V. M. Savchenko and O. V. Mnushka

MODERN TECHNOLOGIES OF SECURE PROGRAMMING

MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE

NATIONAL TECHNICAL UNIVERSITY
“KHARKIV POLYTECHNIC INSTITUTE”

V. M. Savchenko and O. V. Mnushka

MODERN TECHNOLOGIES OF SECURE PROGRAMMING

Educational and Methodological Guide
for Independent Study by Second (Master’s) Level Students
of Full-Time, Part-Time, and Dual Study Forms
in Specialty 123 “Computer Engineering”

Approved by
the Editorial and Publishing
Board of the NTU “KhPI”,
Protocol No. 1 dated 13.02.2025.

Kharkiv
NTU “KhPI”
2025

UDC 004.42/.49:004.056(075)
S26

Reviewers:

V. D. Kovalov, Doctor of Technical Sciences, Professor, Rector of Donbas State Machine-Building Academy, laureate of the State Prize of Ukraine in Science and Technology;

K. A. Trubchaninova, Doctor of Technical Sciences, Professor, Ukrainian State University of Railway Transport

Savchenko V. M.

S26 Modern Technologies of Secure Programming: Educational and Methodological Guide for Independent Study by Second (Master's) Level Students of Full-Time, Part-Time, and Dual Study Forms in Specialty 123 "Computer Engineering" / V. M. Savchenko, O. V. Mnushka. – Kharkiv: NTU "KhPI", 2025. – 102 p.

ISBN 978-617-05-0554-5

This educational and methodological guide covers key concepts of cybercrime, methods of protection against threats, and secure programming standards. It contains the essential theoretical minimum and assignments for students' independent work.

Intended for independent work by second (Master's) level students in full-time, part-time, and dual forms of study in specialty 123 "Computer Engineering".

Fig. 14. Tab. 8. Refs. 36 titles.

UDC 004.42/.49:004.056(075)

ISBN 978-617-05-0554-5
DOI 10.20998/978-617-05-0554-5

CONTENTS

Кафедра **«Комп'ютерна інженерія і програмування»** НТУ «ХПІ»
Department of **Computer Engineering and Programming** NTU «KhPI»

123 COMPUTER ENGINEERING
The program encompasses the study of computer hardware and software architecture, computer circuitry, the development of local and global network components, computer games, data mining, web development, and front-end design, with a strong emphasis on the mathematical foundations of information processing.

123 COMPUTER ENGINEERING
This specialty combines software engineering, computer science, and the engineering of computer components and systems. Computer engineering is a discipline that requires an in-depth understanding of the theoretical foundations of information technology, engineering principles, and computer science.

MODERN PROGRAMMING, MOBILE DEVICES AND COMPUTER GAMES
Training specialists in the development of software, hardware, and integrated systems in the computer industry, robotics, control, and process management, as well as for creating computer games, is a creative and complex process. It demands expertise in programming, algorithms, hardware, and networking to produce unique and widely popular solutions. Educational trajectories include: «***Programming of Mobile Devices and Computer Systems***» and «***Innovation Campus***»

APPLIED COMPUTER ENGINEERING
Training specialists in designing websites and web applications, including their interfaces, involves developing creative site concepts, creating site designs and page layouts, integrating multimedia objects, programming functional tools, or integrating them into a content management system. It also includes optimizing and managing site materials, testing and maintaining websites, hosting project publications, 3D modeling, and developing software and information for smart objects. Educational trajectories include: «***Web Design and Internet Programming***» and «***Engineering of Secure Systems and Networks***»

INTRODUCTION

The development of modern software requires not only a profound understanding of methods for ensuring its quality and efficiency, but also awareness of security issues at all stages of the life cycle – from design to maintenance. Mastering the principles of secure programming is an important task for future specialists in the field of *computer engineering*, and its successful assimilation largely depends on the student's independent work.

The eductional and methodological guide "Modern Technologies of Secure Programming" has been created to facilitate self-study. It focuses on developing practical skills for working with scientific publications, professional literature, and on applying methods, tools, and technologies for secure software development. The guide covers methods of analyzing vulnerability databases, as well as principles of coding, hashing, encryption, and the use of digital signatures.

Independent study of this guide enables students to examine in detail the issues of analyzing common application security threats based on the *OWASP Top 10* rating. This rating compiles generalized research findings on software vulnerabilities in recent years and is recognized regardless of technology or programming language. Understanding the mechanisms by which threats emerge and methods to eliminate them fosters the ability to design secure software in accordance with the *Secure Software Development Life Cycle (SSDLC)* principles. Working independently with vulnerability databases (*Common Vulnerabilities and Exposures (CVE)*) gives students experience in searching for, analyzing, and summarizing information about potential threats to software systems, as well as examining the history of software development.

Additionally, the guide discusses the basics of working with code analysis tools (*YARA*, *radare2*, etc.) and introduces the foundations of cryptography, particularly the use of cryptographic libraries like *OpenSSL*, along with the implementation of *HMAC* (Hash-Based Message Authentication Code) and *electronic digital signatures (EDS)*. Independent study of these materials enables students to learn how to implement hashing and digital signature mechanisms, which form the cornerstone of modern information security.

Topic 1
INTRODUCTION TO THE COURSE

The objective is to become familiar with the basics of secure programming and gain practical skills in installing and configuring toolchains.

Tasks for Independent Study

1. *Install and configure Oracle VirtualBox* for working with virtual machines (create a new virtual machine, select the desired operating system (OS), configure the network).
2. *Install QEMU* and run a virtual machine, compare its performance with *VirtualBox* (use basic settings).
3. *Create a snapshot* in *VirtualBox* or *QEMU* to enable quick recovery of the virtual machine's state.
4. *Explore basic commands* for managing virtual machines (`start`, `stop`, `pause`, `clone`, etc.) in *VirtualBox* and *QEMU*.
5. *Prepare a test environment*: install all necessary tools (compilers, interpreters, networking utilities) in the virtual machine for upcoming tasks.

Brief Theoretical Background

1. General overview of the course.
2. Cybercrime and cybercriminals.
3. Computer security and technical security.
4. The impact of generative artificial intelligence on secure programming technologies.

The course on modern secure programming technologies covers the basic principles and practices related to the development of software that minimizes vulnerabilities to cyber threats. Software often becomes a target for attacks from cybercriminals, making it essential to implement security mechanisms at all stages of development. The importance of a proactive approach to identifying and eliminating potential vulnerabilities is key to creating reliable software.

The course also examines *Software Development Life Cycles (SDLC)*, which include steps for implementing security such as risk analysis and vulnerability testing. Studying examples of attacks helps to avoid similar situations in the future.

The following literature is recommended: [1–9]

Cybercrime and Cybercriminals

Cybercrime includes various activities such as unauthorized access to information systems, theft of confidential data, distribution of malicious software, DDoS attacks (denial-of-service attacks), and other online crimes. According to the report [10], the number of cybercrimes continues to grow rapidly. Losses from such crimes in 2022 are estimated to exceed 10 billion dollars, demonstrating the scale of the threats.

Cybercriminals actively exploit vulnerabilities in software, such as outdated systems or unsecured server configurations. One notable example was the *WannaCry attack* in 2017, which exploited a vulnerability in the *SMB protocol* to spread malware to thousands of computers in over 150 countries [11].

Secure programming plays a key role in reducing risks associated with cybercrime. *Proper system design* and regular software updates significantly reduce the likelihood that criminals can exploit vulnerabilities.

Modern cybercriminals use various approaches and techniques, among which the following can be highlighted:

• *Phishing* – distributing fake emails or using fake websites to steal user credentials. According to statistics, almost 1.2% of all sent emails are malicious, which means *3.4 billion phishing emails are sent daily*. It is expected that by 2024, over 33 million records will be stolen, and phishing attacks demanding ransom will occur every 11 seconds [1];

• *Supply chain attacks* – hacking software or hardware during the production process by exploiting trust between the manufacturing company and its clients. A well-known example is the *SolarWinds attack* in 2020, which affected thousands of organizations, including U.S. government agencies [12];

• *Exploitation of zero-day vulnerabilities*, which have not yet been discovered by developers and therefore do not have security patches. For instance, the *Microsoft Exchange attack* in 2021 exploited several zero-day vulnerabilities to steal confidential data [13].

Software development with a focus on security helps minimize the risks of such attacks. Implementing *code-level protection*, such as input validation, exception handling, and adherence to the principle of least privilege, significantly reduces system vulnerabilities.

Computer Security and Technical Security

In English, there are two terms “safety” and “security”, both of which can be translated as “безпека” in Ukrainian.

Computer security or security refers to a set of measures aimed at protecting

[1] Astra Security: 81 Phishing Attack Statistics 2024: The Ultimate Insight

information and computer systems from unauthorized access, modification, or theft of data. The main principles of computer security are known as the *CIA triad* (*Confidentiality*, *Integrity*, *Availability*):

• *Confidentiality* ensures that data is accessible only to those who have the right to access it. This is achieved through data encryption and proper access policies;

• *Integrity* guarantees that data cannot be altered without permission. Methods such as hashing and digital signatures allow verifying the integrity of the data;

• *Availability* ensures that information systems are accessible whenever needed. Protection against DDoS attacks and server setups with backups help maintain availability.

Technical security or safety includes hardware and software tools that help protect systems:

• *Network firewalls* protect systems from unauthorized network access by filtering incoming and outgoing traffic;

• *Antivirus and anti-spyware software* detect and remove malicious software;

• *Intrusion Detection Systems (IDS)* and *Intrusion Prevention Systems (IPS)* help detect and prevent attacks at early stages.

Specialists in the field of *computer engineering*, such as hardware and software developers, should implement these tools and apply modern encryption, authentication, and access control methods to ensure reliable protection of data and systems.

The Impact of Generative Artificial Intelligence on Secure Programming Technologies

Generative Artificial Intelligence (AI), especially based on models like *GPT-3* and *GPT-4*, has significant potential in programming due to its ability to quickly solve typical tasks and automatically generate code, which can significantly accelerate the development process. However, this *AI* can also generate code with vulnerabilities. For example, code generated by *GPT-3* and *GPT-4.x* without proper security measures may contain vulnerabilities such as *SQL injection* or *buffer overflow*.

Moreover, *AI* can be used to automate attacks, for example:

• *Deepfakes* can be used to create fake video or audio, which are then employed in phishing campaigns [14; 15].

• *AI* can help automatically scan applications for vulnerabilities.

• *AI* can generate exploits.

Therefore, one of the current challenges in the field of computer science and cybersecurity is the development of new approaches to software security, taking into account the capabilities and risks posed by generative AI. Research directions may include:

• *Automated code testing* to detect vulnerabilities;

• *AI models for detecting network attacks*, for example, to analyze network traffic and detect anomalous activity that may indicate an attack;

• *Continuous monitoring and training of AI models* for real-time updates to protection systems.

It is worth noting that a potential and serious issue with the use of *AI* is *data privacy*, which manifests in the use of user data to train *AI* models, as well as possible access to such data by developers and researchers.

• Most modern *AI* models, including large language models (such as *GPT*), need to be trained on very large datasets, including data collected from the Internet. This raises serious privacy concerns, as these datasets may contain personal or sensitive user information. There are numerous examples where *AI* models have generated content that included personal user data not intended for public use;

• Personal user data collected during the training of models may be accessible to developers or researchers, which violates privacy. According to studies [16; 17], machine models may "remember" fragments of confidential information;

• The terms of use of language models directly address the possibility of using private data for training. *OpenAI*, the developer of *GPT-4*, has a dedicated section discussing data privacy and security [2].

Thus, generative *AI* can be not only a powerful tool for developers across various fields but also a serious threat to cybersecurity and privacy.

Review Questions

1. What is *secure programming*, and why is it important?
2. How do different stages of the *SDLC* integrate with security methods?
3. What are the most common types of *cybercrime* today?
4. How do *cybercriminals* exploit software vulnerabilities to conduct attacks?
5. What are *zero-day vulnerabilities*, and how can they be avoided during software development?
6. What are the main principles of *computer security* that should be implemented during software development?
7. How do technical *security measures* help reduce *cyber threats*?
8. How does generative *AI* affect the process of developing *secure software*?
9. What are the risks of using generative *AI* for software development?
10. What *security measures* can be applied to protect against *AI*-driven attacks?

[2] https://openai.com/policies/privacy-policy/

Topic 2

OVERVIEW OF SECURE PROGRAMMING METHODS

The objective is to study the main approaches, methods, and principles of secure programming, as well as gain skills in source code analysis.

Tasks for Independent Study

1. *Analyze two open-source projects* and identify which secure programming methods are used (refer to *GitHub* repositories and examine security practices described in the `README` or `CONTRIBUTING` files).
2. *Write a short security guide with five key practices* for beginners (e.g., principle of least privilege, regularly updating dependencies, etc.).
3. *Compare static and dynamic code analysis* in the form of a summary table (consider tools, advantages, and disadvantages of both approaches).
4. *Develop a training scenario for a developer team* on implementing security methods (training plan, topics, code examples).
5. *Create a checklist for code security auditing* that can be used before each release (availability, presence of tests, use of linters, etc.).

Brief Theoretical Background

1. Secure programming methods.
2. Data protection principles and standards.
3. Legal and ethical aspects.

Secure Programming Methods

Secure programming includes several approaches and methods that help reduce the risks of vulnerabilities and attacks on software. One of the key principles of secure programming is input validation. All external data must be thoroughly checked to prevent code injections, such as *SQL injections*, or other types of attacks. It is important to consider the possibility of embedding code into commands or queries, especially for web applications where user input can become a source of malicious code, and data processing takes place, particularly on the server side.

Another important method is output data filtering. Correct encoding of data before its use or transmission to external systems reduces the risk of *XSS* (*Cross-Site Scripting*) attacks and other vulnerabilities related to unsafe output data.

The use of *cryptography* is also a fundamental element of secure programming. *Data encryption* ensures its confidentiality and integrity, especially during transmission over the network, provided that modern and reliable encryption algorithms, such as *AES* or *RSA*, are used [18].

In general, the following key secure programming methods are essential for specialists in the field of computer engineering:

- Input validation;
- Output data encoding;
- Authentication and password management;
- Session management;
- Access control and resource management;
- Use of cryptographic methods, algorithms, and protocols;
- Error handling and logging;
- Data protection;
- Data transmission security;
- Code review and static analysis;
- Code testing, including security testing (*pen-testing*).

Data Protection Principles and Standards

Secure programming must adhere to clearly defined principles and standards. For example, the *OWASP* (*Open Web Application Security Project*) standard provides developers with practical recommendations for identifying and mitigating the most common vulnerabilities in web applications. The *OWASP Top 10 list* (Table 2.1) contains the most common threats, such as *SQL injections* and *XSS*, and offers recommendations to prevent them [18].

OWASP Top 10 introduced some changes in the 2021 edition (Fig. 2.1[1]). The 2025 Top 10 is expected to be published in the first half of 2025.

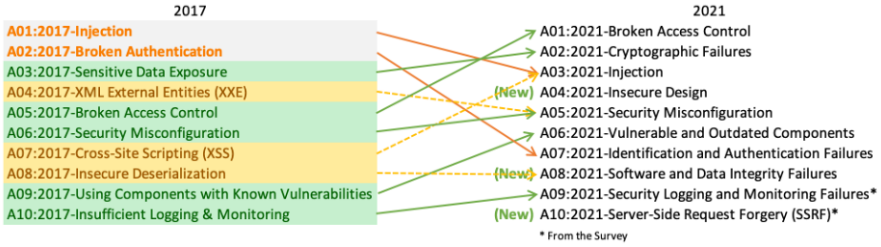

Figure 2.1 – Mapping of *OWASP Top 10* Categories (2021)

[1] https://owasp.org/Top10/assets/mapping.png

Таблиця 2.1 – Comparison of *OWASP Top 10* for 2017 and 2021

OWASP Top 10 2017	**OWASP Top 10 2021**
A01. Injections	A01. Access Control Failures
A02. Authentication Failures	A02. Cryptographic Failures
A03. Sensitive Data Exposure	A03. Injections
A04. XML External Entities (XXE)	A04. Insecure Design
A05. Broken Access Control	A05. Security Misconfiguration
A06. Security Misconfiguration	A06. Vulnerable and Outdated Components
A07. Cross-Site Scripting (XSS)	A07. Identification and Authentication
A08. Insecure Deserialization	A08. Software and Data Integrity
A09. Using Components with Known Vulnerabilities	A09. Security Logging and Monitoring Failures
A10. Insufficient Logging and Monitoring	A10. Server-Side Request Forgery (SSRF)

In addition to *OWASP*, the *SEI CERT* (*Software Engineering Institute's Computer Emergency Response Team*) provides guidelines for secure coding. *SEI CERT* develops standards for secure programming in various languages, such as *C*, *C++*, and *Java*, including the *SEI CERT C Coding Standard* [19], *SEI CERT C++ Coding Standard* [20], and *SEI CERT Oracle Coding Standard for Java* [21], which provide rules and recommendations for avoiding common security errors during code development. These standards cover issues such as memory management, buffer overflow prevention, file and process handling, exception handling, and more. The latest versions of these standards can be found on the *SEI* website [22].

These rules and recommendations, along with other standards such as *MITRE* and *ISO/IEC TS 17961:2013*, are used in static code analyzers like *Cppcheck* or *clang-tidy*, enabling the detection of potential issues at early stages of development.

The implementation of secure coding standards improves code quality, system security, and reduces the number of vulnerabilities in software [23].

Other important standards include *ISO/IEC 27001*, which sets requirements for information security management systems, and *PCI DSS* (*Payment Card Industry Data Security Standard*), which regulates the security of payment card data [24].

Legal and Ethical Aspects

Secure programming also involves important legal and ethical aspects. Laws in various countries, such as the *General Data Protection Regulation (GDPR)* in the European Union, require developers to adhere to strict requirements for protecting users' personal data [23].

Developers must not only comply with these laws but also act ethically. Respon-

sible behavior includes promptly fixing vulnerabilities and responsibly disclosing information about them. For example, ethical hacking (or white-hat hacking) helps improve software security by identifying vulnerabilities before they are exploited.

Review Questions

1. What are the key methods of *secure programming*?
2. Why is *input validation* necessary, and how does it affect security?
3. What is *output data filtering*, and what attacks does it help prevent?
4. What role does *cryptography* play in ensuring data security?
5. What is the *OWASP Top 10*, and what threats does it include?
6. What are the main recommendations of *SEI CERT* for *secure programming*?
7. How do legal regulations like *GDPR* affect *secure programming*?
8. Why are ethical principles important in the context of *secure programming*?
9. What are the main legal requirements for *data protection* in different jurisdictions?
10. How should developers act responsibly when vulnerabilities are discovered?

Topic 3

MEMORY CORRUPTION AND BUFFER OVERFLOWS

The objective is to study vulnerabilities related to buffer overflows and memory management errors, and to gain practical skills in analyzing software code for security flaws.

Tasks for Independent Study

1. *Find an example of buffer overflow code* (in *C* and *C++*) and fix it, explaining where the error was and how to prevent it.
2. *Write a program in C and C++* that demonstrates array bounds checking (use index validation when accessing array elements).
3. *Test your code using Valgrind* to detect *memory leaks* and *buffer overflows*.
4. *Explore how ASLR (Address Space Layout Randomization)* helps prevent buffer overflow exploitation (explain the mechanism and its advantages).
5. *Create a presentation on the most common memory management errors* and provide recommendations on how to avoid them.

Brief Theoretical Background

1. Memory corruption.
2. Buffer, stack, and heap overflows.
3. Static and dynamic analysis methods.

Memory Corruption

Memory corruption is one of the most severe security vulnerabilities caused by improper memory management in software. Such vulnerabilities can lead to unpredictable program crashes or allow attackers to execute attacks, including arbitrary code execution or privilege escalation. Memory corruption commonly occurs due to access to memory that was either not correctly allocated or already freed. This leads to typical vulnerabilities such as *buffer overflow*, *stack overflow*, and *heap overflow*.

These vulnerabilities can be exploited due to flaws in programming libraries or runtime libraries of operating systems, such as *libc* or *libc++*. For example, classic buffer overflow attacks allow attackers to overwrite memory data, which can result in the execution of malicious code at the operating system level. Attacks like *Return-Oriented Programming (ROP)* or *Jump-Oriented Programming (JOP)* manipulate the

sequence of program instructions to carry out malicious actions, potentially giving full control over the system.

Memory corruption attacks are among the most dangerous because they allow attackers to manipulate critical data or alter the program's execution flow. This makes the system vulnerable to exploits that can be used for executing privileged code or leaking confidential data. Many modern security systems, such as *Data Execution Prevention (DEP)* and *Address Space Layout Randomization (ASLR)*, were developed to prevent such attacks. However, bypassing these protections is possible in the case of complex exploits that combine multiple vulnerabilities.

Buffer, Stack, and Heap Overflow

When a function is called, the stack is used to store additional arguments that do not fit into the allocated registers (Fig. 3.1[1]). The stack also always stores the `return address`, which allows the function to return to the point of the call after it completes. The stack is aligned to 16 bytes to ensure the correctness of function calls and data access.

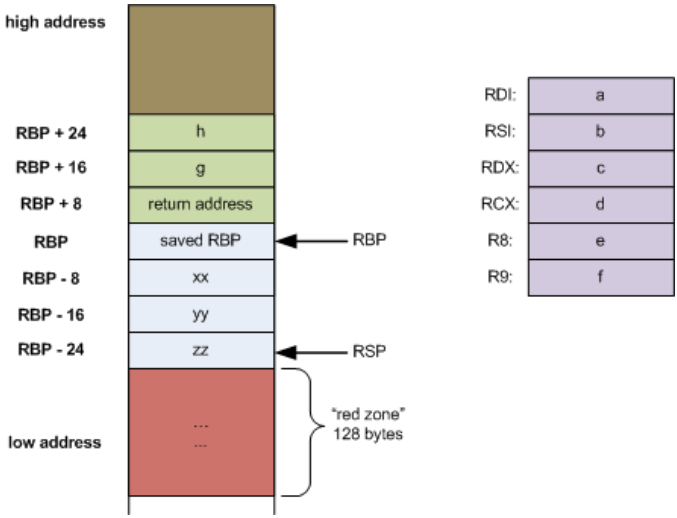

Figure 3.1 – *x86_64 Stack* (*Linux*)

Calling convention defines how function arguments are passed and how functions return values at a low level in programming. In *Windows x64* (Microsoft *ABI*), the

[1] Eli Bendersky. Stack frame layout on x86-64

first 4 arguments are passed via the registers `RCX`, `RDX`, `R8`, `R9`, and the rest are passed via the stack. Additionally, 32 bytes of "shadow space" are allocated on the stack to store these registers, even if they are not used. Return values are passed via `RAX` for integers and `XMM0` for floating-point numbers. The stack is aligned to 16 bytes before a function call.

Windows x64 (Microsoft ABI)

1. The first 4 integer or pointer arguments are passed via `RCX`, `RDX`, `R8`, `R9`;
2. Arguments after the 4th are passed via the stack;
3. 32 bytes of "shadow space" are allocated on the stack for the first 4 arguments, even if they are not used;
4. Return values: `RAX` for integers and `XMM0` for floating-point values;
5. The stack is aligned to 16 bytes before a function call.

Linux x64 (System V ABI)

1. The first 6 integer or pointer arguments are passed via `RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9`.
2. Arguments after the 6th are passed via the stack.
3. No "shadow space".
4. Return values: `RAX` for integers and `XMM0` for floating-point values.
5. The stack is also aligned to 16 bytes.
6. In Linux, a *128-bit red zone* is used below `RSP`, which can be utilized for temporary variables without modifying `RSP`. It is not used during function calls.

Stack overflow is a common type of vulnerability related to improper memory management, but it is not directly caused by *buffer overflow*. *Stack overflow* occurs when the amount of data or the number of functions stored in the stack exceeds its size, which can lead to logic execution errors or security attacks, particularly by overwriting return pointers (*return pointers*). This can allow attackers to gain control over the program's execution flow.

Buffer overflow is a vulnerability where data written to a buffer exceeds its size, causing an overwrite of memory outside the allocated space (Fig. 3.2[2]). This can lead to changes in critical program structures such as variables, pointers, or return addresses, which can be exploited for attacks such as *arbitrary code execution*. Such vulnerabilities typically pose significant security threats, including *stack smashing* or *heap corruption* attacks.

Heap overflow occurs in programming languages like *C* and *C++*, which allow direct memory access. The heap is used for dynamic memory allocation during program execution, and improper memory management (e.g., failure to correctly allocate

[2] Wallarm: What is a Buffer Overflow Attack?

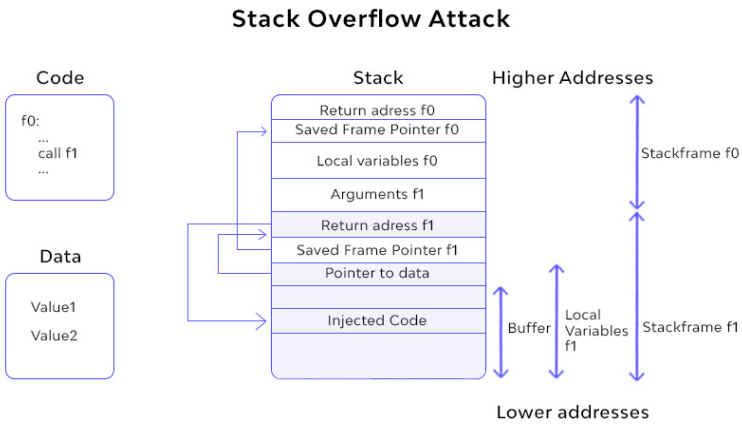

Figure 3.2 – Stack Overflow

and free dynamic memory) can lead to vulnerabilities such as *use of uninitialized pointers* or *double free* (releasing memory more than once).

Common types of attacks:

• *Stack smashing* – a method where an attacker alters the stack content via buffer overflow, potentially leading to the execution of malicious code.

• *Heap corruption* – the corruption of data integrity in the heap, typically due to improper dynamic memory management.

• *ASLR* (Address Space Layout Randomization) and *Stack Canaries* are protection mechanisms that reduce the risk of successful stack and buffer overflow exploits.

Stack Overflow Example

Consider a simple *C function* example that causes a stack overflow because all local variables and return addresses are typically stored in the stack during function execution:

```
void vulnerable_function() {
        char buffer[10]; // C-string
        gets(buffer);    // unsafe function
    }
```

If the user inputs more than 10 characters, this will cause a buffer overflow,

potentially allowing an attacker to overwrite the return pointer and execute malicious code.

Heap Overflow Example

In the case of heap overflow, an attacker can exploit improper dynamic memory allocation to alter critical data in the program. For example:

```
void vulnerable_heap() {
        char *buffer1 = (char *)malloc(10);
        char *buffer2 = (char *)malloc(10);
        strcpy(buffer1, "123456789012345");  // buffer1 overflow
        free(buffer1);
    }
```

Here, the `strcpy()` function writes more data into `buffer1` than was allocated, potentially corrupting other variables or pointers in the heap.

Note that the *C* and *C++* language standards do not ignore the issue of buffer overflow. Some unsafe functions, such as `gets`, were removed in the *C11 standard* due to their inherent dangers. Other functions have "safe" variants with buffer size control. For example, instead of `strcpy`, it is recommended to use `strncpy`, and instead of `sprintf`, `snprintf`, which allow specifying the maximum buffer size and prevent memory overwrites beyond the buffer's boundaries (Table 3.1[3]).

Additionally, some functions have been marked as *deprecated*. For instance, in the *C++ standard*, `gets` was also marked as deprecated and removed due to the inability to control the input size. Functions like `strcat` can also be unsafe, and it is recommended to use their "safe" variants (`strncat`) to prevent buffer overflow.

The *C11* and *C++11* standards significantly improved memory management security by introducing new safe functions and removing old, unsafe approaches to string handling.

Static and Dynamic Analysis Methods

To detect and prevent memory corruption and overflows, both static and dynamic analysis methods are used. *Static analysis* involves inspecting the program's source code without executing it. Static analysis tools can identify potential issues such as unsafe memory management or the use of unchecked data. This allows errors to be corrected before they lead to vulnerabilities in real-world environments.

[3]List depends on standard and compiler version

Таблиця 3.1 – Unsafe functions and their safe counterparts in*C* and *C++*

Unsafe function	ANSI C and C++	Microsoft VS C and C++
`strcpy`	`strncpy`	`strcpy_s`, `strlcpy`
`strcat`	`strncat`	`strcat_s`, `strlcat`
`sprintf`	`snprintf`	—
`vsprintf`	`vsnprintf`	—
`gets`	`fgets`	`gets_s`
`makepath`	—	`_makepath_s`
`_splitpath`	—	`_splitpath_s`
`scanf`	—	`sscanf_s`
`sscanf`	—	`sscanf_s`
`snscanf`	—	`_snscanf_s`
`strlen`	—	`strnlen_s`

Dynamic analysis, unlike static analysis, is performed while the program is running. This method helps to identify issues in real-time, such as improper memory deallocation or the use of corrupted pointers. Tools like *Valgrind* or *AddressSanitizer* are widely used to detect memory corruption issues during software testing. Combining static and dynamic analysis is an effective approach to creating secure programs.

Review Questions

1. What is *memory corruption*, and what are its primary causes?
2. How can *memory corruption* be exploited by attackers to compromise software?
3. What is the difference between *stack overflow* and *heap overflow*?
4. How can *buffer overflow* lead to security breaches in a program?
5. What safe *memory management* practices can prevent *memory corruption*?
6. What is *static analysis*, and how does it help in identifying software vulnerabilities?
7. What tools are used for *static code analysis*?
8. What is *dynamic analysis*, and in what scenarios is it effective?
9. What *dynamic analysis* tools help detect *memory management* issues?
10. How does combining *static* and *dynamic analysis* enhance software security?

Topic 4

CODE INJECTIONS

The objective is to analyze the nature of common code injection attacks (*SQL*, *XSS*, etc.) and methods of their prevention, as well as to gain practical skills in detecting code injections.

Tasks for Independent Study

1. *Write an SQL query using parameterized inputs (Prepared Statement)* to prevent *SQL* injection attacks.
2. *Analyze web server logs* (e.g., *Apache* or *Nginx*) and identify potential *SQL* injection attempts.
3. *Fix a vulnerable Python script* that is susceptible to command injection (review how system commands are executed and replace with a safer method).
4. *Demonstrate an XSS attack* on a test web application (create a simple page with a form that reflects user input).
5. *Develop an input validation filter* that blocks suspicious characters and checks input for validity before processing (use regular expressions, whitelisting, etc.).

Brief Theoretical Background

1. Overview of code injection.
2. *SQL* injections.
3. Security in scripting languages and shell environments.

Code Injections – General Overview

Code injections are one of the most common and dangerous types of attacks on software. These attacks occur when an attacker injects malicious code into an input field that has not been properly secured, and this code is executed by the system. The most common examples of such attacks are operating system *command injections* and *SQL injections*.

Operating system command injections (Fig. 4.1[1]) allow an attacker to execute unauthorized commands directly in the operating system, while *SQL* injections target databases by executing malicious *SQL* queries.

[1] MITRE: CWE-78

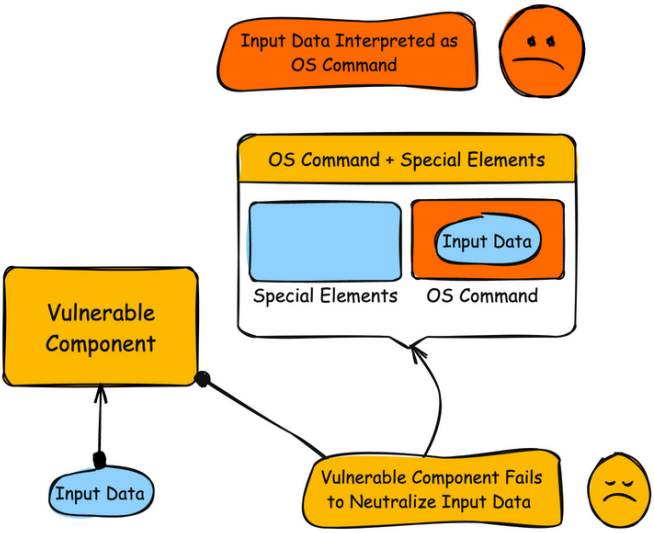

Figure 4.1 – *OS command injection*

An example of an operating system command injection could be a situation where a web application accepts user input to execute a system command, such as a ping request:

```
ping 127.0.0.1
```

If the input is not properly validated, an attacker may append their own malicious code to the command:

```
ping 127.0.0.1; rm -rf /
```

This would result in the execution of a file deletion command on the server.

SQL Injections

SQL injection is a technique where an attacker adds or modifies *SQL queries* via input in web forms or *URL parameters*, leading to the execution of unauthorized actions on the database (Fig. 4.2[2]).

[2] xkcd: Exploits of a Mom

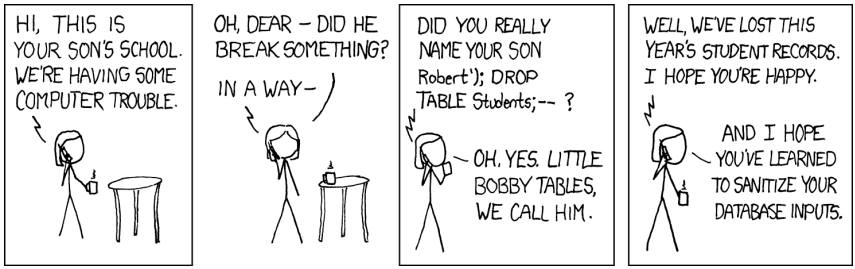

Figure 4.2 – *SQL Injection*

For example, if an SQL query looks like this:

```
SELECT * FROM users WHERE username = '$username' AND password =
↪  '$password';
```

An attacker can input the following code into the input field:

```
' OR '1'='1
```

This transforms the query into:

```
SELECT * FROM users WHERE username = '' OR '1'='1' AND password = '';
```

In this case, the query will return all records from the database because the condition will always be true. This allows the attacker to bypass authentication and gain access to confidential data.

To prevent *SQL* injections, it is essential to use parameterized queries and avoid directly embedding user input into *SQL* queries [3]. Here is an example of secure SQL usage in PHP with prepared statements:

```
$stmt = $pdo->prepare('SELECT * FROM users WHERE username = :username
↪  AND password = :password');
$stmt->execute(['username' => $username, 'password' => $password]);
```

[3] OWASP: SQL Injection Prevention Cheat Sheet

Now we can consider the example shown in Fig. 4.2. Suppose the following command is used to insert data into a table:

```
INSERT INTO Students (firstname) VALUES ('Elaine');
```

If the name is `name = 'Robert'); DROP TABLE STUDENTS; -- '` as shown in Fig. 4.2, the following query will be executed:

```
INSERT INTO Students (firstname) VALUES ('Robert'); DROP TABLE
↪  STUDENTS;  -- '
```

Which transforms into three SQL commands, resulting in the deletion of the `Students` table:

```
INSERT INTO Students (firstname) VALUES ('Robert');
DROP TABLE Students;
 -- ';
```

Security of Scripting Languages and Command Shells

Scripting languages such as *PHP*, *Python*, and *JavaScript* are often targeted for code injections due to their flexibility and wide use in web applications. The danger lies in the ability to execute malicious code when interacting with external data. For instance, a vulnerability to *XSS* (*Cross-Site Scripting*) can arise if unverified user input is displayed on a webpage.

Command injections through command shells like *Bash* or *PowerShell* can also lead to serious consequences. Attackers can gain access to the system by executing dangerous commands through vulnerable scripts. One way to protect against this is to avoid executing external commands directly, use functions to isolate user input, or eliminate command execution on the server side altogether.

Review Questions

1. What is *code injection*, and how is it used in attacks on software?
2. How do operating system *command injections* work? Provide an example.
3. What potential consequences can *SQL* injections have for databases?
4. How can you protect against *SQL* injections? What techniques are most effective?

5. What risks arise when working with scripting languages and *command shells*?
6. What are *Cross-Site Scripting (XSS)* attacks, and how are they related to *code injections*?
7. How can prepared statements in *SQL* prevent *code injections*?
8. What general approaches to *script security* should be used when interacting with user input?
9. What methods can be used to protect *command shells* from *injections*?
10. How can *code injections* be detected using *static* and *dynamic analysis*?

Topic 5

CONCURRENCY ISSUES (RACE CONDITIONS)

The objective is to study threats related to race conditions and ways to prevent them in multithreaded programs, and to gain practical skills in analyzing concurrent applications.

Tasks for Independent Study

1. *Find a race condition in Java code* and fix it using the `synchronized` keyword or other synchronization mechanisms.
2. *Write an example using mutexes* in *C*, *C++*, or *Python* to control access to a shared resource.
3. *Test a multithreaded application* (written in *C*, *C++*, or *Java*) using *ThreadSanitizer*[1] and analyze the results.
4. *Compare race condition prevention approaches in C++ and Python*, explaining key differences (e.g., synchronization primitives, *GIL* in *Python*, etc.).
5. *Create a thread interaction diagram* for a complex multithreaded scenario (e.g., task queue and worker thread coordination).

Brief Theoretical Background

1. General overview of concurrency issues.
2. Methods for preventing race conditions.
3. Consequences of concurrency-related problems.

General Overview of Concurrency Issues

Concurrency issues arise when multiple threads or processes simultaneously access the same resource (e.g., memory or a file) without proper synchronization. This can lead to incorrect program behavior and serious vulnerabilities, such as data corruption or leakage of sensitive information (Table 5.1).

TOCTOU (Time-of-Check to Time-of-Use) Attack

TOCTOU (Time-of-Check to Time-of-Use) is a class of vulnerabilities that arise in multitasking and multithreaded systems due to a window of time between the resource

[1] https://github.com/google/sanitizers/wiki/threadsanitizercppmanual

Таблиця 5.1 – Main Vulnerabilities Related to Concurrency Issues

Vulnerability Type	Description	CWE
Race Conditions	Two or more threads or processes try to modify or access a shared resource (memory, files, variables) without proper synchronization.	*CWE-362*
Data Races	Two or more threads modify the same variable or memory region simultaneously without proper synchronization, leading to unpredictable results.	*CWE-367*
Deadlocks	Two or more threads or processes block resources that they are waiting for from each other, leading to a complete halt of their operations.	*CWE-833*
Live Locks	A situation where processes continue performing actions that block each other without being able to complete their tasks.	*CWE-667*
Priority Inversion	A lower-priority process holds a resource needed by a higher-priority process, causing delays in the system's performance.	*CWE-410*
Double-Fetch	Data from a resource is read twice, and an attacker can modify the data between the operations, leading to unpredictable results.	*CWE-367*
Lock Contention	Lock contention occurs when multiple threads compete for the same resource, causing delays and reduced performance.	*CWE-667*
Permission Races	Two or more processes attempt to modify or use resources without proper access rights.	*CWE-275*
Races with Temporary Files	Vulnerabilities arise when temporary files are created and used without proper synchronization.	*CWE-377*
Directory Position Race	A race occurs when directories are changed between the time of checking and use.	*CWE-346*

check (time of check) and its use (time of use). This allows an attacker to change the state of a resource between the check and use, which can lead to undesirable consequences.

TOCTOU Concept:

Time of Check – the moment when a program checks the availability or state of a resource (file, memory, network connection, etc.) and makes a decision based on this check. *Time of Use* – the moment when the program performs an operation on the resource, assuming that its state has not changed since the check. *The Problem* – between these two moments, another process or thread can change the state of the resource, leading to unintended or dangerous actions.

Imagine a program that checks if a file exists and then opens it for writing:

```
if (access("somefile.txt", W_OK) == 0) {
    // file exists and is writable
}
fd = open("somefile.txt", O_WRONLY);
```

Between these operations (the check and the open), another process may replace the file with a symbolic link to another file that the program should not have access to, which can lead to incorrect data writing.

TOCTOU vulnerabilities can lead to:

- Incorrect access to data;
- Exploiting the system by altering resource states;
- Potential privilege escalation.

Methods for Preventing TOCTOU

1. *Atomic Operations*. Use system calls that perform both the check and the use of the resource in a single operation. For example, you can ensure that a file is created only if it does not exist, avoiding a TOCTOU attack:

```
fd = open("somefile.txt", O_WRONLY | O_CREAT | O_EXCL);
```

2. *Principle of Least Privilege* by restricting access to files and resources.
3. *Inter-process Synchronization* by locking resources to prevent improper access between processes or threads.
4. *Path Fixation* by directly defining file paths (`fdstat()` is safer than `stat()`).

This group of vulnerabilities also includes *Permission Races*, *Races with temporary files*, *Directory position race*, and others (Table 5.2).

Data Races

One of the most common concurrency scenarios is a *race condition*. It occurs when two or more threads modify a shared resource simultaneously, and the result

Таблиця 5.2 – Main Vulnerabilities Related to Concurrency Issues

Vulnerability	Description
CWE-367: Time-of-check Time-of-use (TOCTOU)	A file or resource is checked before being used, but is changed before the use operation.
CWE-362: Concurrent Execution using Shared Resource	Resources are used simultaneously by multiple threads without proper synchronization.
CWE-364: Signal Handler Race Condition	A signal handler is called while a global state is being modified, potentially causing data corruption or crashes.
CWE-363: Race Condition Enabling Link Following	A file or directory state is checked before access, but the resource can be replaced with a link before it is used.
CWE-368: Context Switching Race Condition	Occurs during context switching between threads, enabling attacks on data integrity or confidentiality.

depends on the order in which the threads execute. Since the order is not always predictable, this can lead to unpredictable results.

Access to a shared variable by multiple threads may lead to a data race if:

1. Access occurs (potentially) simultaneously;
2. At least one access is a write operation.

A *data race* is a memory-level race condition involving atomic operations. It is the root cause of many complex bugs in multithreaded programs.

Data races are typically sporadic errors:

- They lead to nondeterministic behavior.
- The incorrect behavior may be very rare.
- Reproducing the error can be challenging.

These approaches are usually applied only in expert library code or OS kernel development. Ordinary application developers should strive to write race-free programs.

Race Condition Example

Consider a *C++* example where two functions access the same variable simultaneously:

```
int shared_resource = 0;

void increment() {
    shared_resource++;
}
```

```
void decrement() {
    shared_resource -- ;
}

int main() {
    std::thread t1(increment);
    std::thread t2(decrement);
    t1.join();
    t2.join();
    printf("%d\n", shared_resource);
}
```

In this example, the threads simultaneously access the `shared_resource` variable. Due to the lack of synchronization, the result can be unpredictable – the value of the variable may be 0, 1, or -1, depending on the order of execution.

If a race condition occurs in the operating system, attackers can exploit it to cause incorrect calculations or inconsistent values by:

- Influencing thread scheduling;
- Repeatedly executing a specific action.

Methods for Preventing Race Conditions

Various synchronization methods, such as *mutexes*, *semaphores*, or locks, are used to prevent race conditions. Mutexes ensure that only one thread can access a resource at a time, while other threads must wait.

Example of Using a Mutex to Prevent Race Conditions

Let's consider the same example, but with a mutex for synchronizing access to the shared resource:

```
#include <mutex>

int shared_resource = 0;
std::mutex mtx;

void increment() {
    mtx.lock();
    shared_resource++;
```

```
        mtx.unlock();
    }

    void decrement() {
        mtx.lock();
        shared_resource -- ;
        mtx.unlock();
    }

    int main() {
        std::thread t1(increment);
        std::thread t2(decrement);
        t1.join();
        t2.join();
        printf("%d\n", shared_resource);
    }
```

In this example, the mutex (`mtx`) ensures that only one thread can modify `shared_resource` at any given time. This prevents race conditions and guarantees the correct result.

Other methods, such as semaphores and read/write locks, can also be used to address concurrency issues depending on the specific situation. Semaphores provide limited access to resources, while read/write locks allow multiple threads to read data simultaneously but only one thread to write.

Consequences of Concurrency Issues

If concurrency issues are not addressed properly, they can lead to serious vulnerabilities in software. For example, in multithreaded web servers or databases, the lack of synchronization may result in incorrect request handling, data loss, or leakage of sensitive information. Attackers can exploit these vulnerabilities to launch attacks or gain unauthorized access to resources.

Example of an Attack Exploiting a Race Condition

An attacker may attempt to exploit a file access vulnerability where two threads simultaneously try to modify the same file:

```
    open("/tmp/file", O_CREAT | O_WRONLY, S_IRUSR | S_IWUSR);
    write(fd, buffer, sizeof(buffer));
```

```
    close(fd);
```

If the attacker can modify the file during these operations, they can trick the program into working with a different file, potentially leading to data loss or theft.

Review Questions

1. What are *race conditions*, and how do they occur in software?
2. How can *concurrency issues* lead to incorrect program behavior?
3. What methods can be used to prevent *race conditions*?
4. What are *mutexes*, and how do they help solve *race condition* issues?
5. How can *semaphores* be used to synchronize resource access?
6. What are the security consequences of improper *concurrency management*?
7. How can *race conditions* be exploited by attackers to compromise software?
8. What other thread synchronization methods can be used to prevent *race conditions*?
9. What problems can arise in *multithreaded programs* due to the lack of proper synchronization?
10. What analysis tools can be used to detect *race conditions* in software?

Topic 6

MALWARE AND SOCIAL ENGINEERING

The objective is to become familiar with types of malicious software and social engineering methods, and to gain practical skills in counteracting these threats.

Tasks for Independent Study

1. *Analyze a malware sample* using a *sandbox* (e.g., *Any.Run*, *Cuckoo*) and prepare a report on its behavior. It is recommended to use safe educational samples such as the *EICAR test file* or controlled samples from projects like *theZoo*, strictly in isolated environments.
2. *Write a training scenario for staff on detecting phishing emails*, including tips for verifying senders, links, and attachments.
3. *Explore how antivirus software detects malicious files* (e.g., signature-based detection, heuristic analysis, behavioral detection).
4. *Create an infographic on the main social engineering techniques* (*phishing*, *pretexting*, *baiting*, *tailgating*, etc.).
5. *Conduct a social engineering attack simulation* within your team (with participants' consent) and analyze the outcomes.

Brief Theoretical Background

1. Malware.
2. Programming techniques for malware resistance.
3. Social engineering and its impact on cybersecurity.
4. Defense mechanisms against social engineering.

Malware

Malware is one of the most common tools used by attackers to compromise systems and networks. It includes *viruses*, *worms*, *trojans*, *rootkits*, *adware*, and *spyware*. The primary goal of such programs is to exploit vulnerabilities in computer systems to gain unauthorized access, steal data, or damage resources.

Viruses can spread through infected files or macros in *Microsoft Office* documents. *Trojan programs* disguise themselves as regular applications but actually grant attackers access to the system. *Rootkits* can hide the activity of other malicious programs, providing attackers with prolonged access to the system.

One of the most dangerous types is *ransomware*. This type of *malware* locks access to files or entire systems and demands a ransom for unlocking them. The virus encrypts user data on local drives, after which the computer is locked, and the attackers demand a ransom (usually in cryptocurrency) to provide the decryption key, typically within a limited time frame. If the demands are not met, the data may be lost permanently. These attacks target both individual users and large corporations, governments, and other organizations, leading to significant disruptions.

One of the most notable ransomware attacks was the *WannaCry* attack in 2017, which affected over 200,000 computers in 150 countries. This virus exploited a vulnerability in *Windows* operating systems to spread rapidly through networks, causing significant damage to companies and government institutions. Another example is the *NotPetya* virus, which attacked corporate networks in Ukraine and spread worldwide, causing billions of dollars in damage.

Programming Methods for Resilience Against Malicious Code

When designing application architecture, software architects must consider all potential risks, including cybersecurity threats. Secure programming generally involves adhering to common practices such as:

- proper error and exception handling;
- applying the principle of least privilege;
- using modern and secure libraries and frameworks;
- maintaining quality documentation and code commenting;
- conducting thorough code reviews.

Codebase Analysis and Testing

To ensure a high level of cybersecurity for an *IT product*, the following types of analysis are used:

- *Integration and functional testing*, which checks the interaction between different components of the program and allows for identifying problems related to integration and program logic;
- *Static code analysis*, performed before the program is compiled, to detect common errors like memory leaks, buffer overflows, or failure to follow programming standards. Specialized tools, such as static analyzers, are used for this purpose;
- *Dynamic code analysis*, which tests the program during execution, allowing for an assessment of its actual performance under load, resource usage, and memory leaks. This analysis also includes code instrumentation and the emulation of cyberattacks to test the program's resilience against attacks like *SQL Injection*, *XSS*, and *CSRF*.

To prevent, detect, and timely address vulnerabilities in software, it is important to follow certain guidelines and standards:

Adhering to Cybersecurity Standards

The development and maintenance of IT products must comply with international security standards, including:

• Using development technologies such as:

1. *Open Web Application Security Project (OWASP)* for web application development;

2. *Payment Card Industry Data Security Standard (PCI DSS)* for electronic payment systems.

• Establishing processes with cybersecurity in mind:

1. *NIST Cybersecurity Framework*, general security guidelines from the U.S. National Institute of Standards and Technology;

2. *ISO 27001*, an international standard for information security management that allows organizations to build secure processes;

3. *GDPR (General Data Protection Regulation)*, which sets out regulations for personal data protection in the European Union.

Adhering to these standards helps mitigate risks and ensure code security in the face of modern cyber threats.

It is also essential to emphasize the importance of *testing* to ensure product security. Testing must be an ongoing process, regardless of the product's lifecycle stage. Testing based on the use of CI/CD[1] aims to check for security and fix vulnerabilities both before and after the product's release, as threats constantly evolve, and new vulnerabilities can be found even in well-tested code.

Social Engineering and Its Impact on Cybersecurity

Social engineering is a technique used by attackers to manipulate people into providing access to confidential information or systems. Instead of attacking the technical side of a system, attackers may exploit the human factor as a weak point in security. Common examples of social engineering include *phishing*, where users are tricked into revealing passwords or other sensitive information through fake websites or emails.

Phishing

Over the years, *phishing* attacks have evolved significantly, and cybercriminals have developed various methods to exploit it, continuously improving them[2]:

1. *Email phishing*, where attackers disguise links in emails to mimic well-known companies, banks, or the victim's own company;

[1] Continuous Integration/Continuous Delivery

[2] Microsoft: What is phishing?

2. *Malware phishing*, where malicious software is disguised as safe files, such as resumes or bank statements in email attachments. Opening such files can harm IT systems;
3. *Spear phishing*, which targets specific individuals whose work or personal data has been pre-collected. This attack may bypass security measures as it is highly targeted;
4. *Whaling* targets business executives, celebrities, or others with significant assets. Attackers carefully analyze the victim and wait for the right moment to steal credentials or other confidential information;
5. *Smishing* – phishing through *SMS messages*, where scammers pose as well-known service providers like Amazon, Nova Poshta, or Monobank. These messages often appear very personal, making them especially effective;
6. *Vishing*, an attack through phone calls where attackers pose as customer service representatives to trick victims into revealing personal information.

Phishing Attack Example

A user receives an email pretending to be from a bank:

```
Dear Customer,

Please click on the link below to verify your account information:
http://fakebank.com/login

Thank you,
Your Bank
```

After following the link, the user lands on a fake website that looks legitimate. If the user enters their login credentials, they will fall into the hands of the attackers.

Methods to Protect Against Social Engineering

To protect against social engineering, it is crucial to educate users about threats and attack methods. Additionally, implementing multi-factor authentication complicates the attackers' ability to gain access even if they have stolen user credentials.

Example of Multi-Factor Authentication

```
if (user_authenticated && sms_code_verified) {
        // Grant access to the system
```

```
        }
```

Even if an attacker obtains the user's password, they would still need the confirmation code sent to the user's phone.

Cyber Kill Chain Framework

The *Cyber Kill Chain* framework (developed by Lockheed Martin[3]) is a model for detecting and preventing cyber intrusion activities. The model defines the steps an adversary must take to achieve their goal. The main idea is to help better understand the roles of the attack and defense parties (Table 6.1).

The following cybersecurity frameworks and models are worth studying:

1. *MITRE ATT&CK* (Adversarial Tactics, Techniques, and Common Knowledge) – a knowledge base that provides information on the tactics, techniques, and procedures (TTPs) used by cybercriminals. It is used to analyze threats, assess security, and develop defensive measures;
2. *Diamond Model of Intrusion Analysis* – a model of cyberattack analysis that describes four main components of any attack: adversary, victim, infrastructure, and methods. The model helps identify connections between these components for better threat response;
3. *OODA Loop* (Observe, Orient, Decide, Act) – a decision-making model used in cybersecurity. It describes the process of observing, orienting, deciding, and acting when responding to attacks.
4. *Purdue Enterprise Reference Architecture* (PERA) – an architectural model for describing the interactions of various levels in industrial networks. It helps design and secure automation systems at all levels.
5. *NIST Cybersecurity Framework* – a cybersecurity standard from the *National Institute of Standards and Technology* (*NIST*). The framework includes the stages of *identifying*, *protecting*, *detecting*, *responding*, and *recovering* to manage *cybersecurity risks*.

Review Questions

1. What is *malware* and what are its main types?
2. How do *viruses* or *trojan programs* work, and how can they exploit vulnerabilities?
3. What programming methods can ensure resilience against malicious code?
4. How can *encryption* help protect information from attacks?
5. What is *social engineering* and how is it used in attacks on users?

[3] https://www.lockheedmartin.com/en-us/capabilities/cyber/cyber-kill-chain.html

Таблиця 6.1 – Stages of the *Cyber Kill Chain*

Stage	Description
Reconnaissance	Gathering as much information about the target as possible.
Weaponization	Creating a malicious tool, such as a virus or worm.
Delivery	Delivering malware to the victim through system vulnerabilities.
Exploitation	Exploiting vulnerabilities to execute malicious code on the victim's system.
Installation	Installing the malware and spreading it within the environment.
Command and Control (C2)	Creating a back channel to the attacker's server.
Actions on Objectives	Exfiltrating data, destroying or encrypting it for ransom.

6. What are the most common examples of *social engineering* attacks?
7. How can *phishing* be used to steal confidential information?
8. What methods can be used to protect against *social engineering*?
9. How does *multi-factor authentication* work and why is it important for security?
10. How can users be trained to counter *social engineering* attacks?

Topic 7

SECURE PROGRAMMING METHODS IN PROGRAMMING LANGUAGES

The objective is to study the security features of different programming languages and to develop skills in analyzing secure programming techniques in those languages.

Tasks for Independent Study

1. *Compare built-in security mechanisms in Rust and C++* (e.g., the borrow checker in *Rust*, exception handling in *C++*, etc.).
2. *Write Java code using the SecurityManager* and explain how it restricts access to system resources.
3. *Use a static code analyzer (e.g., ESLint)* to check *JavaScript* code for common mistakes and vulnerabilities.
4. *Implement error handling in Python* using `try-except` blocks and observe how it affects the program's stability.
5. *Explore how the Go programming language prevents buffer overflows*, focusing on its memory management features.

Brief Theoretical Background

1. Secure programming practices across different languages:
- *C* and *C++*;
- *Java*;
- *Python*;
- *JavaScript*;
- *Rust*;
- *Go*.

2. Comparison of approaches to writing secure code.

Overview of Specific Secure Programming Methods for Different Languages

Each programming language has its own characteristics and vulnerabilities, which means that approaches to ensuring security may differ. In this section, we will review secure programming methods for languages such as *C*, *C++*, *Java*, *Python*, and *JavaScript*. A comparison of these methods will help to better understand the specifics of each language and the most effective approaches to prevent vulnerabilities.

C and *C++*: Memory Management and Buffer Overflow Protection

C and *C++* give programmers low-level memory access, making them powerful but also vulnerable to memory management errors. One of the biggest problems in these languages is *buffer overflow*, which can lead to memory corruption or even malicious code execution.

The main documents defining secure programming methods in these languages are the *SEI CERT C Coding Standard* [19] and *SEI CERT* C++ *Coding Standard* [20]. As noted in [20], "the goal …is to develop safe, reliable, and secure systems, for example, by eliminating undefined behaviors that can lead to unpredictable program actions and vulnerabilities that attackers could exploit."

The *SEI CERT* C++ *Coding Standard* includes the following groups of rules that apply to specific features of the *C++* programming language:

1. *DCL* – Declarations and Initialization;
2. *EXP* – Expressions;
3. *INT* – Integers;
4. *FLP* – Floating Point Numbers;
5. *ARR* – Arrays;
6. *STR* – Strings;
7. *MEM* – Memory Management;
8. *FIO* – Input/Output;
9. *ENV* – Environment;
10. *SIG* – Signals;
11. *ERR* – Error Handling;
12. *OBJ* – Object-Oriented Programming;
13. *CON* – Concurrency;
14. *MSC* – Miscellaneous;
15. *API* – Application Programming Interface.

Let's look at an example requirement from the *SEI CERT C++ Coding Standard* related to memory management.

MEM53-CPP. Explicitly create and destroy objects when manually managing object lifecycles. The creation of dynamically allocated objects in *C++* occurs in two stages. The first stage allocates enough memory to store the object, and the second stage initializes the newly allocated memory block based on the type of object being created.

This type of error is one of the most common, especially when an object contains data fields, with resources allocated using `new()`.

Risk Assessment for MEM53-CPP[20]

- *Rule*: MEM53-CPP

- *Severity*: High
- *Likelihood*: Likely
- *Remediation Cost*: Medium
- *Priority*: P18
- *Level*: L1

For developers, examples are particularly valuable, as they are divided into two types: compliant and non-compliant examples, which help better understand the essence of the rule.

The *SEI CERT C Coding Standard* contains similar rules and recommendations as the *C++* variant, with a focus on memory management, but with rules and guidelines tailored to the *C* language. Here, the distinctions between rules and recommendations are significant:

- *Rules* are mandatory requirements that *must* be followed to avoid vulnerabilities and undefined behavior in the program. These rules can often be automatically checked by tools, and breaking them has severe consequences for security and program stability.
- *Recommendations* are *advisable* practices that improve the quality and security of the code, though they are not always mandatory. Violating these recommendations may degrade code quality but does not pose the same critical threat as violating rules.

Both standards for *C* and *C++* are freely available, with the 2016 version being the most recent.

Java

Java is a language with a managed runtime environment (*JVM*), making it more secure in terms of memory management since automatic garbage collection prevents memory leaks. *Java* is one of the main programming languages for business applications, so mistakes in *Java* programs can have critical consequences for the user. That's why the *SEI CERT Oracle Coding Standard for Java* [21] exists.

The original version was released in 2011, and the current version is available on the SEI CERT website [22]. Its structure is similar to that of the *SEI CERT C Coding Standard*. Rules are also divided into different categories, each covering a particular aspect of programming. These categories cover topics from resource and memory management to control flow and secure data handling. The rules are mandatory for compliance, and violating them can lead to serious vulnerabilities that attackers can exploit.

Java also has its specific security challenges, due to its widespread use.

Let's look at an example rule from the *Input Validation and Data Sanitization* (*IDS*) section[1].

[1] SEI: IDS00-J. Prevent SQL injection

IDS00-J. Prevent SQL injection vulnerabilities. *SQL* injection vulnerabilities arise in applications where *SQL* query elements come from untrusted sources. Without proper security measures, untrusted data can maliciously alter a query, leading to data leakage or modification. The primary means of preventing SQL injection include data sanitization and validation, typically implemented through parameterized queries and stored procedures.

The following code fragment from the *SEI CERT Oracle Coding Standard for Java* demonstrates how to prevent *SQL* injection. A key feature of this method is the use of `set*()` methods of the `PreparedStatement` class to ensure strict type-checking. This reduces the risk of SQL injection since input data is automatically escaped. Prepared statements can be used to insert data into a database:

```
...
String pwd = hashPassword(password);
// Validate username length
if (username.length() > 8) {
        // Handle error
    }
String sqlString =
"select * from db_user where username=? and password=?";
PreparedStatement stmt = connection.prepareStatement(sqlString);
stmt.setString(1, username);
stmt.setString(2, pwd);
ResultSet rs = stmt.executeQuery();
if (!rs.next()) {
        throw new SecurityException("User name or password
        ↪  incorrect");
    }
...
```

Risk Assessment for IDS00-J [22]:

- *Rule*: IDS00-J.
- *Severity*: High.
- *Likelihood*: Likely.
- *Remediation Cost*: Medium.
- *Priority*: P18.
- *Level*: L1.

Python

Similarly to *C*, *C++*, and *Java*, *Python* and applications built using it, due to its popularity, can also become targets of attacks. A secure coding standard has been developed for *Python* as well [25], under the aegis of the *Open Source Security Foundation* (*OpenSFF*), which continued the work initiated by Ericsson to improve secure coding practices in *Python* (for educational purposes). The structure of the standard is directly based on the *CWE Pillar Weaknesses* [mitre.org 2023]. The project is in the development stage and contains only code examples.

Let's consider an example [2].

CWE-197. Numeric Truncation Error. Ensure predictable results in loops by using `int` instead of `float` as a counter. Floating-point arithmetic can only represent a finite subset of real numbers in *IEEE 754*, for example, 0.555... is represented as 0.5555555555555556, which is also discussed in *CWE-1339: Insufficient Precision or Accuracy of Real Numbers*.

This error is also relevant for *Java* (NUM09-J) and *C* (FLP30-C).

Below are code fragments showing both compliant and non-compliant approaches when using loops.

```
""" Non-compliant Code Example """
counter = 0.0
while counter <= 1.0:
    if counter == 0.8:
        print("we reached 0.8")
        break  # never going to reach this
    counter += 0.1
```

```
""" Compliant Code Example """
counter = 0
while counter <= 10:
    value = counter / 10
    if value == 0.8:
        print("we reached 0.8")
        break
    counter += 1
```

[2] OpenSFF: CWE-197: Numeric Truncation Error

JavaScript

JavaScript is widely used in web applications, which makes it a target for attacks, particularly cross-site scripting (*XSS*). XSS occurs when an attacker injects malicious code into a webpage, which is executed in the user's browser.

Today, there is a secure coding standard for *JavaScript* from Checkmarx [26]. The *JavaScript Web Application Secure Coding Practices* standard is organized into sections that cover critical security aspects of web development using *JavaScript*. These sections include *input validation*, *output encoding*, *authentication management*, *session handling*, and *error handling*. The standard proposes a logical approach to ensuring security: starting from input validation to secure coding patterns for handling sensitive information and interacting with external systems.

One of the key focuses of the standard is *input validation*, which ensures that all user input is treated as untrusted by default. The importance of distinguishing between trusted and untrusted data sources is emphasized, along with recommendations for secure validation practices. This section covers protection against common vulnerabilities such as *SQL injection* and *cross-site scripting (XSS)*, providing appropriate methods of validation and encoding.

The standard describes the main types of vulnerabilities that occur in web applications:

• *SQL injection*: the risk arises from incorporating untrusted data into *SQL* queries. To prevent this, it is recommended to use *parameterized queries* and *stored procedures*.

• *Cross-site scripting (XSS)*: to prevent *XSS* vulnerabilities, *output encoding* should be used so that data is safely displayed on pages without executing malicious scripts.

• *NoSQL injection*: the best practices for securing *NoSQL* databases are provided to protect against NoSQL injection vulnerabilities.

A separate section is dedicated to *authentication* and proper session management. The importance of securely storing credentials and following secure password policies is emphasized. Methods for session management, such as secure cookie handling and session termination to protect against session hijacking, are also detailed.

The standard also covers *cryptographic practices* related to using secure algorithms and key management to protect sensitive information. In addition, the standard emphasizes proper error handling, particularly avoiding exposing system information through error messages and applying robust logging mechanisms.

Comparison of Approaches to Ensuring Secure Code

• Languages like *C* and *C++* provide the most options for low-level memory access, but this also makes them the most vulnerable to memory management issues

(buffer overflows, memory leaks). Using modern memory management functions and validating user input is critically important.

• *Java* is less vulnerable to memory issues due to automatic memory management, but it requires careful exception handling and serialization to avoid information leaks and unauthorized access.

• *Python* with its dynamic typing is convenient but requires additional caution when working with libraries to prevent vulnerabilities.

• *JavaScript* is particularly vulnerable to XSS attacks, so developers must ensure proper escaping of user input and avoid unsafe constructs that allow the execution of untrusted code.

Review Questions

1. What are the main security issues when working with C and *C++*?
2. How does Java handle memory, and what security methods should be considered?
3. Why is using `eval` in Python dangerous, and how can it be avoided?
4. What are *XSS* attacks in *JavaScript*, and how can they be prevented?
5. How can parameterized queries help avoid *SQL injection*?
6. How can serialization in Java lead to vulnerabilities, and how can this be avoided?
7. Why is memory management important in C and *C++*, and what methods can be used to prevent memory leaks?
8. What are the advantages and disadvantages of using automatic garbage collection in *Java*?
9. How can security be ensured when working with user input in *Python*?
10. What authentication and authorization methods can be used to protect cloud applications?
11. What risks arise from using unsafe functions in JavaScript, and how can they be avoided?

Topic 8

VIRTUALIZATION AND CLOUD APPLICATIONS

The objective is to become familiar with security issues in cloud technologies and virtualization, and to develop practical skills using tools for vulnerability analysis.

Tasks for Independent Study

1. *Set up an isolated environment using Docker* for application testing (create a `Dockerfile`, use `Docker Compose`, and build an image).
2. *Analyze security risks* in cloud infrastructure (*AWS* or *Azure*) and compile a list of possible attack vectors.
3. *Explore access control policies for virtual machines in VMware* using role-based access control (*RBAC*).
4. *Perform a penetration test of a cloud application* using *OWASP ZAP* and document the discovered vulnerabilities.
5. *Compare the security of Docker containers and virtual machines*, focusing on isolation levels and hypervisors.

Brief Theoretical Background

1. *Virtual machines* and their security mechanisms.
2. *Process isolation* and permissions as protection methods.
3. *Cloud application architecture*.
4. *Data privacy issues* in cloud services.

Virtual Machines and Their Security Measures

Virtualization is a technology used to create virtual representations of servers, storage, networks, and other physical machines (Table 8.1, Figure 8.1). Virtual software simulates the functions of physical hardware to run multiple virtual machines on a single physical machine simultaneously. The use of virtualization allows modern companies to optimize their computing resources, including transitioning to full use of cloud services. With virtualization in the form of cloud services, companies can completely abandon the use of physical servers, simplifying infrastructure maintenance [27].

Virtual Machines (VM) are tools for creating isolated execution environments on a single physical computer. They allow multiple operating systems (*guest OS*)

Таблиця 8.1 – Types of Virtualization

Type of Virtualization	Brief Description
Application Virtualiza-tioz	Allows running applications in isolated environments, inde-pendent of the OS or hardware.
Network Virtualization	Creating virtual network infrastructures based on physical networks for flexible traffic management.
Desktop Virtualization	Allows users to access remote desktops from any device.
Storage Virtualization	Combines physical storage resources into virtual ones to sim-plify data management.
Server Virtualization	Enables running multiple virtual servers on a single physical server for increased efficiency.
Data Virtualization	Allows access to data regardless of its physical location or format.

to run simultaneously on a single server (*host OS*). Each guest OS operates in its own isolated environment, provided by a hypervisor. Virtualization offers certain security advantages, as the isolation between virtual machines prevents one system from interfering with another's operation (Fig. 8.1).

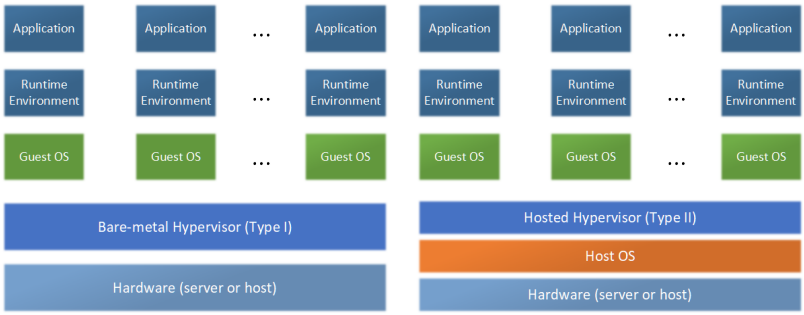

Figure 8.1 – Virtualization

Technical aspects of isolation include using different levels of virtualization, such as type 1 hypervisors (*bare-metal*) and type 2 hypervisors (running on top of the host operating system). Type 1 hypervisors, like *VMware ESXi* or *Microsoft Hyper-V*, provide more efficient isolation and performance due to direct access to hardware. In contrast, type 2 hypervisors, such as *VirtualBox* or *VMware Workstation*, run on top

of the host OS, which can introduce additional overhead.

Additionally, hardware virtualization supported by modern processors, such as Intel VT-x or AMD-V, provides an even higher level of security and isolation between virtual machines, minimizing the risk of escaping the isolated environment (*VM escape*).

Key security measures for virtual machines include:

- *Hypervisor security*, including protection against breakout attacks.
- *Isolation*, segmentation, and micro-segmentation of resources, ensuring that each *VM* uses a specific portion of processor time, memory, and other resources, making direct access between *VMs* impossible.
- *Network security* through the use of intrusion detection (*IDS*) and intrusion prevention systems (*IPS*).
- *Secure data storage* and recovery mechanisms.
- Regular updates of security patches.

These methods ensure the fine-tuning and administration of security in virtual environments and their stable operation.

Process Isolation and Memory Management in Protected Mode

Process isolation is a fundamental element of ensuring security both in virtualized environments and on physical servers. In operating systems, each process runs in its own address space, protecting it from unauthorized access by other processes. This means that each process does not have access to the memory of other processes, and access to hardware resources is strictly controlled.

Process isolation is supported both at the hardware and software levels. Modern processors have multiple *protection rings*, providing processes with different levels of privileges. For example, in x86 architecture processors, there are four rings:

- *Ring 0* – the most privileged level, where the operating system kernel runs. This level has full access to all hardware resources.
- *Rings 1 and 2* – used for drivers and services with intermediate privilege levels.
- *Ring 3* – designated for unprivileged user processes, which do not have direct access to hardware and can interact with resources only through system calls.

Memory Management

To implement process isolation in operating systems, technologies such as *virtual memory* are used. Virtual memory allows each process to have its own *virtual address space*, mapped to physical memory using page tables. This scheme protects one process's memory from being accessed by other processes and allows for efficient use of the available physical memory.

In *protected mode*, supported by most modern x86-64 architecture processors, operating systems have full control over memory allocation and management. Processes execute in isolated environments, and access to critical resources (e.g., input/output devices or system resources) is only performed through system calls to the *OS kernel*.

Processes in protected mode run with different privilege levels, regulated through *protection rings*. This provides reliable isolation between privileged and unprivileged processes. For example, if a user process requires access to privileged resources, it must invoke the appropriate kernel function operating at *ring 0*, ensuring controlled access.

In the context of virtualization, such as with *hypervisors* and *containers* (e.g., *Docker*), this isolation mechanism is complemented by additional levels of resource allocation at the hardware level. Containers enable the execution of isolated processes with their own system resources, such as file systems and network interfaces, providing an extra level of security.

In addition to isolation, correctly setting permissions for users and processes is crucial to avoid privilege escalation.

Methods for ensuring isolation include:

• *Containers* (e.g., *Docker*) provide an isolated environment that allows each container to operate independently of others. Containers also ensure a consistent environment and a ready set of tools for the developer.

• *Application virtualization*, such as *VMware ThinApp, Turbo Studio* (formerly *Spoon Studio* and *Xenocode Studio*) on *Windows*, and *AppImage* on *Linux*. These applications run in a container but appear as regular programs and do not require installation.

• *Access Control Lists (ACLs)* are used to set permissions for files and processes. This ensures that only authorized users can access specific resources.

Despite the advantages of virtualization, this technology is also constantly under attack. For example, Table 8.2 lists some typical vulnerabilities of various hypervisors over recent years.

Cloud Application Architecture

Cloud applications and services can be built on multiple levels, which significantly impacts the capabilities of such applications as well as the qualifications of the application or service user. At the lowest, physical level (*Physical layer*), real hardware is used, including servers, data storage, network infrastructure, etc. At the next, infrastructure level (*Infrastructure layer*), basic computing resources necessary for cloud application functionality are provided, such as the number of processors, memory capacity, and disk space. At the platform level (*Platform layer*), a ready-to-deploy system serves as the foundation for development, such as a configured

Таблиця 8.2 – Virtualization Software Vulnerabilities

CVE	Product	Description
CVE-2024-43575	*Hyper-V*	Denial of Service (DoS) vulnerability in *Microsoft Hyper-V*, which can cause a system crash due to specially crafted requests.
CVE-2023-21990	*VirtualBox*	Vulnerability in *Oracle VirtualBox*, allowing a privileged local user to compromise *VirtualBox*, potentially leading to full control over the system.
CVE-2022-39423	*VirtualBox*	Vulnerability in *VirtualBox* (versions up to 6.1.38), allowing unauthorized access to data or full system takeover.
CVE-2021-28476	*Hyper-V*	Critical vulnerability in *Hyper-V*, enabling remote attackers to execute arbitrary code through the vmswitch component.
CVE-2022-23825	*Hyper-V*	Speculative execution vulnerability (similar to Spectre) that may lead to information leakage in *Hyper-V* virtual machines.
CVE-2023-2085	*VirtualBox*	Vulnerability in *VirtualBox*, allowing a VM escape and access to the host system.
CVE-2022-21900	*Hyper-V*	Vulnerability in *Hyper-V*, allowing security bypass and data leakage between virtual machines.
CVE-2023-21982	*VirtualBox*	Vulnerability in *VirtualBox*, allowing security restrictions to be bypassed and unauthorized access to the host system from the guest OS.
CVE-2020-4006	*VMware*	Remote code execution vulnerability in *VMware ESXi*, allowing an attacker to execute arbitrary commands via the OpenSLP service.
CVE-2022-29900	*Hyper-V*	Vulnerability related to speculative execution (*RETbleed*), potentially leading to information leakage through side-channel attacks.

operating system with installed libraries and frameworks for developers. Finally, at the highest level – the application layer (*Application layer*) – user applications operate.

It should be noted that cloud computing architecture is complex and includes not only the services we will discuss below but also many technical implementation details (Fig. 8.2, [28]).

Platforms can be classified based on the service delivery model:

• *IaaS* (*Infrastructure as a Service*) – provides basic computing resources. Examples of such platforms include *Amazon Web Services (AWS)* by Amazon, *Microsoft Azure* by Microsoft, and *Google Cloud Platform* by Google.

• *PaaS* (*Platform as a Service*) – creates an environment for software development and management. Examples of *PaaS* platforms include *AWS Elastic Beanstalk*

by Amazon, *Azure App Service* by Microsoft, and *Google App Engine* by Google.

• *SaaS* (*Software as a Service*) – provides ready-to-use applications that can be used without the need for installation or management, such as *Microsoft 365* and *Google Workspace*.

It is worth noting that the provided list of services is not exhaustive – there is a trend today to move most services to the cloud.

Cloud Applications (SaaS): Multi-layered Architecture and Security

Cloud applications (*SaaS*) are typically built using a multi-layered architecture that includes the *frontend* (user interface), *backend* (data processing servers), *databases*, and *network resources*. The main element of security in cloud applications is a clear division of roles and permissions, as well as the provision of proper authentication and encryption.

A typical architecture of SaaS cloud applications:

• *Frontend* – the interface through which the user interacts with the cloud application, usually via a web browser or mobile app. *Frontend* security is ensured through the use of *TLS* (*Transport Layer Security*) to protect traffic.

• *Backend* – servers that process user requests, manage business logic, and access *databases*. *Backend* security involves access control, authentication, and the use of isolated environments for code execution.

• *Databases* – store users' sensitive data. Data should be encrypted both at rest and in transit.

Example Architecture on AWS

On the *AWS* platform, a cloud application may use an *Elastic Load Balancer* to distribute traffic among *EC2* servers, *Amazon RDS* for data storage, and *S3* for static files. Each component is isolated and has its own security policies to ensure an appropriate level of protection.

Data Privacy Issues on Cloud Services

One of the main challenges of using cloud services is ensuring the privacy and security of data. User data may be stored on remote servers located in different parts of the world, raising questions about who has access to this data and how it can be protected.

Key privacy issues include:

• Control over data in the cloud environment is entrusted to the cloud service provider, which may pose risks of unauthorized access.

• Encryption of data both during transmission (*TLS*) and at rest (*AES*) can ensure data confidentiality.

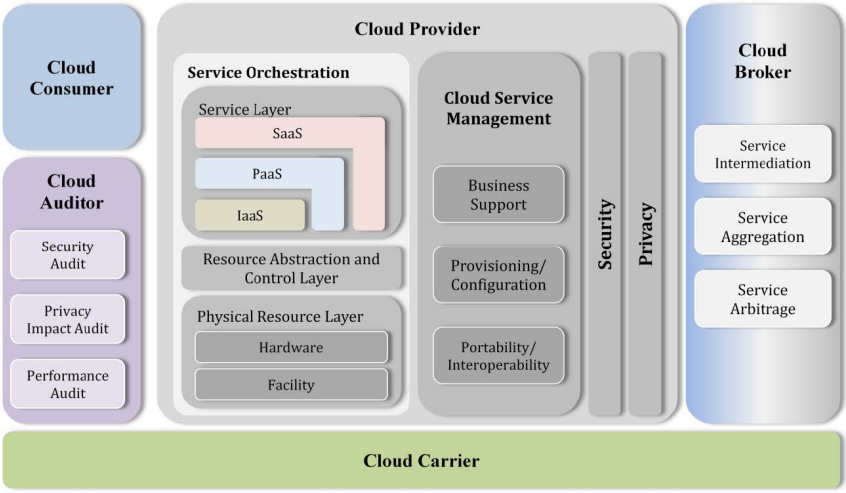

Figure 8.2 – NIST Cloud Computing Reference Architecture [28]

• Different countries have specific laws regarding the processing and storage of personal data, so cloud service providers must comply with standards such as GDPR.

Example of Data Encryption on AWS

On *AWS*, *KMS* (Key Management Service) can be used to encrypt sensitive data:

```
aws kms encrypt  -- key-id alias/my-key  -- plaintext fileb://data.txt
↪  -- output text  -- query CiphertextBlob
```

This example shows how to encrypt a file using keys stored in *KMS*. This helps protect the data even in the case of unauthorized access to physical media.

Review Questions

1. What are *virtual machines*, and how do they help ensure security?
2. What are the advantages and disadvantages of *process isolation* compared to *virtual machines*?
3. What are the main components of a typical *cloud application architecture*, and how do they interact?
4. What privacy issues arise when using *cloud services*?

5. How does *data encryption* help protect confidential information in the cloud?
6. What technologies can be used for *data encryption* on *cloud platforms*?
7. What are the main *authentication methods* used in *cloud applications*?
8. What are the advantages and risks of using *containers* compared to *virtual machines*?
9. How can *cloud applications* be made compliant with regulatory requirements (e.g., *GDPR*)?
10. How can isolation and security between *virtual machines* be configured on a *cloud platform*?

Topic 9

GENERAL ISSUES IN WEB APPLICATION SECURITY

The objective is to study common web application vulnerabilities and methods for their mitigation, and to gain practical skills in analyzing web security issues.

Tasks for Independent Study

1. *Configure HTTPS* for a local web server using *Let's Encrypt*. For example, use the *Certbot* utility to automatically obtain and renew a *TLS* certificate.
2. *Test a web application for vulnerabilities* according to the *OWASP Top 10*. Perform testing for common threats such as *SQL* Injection (*SQLi*), Cross-Site Scripting (*XSS*), Cross-Site Request Forgery (*CSRF*), etc. Prepare a report with identified issues and mitigation recommendations.
3. *Implement CORS (Cross-Origin Resource Sharing) policies* for your *API*. Configure appropriate headers, including *Access-Control-Allow-Origin*, *Access-Control-Allow-Methods*, *Access-Control-Allow-Headers*, etc.
4. *Analyze the security headers* of a website. Check for the presence and correctness of headers such as *Content-Security-Policy*, *X-Frame-Options*, *X-Content-Type-Options*, and others. Suggest improvements based on current security best practices.
5. *Write a database backup script* for a web application. Automate the backup process using a *Bash* or *Python* script or a suitable tool. Ensure daily backups with integrity checks and recovery capability.

Brief Theoretical Background

1. Web development platforms and their security features.
2. The *HTTP* protocol and its secure use.
3. Forms and protection against attacks.
4. Client-side and server-side threats.
5. Methods for preventing security risks.

Web Development Platforms and Methods for Securing Them

Web development involves using platforms and frameworks such as *Django* (*Python*), *Spring* (*Java*), *Express* (*Node.js*), and *Laravel* (*PHP*). Regardless of the

platform chosen, the key elements of security include user authentication, data encryption, and protection against common attacks such as *SQL injections* and *XSS*. Most modern frameworks have built-in security features, such as automatic *CSRF* protection and support for parameterized *SQL* queries to prevent injections.

The main sources of information on security threats are *OWASP Top 10* (Table 2.1), *MITRE Common Weakness Enumeration (CWE)* [29], *MITRE CVE Program* [30], etc. Web applications are constantly under attack, as evidenced by the data in Table 9.1, which lists some selected vulnerabilities.

Таблиця 9.1 – Web Framework Vulnerabilities

CVE	Framework	Description
CVE-2023-37979	*Ninja Forms (WordPress)*	Cross-Site Scripting (*XSS*) vulnerability in the *Ninja Forms* plugin (≤ 3.6.26), allowing JavaScript injection in the frontend, potentially compromising the site.
CVE-2023-28708	*Apache Tomcat*	Session cookies may be exposed over insecure connections if the `secure` flag is not set while using the `X-Forwarded-Proto` header.
CVE-2023-29450	*Zabbix*	*Zabbix* (≤ 6.4.1) allows unauthorized access to confidential information by processing JavaScript before authentication.
CVE-2022-47148	*WooCommerce (WordPress)*	*CSRF* vulnerability in the *WooCommerce PDF Invoices & Packing Slips* plugin (≤ 3.2.6), allowing unauthorized modification of user data.
CVE-2023-3247	*PHP*	*SOAP Digest Authentication* in *PHP* (≤ 8.1.20) may leak uninitialized memory between the client and the server.
CVE-2023-3824	*PHP*	Buffer overflow when parsing *PHAR* files, which may lead to memory corruption or arbitrary code execution.
CVE-2022-41040	*Microsoft Exchange*	*SSRF* vulnerability in *Microsoft Exchange* that can be chained to achieve remote *PowerShell* execution.
CVE-2022-41082	*Microsoft Exchange*	*RCE* vulnerability in *Microsoft Exchange*, exploiting *ProxyNotShell* to execute *PowerShell* commands remotely.
CVE-2023-3823	*PHP*	Improper global state handling in *libxml* can cause leakage of configuration data across *PHP* modules in the same process.

The development process must account for known security issues and adhere to secure development and deployment practices throughout the entire software lifecycle, such as by following the *Secure Development Lifecycle (SDLC)*.

Ensuring the security of web applications requires the use of tools to detect and eliminate vulnerabilities. *Static Application Security Testing (SAST)* tools analyze the source code to identify issues during development. Dynamic tools (*DAST*), such as *OWASP ZAP*, simulate real-world attacks to test applications in real-time. *Web Application Firewalls (WAF)*, such as *Cloudflare WAF*, provide protection against network threats, while tools for managing *SSL/TLS* certificates simplify traffic encryption, ensuring confidentiality.

It is essential to regularly update software, including third-party libraries. Updates contain patches to fix known vulnerabilities.

Using strong passwords and two-factor authentication is important for account protection. Server security should include firewalls, intrusion detection systems (*IDS*), and *Web Application Firewalls (WAF)*.

To maintain security relevance, one should constantly monitor for new cyber threats and vulnerabilities using appropriate resources and tools.

The HTTP Protocol and Its Secure Use

HTTP (Hypertext Transfer Protocol) is the basic protocol for data transmission in web applications. However, it does not provide data encryption. Using *HTTPS* (*HTTP Secure*) with *SSL/TLS* encryption is mandatory to protect sensitive information during transmission, preventing *Man-in-the-Middle* attacks.

HTTP GET and POST in Terms of Security

HTTP GET and *POST* requests have important security differences. The *GET* request transmits data via the URL, making it insecure for sensitive information since the URL can be stored in the browser history or transmitted in referrers. *GET* requests also have limitations on the amount of data that can be sent. For example:

```
GET /search?q=keyword HTTP/1.1
```

The *POST* request uses the request body to transmit data, making it more suitable for sensitive data such as passwords or user information, as the data does not appear in the URL. Here is an example of a POST request:

```
POST /login HTTP/1.1
```

```
Host: example.com
Content-Type: application/x-www-form-urlencoded

username=user&password=pass
```

Security issues with *GET*:

- Data is transmitted via the URL and may be stored in the browser history.
- Data may be accessible through referrers or server logs.
- There are limitations on the size of the transmitted data.

Security issues with *POST*:

- Although data is not stored in the *URL*, it can still be intercepted if HTTPS is not used.
- CSRF attacks are possible, so CSRF tokens should be used for protection.

It should be noted that using *HTTPS* for request encryption is mandatory for both methods. It is also important to implement input validation and *CSRF* protection.

Configuring HTTPS in Apache

```
<VirtualHost *:443>
ServerName example.com
SSLEngine on
SSLCertificateFile /etc/ssl/certs/example.com.crt
SSLCertificateKeyFile /etc/ssl/private/example.com.key
</VirtualHost>
```

Base64 encoding is widely used in web development and cybersecurity to convert binary data into a text format. This enables the safe transmission of data over text-based protocols such as *HTTP*. *Base64* is often used to encode images, files, and other binary information in formats like *JSON*, *HTML*, and emails to prevent data corruption during transmission.

Non-Latin Characters in URLs and Phishing

Encoding non-Latin characters in URLs using *UTF-8* and percent-encoding (for example, encoding the Cyrillic letter “к” as `%D0%BA`) allows the transmission of characters through systems that do not support them directly. However, this can also be exploited for phishing attacks. Attackers can create URLs with non-Latin characters that look similar to legitimate sites (e.g., using the Cyrillic letter “а” instead of the Latin letter “a”) to deceive users and lure them to malicious resources. To protect against this, users should inspect URLs before clicking and use antivirus software.

Forms and Their Protection Against Attacks

Forms are a key element of user interaction with web applications, making their protection critical. The main threats are *SQL injection*, *XSS*, and *CSRF*. To ensure security, it is necessary to:

- Validate all input data.
- Protect against *CSRF* using tokens.
- Escape input data to prevent *XSS*.

Protection Example in Django

Django provides automatic *CSRF* protection through built-in mechanisms:

```
from django.views.decorators.csrf import csrf_protect

@csrf_protect
def my_view(request):
return HttpResponse('Protected page')
```

XSS Protection Example in PHP

```
<?php
function escape($input) {
            return htmlspecialchars($input, ENT_QUOTES, 'UTF-8');
        }
    ?>
    <form method="post" action="submit.php">
    <input type="text" name="username" value="<?php echo
    ↪  escape($_POST['username']); ?>">
<input type="submit" value="Submit">
</form>
```

Threats on the Client and Server Sides

The main threats on the client side are *XSS*, phishing, and untrusted scripts. To protect against these, use *Content Security Policies (CSP)* and avoid dynamically generating *HTML* without proper validation.

The main threats on the server side include *SQL injection*, file attacks, and exploitation of database vulnerabilities. Parameterized queries and access control restrictions help protect servers.

Example of a Parameterized SQL Query in PHP

```
$stmt = $pdo->prepare('SELECT * FROM users WHERE username =
↪  :username');
$stmt->execute(['username' => $username]);
```

Methods for Avoiding Threats

To avoid common threats on both the client and server sides, the following methods should be used:

- Use *HTTPS* to encrypt transmitted data.
- Protect against *CSRF* using tokens.
- Use parameterized *SQL* queries to prevent *SQL injection*.
- Regularly update frameworks and libraries to eliminate vulnerabilities.
- Use password hashing (e.g., *bcrypt*, *Argon2*) and multi-factor authentication.

Review Questions

1. What are the main security threats on web platforms like *Django* or *Spring*?
2. How does *HTTPS* ensure data security in web applications?
3. What are *CSRF* tokens, and how do they help protect web forms?
4. How do *XSS* attacks work, and what client-side protection methods can be used?
5. What methods can be applied to secure server databases from *SQL injection*?
6. How can web forms be protected from malicious user inputs?
7. What server security measures should be implemented to prevent unauthorized access?
8. What authentication methods can help increase web application security?
9. How does regular framework updating help ensure web application security?
10. How does multi-factor authentication help prevent unauthorized access to web applications?

Topic 10

WEB APPLICATION SECURITY: COOKIES, SESSIONS, AND ATTACKS

The objective is to study authentication mechanisms, cookies, sessions, and related attacks, and to gain practical skills in identifying and mitigating session management vulnerabilities.

Tasks for Independent Study

1. *Configure secure cookies* using the `HttpOnly` and `Secure` flags and explain their advantages.
2. *Identify vulnerable sessions* in a web application using Burp Suite (e.g., check for session ID reuse and predictability).
3. *Implement CSRF tokens* in a login form and verify that the protection works.
4. *Perform a session hijacking attack* in a test environment (explain how the attack works and how to defend against it).
5. *Explore how JWT (JSON Web Tokens)* enhances session security compared to traditional cookie-based sessions.

Brief Theoretical Background

1. Cookies and sessions: fundamentals and security.
2. *Cross-Site Request Forgery* (*CSRF*) attacks.
3. *XML External Entity* (*XXE*) attacks.
4. Redirection and its security implications.

Cookies and Sessions: Basic Concepts and Security

Cookies are small text files stored in the user's browser that contain information used for authentication, preferences, or tracking activity. Sessions are a mechanism that allows the server to identify the user across multiple requests. Cookies are often used to store session identifiers.

Google plans to gradually phase out third-party cookies in its *Chrome* browser as part of the *Privacy Sandbox* initiative. The primary goal is to protect users' privacy and give them more control over how their data is used. The phase-out will start in 2024. While third-party cookies will be phased out, *first-party cookies* will remain active to support internal website functions such as maintaining sessions and user preferences. Additionally, Google is developing new technologies, such as *FLoC*

(Federated Learning of Cohorts) and *Topics API*, to support advertising without compromising user privacy [31–33].

The main security issues related to cookies and sessions include:

• *Session Hijacking* – Attackers can steal session identifiers and gain access to the user's account.

• *Insecure Cookies* – Unprotected cookies can be transmitted over an unencrypted connection or used for malicious purposes.

Example of Secure Cookie Configuration in PHP

```
setcookie('session_id', $sessionId, [
        'secure' => true,        // HTTPS only
        'httponly' => true,      // Not accessible via JavaScript
        'samesite' => 'Strict'   // Prevents CSRF
    ]);
```

This example shows how to configure cookies for security: the `secure`, `httponly`, and `samesite` attributes help prevent cookie-related attacks.

In terms of cybersecurity, using cookies can compromise the security of web applications:

• *CVE-2023-26136*: Vulnerability in the tough-cookie package allows "Prototype Pollution" through improper handling of cookies when `rejectPublicSuffixes=false`. This may enable attackers to manipulate objects in the system, leading to unpredictable consequences during code execution on the server.

• *CVE-2023-46218*: Vulnerability in *curl* allows setting "super-cookies" using mixed uppercase and lowercase letters in the domain. This may result in cookies being transmitted to unrelated sites and domains, creating serious security issues.

• *CWE-539: Use of Persistent Cookies Containing Sensitive Information*: Vulnerability related to storing sensitive information in persistent cookies, which may include session identifiers or even passwords, leading to data leakage if the cookies are intercepted or compromised.

• *CWE-1275: Sensitive Cookie with Improper SameSite Attribute*: Vulnerability associated with improper configuration of the *SameSite* attribute, allowing *CSRF* attacks. If the *SameSite* attribute is not configured correctly, cookies may be used to send malicious requests on behalf of the user.

CSRF (Cross-Site Request Forgery) Attacks

CWE-352: Cross-Site Request Forgery (CSRF). If a web server is designed to accept client requests without a mechanism for verifying whether the request was

intentional, an attacker may trick a client into unintentionally sending a request to the web server that will be processed as authentic. This can be done via URLs, image loading, *XMLHttpRequest*, etc., potentially leading to data leakage or unintended code execution[1].

Thus, *CSRF* is an attack where an attacker tricks a user into performing unwanted actions on a website where the user is already authenticated. For example, an attacker may create a fake form that submits a password change request without direct access to the user's account.

CSRF Attack Example

```
<form action="http://victim.com/change-password" method="POST">
<input type="hidden" name="password" value="newpassword">
<input type="submit" value="Submit">
</form>
```

A user who is already authenticated on `victim.com` might accidentally click this link, and their password would be changed to `newpassword` if the site is not protected against *CSRF*.

CSRF Protection Methods

- Use *CSRF* tokens.
- Configure the `SameSite` attribute for cookies.
- Check *HTTP* headers to detect suspicious requests.

Example of CSRF Protection in PHP

```
if ($_POST['csrf_token'] !== $_SESSION['csrf_token']) {
        die('CSRF attack detected!');
    }
```

This code checks for the presence of a *CSRF* token in the request and compares it to the token stored in the session.

[1] https://cwe.mitre.org/data/definitions/352.html

XXE (XML External Entity) Attacks

CWE-611: Improper Restriction of XML External Entity Reference. *XML* documents may optionally contain a *Document Type Definition (DTD)* that allows defining XML entities. An entity can be defined by specifying a substitution string in the form of a *URI*. The *XML* parser may access the content of this *URI* and insert it back into the *XML* document for further processing[2].

An attacker may use an *XML* file with an external entity pointing to a *URI* like `file://` to force the application to access a local file's content. For instance, the *URI* `file:///c:/winnt/win.ini` may refer to a configuration file in *Windows*, or `file:///etc/passwd` may refer to a password file in Unix. It is also possible to use *URIs* of other protocols, such as `http://`, to send requests to remote servers, bypassing restrictions or concealing the attack source. The content of such files may be inadvertently displayed in the application, revealing sensitive information.

Vulnerable XML Example

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
        <!ELEMENT foo ANY >
        <!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
<foo>&xxe;</foo>
```

In this example, the *XML* document attempts to access the `/etc/passwd` file, potentially leading to information leakage.

Protection Against XXE Attacks

Disabling external entities in *XML* parsers and using secure libraries for *XML* processing are key methods to protect against such vulnerabilities.

Example of XXE Protection in Java

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
dbf.setFeature("http://apache.org/xml/features/disallow-doctype-decl",
↪  true);
```

[2] https://cwe.mitre.org/data/definitions/611.html

This code disables the processing of external entities in XML documents, protecting the system from *XXE* attacks.

Redirects and Their Security

Redirects are a common feature of web applications that can be used to direct users to other pages or sites. However, if redirects are not properly controlled, attackers can exploit them for phishing attacks or to redirect users to malicious websites.

CWE-601: URL Redirection to Untrusted Site ('Open Redirect'), refers to a vulnerability where a web application accepts a user-supplied *URL* and uses it for redirection without proper validation. This can lead to phishing attacks, where an attacker manipulates the *URL* to trick users into visiting a malicious site that appears legitimate but steals their data or executes harmful code[3].

Example of Vulnerable Redirect

```
header('Location: ' . $_GET['url']);
```

This code redirects the user to any *URL* specified in the request, which can be exploited for an attack.

Protection Against Unsafe Redirects

- Validate URLs before performing a redirect.
- Use a whitelist of allowed safe URLs.

Example of Safe Redirect

```
$allowed_urls = ['home.php', 'profile.php'];
if (in_array($_GET['url'], $allowed_urls)) {
    header('Location: ' . $_GET['url']);
}
```

In this example, the redirect is performed only to safe URLs from the list of allowed ones.

[3]https://cwe.mitre.org/data/definitions/601.html

Review Questions

1. What are *cookies*, and how can they be configured to secure a web application?
2. What security issues arise with the use of *sessions*?
3. What is a *CSRF* attack, and how can a web application be protected against it?
4. What is an *XXE* attack, and how can it be prevented?
5. How can unsafe *redirects* be used in phishing attacks?
6. How can *session hijacking* be prevented?
7. Which *cookie attributes* can help prevent security-related attacks?
8. What methods can be used to ensure the security of *redirects* in web applications?
9. How can *CSRF tokens* be used correctly in web applications?
10. Which *XML libraries* can be used to prevent *XXE* attacks?

Topic 11

MOBILE APPLICATION SECURITY

The objective is to study mobile application threats and protection mechanisms, and to gain practical skills in analyzing mobile apps based on secure programming principles.

Tasks for Independent Study

1. *Analyze the permissions of an* Android *application* using *adb* (check what permissions the app requests and why).
2. *Write code to securely store passwords in an* iOS *application* using the *Keychain*.
3. *Identify vulnerabilities in a mobile application* using *MobSF* (*Mobile Security Framework*) and provide recommendations for remediation.
4. *Implement biometric authentication* (*Face ID* or *Touch ID*) in a test mobile application.
5. *Compare the security of native apps and* Progressive Web Apps (PWA), focusing on data storage and access to hardware resources.

Brief Theoretical Background

1. Mobile operating system platforms and mobile app architecture.
2. Secure programming for *Android OS*.
3. Secure programming for *iOS*.
4. Network operation protectio.

Mobile Operating System Platforms and Mobile Application Architecture

Mobile devices run on various operating systems (Fig. 11.1), the most popular being *Android* and iOS. These operating systems have different security mechanisms, such as permission control, application isolation, and data encryption. A *mobile platform* is a general term for technology that enables the creation of applications that run on mobile devices under *Android* and iOS.[1].

The typical architecture of a mobile application includes several layers:

[1] One Service (oneservice.in.ua)

Figure 11.1 – Mobile OS usage according to StatCounter.

• *User Interface (UI)* provides a convenient and intuitive interaction with the application through widgets, animations, and other visual elements.

• *Business Logic* is responsible for the functionality of the application and manages core processes for solving applied tasks.

• *Data Access* ensures data storage and retrieval through local databases or interacts with remote servers using *REST*, *GraphQL APIs*, etc.

• *Authentication, Security, and Network Services* provide secure communication through *SSL/TLS* encryption and implement authentication mechanisms such as *OAuth* or *JWT*.

There are at least four typical architectures for mobile applications[2], and the choice of architecture significantly depends on the purpose of the application, developer experience, chosen technologies, etc.:

• *MVC (Model-View-Controller)*: This is a classic model where *Model* is responsible for data, *View* for the interface, and *Controller* manages the interaction logic between them. It is used for clear separation of logic, interface, and data, which facilitates development and code maintenance.

• *MVP (Model-View-Presenter)*: In this architecture, the *Presenter* acts as an intermediary between the *Model* and *View*. The *View* only displays data, while the *Presenter* handles all interaction logic. This allows for testing business logic independently from the interface.

• *MVVM (Model-View-ViewModel)*: Here, the *ViewModel* separates business logic from the interface (*View*) by interacting with the *Model*. This architecture is

[2]IT company WEZOM (wezom.com.ua)

suitable for applications with a lot of data and complex display logic.

• *Clean Architecture*: Based on the principles of separating business logic, processes, and technologies. The goal is to create an independent architecture that does not depend on interfaces, databases, or external services, making it flexible and easy to test.

Key security issues for mobile applications include controlling access to device resources, protecting locally stored data, and securely performing network operations.

Secure Programming for Android OS

Android is an open-source operating system that uses permission models to protect access to critical device resources (camera, microphone, contacts, etc.). *Android* applications run in isolated environments, preventing unauthorized access to the data of one application by another.

Android applications are developed in several languages, such as *Java*, *Kotlin*, *C*, and *C++*, as well as *HTML5* (for mobile web applications). *Android* apps may use native code written in *C* and *C++* to implement specific functionalities, such as working with the Linux kernel. It should be noted that interactions with such applications via the *Java Native Interface (JNI)* are potentially susceptible to vulnerabilities, such as buffer overflow and other issues that may be present in native code implementation and are common to *C* and *C++*.

The *SEI* Android *Secure Coding Standard* provides rules for securely coding applications on the *Android* platform, aiming to create secure, reliable, and attack-resistant software systems. The primary focus is on eliminating undefined behaviors that may lead to software vulnerabilities. Compliance with these rules is necessary for system security, but not sufficient – secure system design is also crucial. For critical systems where security is a priority, stricter requirements, such as static memory allocation, are stipulated. The *CERT standard* for *Android* partially integrates rules from other programming languages, such as *C* and *Java*, while considering the specifics of the *Android* platform [34].

The standard distinguishes rules and recommendations in four categories:

- Rules specific only to *Android*.
- Rules and recommendations for *C*.
- Best practices (recommendations) for secure coding in *Java*.
- Rules for *Java*.

Example of *SSL/TLS* Certificate Verification in *Android* Applications

Rule 04. Network - *SSL/TLS* (*NET*). DRD19. Properly verify server certificate on *SSL/TLS*.

Android applications that use *SSL/TLS* protocols for secure communication must properly verify server certificates. Basic verification includes:

- Checking that the subject (*CN*) of the *X.509* certificate matches the *URL*.
- Verifying that the certificate is signed by a trusted *Certification Authority* (*CA*).
- Validating the signature correctness.
- Ensuring the certificate is not expired.

There are also standards for *Android* developers by the Japan Smartphone Security Association (JSSEC) [35; 36].

The *Android Application Secure Design/Secure Coding Guidebook*, like the *SEI* Android *Secure Coding Standard* mentioned above, contains rules, recommendations, and best practices for secure design and coding specifically for *Android* application developers.

Considering the rapid growth of the smartphone market and the increasing openness of mobile platforms like Android, developers now have access to features that were previously restricted. However, such openness brings increased responsibility for developers since improperly designed or coded applications can lead to data leaks or create security vulnerabilities. Given the open nature of Android, developers need to be especially vigilant regarding security issues, as there are fewer restrictions on app releases compared to iOS.

The *Android Application Secure Design/Secure Coding Guidebook* contains a significant number of examples for the correct design of *Android* applications with detailed examples. All information is divided into four major groups:

- Basic Knowledge of Secure Design and Secure Coding.
- Using Technology in a Safe Way.
- How to Use Security Functions.
- Difficult Problems.

Permission Control in Android

Starting from *Android* 6.0 (API level 23), users can allow or deny access to certain application features during its use. This gives users better control over which data the application can access.

Example of Requesting Camera Permission

```
if (ContextCompat.checkSelfPermission(this,
↪  Manifest.permission.CAMERA)
!= PackageManager.PERMISSION_GRANTED) {
    ActivityCompat.requestPermissions(this,
    new String[] { Manifest.permission.CAMERA }, REQUEST_CAMERA);
```

```
    }
```

This code checks if the application has permission to access the camera, and if not, it requests the user's permission.

Encryption of Local Data in Android

To protect confidential data stored on the device, applications can use the *Android* Keystore to securely store encryption keys.

```
KeyGenerator keyGenerator =
↪  KeyGenerator.getInstance(KeyProperties.KEY_ALGORITHM_AES,
↪  "AndroidKeyStore");
keyGenerator.init(new KeyGenParameterSpec.Builder("myKeyAlias",
KeyProperties.PURPOSE_ENCRYPT | KeyProperties.PURPOSE_DECRYPT)
.setBlockModes(KeyProperties.BLOCK_MODE_GCM)
.setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_NONE)
.build());
SecretKey key = keyGenerator.generateKey();
```

This example shows how to create a key in the Keystore for encrypting and decrypting data.

Example of Using HTTPS in *Android* with HttpURLConnection

```
URL url = new URL("https://example.com/api");
HttpsURLConnection connection = (HttpsURLConnection)
↪  url.openConnection();
connection.setRequestMethod("GET");
InputStream inputStream = connection.getInputStream();
// Process the response
```

Secure Programming for iOS

iOS is a closed-type operating system with a high level of access control to system resources. Apple provides developers with strict guidelines for protecting user data and uses application isolation models to ensure security.

Permission Control in iOS. Like *Android*, iOS requires applications to request permissions to access sensitive resources, such as the camera or location. Permission requests are made through the `Info.plist` configuration file, where an explanation is provided for the user as to why the application needs access.

Example of Configuring Location Access Permission

```
<key>NSLocationWhenInUseUsageDescription</key>
<string>We need access to your location for navigation
↪  purposes.</string>
```

Data Encryption in iOS

Apple provides data encryption for data stored in applications through the *Data Protection mechanism*. Additionally, developers can use `Keychain` to store sensitive information such as passwords.

Example of Storing Data in Keychain

```
let keychain = Keychain(service: "com.example.myapp")
keychain["password"] = "supersecretpassword"
```

This code saves a password in the *Keychain*, which uses *encryption* to protect sensitive information.

Protecting Network Operations

Both platforms, *Android* and *iOS*, support the use of *HTTPS* to ensure secure data transmission over the network. Using *HTTPS* prevents data interception by attackers during transmission between the application and the server.

Example of Using HTTPS in iOS with URLSession

```
let url = URL(string: "https://example.com/api")!
let task = URLSession.shared.dataTask(with: url) { data, response,
↪  error in
    // Process the response
}
```

```
    task.resume()
```

Review Questions

1. What is the main architecture of a mobile application, and what are its main levels?
2. What permission control methods are used in *Android* and *iOS*?
3. How can you ensure the protection of data stored locally on an *Android* device?
4. What is the *Android* Keystore, and how does it help ensure security?
5. How can the *Keychain* be used in *iOS* to securely store sensitive information?
6. How can you ensure the security of network operations in mobile applications?
7. What are the advantages and disadvantages of application isolation in *Android* and *iOS*?
8. How can you protect data during transmission between a mobile application and the server?
9. How does the permission system work in *Android* 6.0 and above?
10. What are the main security mechanisms of mobile platforms *Android* and *iOS* to prevent unauthorized access?

Topic 12

SECURE SOFTWARE DEVELOPMENT LIFECYCLE (SSDLC)

The objective is to study the concept of the *Secure Software Development Lifecycle (SSDLC)* and its phases, and to gain practical skills in applying the *SSDLC* methodology.

Tasks for Independent Study

1. *Create an SSDLC implementation plan* for a fictional company (define the phases, responsible parties, and security testing tools).
2. *Conduct Threat Modeling* for a new application feature (use approaches such as *STRIDE* or *DREAD*).
3. *Write a secure code review policy* for the development team (who performs reviews, what is reviewed, which tools are used).
4. *Analyze the SSDLC phases in the context of the* Agile *methodology*, considering how frequently security checks should be performed.
5. *Develop a security testing plan* for each SSDLC phase (planning, design, development, testing, release).

Brief Theoretical Background

1. Overview of *SSDLC* structure and components.

Overview of SSDLC

The *Secure Software Development Life Cycle (SSDLC)* integrates security practices at each stage of software development. *SSDLC* helps identify, avoid, and fix vulnerabilities early in the development process, reducing the security risks (Fig. 12.1[1]).

The main stages of SSDLC are:

- Requirements Definition.
- Design.
- Development.
- Testing.
- Deployment.
- Maintenance and Monitoring.

[1] Checkmarx: Secure SDLC

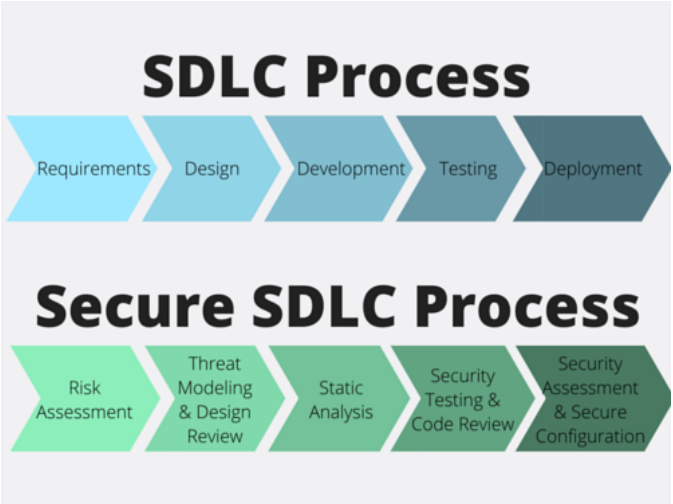

Figure 12.1 – *SSDLC* process stages

Requirements Definition

In this initial stage, software requirements are defined, including security requirements. It is important to identify potential threats and risks, and security experts should participate in the requirements definition process. Key actions include:

- Identifying threats and vulnerabilities.
- Defining security requirements for functionality and data.

Design

This stage involves designing the software architecture with security requirements in mind. Threat analysis is conducted, and protective mechanisms are defined:

- Conducting Threat Modeling.
- Designing security measures: encryption, authentication, access control.

Development

During development, secure coding practices are applied, such as input validation, permissions management, and using trusted libraries:

- Using secure coding patterns to prevent injections.
- Performing static code analysis to identify vulnerabilities.

Testing

Various types of testing are conducted at this stage to detect vulnerabilities, including penetration testing and automated analysis:

- Penetration Testing to simulate real-world attacks.
- Using dynamic analysis tools to identify vulnerabilities during execution.

Deployment

Deploying the software should include securing the runtime environment, configuring encryption, and access control:

- Using DevSecOps to automate secure deployment.
- Configuring firewalls, VPNs, and other network security tools.

Maintenance and Monitoring

After deployment, continuous monitoring for new vulnerabilities is essential. Updates and patches are applied to maintain security:

- Regular updates and patching of software.
- Using threat monitoring tools (*IDS/IPS*).

SSDLC Tools and Methods

- *Code Review* – manual inspection of the code for potential vulnerabilities.
- *Dynamic Analysis* – testing the application during runtime to identify vulnerabilities.
- *Penetration Testing* – simulating attacks on the application to find weaknesses.
- *Patching and Updates* – regular updates to close vulnerabilities.

CVE and CWE in the Context of SSDLC

CVE (Common Vulnerabilities and Exposures) is a system of unique identifiers for known vulnerabilities, helping to identify and fix software vulnerabilities.

CWE (Common Weakness Enumeration) is a classification of types of vulnerabilities, aiding developers in avoiding common mistakes during software development.

Principles of SSDLC

- *Security by Default* – software should be secure without additional configuration
- *Least Privilege Principle* – each user or process should have the minimum necessary privileges

• *Incident Response* – the need to have a response plan in place if a threat is detected

• *Continuous Improvement* – security threats constantly evolve, so SSDLC processes must continuously improve

Review Questions

1. What are the stages of *SSDLC*, and why are they important?
2. What is the role of static code analysis in the *SSDLC* process?
3. What is *CVE*, and how does it affect software security maintenance?
4. How can security be integrated at the software deployment stage?
5. What is *Penetration Testing*, and how do its results help improve security?
6. What are the main security principles in the context of *SSDLC*?
7. How does *Threat Modeling* assist in designing secure software?
8. What monitoring methods can be applied during the system maintenance stage?
9. What is the difference between *CVE* and *CWE*, and how are they applied in *SSDLC*?
10. How can automated *DevSecOps* tools enhance security during the development process?

Topic 13

HIGH-LEVEL SECURITY PROGRAMMING

The objective is to study methods of integrating security at the software architecture level and gain skills in working with encryption protocols.

Tasks for Independent Study

1. *Write a data encryption module using AES-256* (e.g., in *Python* with the *PyCryptodome* library).
2. *Explore the implementation of the OAuth 2.0 protocol* in a web application (understand the authorization flow and token issuance).
3. *Implement a Role-Based Access Control (RBAC) system* in an application (define roles, permissions, and validation mechanisms).
4. *Compare password hashing algorithms: bcrypt*, *scrypt*, and *Argon2* (performance, security, configuration).
5. *Create a multi-factor authentication scheme* for an *API* (e.g., passwords + *OTP* or hardware security keys).

Brief Theoretical Background

1. High-level security assurance methods.
2. Hashing and cryptography using cryptographic libraries.
3. Authentication via *GSSAPI*.
4. Use of digital signatures.

High-Level Security Methods

High-level security in corporate systems involves using advanced protective measures to ensure the integrity, confidentiality, and authenticity of data. This includes *hashing*, *encryption*, *digital signatures*, and *authentication* and authorization protocols.

Key security methods include:

• *Hashing* – a one-way process that transforms data into a fixed-length value, used to verify data integrity.

• *Encryption* – the process of converting data into an encrypted form to ensure confidentiality.

• *Digital Signatures* – using cryptographic algorithms to ensure the authenticity and integrity of data.

• *Authentication* – verifying the authenticity of a user or service, including protocols such as *GSSAPI*.

Hashing and Cryptography Using Cryptographic Libraries

Hashing is essential for ensuring data integrity. One of the most commonly used hashing algorithms is *SHA-256*, which provides a high level of security. Cryptographic libraries like *OpenSSL* offer tools to use various hashing and encryption algorithms.

Example of Hashing Using OpenSSL (SHA-256).

```
#include <openssl/evp.h>
#include <string.h>

unsigned char hash[EVP_MAX_MD_SIZE];
unsigned int hash_len;

EVP_MD_CTX *mdctx = EVP_MD_CTX_new();
EVP_DigestInit_ex(mdctx, EVP_sha256(), NULL);
EVP_DigestUpdate(mdctx, data, strlen(data));
EVP_DigestFinal_ex(mdctx, hash, &hash_len);
EVP_MD_CTX_free(mdctx);
```

This code demonstrates how to create a hash for a data array using *SHA-256* with *OpenSSL*.

Encryption is another crucial tool for ensuring data confidentiality. Cryptographic algorithms like *AES* (*Advanced Encryption Standard*) are widely used to protect data in corporate systems.

Example of Encryption Using AES.

```
EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
EVP_EncryptInit_ex(ctx, EVP_aes_256_cbc(), NULL, key, iv);
EVP_EncryptUpdate(ctx, ciphertext, &len, plaintext, plaintext_len);
EVP_EncryptFinal_ex(ctx, ciphertext + len, &len);
EVP_CIPHER_CTX_free(ctx);
```

This example uses *AES-256* to encrypt data in `CBC mode`, providing a high level of protection.

Authentication via GSSAPI

GSSAPI (*Generic Security Services Application Programming Interface*) is a standard for authentication and establishing secure sessions between users and services. This *API* is widely used in corporate networks to support secure authentication using protocols such as *Kerberos*.

GSSAPI allows applications to use various security mechanisms for authentication without the need to configure cryptographic parameters in detail. For example, *GSSAPI* can be used to implement user authentication via *Kerberos*, enabling reliable identification of users in corporate networks.

Example of Authentication via GSSAPI (Kerberos).

```
OM_uint32 major_status, minor_status;
gss_name_t server_name;
gss_buffer_desc name_buf;
gss_ctx_id_t context = GSS_C_NO_CONTEXT;

name_buf.value = "service@hostname";
name_buf.length = strlen((char *)name_buf.value);
major_status = gss_import_name(&minor_status, &name_buf,
GSS_C_NT_HOSTBASED_SERVICE, &server_name);

major_status = gss_init_sec_context(&minor_status,
↪  GSS_C_NO_CREDENTIAL,
&context, server_name, GSS_C_NO_OID,
GSS_C_MUTUAL_FLAG | GSS_C_REPLAY_FLAG,
0, NULL, GSS_C_NO_BUFFER, NULL, NULL, NULL);
```

This example demonstrates the process of initializing an authentication context using *GSSAPI* to establish a secure session.

Using Digital Signatures

Digital signatures provide authenticity and integrity of data. They use *asymmetric encryption* to verify the data source and ensure that the data has not been altered during transmission. In corporate systems, digital signatures are widely used for signing electronic documents or protecting software.

Example of Creating a Digital Signature Using OpenSSL.

```
EVP_PKEY *private_key = load_private_key();
EVP_MD_CTX *mdctx = EVP_MD_CTX_new();
EVP_SignInit(mdctx, EVP_sha256());
EVP_SignUpdate(mdctx, message, message_len);
EVP_SignFinal(mdctx, signature, &sig_len, private_key);
EVP_MD_CTX_free(mdctx);
```

This code shows how to create a *digital signature* for a message using the *SHA-256* algorithm and a private key.

Review Questions

1. What is *hashing*, and what is its primary purpose in security systems?
2. Which cryptographic algorithms are used for *data encryption* in corporate systems?
3. How does *GSSAPI* work, and what is its significance in the authentication process?
4. What role do *digital signatures* play in ensuring data security?
5. How can *cryptographic libraries* be used to implement *encryption* and *hashing*?
6. What risks may arise from improper use of *encryption*?
7. What is *AES*, and why is this algorithm widely used for *data protection*?
8. How can *GSSAPI* be used to secure corporate networks?
9. What types of *digital signatures* are used in corporate systems, and how do they help ensure *data integrity*?
10. How does *OpenSSL* help implement *cryptographic solutions* in software?

Topic 14

ANALYZING SOFTWARE WITHOUT ACCESS TO SOURCE CODE

The objective is to understand the principles of reverse engineering and its application in security, and to gain practical skills in using reverse engineering tools.

Tasks for Independent Study

1. *Analyze a binary file* using *IDA Pro* or *Ghidra* (identify entry points and main functions).
2. *Locate encryption functions* in the reverse-engineered code and try to understand the algorithm being used.
3. *Write a patch* to modify the program's behavior (e.g., bypassing license key verification).
4. *Explore code obfuscation techniques* such as *packers*, *string encryption*, or *code virtualization* used to prevent reverse engineering.
5. *Perform malware analysis* using a *debugger* such as *OllyDbg* or *x64dbg*, focusing on key execution points.

Brief Theoretical Background

1. Definition and significance of reverse engineering.
2. Core methods of software analysis without source code.
3. Tools used for reverse engineering.
4. Ethical and legal aspects of reverse engineering.

Reverse Engineering Methods

Reverse engineering is the process of analyzing software to discover its structure and functionality without access to the source code. This method is used for analyzing suspicious software, finding vulnerabilities, and fixing bugs in compiled programs.

The main reverse engineering methods include:

• *Decompilation* – converting executable code back into high-level code, allowing researchers to understand the program's logic.

• *Disassembly* – converting executable code into low-level assembly code, which enables analyzing the program's operation at the machine level.

• *Binary File Analysis* – examining compiled binary files to detect hidden functions or vulnerabilities.

Example of Using a Java Decompiler.

```
javap -c MyClass.class
```

This example demonstrates the use of the `javap` tool for decompiling Java bytecode, allowing the inspection of class methods.

Packing and Its Analysis

Packing is a process used to compress or encrypt executable files to hide their original code. Packed programs are often used to conceal malware or protect code from reverse engineering. Unpacking such files is a crucial step for conducting analysis.

Example of a Tool for Analyzing Packed Files (UPX).

```
upx -d packed_binary
```

This example shows the use of *UPX* to unpack an executable file, allowing the analysis of its original code.

Decompilation

Decompilation is the process of converting executable code into high-level code. This method helps recover the program's logic and study its behavior. There are decompilers for various programming languages that allow the analysis of programs without access to the source code.

Example of Using a Decompiler for .NET.

```
ildasm MyProgram.exe
```

This example demonstrates the use of the `ildasm` tool for decompiling *.NET applications*, enabling the inspection of the program's code structure and methods.

Network Traffic Analysis

Network traffic analysis allows studying how a program interacts with the network and identifying possible vulnerabilities or malicious actions. For example, it

can detect whether confidential data is transmitted over unsecured connections or if the program communicates with suspicious servers. One of the most common utilities for this is *Wireshark* (Fig. 14.1).

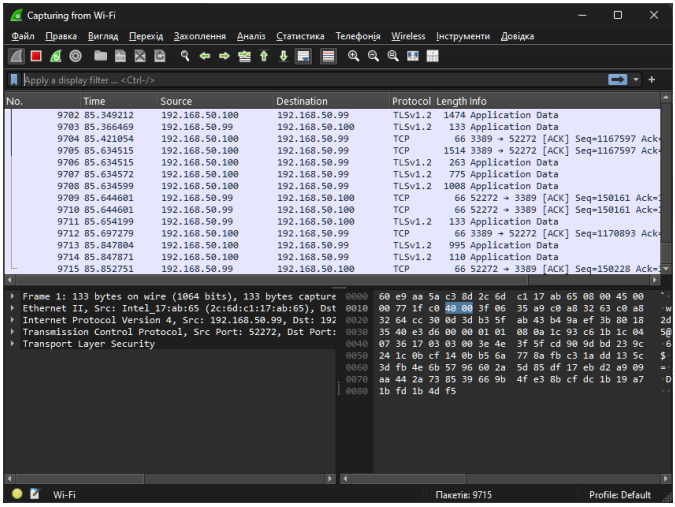

Figure 14.1 – *Wireshark*

Example of Analyzing Traffic Using Wireshark.

```
wireshark -i eth0 -k
```

This example demonstrates how *Wireshark* can be used to monitor network traffic on a network interface, allowing the inspection of packets transmitted or received by the program.

Attempts to Impersonate a Web Server and Detection Methods

Attackers may attempt to impersonate a web server to steal user data or inject malicious code. These attacks can be carried out using *DNS* spoofing or *Man-in-the-Middle (MitM)* attacks. Detecting attempts to impersonate a web server involves analyzing *SSL/TLS* certificates and verifying that the *IP* addresses correspond to the original server.

Example of Checking an SSL Certificate Using `openssl`.

```
openssl s_client -connect example.com:443
```

This command opens an *SSL* connection to the web server and outputs information about its certificate, allowing verification that the certificate matches the expected one.

Methods for Detecting Web Server Impersonation.

- Checking *SSL/TLS* certificates.
- Using network monitoring tools to detect changes in *DNS* records.
- Analyzing *IP* addresses and routes to detect deviations from normal network parameters.

Review Questions

1. What is *reverse engineering*, and what are its main methods?
2. How does *decompilation* work, and what tools are used for this process?
3. What is *software packing*, and how can packed files be analyzed?
4. How can *Wireshark* be used for network traffic analysis and threat detection?
5. What is *DNS* spoofing, and how is it used to impersonate a web server?
6. How does *SSL* certificate checking help detect web server impersonation?
7. What detection methods can be used to protect against *web server impersonation*?
8. What is *disassembly*, and how does it aid in *reverse engineering*?
9. What are the main signs that indicate a possible *web server impersonation*?
10. How can binary file analysis help identify *malware*?

Topic 15

SOFTWARE AND VIDEO GAME PROTECTION METHODS

The objective is to study methods of protecting software and video games from unauthorized access and modification, and to gain skills in analyzing common security threats in gaming software.

Tasks for Independent Study

1. *Implement anti-cheat protection in an online game* (e.g., client integrity checks or detection of known cheat tools).
2. *Ensure data integrity between the client and server* using encryption mechanisms or digital signatures.
3. *Set up a system for detecting modified game files* that alerts when resources or executable files are changed.
4. *Create a licensing mechanism* for desktop software (e.g., key generation and signature validation).
5. *Perform a game stress test for DDoS resistance* by simulating high-load scenarios.

Brief Theoretical Background

1. The importance of software protection.
2. Core methods of software protection.
3. Game protection against piracy and cracking.
4. Legal aspects of software protection.

Methods of Software Protection

Software protection is a set of methods and technologies used to prevent unauthorized use, modification, and reverse engineering of software. These include code signing, obfuscation, watermarking, and anti-tampering mechanisms.

Main protection methods:

• *Code signing (Digital Signing)* – the use of digital signatures to confirm the integrity and authenticity of the software.

• *Code obfuscation (Software Obfuscation)* – a technique that makes the source code difficult to read to complicate reverse engineering.

• *Anti-tampering and Binary Integrity Checking* – technologies that detect changes in the software and can trigger mechanisms to terminate the program.

• *Software protection dongles* – special hardware devices that provide access to the software only when connected to a computer, thus preventing unauthorized copying or use.

• *Cloud licensing servers (Cloud Licensing Location)* – remote servers that manage software licenses and verify their validity during software launch or use.

• *White-box cryptography* – a method of protecting cryptographic algorithms where keys remain secure even when the code and data are accessible, ensuring safe cryptographic operations in potentially malicious environments.

Code Signing

Cryptographic methods play a key role in software protection by providing guarantees of integrity and origin. The use of *hashing* and *digital signatures* allows verifying that the software has not been modified and that its origin is authentic.

Key Protection Mechanisms

• *Hashing* uses algorithms such as *SHA-256* to generate a unique hash, allowing verification of a file's integrity.

• *Digital signatures* enable the verification of the authenticity of the software's source.

Code signing is the process of digitally signing executable files and scripts to confirm their authenticity to the operating system and user. This method uses a cryptographic signature and certificate to verify the authenticity of the software but does not guarantee protection against modifications in the field.

Code signing usually involves a digital signature to confirm the author or build system and a checksum to detect modifications.

The signature is always verified by the operating system at startup, while plugins and add-ons are checked by the host application during loading. Code signing is mandatory in many ecosystems, as it ensures that the software comes from a trusted source. However, it should not be the sole protection measure, as it can be bypassed if the system is compromised.

Example of Code Signing with OpenSSL

```
openssl dgst -sha256 -sign private.pem -out signature.bin program.exe
openssl dgst -sha256 -verify public.pem -signature signature.bin
↪  program.exe
```

This code demonstrates the use of *OpenSSL* for signing and verifying an executable file, ensuring its integrity and authenticity. The first line signs the `program.exe` file using the private key `private.pem`, creating a signature file `signature.bin`. The second line verifies the signature using the corresponding public key `public.pem`.

Protection Against Reverse Engineering

Software protection includes methods that make reverse engineering economically infeasible or difficult. *Code obfuscation*, encryption algorithms, or special instructions can significantly complicate reverse engineering. This also includes methods to protect against dynamic analysis.

Code Obfuscation

Code obfuscation is a protection technique that changes the structure and appearance of the source code, maintaining its functionality but making it hard to analyze and reverse engineer (Fig. 15.1[1]). It is a popular way to protect commercial software from unauthorized access and hacking.

```
var myStr = "document.write('My
             Code')";
eval(myStr);
```

(a) original code

```
var yq = "de')";
var sq = "doc";
var sm = "ument";
var kw = "('My Co";
var myStr = sq + sm + ".write" + kw +
            yq;
eval(myStr);
```

(b) obfuscated code

Figure 15.1 – Code Obfuscation

Main Obfuscation Techniques:

- *Symbol renaming*: names of classes, methods, and variables are changed to random or short values (e.g., `Class123`, `MethodXYZ`, or `varA1`), making them incomprehensible to humans. For example, the variable `customerBalance` may be renamed to `a1`, complicating the understanding of its function.
- *Metadata removal*: the obfuscator can remove metadata, reducing the amount of context available for source code analysis.

[1] JSObfusDetector

• *Structural obfuscation*: the logic of the code is changed to a more complex form. For example, conditional expressions may be transformed into more intricate but equivalent versions.

• *Inlining and abstraction removal*: abstractions such as functions or classes can be "inlined" or merged, making analysis and reverse engineering more difficult.

• *Polymorphism and dynamic control flow*: the logical structure becomes dynamic, making it harder to predict and understand.

• *String encryption*: static strings with important information, such as URLs or messages, may be encrypted and decrypted only during execution.

Popular Obfuscation Tools:

• *ProGuard* – a tool for obfuscating Java and Android applications.
• *Dotfuscator* – a tool for obfuscating .NET applications.
• *JavaScript Obfuscator* – a tool for obfuscating JavaScript code.
• *Stunnix C and C++ Obfuscator* – a tool for obfuscating C and C++ code.

International Obfuscated C Code Contest

The International Obfuscated C Code Contest (IOCCC) is an interesting resource for exploring code obfuscation techniques. The results of the winners demonstrate an endless number of creative ideas and unconventional approaches to programming in *C*, providing insights into how code can be intentionally altered to be unrecognizable and challenging to read and understand.

By analyzing the winners' submissions, researchers, developers, and programming enthusiasts can gain a deeper understanding of obfuscation methods, including complex control flow, unconventional syntax, and non-obvious use of language features.

It is worth noting that such obfuscation techniques are made possible by the unique capabilities and flexibility of the *C* programming language.

Below is a program that "solves the classic problem of placing n queens on a chessboard of up to 99x99, keeping the program as short as possible and using only `for` loops"[2]:

```
v,i,j,k,l,s,a[99];
main()
{
```

[2] https://www.ioccc.org/1990/baruch.c

```
    for(scanf("%d",&s);*a-s;v=a[j*=v]-a[i],k=i<s,j+=(v=j<s&&(!k&&!!pr
    ↪  intf(2+"\n\n%c"-(!l<<!j),"
    ↪  #Q"[l^v?(l^j)&1:2])&&++l||a[i]<s&&v&&v-i+j&&v+i-j))&&!(l%=s),
    ↪  v||(i==j?a[i+=k]=0:++a[i])>=s*k&&++a[ -- i])
        ;
}
```

This is an obfuscated C program that performs a form of backtracking algorithm to generate a numeric arrangement based on input `s`. It uses minimal syntax, dense logic expressions, and inline `printf` to display progress.

```
#include <stdio.h>
int v, i, j, k, l, s, a[99];
int main() {
    scanf("%d", &s);
    while (*a - s) {
        v = a[j *= v] - a[i];
        k = (i < s);
        j += (
            v = (j < s) &&
            (
                !k &&
                !!printf(2 + "\n\n%c" - (!l << !j), " #Q"[(l ^ v) ?
                ↪  ((l ^ j) & 1) : 2]) &&
                ++l || (a[i] < s && v && v - i + j && v + i - j)
            )
        ) && !(l %= s);
        if (!v) {
            if (i == j)
                a[i += k] = 0;
            else if (++a[i] >= s * k)
                ++a[ -- i];
        }
    }
    return 0;
}
```

Another interesting example of obfuscated code from one of the winners of the

IOCCC 2020 contest[3]:

```
  #define/**/Q(x,y)char*/*                                */q=y#x","#y")",*p,s[x;}
  /*IOCCC'20*/#include/*                                  */<stdio.h>/*-Qlock-*/
    int(y),x,i,k,r;Q(9/*              12                 */<<9];float(o)[03];
      void(P)(){*o=r<0/*         11         1             */?r:-r;o[1]=39.5;
       o[2]=22.5;for(k/*      10               2         */=0;++k<39;*o*=i
        /6875.5/(k%2?k/*                                  */:-k))y=o[1+k%2
         ]+=*o;k=o[2];/*     9          o----->  3       */p=s+y+k/2*80;
          }int(main)()/*               /                  */{for(p=s;+i<
          1839;*q>32?k/*       8     L          4         */=i++/80-11,y
           =(750>r*r+k/*          7          5             */*k*4)*4+y/2
           ,*p++=r<41?/*               6                  */y?"0X+0X+!"
           [y-1]-1:+*q/*                                  */++:10:*q++)
           r=i%80-38;;/*                                  */;for(x=13,r
           =20;i=3600*/*     \        /    -----+         */--x,i;*p+ +=
           "OISEA2dC8e"/*     \     /     ---- |         */[x%10],*p+ =x
           /10*41)P();r/*      \   /      ---- |         */=10;;sscanf(
          __TIME__,"%d"/*       \ /       ---- |         */":%d:%d",&k,&
         x,&i);for(i+=(/*        X        ---- |         */k*60+x)*60;18+
        r;*p=k%2?*p%2?+/*      __/ \__         |  |      */59:44:*p>39?59:
       39,i=!r--?i%3600/*     /  \ /  \        |  |      */*12:i)P();puts(s
    ),"#define/**/Q(x"/*     \__/ \__/        +--+      */",y)char*q=y#x\","
  "\"#y\")\",*p,s[x;}"/*                                 */"/*IOCCC'20*/#inclu"
"de<stdio.h>/*-Qlock-"/*                                 */"*/int(y),x,i,k,r;Q(")
```

Methods for Detecting Modifications and Halting Execution

Another task is to protect against unauthorized access and modifications to binary code, aiming to safeguard software from being altered and used in ways not intended by the developers. *Anti-tampering* protection is designed to cause a smooth failure during program execution without providing clues about why the modified code is not working. Although it does not prevent other developers from studying the executable code, this method provides effective protection and significantly complicates *dynamic analysis* of the software.

Technologies for detecting code modifications allow identification of software changes. Programs can periodically check the integrity of their files or use *hashing*

[3]https://www.ioccc.org/2020/endoh3/index.html

mechanisms to verify critical components. If changes are detected, such systems can terminate the program or block access to certain functions.

Protection against unauthorized access can be implemented both inside and outside the protected application.

Software for protecting against unauthorized access is used in many fields: embedded systems, financial applications, mobile device software, network device systems, fraud prevention in games, and more.

It should be noted that the same methods are often used by developers of malicious software for the same purpose — to complicate the *dynamic analysis* of viruses, rootkits, worms, etc.

Example of Detecting Changes in a File

```
String originalHash = "abcd1234...";
String currentHash = calculateHash("program.exe");
if (!originalHash.equals(currentHash)) {
    System.exit(1);  // Terminate the program
}
```

Software Protection Dongles

Hardware security dongles, such as flash drives, are dual-interface security tokens used in systems to protect software from unauthorized use. Without these keys, the software may operate in a limited mode or not work at all.

Modern typical dongles contain non-volatile memory, which allows storing and executing certain parts of the software on the dongle. They also feature built-in robust encryption and use manufacturing technologies that make reverse engineering impossible.

Theoretically, such dongles can be cloned using hardware cloning methods. To prevent this, special smart cards can be used.

Cloud Licensing Location

Considering the shortcomings of traditional protection methods, the transition to cloud technologies in the field of licensing provides a higher level of security than systems with hardware dongles. *Cloud licensing* eliminates the risk of losing or damaging keys or local licensing servers and significantly complicates unauthorized use. However, despite all the advantages, cloud licensing can be compromised on client computers, so client software also requires protection against reverse engineering and modification.

White-box Cryptography

The main idea of white-box cryptography methods is to combine the key and cryptographic algorithm code into a new, transformed code. The key is effectively hidden in the code and cannot be easily extracted. These methods focus on protecting cryptographic algorithms and keys in environments where the executable code is accessible. Their goal is to complicate the analysis of cryptographic operations to preserve the confidentiality of keys and protect algorithms, even if an attacker has access to the code.

Currently, commercial implementations of white-box cryptography methods are available for major symmetric block ciphers (*AES* and *DES*). Additionally, they may include implementations of *hashing*, *RSA*, and elliptic curve cryptography methods.

These methods can be considered advanced techniques of code obfuscation.

Protecting Computer Game Software

There are two main issues when it comes to protecting computer games – the protection of intellectual property rights (legal matters) and the protection of the game ecosystem (technical and organizational matters).

The protection of intellectual property covers such components of games as:

- the multimedia subsystem of the game – audiovisual effects, musical accompaniment, video sequences, audio sequences, animation, graphical elements (logos, settings, characters, objects, interfaces, fonts, etc.);
- the storyline;
- characters;
- databases, including their structures;
- user manuals;
- …and many other components.

To protect the game ecosystem, the above methods are used, but today, with the rise of online games and mobile game applications, there are additional specific challenges related to the gameplay.

Historically, complex hardware protection methods were used to protect computer games, which limited the cloning of game media by manipulating the medium, as well as hardware protection keys that prevented unauthorized game launches.

Network games are vulnerable to network attacks, so it is important to apply secure programming principles:

- *Encryption of network traffic* – using encryption protocols such as *TLS* or *DTLS* to protect game transactions from interception or modification;
- *Player authentication* – verifying the authenticity of each player to prevent unauthorized access to game servers and the use of fake accounts;

• *Protection of personal data* – ensuring compliance with local laws, such as *GDPR*, to maintain the confidentiality and security of players' personal information;

• *Detection of anomalous activity* – using logs, monitoring systems, and behavior analysis tools to detect suspicious activities, such as fraud or *DDoS* attacks.

Legal Aspects of Software Protection

• *Copyright* protects software as a work, granting exclusive rights to reproduce, distribute, and modify it.

• *Software patents* allow protecting new and useful technical solutions if they meet patent law requirements.

• *License agreements (EULA and SLA)* regulate the use of software, establishing the rights and responsibilities of the parties.

• *Data protection laws* require ensuring the confidentiality of user data in accordance with local regulations (e.g., *GDPR*).

• *Trade secret protection* involves keeping unique algorithms or technologies confidential as commercial information.

• *Anti-piracy laws and measures* provide for penalties for illegal copying and distribution of software. In many countries, there is liability for using torrents as a tool for piracy.

• *Digital Rights Management (DRM)* restricts access and controls the use of software and multimedia.

Review Questions

1. What is *code signing*, and how does it protect software?
2. How does *obfuscation* work, and how does it complicate reverse engineering?
3. What are *watermarks* in software, and how do they help protect programs?
4. What cryptographic guarantees ensure the *integrity* and *origin* of the software?
5. What methods of detecting modifications are used to protect software?
6. How can network computer games be protected from *network attacks*?
7. What methods are used to protect computer games from *piracy*?
8. What is *DRM*, and how does it help protect software from unauthorized use?
9. How do *anti-cheat systems* protect online games from third-party software?
10. What *secure programming principles* should be applied to multiplayer games?

Topic 16

SOFTWARE PROTECTION BASED ON HMAC AND DIGITAL SIGNATURES

The objective is to become familiar with *HMAC* and *digital signature* mechanisms for ensuring data integrity and authenticity, and to gain practical skills in working with digital signatures in programming languages.

Tasks for Independent Study

1. *Implement* HMAC *(Hash-Based Message Authentication Code)* in a programming language of your choice (e.g., *Python*). Explain the principle behind *HMAC*, focusing on key usage and the signature generation process.
2. *Generate a digital signature for a file using* OpenSSL:
   - Create a key pair (private and public keys);
   - Sign a selected file using the private key;
   - Verify the signature using the public key;
   - Document all commands and options used.
3. *Implement a simple digital signature mechanism in Python* (using the `cryptography` or *PyOpenSSL* library). Demonstrate the process of signing and verifying (e.g., for a text file).
4. *Compare digital signature algorithms – RSA, ECDSA, EdDSA* – focusing on:
   - Performance (signing and verification speed);
   - Key and signature sizes;
   - Security level (resistance to cryptanalysis).

Summarize the suitability of each algorithm for different scenarios.

5. *Explain the difference between HMAC and digital signatures*, focusing on:
   - Method of signature creation (symmetric vs. asymmetric);
   - Security advantages and limitations;
   - Common application areas (e.g., securing web requests, document signing).

Provide examples of when to choose *HMAC* or a digital signature.

Brief Theoretical Background

1. Overview of *HMAC* (hash code with secret key).
2. Using *HMAC* to ensure integrity and authenticity.
3. Overview of *digital signatures*.

4. Using digital signatures to protect software.
5. Comparison of *HMAC* and digital signatures for different application types.

HMAC: Ensuring Data Integrity and Authenticity

HMAC (*Hashed Message Authentication Code*) is a message authentication method that uses a hash function along with a secret key to ensure data integrity and authenticity. *HMAC* guarantees that the message has not been altered during transmission and that the sender is authenticated (Fig. 16.1[1]).

Key properties of *HMAC*:

• *Data integrity – HMAC* ensures the detection of any changes to the data during transmission.

• *Authenticity* – only parties that share the secret key can create or verify the *HMAC*.

HMAC works by combining the message with a secret key and applying a hash function such as *SHA-256* or *MD5* to generate a unique authentication code for the message.

HMAC is widely used for securing API requests, message authentication in protocols (e.g., *TLS*), and protecting sensitive data from tampering.

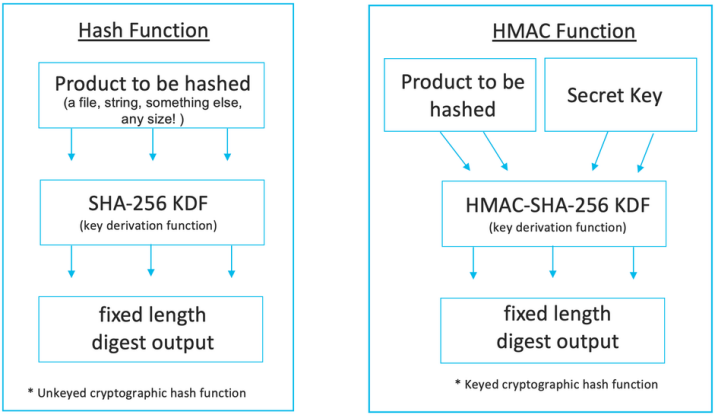

Figure 16.1 – *HMAC*

[1]Configuring & Understanding OSPF HMAC Authentication (community.cisco.com)

Example of HMAC with SHA-256

```
#include <openssl/hmac.h>

unsigned char *result;
unsigned int len = 32;
result = HMAC(EVP_sha256(), key, strlen(key), data, strlen(data),
↪  NULL, NULL);
```

This code demonstrates the use of *HMAC* with the *SHA-256* algorithm to compute a hash of the message using a secret key.

HMAC is widely used for securing *API* requests, message authentication in protocols (e.g., *TLS*), and protecting sensitive data from tampering.

Digital Signatures: Authenticity and Non-Repudiation of Messages

A Digital Signature is a cryptographic method that ensures the authenticity, integrity, and non-repudiation of electronic documents or messages (Fig. 16.2[2]). This method uses a pair of keys: a private key to sign the document and a public key to verify the signature.

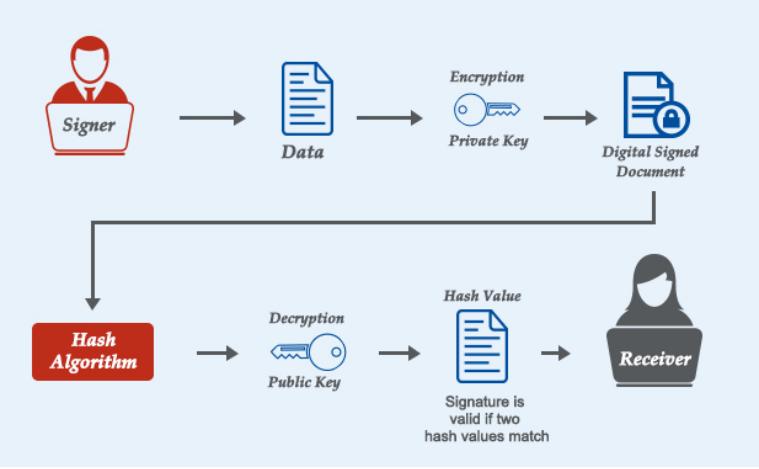

Figure 16.2 – Digital Signature

[2]Comodo: What is Digital Signature? How does it Work? (comodosslstore.com)

Key properties of Digital Signatures:

• *Authenticity* – the signature verifies that the document was signed by the owner of the private key.

• *Integrity* – the signature also ensures that the document has not been altered since it was signed.

• *Non-repudiation* – the sender cannot deny having signed the document, as the signature is unique and tied to the sender's private key.

The process of creating a *digital signature* involves computing a hash of the message and encrypting this hash with the sender's private key. The recipient can verify the signature by decrypting it with the sender's public key and comparing the obtained hash with the hash of the message itself.

Example of Creating a Digital Signature with OpenSSL

```
EVP_MD_CTX *mdctx = EVP_MD_CTX_new();
EVP_PKEY *private_key = load_private_key("private_key.pem");

EVP_SignInit(mdctx, EVP_sha256());
EVP_SignUpdate(mdctx, message, strlen(message));
EVP_SignFinal(mdctx, signature, &sig_len, private_key);
EVP_MD_CTX_free(mdctx);
```

Digital signatures are widely used in areas such as *e-commerce*, *legal documents*, and *software protection* to prevent unauthorized modifications. For example, software can be signed by a developer, and users can verify the signature before installation to ensure its *authenticity* and *integrity*.

This code demonstrates how to create a *digital signature* for a message using *SHA-256* and a private key.

Digital signatures are widely used in areas such as e-commerce, legal documents, and software protection to prevent unauthorized modifications. For example, software can be signed by a developer, and users can verify the signature before installation to ensure its authenticity and integrity.

The Role of Digital Signatures and HMAC in Security

Both *HMAC* and *digital signatures* are tools for ensuring software and data security. *HMAC* is effective in scenarios where both parties share a secret key and need to ensure the *authenticity* and *integrity* of messages without requiring *asymmetric encryption*. On the other hand, *digital signatures* provide guarantees of *non-repudiation* and *authenticity* through the use of *asymmetric keys*.

HMAC and digital signatures are used in different scenarios:

• *HMAC* – suitable for message *authentication* in secure communication channels (e.g., VPN or TLS, see Fig. 16.3[3]).

• *Digital Signatures* – suitable for document and *software authentication* and *long-term authenticity* assurance.

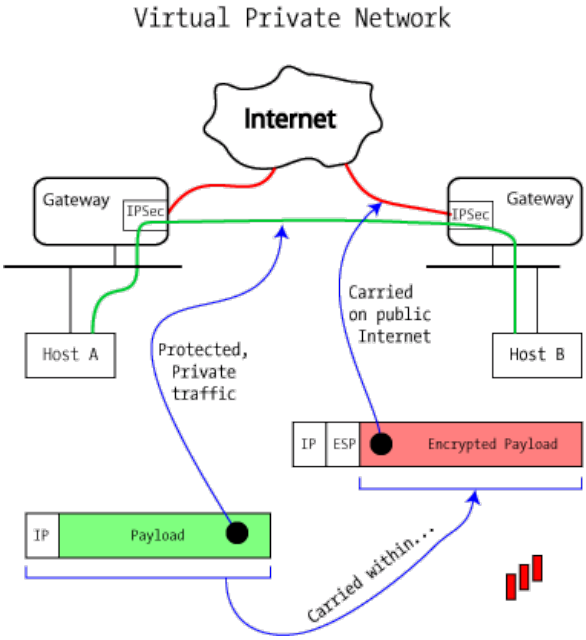

Figure 16.3 – *IPSec*

Review Questions

1. What is *HMAC*, and how does it ensure *data integrity* and *authenticity*?
2. How is *HMAC* used in secure communications such as *TLS* or *VPN*?
3. What is a *digital signature*, and how does it ensure *authenticity* and *non-repudiation* of messages?
4. What is the difference between *HMAC* and *digital signatures*?
5. How does *asymmetric encryption* work in the context of a *digital signature*?
6. How is the integrity of a signed message or document verified?
7. What hashing algorithms can be used in *HMAC* and *digital signatures*?

[3] An Illustrated Guide to IPsec (www.unixwiz.net)

8. In what scenarios is *HMAC* more appropriate than a digital signature?

9. How does *OpenSSL* assist in creating *HMACs* and *digital signatures*?

10. What are the advantages and disadvantages of using *digital signatures* compared to other authentication methods?

11. How does key management differ between *HMAC* and *digital signature* systems?

12. What role do certificate authorities (*CAs*) play in the validation of *digital signatures*?

13. What are common vulnerabilities or implementation mistakes with *HMAC* or *digital signatures*?

14. How can timestamps or counters be used to prevent replay attacks with *digital signatures* or *HMAC*?

15. What libraries or *APIs* are commonly used in programming languages like *Python* or *Java* to implement *HMAC* and *digital signatures*?

LIST OF REFERENCES

1. *Savchenko V. M., Mnushka O. V.* Modern Secure Programming Technologies : Textbook. – Kharkiv : NTU ”KhPI”, 2024. – 180 p.
2. *Savchenko V., Mnushka O.* Modern Secure Programming Technologies. Workshops : Educational and methodological guide. – Kharkiv : FOP Brovin O. V., 2024. – 136 p. – ISBN 978-617-8238-76-6. – URL: https://repository.kpi.kharkov.ua/handle/KhPI-Press/85833 (visited on 11/26/2024).
3. *Остапов С., Євсеєв С., Король О.* Кібербезпека: сучасні технології захисту : Навчальний посібник. – Харків : Новий світ, 2020. – 678 с. – URL: http://library.kpi.kharkov.ua/en/inftechnologies/кібербезпека-сучасні-технології-захисту (дата зверн. 26.11.2024).
4. *Лісовська Ю.* Кібербезпека: ризики та заходи : Навчальний посібник. – Київ : Видавничий дім «КОНДОР», 2019. – 272 с. – URL: http://library.kpi.kharkov.ua/files/new_postupleniya/kibrtz.pdf (дата зверн. 26.11.2024).
5. *Тарнавський Ю. А.* Технології захисту інформації : Підручник. – Київ : КПІ ім. Ігоря Сікорського, 2018. – 162 с. – URL: https://ela.kpi.ua/handle/123456789/23896 (дата зверн. 26.11.2024).
6. *Helfrich J. N.* Security for Software Engineers. – Boca Raton : CRC Press, Taylor & Francis Group, 2020. – URL: http://repo.darmajaya.ac.id/4635/1/SecurityforSoftwareEngineers(PDFDrive).pdf (visited on 11/26/2024).
7. *Winkler I., Gomes A. T.* Advanced Persistent Security. – Amsterdam : Syngress / Elsevier, 2017. – URL: https://www.sciencedirect.com/book/9780128093160/advanced-persistent-security.
8. *Paulsen C., Byers R.* Glossary of Key Information Security Terms. – National Institute of Standards, Technology, 2019. – URL: https://nvlpubs.nist.gov/nistpubs/ir/2013/nist.ir.7298r2.pdf (visited on 11/25/2024).
9. *Committee on National Security Systems*. CNSS Glossary. – 2022. – URL: https://rmf.org/wp-content/uploads/2017/10/CNSSI-4009.pdf (visited on 11/25/2024).
10. *Federal Bureau of Investigation*. Internet Crime Report 2023. – 2023. – URL: https://www.ic3.gov/Media/PDF/AnnualReport/2023_IC3Report.pdf (visited on 04/30/2024).

11. *Samani R.* An Analysis of the WannaCry Ransomware Outbreak. – 2017. – URL: https://www.mcafee.com/blogs/other-blogs/executive-perspectives/analysis-wannacry-ransomware-outbreak/ (visited on 04/30/2024).

12. *Microsoft Security Response Center*. Analyzing Solorigate, the compromised DLL used in the SolarWinds attack. – 2020. – URL: https://www.microsoft.com/en-us/security/blog/2020/12/18/analyzing-solorigate-the-compromised-dll-file-that-started-a-sophisticated-cyberattack-and-how-microsoft-defender-helps-protect/ (visited on 04/30/2024).

13. *Microsoft Security*. Guidance for responders: Investigating and remediating on-premises Exchange Server vulnerabilities. – 2021. – URL: https://msrc.microsoft.com/blog/2021/03/guidance-for-responders-investigating-and-remediating-on-premises-exchange-server-vulnerabilities/ (visited on 04/30/2024).

14. *Miller M.* Deepfakes: A Real Threat to Cybersecurity. – 2023. – URL: https://kpmg.com/kpmg-us/content/dam/kpmg/pdf/2023/deepfakes-real-threat.pdf (visited on 04/30/2024).

15. *European Union Agency for Law Enforcement Cooperation*. Facing reality?: law enforcement and the challenge of deepfakes : an observatory report from the Europol innovation lab. – Publications Office, 2024. – DOI: 10.2813/158794.

16. Proceedings of the 32nd USENIX Security Symposium : USENIX Security Symposium. – [Berkeley, CA] : USENIX Association, 2023. – 490 p. – ISBN 978-1-939133-37-3.

17. Language Models are Few-Shot Learners / T. B. Brown [et al.]. – 2020. – DOI: 10.48550/ARXIV.2005.14165.

18. *OWASP Foundation*. OWASP Top 10 - 2021. – 2021. – URL: https://owasp.org/www-project-top-ten/ (visited on 10/30/2024).

19. *Software Engineering Institute*. SEI CERT C Coding Standard. – Version 01. – Carnegie Mellon University, 2016. – URL: https://resources.sei.cmu.edu/downloads/secure-coding/assets/sei-cert-c-coding-standard-2016-v01.pdf (visited on 05/15/2024).

20. *Ballman A.* SEI CERT C++ Coding Standard. – Version 01. – Carnegie Mellon University, 2016. – URL: https://resources.sei.cmu.edu/downloads/secure-coding/assets/sei-cert-cpp-coding-standard-2016-v01.pdf (visited on 05/15/2024).

21. The CERT Oracle Secure Coding Standard for *Java* / ed. by F. W. Long. – Upper Saddle River, NJ : Addison-Wesley, 2012. – 699 p. – (The SEI series in software engineering). – ISBN 0-13-288284-1.

22. *Software Engineering Institute*. CERT Secure Coding Standards. – 2021. – URL: https://www.securecoding.cert.org (visited on 04/30/2024).

23. *European Union*. General Data Protection Regulation (GDPR). – 2016. – URL: https://gdpr.eu/ (visited on 10/30/2024).

24. *PCI Security Standards Council*. PCI Security Standards Overview. – 2021. – URL: https://www.pcisecuritystandards.org/standards/ (visited on 10/30/2024).

25. *OpenSSF*. Secure Coding Guide for *Python*. – 2024. – URL: https://github.com/ossf/wg-best-practices-os-developers/tree/main/docs/Secure-Coding-Guide-for-Python (visited on 05/12/2024).

26. *Checkmarx*. JavaScript Secure Coding Practices (JS-SCP). – 2024. – URL: https://github.com/Checkmarx/JS-SCP (visited on 05/12/2024).

27. *Amazon Web Services*. What is Virtualization? – 2024. – URL: https://aws.amazon.com/what-is/virtualization/ (visited on 05/15/2024).

28. NIST cloud computing reference architecture / F. Liu [et al.]. – 2011. – DOI: 10.6028/nist.sp.500-292.

29. *The MITRE Corporation*. Common Weakness Enumeration (CWE). – 2024. – URL: https://cwe.mitre.org/index.html (visited on 05/15/2024).

30. *The MITRE Corporation*. Common Vulnerabilities and Exposures (CVE). – 2024. – URL: https://www.cve.org/ (visited on 05/15/2024).

31. *Google*. Prepare for the phaseout of third-party cookies. – 2024. – URL: https://support.google.com/google-ads/answer/1033961 (visited on 05/15/2024).

32. *Wagner K.* Google Third Party Cookies Phase Out in 2024: Timeline and Tips. – 03/2024. – URL: https://kattwagner.com/google-third-party-cookies-phase-out-in-2024 (visited on 05/15/2024).

33. *Google*. A more private web: Privacy Sandbox. – 2023. – URL: https://privacysandbox.com/ (visited on 05/15/2024).

34. *Software Engineering Institute*. Android Secure Coding Standard. – 2024. – URL: https://wiki.sei.cmu.edu/confluence/display/android/Android+Secure+Coding+Standard (visited on 05/25/2024).

35. *JSSEC*. Android Secure Coding. – 2018. – URL: https://www.jssec.org/dl/android_securecoding_en.pdf (visited on 05/15/2024).

36. *JSSEC*. Security Guidelines 2012. – 2012. – URL: https://www.jssec.org/dl/guidelines2012Enew_v1.0.pdf (visited on 05/15/2024).

Навчальне видання

САВЧЕНКО Володимир Миколайович
МНУШКА Оксана Василівна

СУЧАСНІ ТЕХНОЛОГІЇ БЕЗПЕЧНОГО ПРОГРАМУВАННЯ

Навчально-методичний посібник
для самостійної роботи студентів другого (магістерського) рівня
денної, заочної та дуальної форм навчання
за спеціальністю 123 “Комп’ютерна інженерія”

Англійською мовою

Відповідальний за випуск проф. Заковоротний О. Ю.
Роботу до видання рекомендував проф. Заполовський М. Й.

В авторській редакції

План 2025 р., поз. 18
Гарнітура Times New Roman. Ум. друк. арк. 4.64

Видавничий центр НТУ “ХПІ”.
Свідоцтво про державну реєстрацію ДК № 5478 від 21.08.2017 р.
61002, м. Харків, вул. Кирпичова, 2

Електронне видання


Department of
Computer
Engineering and
Programming
NTU «KhPI»