Software Security

- Learn about different software vulnerabilities and techniques that can be used to identify, prevent, or reduce adverse effects.

❖ Software can be considered safe when it meets a defined or expressed security requirement. This includes confidentiality, integrity, and availability requirements for system operation and data.

- For Example: The following requirements may be imposed on for a Social Media software
    - The user can only edit their own data. (Integrity)
    - Only the user's friends can see their data. (Confidentiality)
    - The software must be available in 99.9% of the time. (Availability)

❖ The claims can also be mutually invalid:

- Example: an integrity requirement: shut down a system in the event of an attack, but the system would be unavailable.

❖ Security failure: It is a situation where a system does not achieve an information security goal.

❖ Vulnerability can be defined as causing a security incident

- It is assumed that a security objective is defined.
- In practice, many systems do not have defined information security objectives or are ambiguous.

- Vulnerabilities exist even when they are not tied to security objectives.
    - In this case, vulnerabilities are thought to be implementation vulnerability.

❖ Well-defined security objectives also cover implementation-specific vulnerabilities,

- but this is not always the case and
- it is still thought that implementation-specific vulnerabilities are real vulnerabilities.

EU General Data Protection Regulation requirements for software

- Data protection requirements come directly from legislation.
- EU GDPR regulates the requirements for software processing personal data within the EU.
- So, it is important to understand that software development must take into account the EU's GDPR.

- When developing new programs that process personal data and repairing old ones, it is very important that programmers and their supervisors know the requirements of the regulation for data protection.

❖ The organisation's top management should also know that

- meeting the EU GDPR requires resources and
- organisation is financially responsible for any non-compliance with the Regulation.

If the software is available in the EU,

❖ the EU GDPR imposes a number of requirements on the software,:

- Requesting consent as required by law
- Preparing for the reporting of personal data breaches,
  - including collecting and managing log data and notifying those whose personal data has been breached.
- Preparing for information requests from users.
  - For information requests, an automated report of the user's personal data can be generated.

EU GDPR requirements on the software

- Preparing for accountability
- Minimisation of personal data processed
- Deletion of personal data when there is no longer a need and legal basis for storing the data.
- Preparing or avoiding the processing of special categories of data
- Carrying out an impact assessment if there is a requirement under the Regulation (processing of special categories of data or high risk)

Software Security: Classification of vulnerabilities

Vulnerabilities can be **classified** in different ways,

❖ such as frequency, severity of consequences, classification of consequences, and severity of exploitability.

❖ A pre-agreed classification provides a generally understood view of vulnerabilities.

❖ CVE (Common Vulnerabilities and Exposures) is one common way to classify vulnerabilities,

- Another way is Common Weakness Enumeration (CWE).
- At country level, vulnerabilities are communicated and coordinated by the National Cyber Security Centers

The following is a classification of vulnerabilities in accordance with the CYBOK dataset.

- Memory Management Vulnerabilities (Deepening)
- Input generation vulnerabilities (deepening)
- API vulnerabilities (deepening)
- Vulnerabilities in competitive situations (in-depth)
- Side channel vulnerabilities (deepening)
- Comprehensive security requires complex specifications (deepening)
- Vulnerabilities are a subset of failures (deepening)

Memory Management Vulnerabilities (Deepening)

- **Programming** requires **memory allocations** and some vulnerabilities are related to them. In particular, languages without automatic memory allocation are vulnerable, like C and C++.
- If a programmer allocates memory outside allocated memory area, an undefined operation occurs, which can lead to vulnerability.
- These types of vulnerabilities are **buffer overflow vulnerability**.
- **Example:** if a table is allocated space for 10 items, but the program writes to 11 (or more) items, this can lead to vulnerability
- The vulnerability is based on modifying memory addresses at the end of a table so that an attacker's own code is executed. For example, the execution of the program moves inside the table, The situation is dangerous if an attacker is able to fill in the overflowing memory, e.g. From a web form, URL, or other feed. The compiler or system may have been protected, but these should not be trusted, and the program should be implemented safely.
- Vulnerabilities related to **memory reservation**.
- Memory reservations are made when you want to use memory dynamically during program execution. i.e., the amount of memory to be used is not known in advance, so memory is allocated during program execution. An example of a vulnerability: program removes the memory area, yet the program still refers to the same freed memory area that an attacker can now modify.

- A problem with memory management can also arise when a program allocates memory, but it is also accessible by other resources, such as other programs. **Example:** not clearing allocated memory, using a temporary memory area, browser cookies, environment variables or cache. Another resource can change or read memory. Memory reservation and deletion can be corrected in itself, but if stored data is not overwritten, confidential information such as passwords, payment information or personal information may be revealed.

  Memory management vulnerabilities can be **prevented**,

- By code reviews, testing, and proper program design, e.g. Using tables only through interfaces or secure program calls. However, this can be at the expense of performance.
- By choosing a programming language to prevent vulnerabilities. E.g. Java, Python, Perl, Rust, and Ruby include automatic memory management, so buffer overflows and memory allocations don't cause the same problems.
- Translator selection and settings can also prevent vulnerabilities, as compilers also have protections in place that detect and block vulnerabilities.
- However, this is not complete protection, as it is not possible to identify all errors.

Input generation vulnerabilities (deepening)

- Programs often need to dynamically construct new structures, such as queries, form input.
- Examples include SQL queries and html handling.
- For example, when a user logs on to the program, the username and password hash [reference] may be retrieved from the database by SQL queries.
- Similarly, when searching for information using an online form, the data can be retrieved with SQL queries from the database.
- The problem arises if the user is able to enter SQL commands that the program interprets as part of the SQL query.
- SQL commands can be complex and, at worst, such a vulnerability can lead to system takeover, but even at the mildest, obtaining information from the database or denial of service is possible.
- The programmer's job is to ensure secure handling of SQL data.
- It is often possible to prevent user input from being interpreted as SQL queries.
- On the other hand, if there is no blocking, a vulnerability has arisen.

- Similarly, for other vulnerabilities that arise in input generation, the programmer must be familiar with the protection mechanisms. They are often simple, but you need to know how to build them into the program code.
- i.e,, the programmer must be aware and also monitor the development of vulnerabilities.
- One input **vulnerability is incomplete mediation**.
- A program passes some piece of information, a parameter, to another program, which fails to carry out the checks it is capable of.
- There is a risk not only of errors made by the program transmitting information, but also that the program is in the possession of an attacker.

Example: an order sent by a browser to a web store.

- Even if the subscriber is authenticated, the recipient, i.e. the server, should not believe,
- for example, in the customer. The total amount stated with the order, but it must be calculated and an incorrect order must be prevented.

API vulnerabilities (deepening)

- An API refers to how software communicates with each other.
- Example: a program calls the interface of another program, in which case the called program performs pre-programmed actions based on the specifications provided by the calling program.
- For example, **on a phone**, a program may want the location information of the device, by calling the phone's software interface, this information can be obtained, if the software and system rights are sufficient to transmit the information.
- Otherwise, the response from the interface will tell you that location data cannot be provided and perhaps the reason for this.
- Almost all programs use one or more APIs.
- The software designer may make assumptions about the operation of the interface and thus cannot prepare for the exceptional operation of the interface, which may lead to vulnerability.
- There may also be a vulnerability in the implementation of the API.
- It is important for the programmer to find out that the interfaces work safely and to be prepared for the possibility of updating the interface to be called in case vulnerabilities are found in the interface used.
- The programmer must also take into account the correct use of the API.

- This is especially important when implementing cryptography [reference] so that cryptography is executed in the manner required by the standard defined by the cryptographic function and securely.
- In practice, the programmer must be aware of secure algorithms, their implementation and initialization, the different modes and their security, as well as the correct order of calls.
- Of course, ready-made solutions must be used, since they have already been implemented safely, provided that everything has been done correctly.
- Some interfaces know how to take care of things directly correctly, but in others a lot is left to the programmer's expertise. However, some interfaces may contain errors or use weak encryption, so when using cryptography, you need to ensure the security of the interface and implementation.

Vulnerabilities in competitive situations (in-depth)

- Vulnerability in a competitive situation can be avoided with good planning and implementation of the program.
- Competitive situations must be anticipated and implemented in accordance with good programming practices.
- For example, some programming commands prevent a racing situation from occurring.
- In parallelism, rights management becomes important so that the racing situation does not cause problems. For example, after charging the memory area, no other process should write to the memory area.

Side channel vulnerabilities (deepening)

- Side-channel vulnerabilities take advantage of the extra information generated by the execution of the actual software.
- This information is typically noise that can be analyzed. For example, it is possible to analyze the operation of the processor using hardware or software.
- The data processed by the processor can be analyzed, for example, by using a data processor. monitoring execution time or power consumption. If the contents of the key affect the monitored execution time or power consumption, it is possible to find out the key or part of it.
- Side-channel attacks are protected by the correct design and implementation of the software, making analysis difficult. For example, temporal analysis of execution can be prevented by executing software commands so that there is no temporal difference in program execution depending on the encryption key.
- In addition to reading information, one form of side-channel attacks is influencing program execution, which can be called fault injection attacks. The fault can be caused by

e.g. with temperature changes, clock changes or electromagnetic radiation. One example is the so-called. Row hammer attack, in which numerous repeated memory references are used to change the positions of adjacent bits in DRAM.

Comprehensive security requires complex specifications (deepening)

- Implementation-specific vulnerabilities have been discussed above. However, the system should be viewed as a whole. For example, an implementation-specific vulnerability does not pose a problem as a whole if it has been prepared for elsewhere in the system. In this case, layers of security have been built to prevent the problem caused by a single vulnerability.
- It may also be the case that even though there are no implementation-specific vulnerabilities in the system, the system's security objective is still not met. It is therefore important to consider safety as a whole. The system can be divided into subsystems, the safety of which is also considered as part of the whole. For example: How to prepare for a problem in one subsystem as a whole and how do the subsystems support the overall safety objective?
- The security objective for implementation-specific vulnerabilities is easy to define: either it is met or it is not met, e.g. either there is a vulnerability or there is none. However, there are more complex security configurations. One of these is information flow security. In this case, information security is examined from the perspective of data transfer, in which case publicity and confidentiality requirements are defined for each dataflow. An example of the requirements in practice is that confidential input does not affect public output in any way. For example, by searching for sales prices in a service, you cannot influence what others see as public sales prices of services in their own searches.
- Dataflow configurations should also evaluate input repetition. For example, an attacker could repeat searches and perform them in different formats. In this way, it may be possible to carry out a repeat attack, or it may be possible to combine information in such a way as to breach the confidentiality requirement.

Vulnerabilities are a subset of failures (deepening)

- The concept of vulnerability is useful but does not cover all software security. Software security can be promoted through good programming principles, testing and, in particular, certification to meet high quality requirements [reference]. Vulnerabilities can be seen as a subset of faults. A defect, in turn, is a mistake in design or implementation. The taxonomy, or scientific classification of defects, has been developed in research in the area of dependable computing.

Software Security: Detection of vulnerabilities

- The earlier a vulnerability is detected and remedied, the more cost-effective it is, but in any case, vulnerabilities need to be identified at different stages of software development: design, implementation and maintenance.
- A detection technique is said to be **sound** if it can perfectly infer that the specified vulnerability (or a set of vulnerabilities) does not exist in the object studied. Insecure detection, on the other hand, does not find all defined vulnerabilities.
- A detection technique is said to be **complete** if the vulnerability found by the technology is indeed a vulnerability. In other words, there will be no false positives about vulnerabilities that are not actually vulnerabilities.

Static analysis (in-depth)

- In static analysis, program code or binary code is studied without executing it. In this case, the advantage is that the program does not have to be ready and, on the other hand, it is possible to check the entire program code. Static analysis can also be considered to include **human code reviews**. Code review has its place, because a person can perceive the operation of the program as a whole better than the program. On the other hand, ready-made programs, i.e. static analyzers, are more effective than humans at finding common known problems. Static analysis can be divided into heuristic static analysis and sound static analysis.
- Heuristic identification is based on rules aimed at identifying errors in the code. For example, certain insecure coding methods, over-indexing of tables, or insecure interface calls may be detected. The analyzer can also build a model of the program and its data flows, thus aiming to identify larger entities than individual errors.
- Whereas heuristic identification aims to indicate a known error or variation thereof, certainty analysis aims to show that there is no error. Assurance analysis aims to demonstrate the security of some aspects of the code using formal methods, such as correct memory processing. The idea behind safety analysis is that if there is a certain known error in the program, it will also be found. In practice, however, compromises have to be made on these lofty goals, so it is not possible to find all the flaws. Take a closer look at [reference].
- Thus, static analysis does not find all errors. They can also give a lot of false positives, which can make using the program tedious and error-prone. Much depends on the program and its use. Static analyzers are naturally tied to a specific programming language. In any case, static analyzers are one very important tool in secure software development. For example, a company may require a program to pass certain selected static analyzers without serious errors.

Dynamic analysis (advanced)

- In dynamic sensing, observation is made during program execution. With dynamic detection, it is possible to prevent a problem caused by a vulnerability even during program execution. However, when this is done, the performance of the program decreases.

Monitoring(advanced)

- Monitoring monitors the performance of the program while driving. In principle, monitoring makes it possible to prevent vulnerabilities completely. However, this would happen with such a severe drop in performance and memory that in practice compromises would have to be made.
- Current C/C++ compilers include the ability to increase monitoring during compilation, allowing the program to automatically detect and block runtime memory management errors.
- The problem with input monitoring is that the input and output are often undefined for the monitoring program from a security perspective.
- Monitoring racing situations is difficult, but in some cases it is possible to monitor memory allocation and check that the rights do not cause a conflict.
- One important way to find vulnerabilities is to find program execution paths and correct input. This is helped by Fuzz testing, which can automatically generate a very large amount of test feed. The input received by the program comes with a large number of corrupt and unpredictable input. The testing program also documents any faulty behavior of the program, allowing errors to be traced.
- Fuzz testing can be divided into black and white box tests:
- No source code is used in black box testing, so testing is not based on knowledge of the structure of the program. Black box testing can be divided into three ways:
- Truly random
- Random, but some range of values has been taken. The concept of parameterization is also used.
- Mutation-based, when some initial input is given, which the testing program begins to modify, i.e. form a mutation.
- In white box testing, source code helps the testing program identify input ranges and program branching. This allows the program to find more suitable testing data than black box testing, which better covers potential error inputs.