, cross-site scripting, and insecure deserialization. CVSS can be used to assess these vulnerabilities. These scores are indicative of the intrinsic properties of each vulnerability. At present, the National Vulnerability Database (NVD) does not offer “temporal scores” (metrics that fluctuate over time due to external events associated with the vulnerability) or “environmental scores” (scores adjusted to mirror the impact of the vulnerability on a particular organization). Nevertheless, the NVD provides a CVSS calculator for both CVSS v2 and v3, facilitating the inclusion of temporal and environmental score data (Sidagni, 2022; National Institute of Standards and Technology, 2023a).

Table 8: Classification Table for Vulnerabilities Based on the Common Vulnerability Scoring System (CVSS)

Severity	CVSS v2.0 base score range	CVSS v3.0 base score range
None	0.0	0.0–3.9
Low	0.1–3.9	4.0–6.9
Medium	4.0–6.9	7.0–8.9
High	7.0–10.0	9.0–10.0
Critical	N/A	9.0–10.0

Source: Vladyslava Volyanska (2024), based on the National Institute of Standards and Technology (2023a).

Another type of classification based on the Bugs Framework (BF) was proposed by Adhikari and Wu (2020). The BF defines four dimensions of vulnerabilities: behavior, usage, goal, and scope. Behavior describes what the vulnerability does or is able to do, usage describes how vulnerability can be exploited or triggered, goal describes why a vulnerability exists or what it aims to achieve, and scope describes where the vulnerability affects or applies to.

Table 9: Classification of Vulnerabilities Based on the Bugs Framework

Dimension	Description
Behavior	Describes what the vulnerability does or allows to do
Usage	Describes how vulnerability can be exploited or triggered
Goal	Describes why vulnerability exists or what it aims to achieve
Scope	Describes where the vulnerability affects or applies to

Source: Vladyslava Volyanska (2024), based on Adhikari & Wu (2020).

There is one other type of vulnerability classification, according to which vulnerabilities are divided into three main categories: input validation, memory management, and logic errors.

Table 10: Classification of Vulnerabilites Into Three Main Categories

Category	Description
Input validation	These vulnerabilities occur when an application does not properly validate user input before using it. This can lead to issues such as a Standard Query Language (SQL) injection or cross-site scripting.

Category	Description
Memory management	These vulnerabilities occur when a program does not properly manage memory allocations, which can lead to issues like buffer overflows.
Logic errors	These are bugs in the program's logic that can lead to unintended behavior.

Source: Vladyslava Volyanska (2024).

Input validation is the process of checking the data that are provided by the user or another source before using it in the software. Input validation is important because it can prevent malicious or malformed data from causing unexpected behavior or damage to the system. For example, if a user enters a very long string of characters into a text field that expects a name, the software should reject or truncate the input instead of trying to store it in a variable that has a limited size. Otherwise, the input could overflow the variable and overwrite other parts of the memory, leading to a bug vulnerability known as “buffer overflow.”

Buffer overflow is one of the most common and dangerous types of input validation bugs vulnerabilities. This happens when a program attempts to store more data in a buffer, or temporary storage area, than it was designed to hold. The overflow of data can then leak into and corrupt adjacent memory space. This can result in crashes, data corruption, or code execution. Code execution means that the attacker can inject their own code into the memory and run it on the system, gaining full control over it. Buffer overflow can affect any software that handles user input, such as web applications, network services, or operating systems.

To prevent buffer overflow and other input validation bugs vulnerabilities, software developers should always validate and sanitize the input before using it. They should also adopt secure programming techniques, including using bounded functions, which limit the amount of data that can be written to a buffer; avoiding unsafe functions that do not check the size of the input; and using memory protection mechanisms such as stack canaries or address space layout randomization, which can detect or prevent memory corruption.

In C++, there are several functions that are considered unsafe because they do not perform any input validation or bounds checking. For instance, the function `strcpy` transfers a string from one buffer to another without verifying whether the destination buffer has enough space to accommodate the source string. This could result in a buffer overflow if the source string exceeds the size of the destination buffer. A safer alternative is to use `strncpy`, which copies most `n` characters from one buffer to another, where `n` is specified by the programmer. Another example is the `gets` function, which reads a line of input from the standard input stream and stores it in a buffer without checking if the buffer is large enough to hold the input. This can lead to buffer overflow if the user enters more characters than the buffer can hold. A safer alternative is to use `fgets`, which reads most `n` characters from a stream and stores them in a buffer, where `n` is specified by the programmer.

Figure 19: Examples of Safer Alternatives to Unsafe Functions in C++

Unsafe strcpy function:
<pre>char src[50] = "This is the source string"; char dest[50]; strcpy(dest, src); // This could lead to buffer overflow if src is larger than dest</pre>
Safe strncpy function:
<pre>char src[50] = "This is the source string"; char dest[50]; strncpy(dest, src, sizeof(dest)); // Copies at most sizeof(dest) characters from src to dest</pre>
Unsafe gets function:
<pre>char buffer[50]; gets(buffer); // This could lead to buffer overflow if input is larger than buffer</pre>
Safe fgets function:
<pre>char buffer[50]; fgets(buffer, sizeof(buffer), stdin); // Reads at most sizeof(buffer) characters from stdin to buffer</pre>

Source: Vladyslava Volyanska (2024).

Memory management is the process of allocating and freeing memory for storing data and code in the software. Memory management is important because it affects the performance and reliability of the software. If the software does not manage memory properly, it can cause memory leaks, dangling pointers, or double free errors.

Memory leaks occur when the software allocates memory but does not free it when it is no longer needed. This can result in wasting memory resources and slowing down the system. Memory leaks can also create security risks because they can expose sensitive data that are stored in the memory or make it easier for attackers to exploit other bug vulnerabilities.

Dangling pointers occur when the software frees memory but still keeps a pointer (a variable that stores the address of another variable) to it. This can result in accessing invalid or freed memory locations, causing crashes or data corruption. Dangling pointers can also create security risks because they can allow attackers to manipulate or read data from other parts of the memory or execute arbitrary code.

Double free errors occur when the software frees memory two or more times. This can result in corrupting the memory management structures, causing crashes or data corruption. Double free errors can also create security risks because they can allow attackers to overwrite or control other parts of the memory or execute arbitrary code.

To prevent memory management bug vulnerabilities, software developers should always follow good practices such as using smart pointers (which automatically manage memory allocation and deallocation), avoiding manual memory management (which requires explicit calls to allocate and free memory), and using tools such as debuggers or analyzers (which can detect or prevent memory errors).

Logic errors are bug vulnerabilities that occur due to mistakes or flaws in the logic or algorithm of the software. Logic errors are important because they can affect the correctness and functionality of the software. For example, if a software calculates the average of two numbers by adding them and dividing by two, it will produce incorrect results if one of the numbers is negative or zero.

Logic errors are often hard to detect and fix because they do not cause obvious symptoms such as crashes or error messages. They may only manifest themselves under certain conditions or inputs that are not tested by the developers or users. Logic errors can also create security risks because they can allow attackers to bypass authentication, authorization, encryption, or verification mechanisms that are implemented in the software.

5.2 Sources of Bugs

Bugs are defects or errors in software that cause it to behave in unintended or unexpected ways. Below are some common types of software bugs and their sources:

Table 11: Software Bugs and Their Sources

Bug class	Source
Functional bugs	These bugs are associated with the functionality of a specific software component.
Errors of commissions	These are defects that occur when something is done incorrectly.
Errors of omissions	These occur when something that should have been done is not done.
Errors of clarity	These occur when something is not clear or is too ambiguous.
Error of speed and capacity	These occur when the software does not perform with the desired speed or capacity.

Source: Vladyslava Volyanska (2024).

Bugs can have various impacts, for example, reducing software performance, reliability, usability, or compatibility, and can cause security vulnerabilities that can be exploited by attackers. The sources of bugs can vary widely, ranging from mistakes in code to external factors like changes in application programming interfaces (APIs). It is important to understand the bug sources: specifically, how they arise and whether they were created consciously.

Table 12: Secure Bug Classes and Their Sources

Security bug class	Source
Memory safety bugs	These include buffer overflow and dangling pointer bugs. They can be caused by insecure or poorly engineered code.
Race condition	This occurs when the behavior of software depends on the sequence or timing of uncontrollable events.
Secure input and output handling	Bugs can occur when input and output are not properly secured. This can be due to unsensitized input.
Faulty use of an API	This happens when an API is used incorrectly, leading to security vulnerabilities.
Improper use case handling	This can lead to security bugs when use cases are not properly handled.
Improper exception handling	Bugs can occur when exceptions are not properly handled, leading to potential security vulnerabilities.
Resource leaks	These often occur due to improper exception handling and can lead to security vulnerabilities.

Source: Vladyslava Volyanska (2024).

Security bugs can originate from human, environmental, and technical factors. Within human factors, psychological, social, and cognitive aspects can be distinguished, all of which influence the way humans interact with software. Human factors affect both developers and users and can entail security bugs due to different reasons:

- **lack of knowledge:** It is relevant for developers, who sometimes don't have the required skills to design, implement, or maintain the software securely, and users, who can't use, configure, or update software securely.
- **lack of awareness:** Both developers and users may not be aware of potential risks or may not be motivated to take preventive or corrective actions. For example, developers may not perform adequate testing or verification of the software, or users may not follow the security policies or guidelines for the software.
- **human error:** These are unintentional errors that can be introduced by both groups.
- **human biases or assumptions:** Both developers and users have biases or assumptions that can influence their decisions regarding the software. For example, developers can underestimate risks, and users may trust software too much.
- **human emotions:** External factors can affect both developers and users, for example, they may feel frustrated or be working under pressure, causing them to introduce intentional errors or harmful code into software.

To prevent human factors from generating security bugs in software, developers and users should

- educate themselves about security principles.
- collaborate within the process of sharing information.
- follow the security policies and guidelines.
- systematically review and audit the code and configuration.
- test and verify functionality and security.
- regularly update and patch the software.

Environmental factors can affect the software itself and also hardware or network on which this software operates. Below are some examples of environmental factors that can affect software security:

- The network that the software operates on, such as the internet or local area network (LAN), can affect the availability, integrity, or confidentiality of the software data or communication. These networks may be congested, disrupted, or intercepted.
- The hardware that the software runs on, such as the processor, memory, disk, or peripheral devices, can affect the performance, reliability, or compatibility of the software. For example, the hardware may be faulty, outdated, or incompatible with the software.
- The operating system that the software depends on can affect the functionality, usability, or security of the software system. For example, the operating system may have bugs, vulnerabilities, or updates that can affect the software.

Environmental factors can introduce security bugs due to different reasons:

- **changes or updates:** Over time, software, hardware, and the network on which this software operates can be changed or modernized. This can provide new functionalities or dependencies that may affect security.
- **interaction or integration:** Software can be integrated into or interact with other software, which can introduce new challenges or complexities that affects security. For example, a third-party library may contain malicious code or vulnerabilities that can compromise the system, or a web service may expose sensitive data or functionality that can be exploited.
- **variations:** The software can operate in different environments with different requirements. For example, mobile apps can operate on different devices in different networks.

To prevent environmental factors from causing security bugs in software, developers and users should

- systematically monitor changes and updates to the software and hardware.
- verify the interactions of the software with another software or system to ensure their compatibility.
- optimize the software for different environments.

Technical factors can affect both the design and the implementation of the software system. Technical factors can introduce security bugs due to various reasons:

- **complexity or size:** For example, a complex algorithm may have subtle flaws or errors that can cause unexpected behaviors or outcomes, or a large codebase may have redundant or inconsistent code that can cause confusion or conflicts.
- **features or functionality:** For example, a feature-rich application may have multiple entry points or attack surfaces that can be exploited by attackers, or a functional application may have multiple dependencies or interactions that can introduce vulnerabilities or risks.
- **performance or efficiency:** For example, a high-performance application may use unsafe practices or shortcuts that can compromise its security.

To prevent technical factors from causing security bugs, software developers and users should

- simplify and modularize the design and implementation of the software system as much as possible to reduce its complexity and size.
- prioritize and focus on the essential features and functionality of the software system as much as possible to reduce its scope and impact.
- make trade-offs and compromises between performance and efficiency of the software system as much as possible to ensure its security.

5.3 Severity of Bugs

Bugs that affect security and functionality of the software can also have different levels of severity. The severity of bugs is a measure of their impacts for software, infrastructure, of users. The severity of bugs depends on various factors:

- **the type or class of the bug:** The bug may belong to different types and categories, such as syntax errors, logic errors, memory errors, concurrency errors, or security vulnerabilities. Each type or category of bug may have different levels of severity depending on how it affects the software system or users. For example, a syntax error may cause the software to fail to compile or run, a logic error may cause the software to produce incorrect or unexpected results, a memory error may cause the software to crash or leak sensitive data, a concurrency error may cause the software to deadlock or race condition, or a security vulnerability may cause the software to be compromised or exploited by attackers.
- **the location or scope of the bug:** The bug may occur in different locations or scopes within the software system, such as in a single line of code, in a function or module, in a component or subsystem, or in the entire system. Each location or scope of bug may have different levels of severity depending on how it affects the software or users. For example, a bug in a single line of code may affect only a specific feature or functionality of the software, a bug in a function or module may affect a group of features or functionalities of the software, a bug in a component or subsystem may affect a major part of the software, or a bug in the entire system may affect the software as whole.
- **the frequency or occurrence of the bug:** The bug may occur with different frequencies or occurrences within the software system, such as “rarely,” “occasionally,” “frequently,” or “always.” Each frequency or occurrence of a bug may have different levels of severity

depending on how it affects the software or users. For example, a rare bug may affect only a few users or cases of the software, and a frequent bug may affect many users or cases of the software.

Table 13: Secure Bug Classes and Their Severity

Bug class	Severity
Syntax errors	High: These bugs usually prevent the program from running correctly.
Logic errors	Medium to high: These bugs can cause the program to produce incorrect results, which can be critical depending on the application.
Memory errors	High: These bugs can cause the program to crash or behave unpredictably and can sometimes be exploited to create security vulnerabilities.
Concurrency errors	Medium to high: These bugs can cause unpredictable behavior in multi-threaded applications and can be difficult to reproduce and fix.
Security vulnerabilities	Critical: These bugs can be exploited by attackers to gain unauthorized access to data or systems with severe consequences.

Source: Vladyslava Volyanska (2024).

The method to measure and classify the severity of bugs typically involves the following steps:

1. Identify the bug: This is an important first step used as a basis for all of the steps below.
2. Assess the impact of the bug on the software’s functionality: This involves understanding which part of the software is affected by the bug and how it affects the user’s interaction with the software.
3. Assign a severity level to the bug: Severity levels can range from low to critical. For example, a low-severity bug might be a minor issue that doesn’t significantly impact the overall functionality of the app, while a critical severity bug might result in a complete system failure.
4. Use severity level to prioritize bug fixes: Critical and major bugs usually require immediate attention, as they affect the software’s usability and reliability. Minor and trivial bugs can be addressed later, depending on available resources.
5. Reassess bug severity regularly: This should be done if new information becomes available or when other parts of the software change.

To prioritize bugs, a severity scale should be used. Different organizations use various scales, but below is a list of common ones (QA Madness, 2021):

- blocker (S1): This kind of error makes it impossible to proceed by using or testing the software. For example, it will shut down an application.

- critical (S2): This is an incorrect functioning of a particular area of a business-critical software functionality, such as unsuccessful installation or failure of its main features (QA Madness, 2021).
- major (S3): It is an error that affects a significant part of the software functionality or quality but does not prevent its usage or testing. For example, it causes performance degradation or incorrect calculations.
- minor (S4): It is an error that affects a minor part of the software functionality or quality but does not affect its usability or testability. For example, it causes cosmetic issues or minor deviations from specifications.
- low/trivial (S5): It is an error that has negligible impact on the software functionality or quality and does not affect its usability or testability. For example, it causes spelling mistakes or formatting issues.

The bug severity scale can help to rank and compare the bugs and define requirements for fixing them.

Testing strategy is a plan or approach that defines the methods and techniques for finding and fixing bugs.

It is also necessary to introduce and apply a **testing strategy**. Some common testing techniques are listed below.

Static Analysis

Static analysis is a testing technique that analyzes the source code of the software system without executing it. Static analysis can help to find and fix bugs such as syntax errors, logic errors, and memory errors by applying different tools or methods:

- Compilers are tools that translate the source code of the software system into executable code. Compilers can help to find and fix bugs by checking the correctness and consistency of the source code.
- Linters are tools that check the style and quality of the source code. They can help to find and fix bugs such as formatting errors or naming errors by enforcing the best practices and conventions of the source code.
- Code analyzers are tools that inspect the structure and logic of the source code. They can help to find and fix bugs such as complexity errors or redundancy errors by detecting and reporting the potential problems or defects of the source code.

Dynamic Analysis

Dynamic analysis is a testing technique that executes the software system or its parts under different conditions or scenarios. This type of analysis can help to find and fix bugs such as concurrency errors or security vulnerabilities by applying different tools or methods:

- Debuggers are tools that control and monitor the execution of the software system or its parts. Debuggers can help to find and fix bugs such as runtime errors or logic errors by allowing the developers to examine and modify the state and behavior of the software system or its parts.

- Profilers are tools that measure and analyze the performance and efficiency of the software system or its parts. They can help to find and fix bugs such as memory leaks or resource consumption by providing the developers with detailed and accurate information about the execution time, memory usage, or central processing unit (CPU) usage.
- Testers are tools that simulate and verify the functionality and usability of the software system or its parts. Testers can help to find and fix bugs such as functional errors or non-functional errors by generating and executing different types of tests, such as unit tests, integration tests, and system tests.

Code Review

Code review is a testing technique that involves examining and evaluating the source code of the software system or its parts by other developers or experts. Code review can help to find and fix bugs such as logic errors, design errors, or security vulnerabilities by applying different methods or approaches:

- Peer review is a method of code review that involves reviewing the source code of the software system or its parts by other developers. Peer review can help to find and fix bugs such as syntax errors or style errors by providing feedback and suggestions on how to improve the source code.
- Expert review involves reviewing the source code of the software system or its parts by other developers who have more skills or knowledge. Expert review can help to find and fix bugs such as logic errors or design errors by providing guidance and advice on how to optimize the source code.
- Formal review involves reviewing the source code of the software system or its parts by following a predefined process or procedure. Formal review can help to find and fix bugs such as security vulnerabilities and compliance issues by ensuring that the source code meets the standards and requirements.



SUMMARY

When we discuss software development, we often encounter different types of bugs. These bugs can be categorized or classified based on their nature, such as syntax errors, logic errors, memory errors, concurrency errors, and security vulnerabilities. Each of these bug classes has a source or cause. For example, syntax errors are usually due to mistakes in the code's syntax, while logic errors are due to incorrect logic or algorithms.

The severity of a bug refers to how much it impacts the software or its users. For instance, a syntax error might prevent the software from running at all (high severity), while a minor logic error might only affect a specific feature (low severity).

To avoid these bugs and vulnerabilities, developers follow certain best practices. These include adhering to coding standards and guidelines, using code analysis tools for automatic code review and testing, applying input validation and output encoding techniques to prevent common vulnerabilities, and avoiding code with poor design.

UNIT 6

PROJECT MANAGEMENT

STUDY GOALS

On completion of this unit, you will be able to ...

- understand different project management models.
- define stages of the life cycle management.
- apply information to develop a solid project plan.

6. PROJECT MANAGEMENT

Introduction

Management in any field is a complex and heterogeneous activity, but a major part of it is associated with participation in different projects. A significant part of today's business worldwide is project-oriented. But what exactly is a project and how can we define it?

There are many different definitions of a project, but they all conclude that a project is a set of interconnected activities with a start and end point that leads to a common goal. Typically, a project is a one-time event and its effect is unique, although it can also lead to the development of a new model of mass-oriented solutions. Such an approach is usually used by companies focused on providing their customers with specialized results, which is particularly evident in the information technology (IT) industry, from which many work organization methodologies originate. These methodologies define individual tasks within a project, as well as control and supervision over implementation.

There are no universal methodologies that can be applied to the specific aspects of each project. However, two forms currently dominate: the waterfall and Agile models. The waterfall model is a traditional model with a linear, sequential approach: Tasks completed in one phase must be analyzed and verified before moving on to the next phase. Conversely, Agile methodology is an incremental approach, based on flexible response to changes. Both models have their advantages and disadvantages depending on factors including the customer approach, financial model, scale, and timeframes.

6.1 The Software Life Cycle

Management in any field of IT is a complex and heterogeneous activity, but much of it is associated with participation in different projects. An increasing number of companies focus their efforts on creating new products and services or achieving new results – this is the reason that a major part of today's business is project-oriented.

The most basic component of project management is the project management life cycle (PMLC) – a comprehensive model of the project management process. The most apparent method of making a project more manageable is to separate the process into sequential steps. There are several versions of the PMLC, each with a different set of recommendations for application and a different number of phases/stages. However, the most commonly used version is the model developed by the Project Management Institute (PMI), which describes both the overall project life cycle and the knowledge required in each phase.

The traditional project management according to the PMI model is based on the linear structure (PMI, 2021). You cannot reach the next level without completing the previous one. It is the most widely used project management model, based on the “waterfall” or

“cascading” cycle, where the task is transferred sequentially through different stages, resembling the flow of water. The disadvantage of this approach is that it requires us to determine what should be received at the first stage of the project as a priority. If this is feasible, the waterfall approach brings a level of consistency to the project work, and the strict planning allows us to simplify the execution of the project. The main disadvantage is an inflexibility to react to changes in the project goal.

The PMI model consists of five overarching stages covering the entirety of the project:

1. **Initiation** means setting the purpose of the project based on the business case and demands of the stakeholders.
2. **Planning** is the development of a project plan and scope according to the chosen management process, schedule, and end goals.
3. **Execution** involves completing tasks including resource allocation, regular meetings, and actual project implementation.
4. **Monitoring and controlling** involves the supervision of the work performed using documents such as a burn chart, and estimating according to key purposes.
5. **Closing** means evaluating the efficiency of the work over the project taking the most important conclusions and optimizing for future projects.

These phases may be applied to almost any type of project. They only differ in how phases manifest themselves in the particular project management process. Larger projects almost always require some overlap between phases. Even the initiation and closing phases may overlap, as some planning may be needed to make a project feasible, or part of a project may require controlled closure ahead of others. When we consider the five phases, some of them can be more or less important for different projects. For example, for firms that are producing the same products repeatedly, the planning phase is not as important (limited planning needs) as the control phase.

During the history of project management, many different project management methodologies have been created for almost any need.

Project Management Life Cycle ≠ Software Development Life Cycle

“The Software Development Life Cycle is a process that produces software with the highest quality and lowest cost in the shortest time possible” (Alexandra, 2023). The concept of the software development life cycle (SDLC) appeared when the programming community understood the need to move from “home-made” methods of software development toward industrial production. Historically, the development of life cycle concepts is associated with the search for appropriate models of performance. The difference in the purpose of applying models determines their diversity. There are three main reasons why software life cycle modeling is necessary:

1. It helps understand what can be expected by ordering or purchasing software, as well as what is unrealistic.
2. The software life cycle model is the basis for understanding programming technologies and tools: Any technology builds its methods and tools around the stages of the life cycle, while they are able to define and outline all activities performed on a product starting from its inception to retirement.
3. The general understanding of how a software project is developing provides the foundational information for its planning, meaning spend resources more economically and achieving a higher quality of management.

The SDLC model concept covers the stages that a software product passes through, from the appearance of an idea to its implementation and subsequent support. Typically, the stages include

- planning,
- requirements analysis,
- design,
- implementation and testing, and
- maintenance and support.

The SDLC models largely predetermine the methods of software development. As well as the above-mentioned waterfall and Agile models, the **incremental** and **spiral** models are also useful.

Incremental model
Development, performed in iterations with feedback loops between stages, is carried out on an incremental basis. Each increment adds certain functionality to the system. In the initial stages, all of the basic requirements are determined and divided into more and less important ones.

Spiral model
At each turn of the spiral, the next workable version/fragment of the product is created, the requirements are defined, and the next work is planned. Particular attention is paid to the initial stages, where the technical solutions are tested through the creation of prototypes.

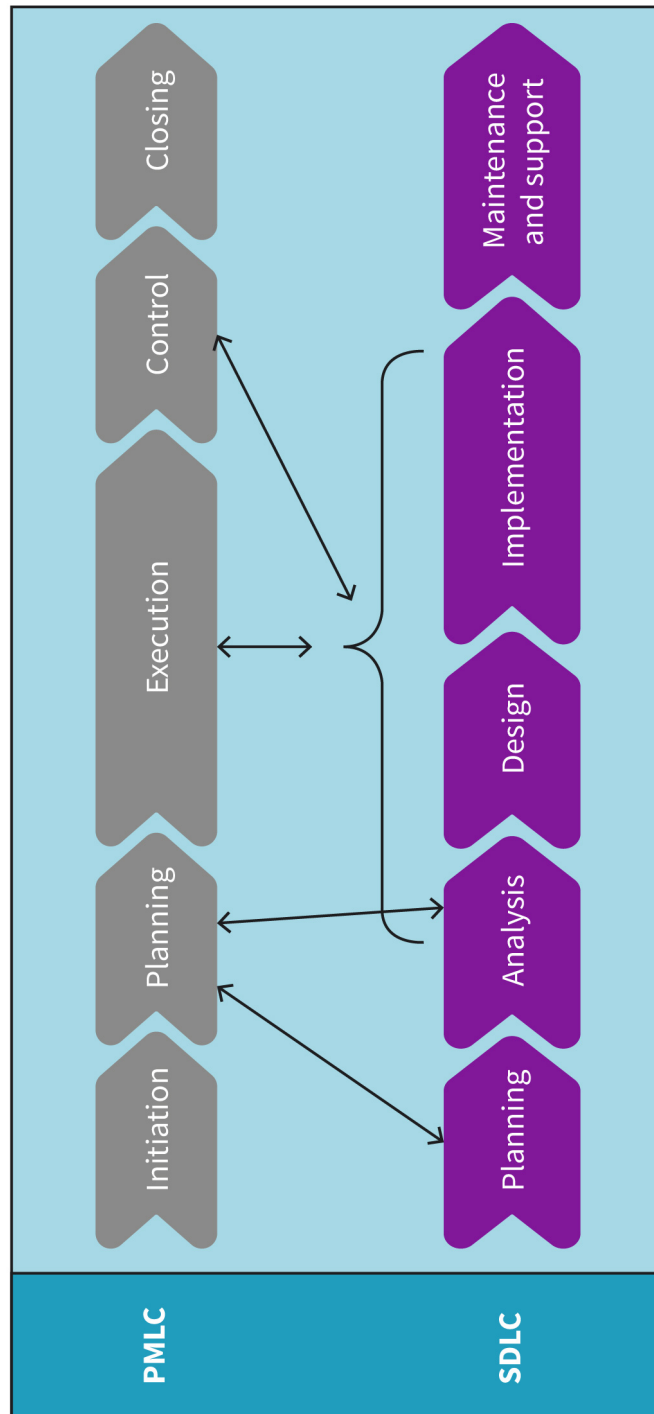
The choice and construction of an SDLC model is based on the conceptual idea of the product being designed, considering its complexity and remaining in accordance with standards. The chosen model is divided into processes, which should include individual works and tasks implemented in it. By choosing a general version of the SDLC model for a specific project, the question of including or not including individual works is very important. Currently, the theoretical basis is the ISO/IEEE 12207 (International Organization for Standardization/Institute of Electrical and Electronics Engineers; ISO, n.d.), as well as the International Electrotechnical Commission (IEC). ISO/IEC/IEEE 12207:2017 describes a complete set of processes, covering all possible types of work and tasks associated with software creation. According to this standard, only those processes that are most suitable for the implementation of exactly this project should be selected. The basic processes that are present in all known SDLC models are mandatory. Depending on the goals and objectives of the project area, they can be supplemented by processes from the group of auxiliary or organizational processes (or sub-processes).

The project management life cycle (PMLC) and SDLC should not be confused with one another. The PMLC is independent from the technology concept and focuses on the “external” side of the project, on management resources such as time, costs, and people and has is prioritized over the SDLC. The PMLC is a high-level approach to managing a project from start to finish and is technology-independent, meaning it can be applied to any type of project, not just software development. It provides a framework for planning, executing, and closing a project, regardless of its nature or complexity.

On the other hand, the SDLC concentrates on the “inner” aspect of the project product – creation and implementation. The SDLC is always dependent on chosen technology and provides a more detailed view by focusing on the development of the product. This differs from the PMLC, which provides a broader overview of the project and is strongly focused on the quality of the product. The SDLC is also dependent on the chosen technology and includes stages like requirements gathering, design, coding, testing, deployment, and maintenance.

In essence, while PMLC provides a broad overview of how to manage a project, SDLC delves into the specifics of how to develop a software product within that project. Both are crucial for successful project management but serve different purposes.

Figure 20: The Relationship Between the PMLC and SDLC



Source: Vladyslava Volyanska (2023), based on Obeidat & Nasereddin (2013).

The Software System Design

The other concept closely connected to the PMLC and SDLC is the software system design. It is a process that can be defined as the process of creating a systems project. The basic aspect of software system design is a software requirement analysis (SRA). In the software system design, an important position belongs to software design principles. These principles can be split into groups (Blancas, 2023):

- SOLID principles:
 - single responsibility principle
 - open/closed principle
 - Liskov substitution principle
 - interface segregation principle
 - dependency inversion principle
- DRY (don't repeat yourself) principles, which advises users to avoid redundant code.
- KISS (keep it simple stupid) principle, which advises users to write simple code.
- YAGNI (you aren't gonna need it) principle, which advises users to avoid unnecessary code.

The systems project ensures that all participants understand the purpose of the development. Depending on the complexity of the software, the design process can be provided manually or be fully automated. Currently, industrial technologies for software creation are widely used. Such technologies are represented by descriptions of the principles, methods, applied processes, and operations and are supported by a set of computer-aided software engineering (CASE) tools, which cover all stages of the SDLC and are used to solve practical problems. Examples of these technologies include Microsoft Solution Framework (MSF), Rational Unified Process (RUP), and Extreme Programming (XP).

During the design process, various graphic tools can also be used, including flowcharts, entity–relationship (ER) diagrams, Unified Modified Language (UML) diagrams, and layouts. They are used for a schematic representation of a process, enabling us to study, plan, and explain complex processes with simple diagrams.

The Development of the Architecture According to the Principles of an Agile Approach

The traditional models of PMLC, like the waterfall model, product manufacturing information (PMI) model, or the more innovative **PRINCE2** model, are good for small or predictable projects. As the work result can only be verified after the completion of a particular stage or the entire project, a negative scenario can lead to a situation where the client discovers that the effect does not meet their expectations. It is the main disadvantage of these models.

For the development of innovative products and “open-end” projects, Agile methodologies are more suitable because they allow a project to be implemented in small steps. This ensures process flexibility, better control, and adaptability to changing needs.

PRINCE2

The full name for this model is “Projects in Controlled Environments version 2” and it focuses on quality. Each team member has a distinct function in each of the seven processes: starting, initiation, directing, controlling a stage, managing product delivery, managing a stage boundary, and closing.

The main idea of Agile implementation is to start working with the standard version of the product and then gradually, iteratively adjust it to the user's specific needs. The Agile method emphasizes close cooperation and customer satisfaction. It defines time-limited stages called "sprints" or "timeboxes." At the beginning of each sprint, a list of tasks is created according to priorities defined together with the client. The client can modify the priority for all system functions. At the end of the sprint, the system provider and customer review and evaluate the progress of the work. The Agile model facilitates cost rationalization.

Rather than being a project management method, Agile is a set of ideas and principles for how projects should be implemented. The separate flexible methods (frameworks), like Scrum, Kanban, Lean, and many others have been developed based on these principles. All of these methods are different, but they follow the same principles and retain the main advantage of Agile, namely flexibility and adaptability.

Scrum

Scrum is one of the most popular frameworks from the Agile family. It has absorbed some elements of the traditional processes and combined them with the ideas of an Agile project management methodology. Scrum is designed for projects that require quick results and are tolerant of changes. In Scrum, the project is broken down into parts (the product backlogs), which can be immediately used by the customer. The most important backlogs are the first to be selected for execution in a sprint, an iteration of Scrum with a typical duration of two to four weeks. The working product increment is presented to the customer at the end of each sprint. For example, we can think of a web page with limited functionality. Afterwards, the project team proceeds to the next sprint; the team decides on the duration of the sprint at the beginning of the project and fixes a timeframe.

Before each sprint begins, the project scope that is yet to be completed is re-evaluated and changes are made to ensure that the project fulfills the expectations of the customer. The project team, the scrum master, and the product owner are involved in this process. The basic structure of scrum processes contains around five main meetings: the backlog refinement meeting (also called "backlog grooming"), sprint planning, daily meetings, sprint debriefing, and sprint retrospective. Scrum demands that the project team be small and cross-functional.

Lean

Agile encourages breaking a task into small, manageable packages, but the way in which the development of this package should be managed remains unclear. Scrum provides processes and procedures, while Lean adds a workflow scheme so that each iteration is executed with the same quality. As in Scrum, the Lean process is divided into small delivery packages that are implemented separately and independently; for the development of each delivery package, there is a workflow with stages. Lean does not have clear stage boundaries in the way that Scrum does with Sprint limits. Lean also allows you to perform several parallel tasks at different stages, which speeds up the project and increases flexibility. Not every part of the project requires the same attention, but the fact that Lean has the same approach to each task and stage makes it unfit for large and heterogeneous

projects. Lean does not offer a clear workflow for the implementation of separate parts, unlike Scrum (National University of Life and Environmental Sciences of Ukraine [NUBiP], 2019).

Kanban

Kanban was created by Toyota engineer Taiichi Ono in 1953 (Hiranabe, 2008). For this reason, Kanban has parallels to the industrial production process: The product increment is passed forward from one stage to another, and a ready-to-ship item results from the process. If the priority of tasks changes, Kanban allows you to leave a stage with an unfinished task so that you can move to another with a higher priority. Kanban is less strict than Scrum: There is no time limit for sprints and there are no roles, except for the product owner. Kanban also allows a team member to execute multiple tasks at the same time. Kanban has four pillars: An individual “card” is generated for each task, which provides all the required information about the task; there is a limit to the number of tasks per stage; continuous flow means that tasks from a backlog enter the flow in order of urgency; there is continuous improvement (or “kaizen”; NUBiP, 2019). Like Scrum, Kanban is suitable for a team with well-established communication.

6.2 Managing Vulnerability Disclosures

Bugs, vulnerabilities, hacking, and application malfunctions regularly appear in news headlines, demonstrating their importance in today’s world. Stories such as those about Log4J/Log4Shell (CVE-2021-44228), a type of remote code execution vulnerability, or security exposures in AMD processors have loomed large in developer’s minds to give just two recent examples (Apache Logging Services, 2023). Claroty’s cybersecurity report shows that most vulnerabilities have a high or critical severity level, leading to remote attacks and resulting in a complete loss of availability (Claroty Team82, n.d.).

Vulnerabilities are defects in software (software flaws), networks, and systems that threaten the security or expose data to disclosure, damage, or deletion. Vulnerabilities in the software are named using certain standards, and information about them is published in a database, for example, the Common Vulnerabilities and Exposures (CVE) database (CVE Details, n.d.). To increase the security of web applications, the Open Worldwide Application Security Project (OWASP) has developed methods for their development and testing, namely the OWASP Top 10. OWASP’s Top 10 Web Application Security Risks contains information about the most important risk categories marked from A01:2021 to A10:2021 (OWASP, 2023a).

The number of **vulnerabilities** detected is increasing. Vulnerabilities can show up in any software, and it is necessary to get information about them as soon as possible and manage them in an automated manner. Vulnerability management consists of four main stages:

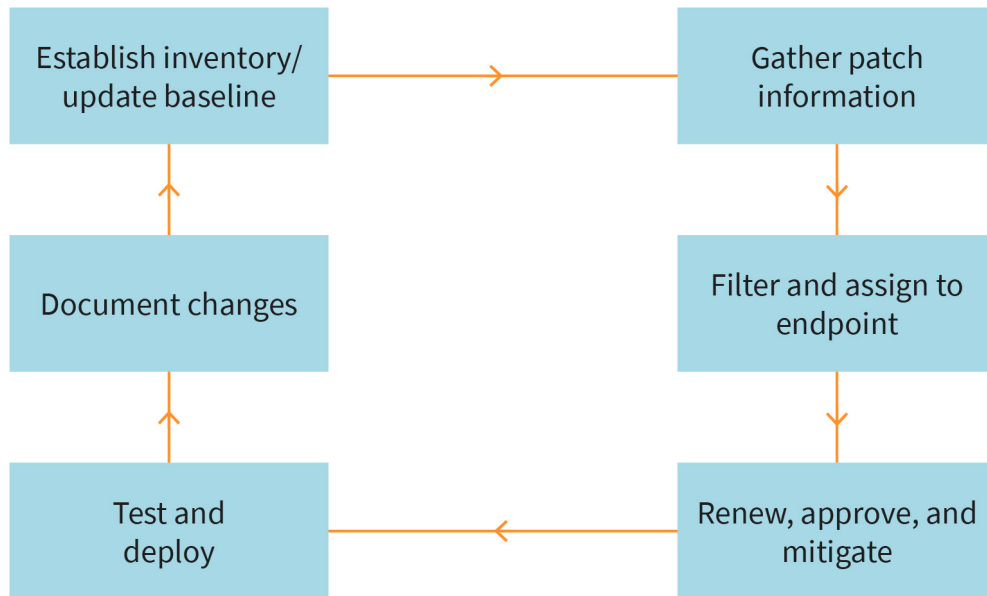
Vulnerability is a weakness in a computer-based system that may be exploited to cause loss or harm (Sommerville, 2016, p. 377).

1. **Scanning:** This is the basis for obtaining reliable information. The vulnerabilities scan may be performed in one of two ways: invasive tests (simulating a real attack in a controlled manner) or non-invasive tests (analysis of the structure and its components without actual vulnerability testing). Both methods are used in vulnerability management, but invasive tests offer a large penetration range.
2. **Risk assessment and categorization of vulnerabilities:** Scanning can indicate a large number of warnings. Therefore, all detected vulnerabilities should be categorized and prioritized to facilitate the creation of initial action plans. Prioritization is performed based on the results from the scanning and information obtained from, for example, the National Vulnerability Database (NVD) CVE or the MITRE Common Weakness Enumeration (CWE) database. Norms and standards are used for this purpose.
3. **Reporting:** The findings are reported along with recommendations.
4. **Implementation of changes:** The recommended corrections are taken to eliminate/patch detected vulnerabilities. The vulnerability management process should consider the operational environment of the software using the Common Vulnerability Scoring System (CVSS) 2.0 and CVSS 3.1 scales. The CVSS 2.0 was used until 2015, while the most advanced CVSS version (3.1) was introduced in 2019. An important point is that “where CVSS 2.0 and 3.0 scores could have been ‘erroneously’ employed as a measure of risk arising from a vulnerability, CVSS 3.1 standard ... explicitly clarifies ‘CVSS measures severity, not risk’” (Sharma, 2020). As Maurice (2020) emphasizes, “the most significant difference between CVSS versions 3.0 and 3.1 is a change in the definition of Attack Complexity.” In version 3.0, attack complexity considered whether the system being attacked could only be exploited if in a certain configuration. If so, attack complexity is considered “high.” Due to many vulnerabilities found to date, not all publicly known vulnerabilities have been assigned a CVSS 3.1 score. This situation disadvantages many organizations in having to move from CVSS standard 2.0 to 3.1. However, there are machine learning algorithms that allow calculation of the missing CVSS 3.1 scores, allowing for many organizations to transition to CVSS 3.1 standard (Nowak et al., 2021).

6.3 Managing Patches/Updating

Vulnerability management is a process that detects, categorizes, and reports vulnerabilities. Patches are updates, usually from the vendor, and can include security fixes or relatively new features. The vendor should document all changes and additions (Verve Industrial, 2022). All patches contain security fixes, and not all patches fix security issues (Bailey, 2019). To simplify and systematize patch management, Verve Industrial Protection has developed a multi-staged approach called operational technology (OT)/industrial control system (ICS) patch management. Patch management is represented as a cycle with stages where known vulnerabilities and available patches are identified and reviewed, strategies are developed, and patches are deployed and tested. Even though this approach was foreseen for industrial systems, its versatility has made it generally applicable to other industries.

Figure 21: End-to-End OT/ICS Patch Management Process



Source: Petra Beenken (2024), based on Verve Industrial (2022).

The main purpose of OT/ICS Patch Management approach is time and complexity reduce by integrating each critical step in a single-flow process consisting of six steps:

1. **Establish baseline OT asset inventory:** At this stage, all plugged assets should be listed.
2. **Gather software patch and vulnerability information:** Because of the variety of applications and the large number of different vendors, it is extremely difficult and time-consuming to collect information on which patches are available and which are required.
3. **Identify vulnerability relevancy and filter to assign to endpoints:** In this step, it is necessary to decide which patches should be applied. Companies present long lists of available patches, and the task of filtering relevant ones from them can become a real challenge. This process should be peremptorily automatized because of its time-consuming nature.
4. **Review, approve, and mitigate patch management:** At this stage, the baselines (strategies) are developed to decide how to filter out the ones that should be deployed among all existing patches.
5. **Test and deploy vulnerability patches:** Here, the patches are tested and deployed. The updates that don't work properly are rolled back.
6. **Profile and document systems pre-and post-patching:** This is the step where the documentation is carried out. Any changes made should be listed to secure old and new configurations and maintain compliance.

The OT/ICS patch management cycle describes a common approach, but proposed solutions are flexible and scalable. The implemented solutions should be adopted to the specific needs of the company.

There is no doubt that patch management must be automated. Currently, patch management tools are developing toward repair processes and delegating tasks to individual users. Such tools have evolved from simple patch installers to solutions that can manipulate security configuration settings, deploy standard software packages, maintain compliance policies, and proactively fix bugs.

6.4 Managing Pentesting and Bug Bounty Programs

Penetration testing (or “pentesting”) is a human-led security assessment process. It can be carried out for many different reasons, for example, for safety assessments or regulatory compliance. The most common reason is to check whether an organization is adequately protected.

Pentests simulate a hacking attack and end with a report, which includes identified gaps and contains solutions that should be implemented. Usually, three types of pentesting are differentiated, based on how much information the tester has about the checked area:

1. **White-box pentest:** The tester has access and information about the IT architecture of the tested area. This type of pentest is used to simulate both an external and internal attack.
2. **Black-box pentest:** The pentester has no information about the area being checked and has no access or permissions. This type of test is used to simulate a hacker attack.
3. **Gray-box pentest:** The tester has partial information about the area being checked.

There is no universal and ideal recipe for pentesting – such an overarching concept would be doomed to failure from the start, since the use of a common pentest algorithm would make the hackers’ job much easier.

Theoretically, pentesting has five stages:

1. Preparation
2. Developing an attack plan
3. Specifying the data type (which data are the object of interest, the aim of the test, etc.)
4. Execution of the test
5. Interpreting the results of the report

A good report should include the following:

- error description
- how it was found
- what tools were used
- additional, optional conditions to be met
- a proof-of-concept file that enables one to reproduce the bug

- impact on the organization (how dangerous this vulnerability is)
- how to fix the error (optional)
- links to other documents explaining what the given error is in the case of lesser-known types of vulnerabilities
- description that shows step by step what should be done to exploit an error (optional)

A bug bounty program is a program initialized by different organizations that rewards discovered and reported vulnerabilities in software, applications, or web services. Today, it is often a part of the security strategy of the company. There are a lot of bug bounty programs and platforms worldwide. IT giants such as Intel, Cisco, Microsoft, and Apple regularly use bug bounty programs (Williams, 2023). The main difference between pentesting and bug bounty programs is that bug bounty programs are not restricted by time.



SUMMARY

Implementing any of the project management methods is usually impossible without the set of technological and organizational tools provided by a project management system. This is a set of methods that can be used to fulfill all tasks. All project management methods have three main goals: to increase employees' efficiency, to make the process of project management more productive and efficient, and to make the management of the project's profile more convenient and transparent to understand.

There are different methodologies for project management but they can be divided into two main approaches: the traditional waterfall model and Agile models. The choice of a suitable methodology depends on the specifics of the project, organization, and the team that will work on a particular project. Project management systems allow one to form a single approach to project management, follow the stages of its implementation at different levels, and control budget and deadlines.

Developing and implementing software is not an easy task as it involves many steps and aspects. Today, many development teams solve this problem in a unique way: The software development process does not end with the implementation of the product. Instead, the development team constantly monitors the software to make sure it works properly and meets the user's needs. If bugs or errors are identified in post-production, developers fix them. To prevent regression, the team re-submits the software to a shortened software development process.

Penetration tests (pentests) are one of the most reliable methods of assessing the security of the software IT environment. Simulated attacks allow one to detect vulnerabilities and take corrective actions.

Vulnerability management is a cyclical process that involves identifying, classifying, mitigating, and managing vulnerabilities. This process includes the identification of vulnerabilities, risk assessment, and the implementation of appropriate measures to mitigate threats.

To enhance security, it is recommended to implement best practices for managing vulnerabilities. These include conducting regular scans of all devices and endpoints within your ecosystem, assigning responsibilities, maintaining records, prioritizing tasks, providing training, setting standards, ensuring smooth operations, increasing program visibility, and tracking progress. By adhering to these practices, organizations can effectively manage vulnerabilities and enhance their overall security posture.

UNIT 7

DEVSECOPS

STUDY GOALS

On completion of this unit, you will be able to ...

- understand the basics of DevOps and DevSecOps.
- identify challenges for information security in the cloud.
- identify possible tools for automated security tests.

7. DEVSECOPS

Introduction

Increasingly shorter development cycles are becoming prevalent in software development. Therefore, development (“Dev”) and operations (“Ops”) teams are being combined into a DevOps team to respond more quickly to customer requests. The continuous consideration of information security (“Sec”) aspects during development and operations leads to the extension of the DevOps approach to DevSecOps, which is the focus of this unit (Kim et al., 2016).

A statistic on DevSecOps from GitLab showed that in 2023, 56 percent of respondents are using DevOps or DevSecOps. This represents an increase from 2022, where the figure was still slightly less at 47 percent (GitLab, 2023).

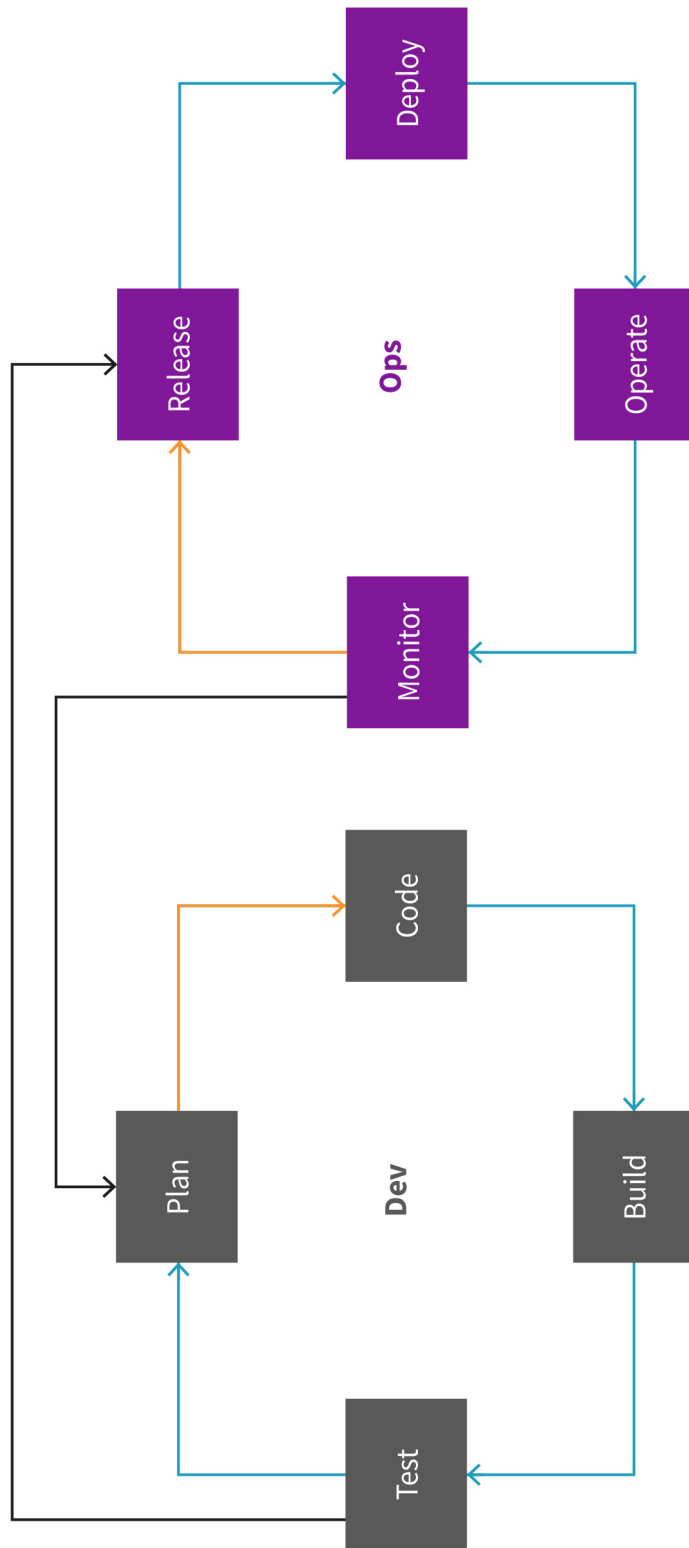
In this unit, we will first explain the DevOps approach. Since development under this approach is increasingly taking place in the cloud, the second section deals with security aspects that are important in the cloud. Software development with all the process components of a CI/CD (continuous integration/continuous deployment or delivery) pipeline is explained in the third section. Such pipelines have far-reaching accesses, so it is important to achieve a good level of access security with the help of short-lived tokens. The final section discusses automation options and possible tools for this purpose.

7.1 DevOps

In the past, different teams were often responsible for the development and operation of software. With the advent of agile software development toward DevOps, this separation was removed and integrated into one team (Hüttermann, 2012, p. 23). The merging is intended to achieve competitiveness through shorter development cycles and faster attention to customer needs (Krief, 2019, p. 4).

The figure below outlines this integration to DevOps. The bottom diagram shows phases of software development: plan, code, build, and test. The top diagram shows the phases of operations: release, deploy, operate, and monitor. The upper black arrows connect the phases of development (Dev) with those of operation (Ops).

Figure 22: DevOps Process



Source: Petra Beenken (2024), based on Mulder (2021).

Software development according to the DevOps model is subdivided into various phases, some of which are presented with different degrees of granularity in the literature. A software project generally starts with the definition of requirements and then the specification of the functions required for this (the “Plan” phase in the figure above). In the next step, these functions are designed and implemented, and the source code is checked in (“Code” phase), usually over an administration tool (repository). The source code is then compiled (“Build” phase) and any necessary software libraries are integrated. In the last step of the software development, the provided software is tested (“Test” phase). If tests are successful, the software is published (“Release” phase) and installed in the target system (“Deploy” phase). The updated software is then put into operation (“Operate” phase) and monitored (“Monitor” phase) with regard to performance, among other things (Mulder, 2021, p. 19).

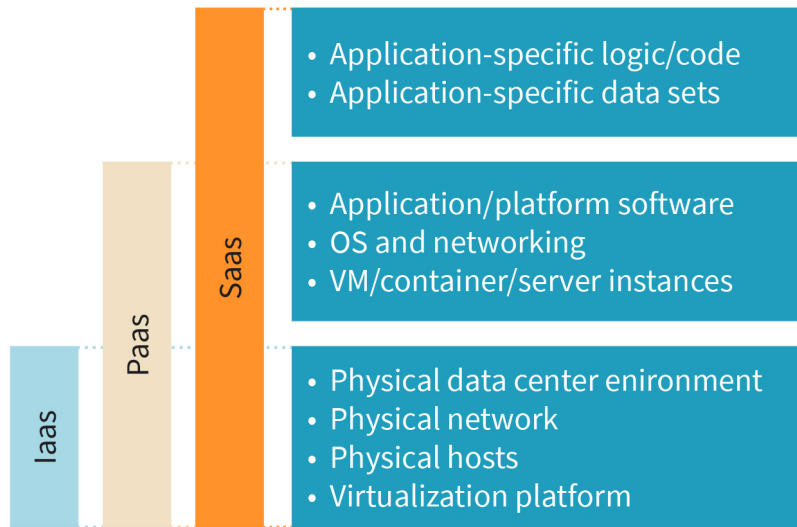
For DevOps to evolve into DevSecOps, security aspects are integrated into every phase. This starts with the integration of threat analyses. In the planning phase, a threat model can already be taken into account in order to set the focus of the required security requirements and implementations. In the next step, where source tests are written (“Code” phase), source code analyses can contribute significantly to improving the source code. Static source code scanners (static application security testing [SAST]) analyze the software without executing it, while dynamic scanners (dynamic application security testing [DAST]) check the behavior of the application during runtime. An analysis of the software components can detect any vulnerabilities of third-party software libraries. A vulnerability scan can also refer to infrastructure components or platforms that are used. These scans can be scheduled in parallel with the build phase. In the test phase, it makes sense not only to test the software functions themselves, but also to perform penetration tests in the sense of DevSecOps. Several security aspects can also be integrated during operation. For example, a compliance check can also be performed for a release (e.g., to verify compliance with any data protection requirements). Before the code is published (deploy phase), it could be digitally signed to prove its authenticity. Moreover, monitoring tools can help to detect attacks (operate and monitor phases).

In principle, software can also be deployed on-premises after DevOps, and software development often takes place in the cloud (Davis & Daniels, 2016, p. 41). For this reason, security aspects that are relevant in connection with software development in the cloud are described in the next subsection.

7.2 Cloud Security

Using a cloud component is technically the use of someone else’s computer. There are different stages of outsourcing into the cloud, which can be differentiated in different ways, as shown in the figure below. If you buy an infrastructure as a server (IaaS), you don’t have to worry about physical security aspects because the entire hardware is somewhere else and the IaaS provider is responsible for securing the physical infrastructure. However, you still need to deal with the operating system and network security by yourself.

Figure 23: Cloud Service Models



Source: Petra Beenken (2024), based on Dotson (2019, p. 8).

If you also want a service for this, a platform as a service (PaaS) provider can take on this responsibility, meaning that you would only need to worry about the middleware and application security. If you want this to be outsourced, a software as a service (SaaS) cloud model will take the responsibility for middleware and application security. The figure above show these different service levels (IaaS, PaaS, and SaaS). All of these are service models for **cloud computing**. If you don't want to build everything on-premises by yourself, you can choose a cloud service. Depending on the cloud service model you choose, you will receive physical servers, networking, storage solutions, or software over the internet (Dotson, 2019). An example for IaaS is Amazon Elastic Compute Cloud (Stallings & Brown, 2018).

Cloud computing enables the use of configurable computing resources (e.g., networks, servers, storage, applications, and services; Stallings & Brown, 2018).

There are different types of cloud: public, private, and hybrid. The difference between them is based on the type of infrastructure, ownership, access, and cost of the cloud services. Public cloud is an environment made available over the internet that anyone can subscribe to and then access. Public cloud offers scalability, reliability, and cost-efficiency, but less control and security. Examples of public cloud providers are Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP). Private cloud is the infrastructure used by just one organization. Private cloud offers more control and security, but less scalability, reliability, and cost-efficiency. Private cloud can be hosted on- or off-premises by a third-party provider. Hybrid cloud is a combination of the two other types, offering the best of both worlds, as it allows users to leverage the benefits of public and private cloud according to their needs and preferences. Hybrid cloud can also enable data and application portability, integration, and management across different clouds (Dotson, 2019, p. 124).

However, while cloud computing has a lot of benefits, it also faces some security challenges. Cloud security is the set of policies, procedures, technologies, and controls that protect data and systems in the cloud from unauthorized access, use, disclosure, modifi-

cation, or destruction (Hamburg & Grosch, 2018). Cloud security faces many problems, including misconfiguration, unauthorized access, insecure interfaces, or hijacking of accounts.

Misconfiguration is the incorrect or improper setting of cloud resources or services, such as user access controls, encryption keys, firewall rules, etc. Misconfiguration can expose cloud data or systems to unauthorized or malicious access or manipulation. For example, in 2019, Capital One suffered a data breach that affected 100 million customers due to a misconfigured firewall in its AWS cloud (Puzas, 2023a).

Unauthorized access is the act of gaining access to cloud data or systems without permission or authorization. Unauthorized access can result from weak or compromised credentials, phishing attacks, and insider threats. Unauthorized access can lead to data theft, tampering, or destruction. For example, in 2018, Tesla's AWS cloud was hacked by cryptojackers who used its computing power to mine cryptocurrency (Check Point, n.d.).

Insecure interfaces are the points of interaction between cloud users and cloud providers, such as web portals, application programming interfaces (APIs), and software development kits (SDKs). Insecure interfaces can have vulnerabilities or flaws that can be exploited by attackers to bypass security measures or perform malicious actions. For example, in 2017, OneLogin's identity and access management service was breached through an insecure API that allowed attackers to access customer data (Puzas, 2023a).

Hijacking of accounts is the act of taking over cloud user accounts or services by stealing or guessing their credentials, tokens, keys, etc. Hijacking of accounts can allow attackers to impersonate legitimate users or services and access their data or resources. For example, in 2014, Code Spaces' AWS cloud was hijacked by an attacker who demanded a ransom and deleted all its data and backups when the company refused to pay (Puzas, 2023a).

These risks or problems have to be taken into account and addressed through mitigating security measures in order to use cloud computing securely. Mitigation is possible, for example, through appropriate access control, encryption, malware prevention, malware behavior identification, and further risk assessments (Stallings & Brown, 2018).

7.3 Continuous Integration, Testing, and Deployment

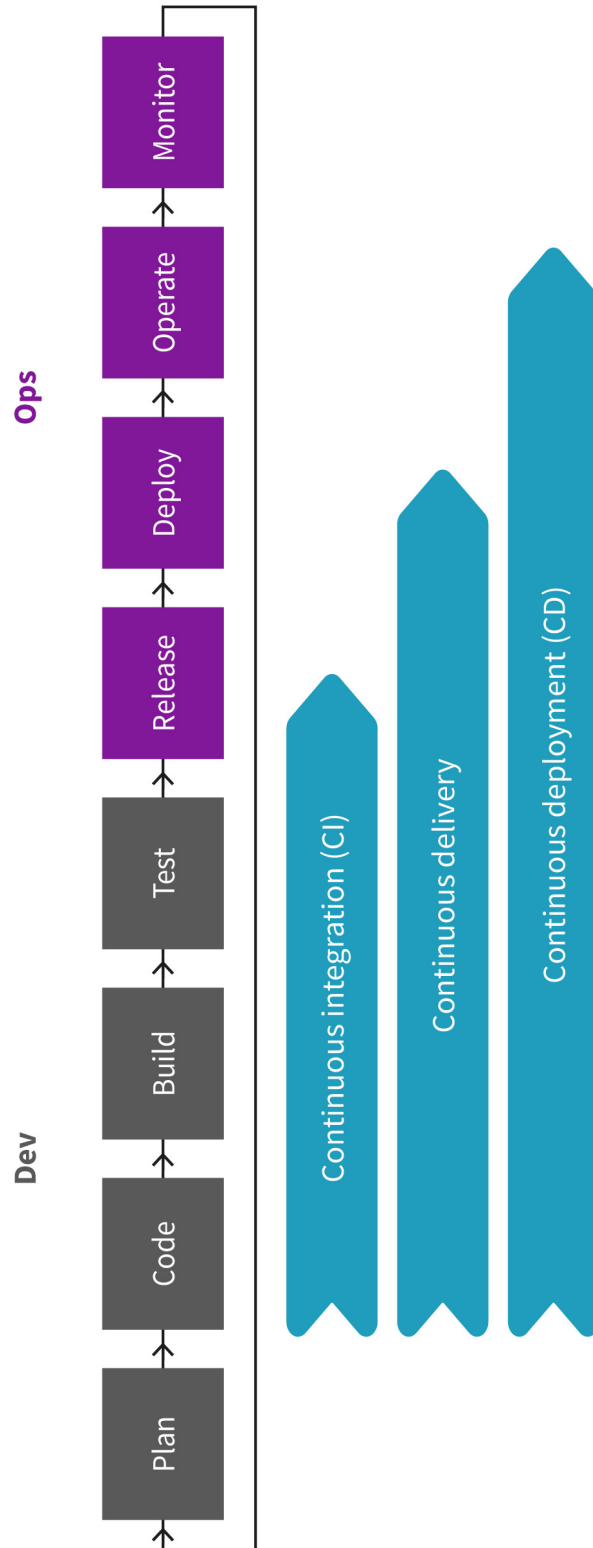
Continuous integration

is a practice where the codebase is continually updated (Conklin et al., 2021, p. 751).

Continuous integration (CI), testing and deployment (CI/CD) is a set of practices that aims to automate and streamline the software development life cycle (SDLC). CI allows testing of even minor changes because its automation means that it doesn't cost organizations too much overhead. Continuous delivery enhances CI so that minor changes go further from releases to production, while continuous deployment enhances continuous delivery, as it is "continuous delivery on autopilot" (Conklin et al., 2021, p. 751). Continu-

ous deployment sends code production without human intervention (Conklin et al., 2021). Continuous deployment and continuous delivery are both abbreviated to “CD” (Davis & Daniels, 2016).

Figure 24: CI/CD Pipeline in DevOps



Source: Petra Beenken (2024).

CI/CD enables developers to deliver software **faster more reliably and with** fewer errors. However, CI/CD also introduces new challenges and risks for software supply chain security, which is the process of ensuring the integrity and quality of software components from source to deployment. Software supply chain security involves protecting software from malicious attacks, unauthorized modifications, vulnerabilities, and defects that could compromise its functionality, performance, or safety. Some of the threats to software supply chain security include code injection, dependency hijacking, build tampering, deployment spoofing, and configuration drift. To address these threats, software developers need to adopt security best practices throughout the CI/CD pipeline, such as code scanning, dependency management, artifact signing, deployment verification, and configuration management. By integrating security into the CI/CD process, software developers can improve their software quality, reduce their attack surface, and enhance their trustworthiness.

Some of the popular CI/CD tools are Jenkins and GitLab. Jenkins is an open-source automation server that supports various plugins and integrations (Jenkins, n.d.). GitLab is a web-based platform that provides end-to-end CI/CD services along with Git repository management (GitLab, n.d.).

There are many other CI/CD tools available, which can be compared based on specific needs and preferences. Choosing a CI/CD tool for the project depends on various factors:

- **functionality:** The tool should support the features and workflows that are needed for the project, such as parallel testing, deployment strategies, and pipeline visualization.
- **extensibility:** The tool should be able to integrate with other tools and platforms that are used, such as code repositories, cloud services, and testing frameworks.
- **programming languages and platforms supported:** The tool should be compatible with the languages and platforms that are used for the project.
- **budget:** The tool should fit the budget of the project and offer a suitable pricing model, such as open source, pay-per-user, and pay-per-build.
- **convenience and ease of use:** The tool should be easy to set up, configure, and use, and offer a user-friendly interface and documentation.

CI/CD is part of GitLab itself and therefore has a good integration. Moreover, GitLab is a good option for version control repositories. An example of a GitLab CI/CD code could be as follows: In a file named “test.gitlab-ci.yml”, one can define each step of a pipeline and if this file is placed in the repository, GitLab will detect it and start the defined pipeline in the file. In the following code example, one can discern two stages. The first stage is a test; if the test goes well, the next stage will be to deploy. The test stage includes the pytest script. This runs during the test stage.

Example for the Use of GitLab CI/CD

```
``yaml
stages:
  - test
  - deploy

test:
```

```
stage: test
script: pytest

deploy:
  stage: deploy
  script: echo "Deploying new pytest"
  environment: production
...`
```

Petra Beenken (2024).

The code above shows a very basic example. GitLab CI/CD can also be used to automate many different tasks such as building Docker images, deploying to Kubernetes clusters, and much more.

7.4 Ephemeral Processes

Ephemeral processes are those that are short-lived and transient, such as temporary files, sessions, or keys. In terms of security, ephemeral processes **can have some advantages and disadvantages**. They can enhance security by reducing the exposure of sensitive data or keys to attackers, since they are deleted or replaced after a short period of time or after a single use. Ephemeral processes can also reduce the risk of data leakage or corruption, since they do not persist on disk or memory and are less likely to be affected by malware, hardware failures, or human errors. Moreover, ephemeral processes can ensure the secure generation, distribution, and destruction of ephemeral keys, protecting the integrity and authenticity of ephemeral messages, and ensuring the availability and reliability of ephemeral services. However, ephemeral processes require careful management and monitoring, such as defining clear policies and procedures for their life cycle, implementing effective backup and recovery mechanisms, and detecting and responding to any incidents or anomalies involving them (Splunk, n.d.).

Ephemeral infrastructure is a term that refers to computing resources or components that are created dynamically and destroyed as needed, rather than being persistent and long-lived. Ephemeral infrastructure can offer benefits, for example, greater flexibility, scalability, and ease of management in cloud-based or other dynamic computing environments. Furthermore, benefits include reduced costs, risks, and complexity associated with maintaining and updating legacy or outdated infrastructure. Additionally, it is possible to have improved security, reliability, and performance by minimizing the exposure of sensitive data or keys, avoiding configuration drift or human errors, and ensuring consistent and reproducible deployments through ephemeral infrastructure.

Some examples of ephemeral infrastructure are described below:

- **containers**, which are isolated and lightweight units of software that can run applications without requiring a full operating system or dedicated hardware.
- **virtual machines**, which are software emulations of physical machines that can run multiple operating systems and applications on the same hardware.

- **infrastructure as code**, which is the practice of using code or scripts to define, provision, configure, and manage infrastructure in an automated and consistent way.
- **ephemeral environments**, which are temporary testing or staging environments that are created on demand for each code change or feature development (Splunk, n.d.).

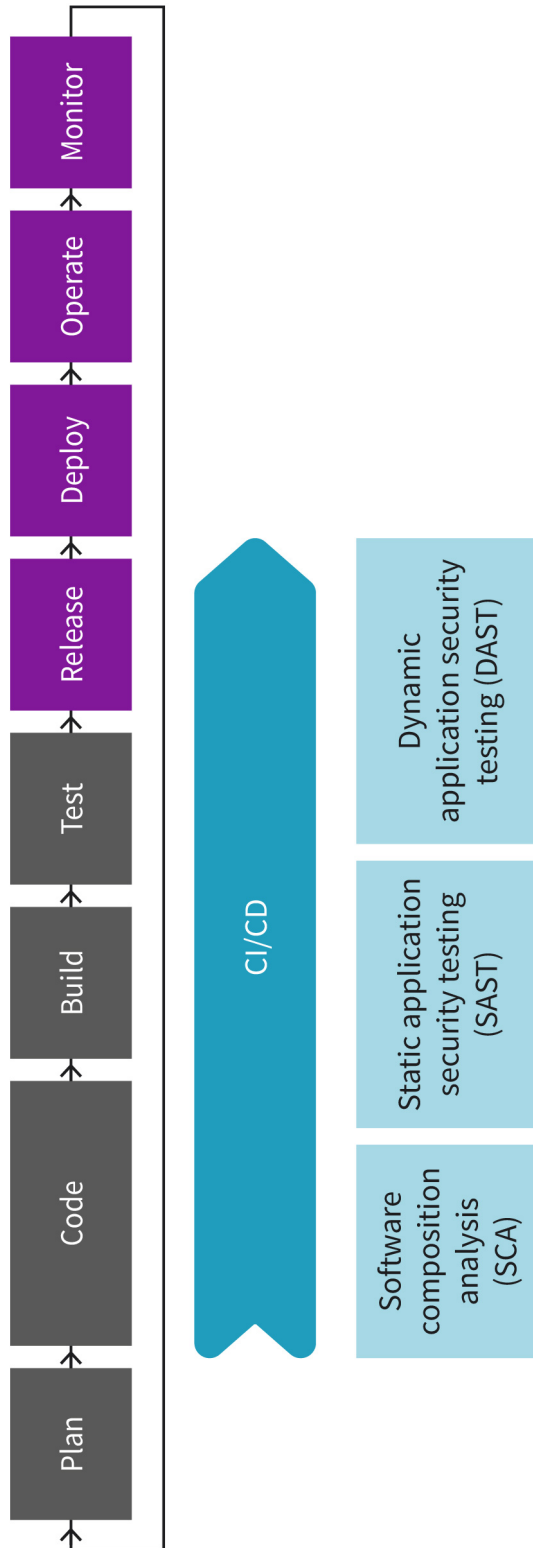
Secure communication in an ephemeral environment is a challenging task, since the communication parties may not have persistent identities, keys, or channels. Secure communication in an ephemeral environment is a challenging task, since the communication parties may not have persistent identities, keys, or channels. Some possible ways to implement secure communication in an ephemeral environment are the use of ephemeral keys or stateless protocols. Ephemeral keys are cryptographic keys that are generated on the fly and discarded after a single use or a short period of time. They can provide forward secrecy, which means that the compromise of a long-term key does not affect the security of past communications. Stateless protocols are communication protocols that do not require the parties to store or update any information about the state of the communication session. Stateless protocols can reduce the complexity and overhead of managing and synchronizing the state in an ephemeral environment.

A server that only exists for a short time is also called “Phoenix server.” It is so named because it is regularly destroyed and then lives again out of its ashes; therefore, it can be used for test runs. This reduces the attack risk because the server exists only for the life of the test run (Bell, Brunton-Spall et al., 2017, p. 281).

7.5 Automation

Automation in DevSecOps can improve the speed, quality, and consistency of security testing, and remediation. Automation is a key element in DevOps and security automation can gain the same for DevSecOps. By implementing security controls and procedures in an automated manner within the CI/CD pipeline, DevSecOps can be partly automated. Automation is done via scripting (Conklin et al., 2021, p. 750).

Figure 25: Security Automation in CI/CD Pipeline



Source: Petra Beenken (2024).

As shown in the figure above, you can embed security testing and scanning into the CI/CD pipeline, such as using SAST, DAST, or software composition analysis (SCA) tools to identify and fix vulnerabilities in code or dependencies. With SAST, you can automatically scan the source code for potential vulnerabilities, so the code is checked during the development. In the next stage, you can use DAST to automatically test running applications. In this way, you can find vulnerabilities in the productive version of the application.

Some best practices for implementing automation in DevSecOps are as follows. Companies should automate early and often, starting from the planning and design phase, to ensure that security is built into the code and not added as an afterthought. Companies should check their code dependencies regularly and use tools that can automatically scan and update them for security issues. A risk-based approach should be adopted, and one should prioritize the most critical and frequent security threats, rather than trying to address everything at once. It is a best practice to use tools and platforms that support security automation and orchestration, such as Red Hat OpenShift, Synopsys, or GitLab, and integrate them with existing CI/CD pipeline. Companies should also perform threat modeling to identify and mitigate the potential attack vectors and scenarios for your application and use tools that can automate or simplify this process. Last but not least, companies should train developers on secure coding practices and standards and use tools that can provide feedback and guidance on how to fix security issues in code (Kohgaidai & Oram, 2022).



SUMMARY

DevSecOps is a paradigm that integrates security into the DevOps culture and practices, aiming to deliver secure and reliable software faster and more efficiently. Cloud security is the set of policies, controls, and technologies that protect the data, applications, and infrastructure of cloud computing from cyber threats. Ephemeral processes are those that run for a short period of time and then terminate, leaving no trace or state behind. Ephemeral processes can be used to implement secure communication in an ephemeral environment, where the communication parties do not have persistent identities, keys, or channels. DevSecOps, cloud security, and ephemeral processes are related topics that can benefit from automation, orchestration, and collaboration.