Security Engineering

Cryptography and Information Security
Master in Informatics

José Bacelar Almeida

Secure Software Development

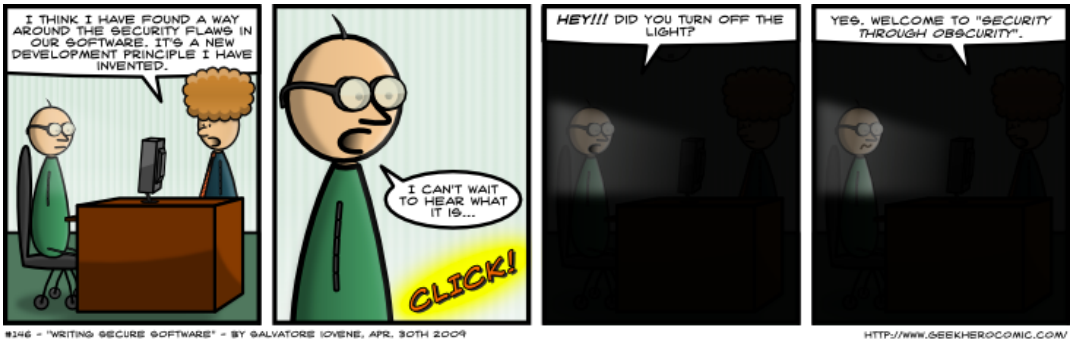
Software Security

Secure Software Development Lifecycle (S-SDLC)

Common Code Vulnerabilities

based on the slides of the 23/24 edition of the course (author: jose.miranda@devisefutures.com)

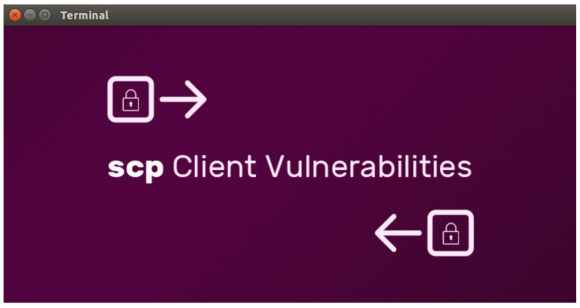
Software Security



Software Security

36-Year-Old SCP Clients' Implementation Flaws Discovered

January 15, 2019 Mohit Kumar



A set of 36-year-old vulnerabilities has been uncovered in the Secure Copy Protocol (SCP) implementation of many client applications that can be exploited by malicious servers to overwrite arbitrary files in the SCP client target directory unauthorizedly.

Demonstrates the need for adequate software development processes that avoid the creation of project vulnerabilities like these.

How to protect your PC against the major 'Meltdown' CPU security flaw

Everything we know so far

By Tom Warren | @tomwarren | Jan 4, 2018, 8:12am EST

Details have emerged on two major processor security flaws this week, and the industry is scrambling to issue fixes and secure machines for customers. Dubbed "Meltdown" and "Spectre," the flaws affect nearly every device made in the past 20 years. The Meltdown flaw primarily affects Intel and ARM processors, and researchers have already released proof-of-concept code that could lead to attacks using Meltdown.

The identified vulnerabilities have an interesting common characteristic:

Existed for more than twenty years

One was in open source (OpenSSH scp), and another in the "closed microcode" of the processor.

Software Security

It is far harder to write solid code than to destroy it (OWASP DevGuide)

- Context:
  - Attacks on financial, medical, government systems, and critical infrastructures do not stop increasing in number and severity;
  - With increasingly complex and interconnected digital infrastructures, the difficulty of providing secure systems increases exponentially;
  - Great ease in the dissemination and exploitation of errors.

Security in Software Development

- Security is one of several goals in software development (sometimes conflicting).
- Some examples:
  - **Functionality** - software products are often evaluated by the number of features provided, which leads to a high degree of complexity, in apparent contradiction with security;
  - **Usability** - ease of a user to take advantage of the software product, and security is often considered as an obstacle to usability, to the point that it is necessary to establish a "compromise" between the two;
  - **Software performance** - cryptographic mechanisms or security checks spend CPU time, with a negative effect on performance;
  - **Simplicity** - reduces production and maintenance costs, not compatible with the need for security mechanisms;
  - **Time-to-market** - quick placement of the product on the market, with security (tests and evaluations) being an obstacle.

Secure Software Development

- A necessary condition but not necessarily sufficient:

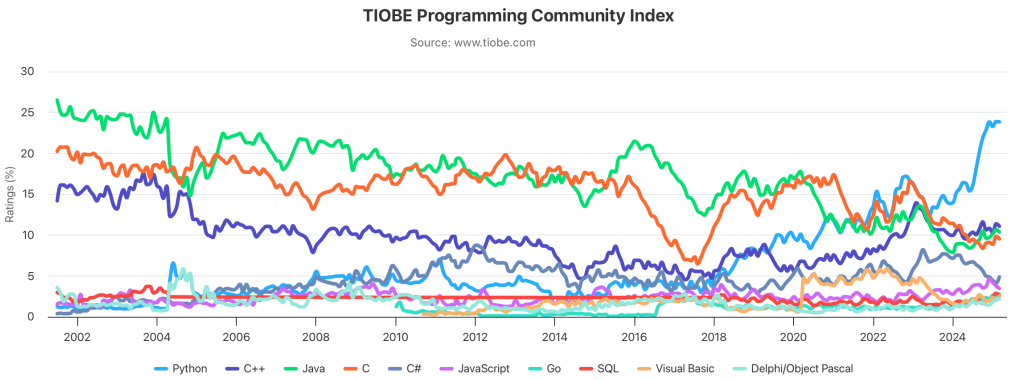
  Competent programmers/teams in secure program development…

  - Programs depend on operating systems, third-party APIs/libraries, and other external services;
  - The security of a computer system always critically depends on its configuration and its execution environment — the mismatch between the intended/planned environment and the deployed one is always significant;
  - It is sometimes unclear whether software houses are willing to pay the price associated with secure coding or whether customers will be willing to pay higher prices and accept longer development times.

- Secure programming depends much more collectively on the development team and the company's culture rather than on the expertise of isolated programmers.

Factors affecting Software Security

- Some factors that influence the degree of exposure to security issues:
  1. Security support of the underlying operational environment/programming language
  2. Open-Source vs. Closed-Source
  3. **Design, coding, and operational vulnerabilities**;
  4. Implemented security mechanisms (or controls);
  5. Organization's maturity level of the development process;



Another interesting PL ranking: https://spectrum.ieee.org/top-programming-languages-2024

Security & Programming Languages

- Programming language features significantly impact the likelihood of security vulnerabilities (e.g., type-safety, scope of type-system; memory safety, etc.)
- Examples:
  - **C/C++** – Compiled language (directly into assembly). Advantage: Excellent performance and flexibility. Disadvantage: Poor memory management that leads to numerous vulnerabilities.
  - **Java** — Compiled into an intermediate language (**JVM**), which is interpreted or optionally compiled at runtime. Advantage: Careful memory management; can be run in a sandbox that imposes access control policies to system resources. Disadvantage: Lower performance.
  - **Rust** — Modern language with a rich type system that guarantees memory-safety and thread-safety, without compromising efficiency.
  - **Perl** — Interpreted language with the possibility of operating in taint mode (security in validating the application input, ensuring that the input is not passed to dangerous commands from a security point of view before being validated).
- But:
  - Using programming language X instead of Y is not a “security feature” in itself!
  - Each programming language typically has its specific application domain…
  - The choice of programming language is typically conditioned by several “external” constraints (e.g., third-party components/libraries; corporate expertise/policy; etc.)

Open-Source vs. Closed-Source

- Controversial subject, which involves ideological charge and economic interests;
- From the point of view of software security, there is a strong argument in favor of open-source security: <u>being open to scrutiny</u> (many eyeballs looking at it).
  - But can thousands of eyes looking at the code discover (often subtle) vulnerabilities?
  - Only those willing to contribute to increasing the security of the system will look at the code?
  - If everybody is able to contribute to the code, won’t it mean that anybody can introduce vulnerabilities in it?
  - Main takeaway: <u>securing open-sourced projects is challenging</u>! (e.g., strict reviewing policy; managing trustworthiness of contributors; CI/CD & testing; etc.)
- On the closed-code side:
  - Aren’t we again on the “security through obscurity” mantra?
  - It’s often very difficult to construct really “closed-code” systems (in the sense that one exerts control over every component/library relied upon).
- In practice, keeping track of all dependencies and their security aspects will play a significant role in the overall security of systems (c.f. Bill-of-Materials)

Software Security (concepts)

- Security aims at ensuring:
  - **Confidentiality** - the absence of unauthorized disclosure of information;
  - **Integrity** - the absence of unauthorized changes to the system or information;
  - **Availability** - readiness of information to be accessed or of the system to provide a service;
  - **Authenticity** - information or service provided is genuine.

Software Security (concepts)

- Security aims at ensuring:
  - **Confidentiality** - the absence of unauthorized disclosure of information;
  - **Integrity** - the absence of unauthorized changes to the system or information;
  - **Availability** - readiness of information to be accessed or of the system to provide a service;
  - **Authenticity** - information or service provided is genuine.

Privacy is the confidentiality of personal data (name, phone number, address, ...).

Software Security (concepts)

- Security aims at ensuring:
  - **Confidentiality** - the absence of unauthorized disclosure of information;
  - **Integrity** - the absence of unauthorized changes to the system or information;
  - **Availability** - readiness of information to be accessed or of the system to provide a service;
  - **Authenticity** - information or service provided is genuine.

The definition of what is or is not authorized constitutes its **security policy**, i.e., the set of security requirements the system must meet.

Software Security (concepts)

- A **Threat** is a potential danger or malicious activity with a potential negative impact on the system's behavior.
- A **Vulnerability** is a weakness or flaw (defect) that a threat could exploit.
- An **attack** is a malicious action that aims to activate one or more vulnerabilities in order to subvert the system's security policy.
- An **intrusion** is a successful attack, i.e. when an attacker successfully activates one or more vulnerabilities to compromise the system.
- An **exploit** is a scheme/code capable of activating a certain vulnerability in a software package or system.
  - Exploit is also used as a verb to name the action of performing an attack.

Attack + vulnerability = intrusion

Risk-based approach to security

- **Threat Mitigation** refers to the strategies and actions taken to reduce the likelihood and impact of security threats.
- **Threat Modeling** is a structured approach to identifying, analyzing, and mitigating security threats.
- A risk-based approach can be used to prioritize threats based on their likelihood and impact

*Risk = Likelihood * Impact*

- ***Aiming at zero vulnerabilities is unrealistic!***

  "*Effective risk management involves not just addressing known risks but also being prepared for unexpected challenges" (Michael Lee, source: Forbes)*

Software Vulnerabilities

- Software vulnerabilities can be classified as:
  - *Project vulnerability* — introduced during the first stages of the software development (requirements or design phases);
  - *Coding vulnerability* (implementation) — introduced during programming (e.g., a bug with security implications);
  - *Operational vulnerability* — caused by the execution environment in which the software runs (e.g. configuration settings).

Software security should address each of these three types of vulnerabilities.

Coding Vulnerabilities

- Coding vulnerabilities have their own life cycle:
  - When a software package begins to be marketed, it contains numerous bugs.
    - It is estimated that any software package has an average of 5 to 50 bugs per 1,000 SLOC (Source Lines Of Code), excluding comment lines, some of which are vulnerabilities.
      - Upper limit (50 bugs per 1,000 SLOC) for normal software;
      - Lower limit (5 bugs per 1,000 SLOC) for software developed using rigorous secure development methods.

Software Package	Files	SLOC
Windows XP (2001)	n/a	45 million (1)
MacOSX 10.4 (2005)	n/a	86 million(1)
Linux kernel pre4.2 (2015)	n/a	20.2 million(1)
MariaDB 10.1	5.014	2.127.699(2)
PostgreSQL 9.4.4	3.864	1.698.471(2)
Python 2.7.10	3.240	998.697(2)
Perl 5.22.0	3.434	903.874(2)

(1) https://en.wikipedia.org/wiki/Source_lines_of_code
(2) https://github.com/aldanial/cloc

Coding Vulnerabilities (…)

- When a vulnerability is discovered, the company that created the software should create a **patch**/ fix.
- The software users (or system administrators) should install the patch to remove the vulnerability.
- Negative aspects of patches:
  - Publishing a patch is a public disclosure that the product has problems;
  - Publishing a patch allows attackers to discover the vulnerability the patch intends to solve and create exploits that take advantage of it.
  - Installing a patch has administration costs;
  - Installing a patch can solve a vulnerability but introduce another one(s).

- To minimize these effects, some software manufacturers publish patches with a fixed periodicity (except for critical vulnerabilities, for which they publish immediate patches).
- Or provide frequent new software versions with new features (which solve known vulnerabilities).

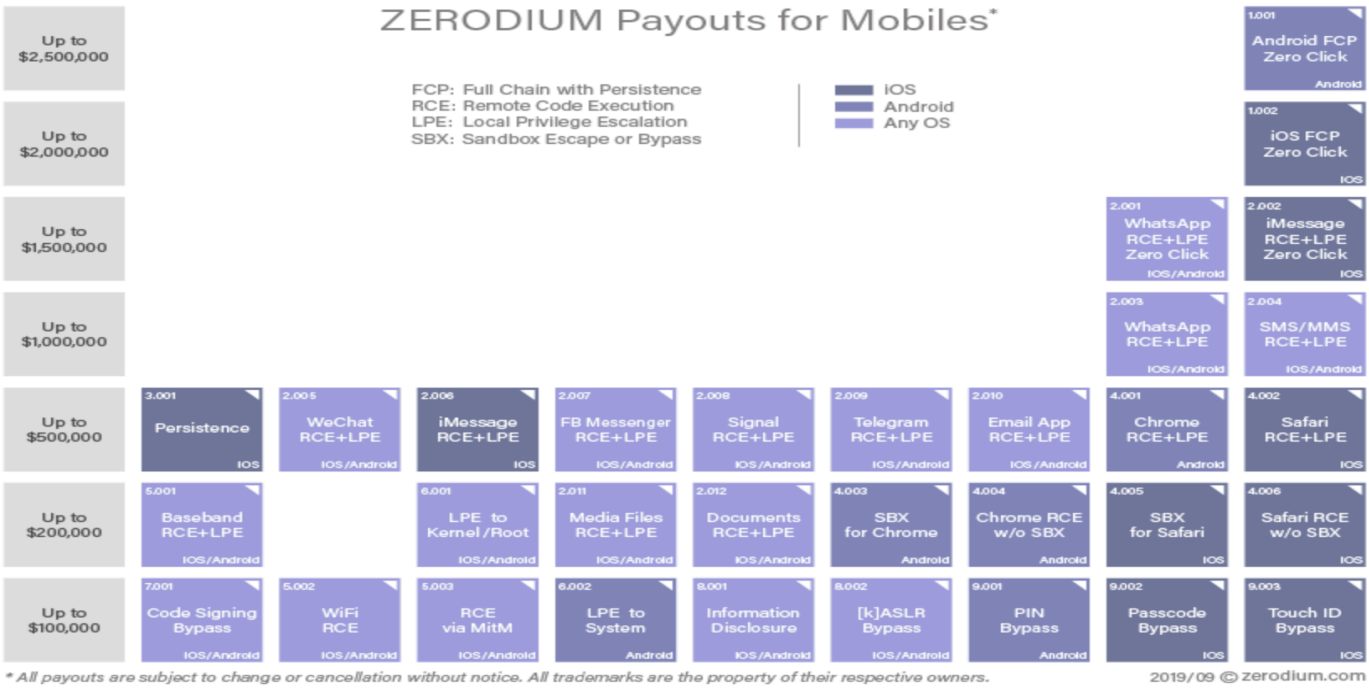
0-day vulnerabilities

- The most harmful vulnerabilities are not known by the computer security community but only by a restricted community. E.g.:
  - Group of hackers (e.g., for sale on the dark web),
  - Military or intelligence agencies of states (cyber-weapons).
- Called **zero-day vulnerabilities**.

Zero-day attacks often allow you to successfully attack systems administered by competent teams with good security knowledge.

Pos	$5k-$25k	Link	Trend	Today	0-day
1	Google Chrome Dev Tools use after free [CVE-2022-1493]	✗		$5k-$25k	$25k-$100k
2	Google Chrome Blink Editing Remote Code Execution [CVE-2022-1492]	✗		$5k-$25k	$25k-$100k
3	Google Chrome Extensions API Remote Code Execution [CVE-2022-1488]	✗		$5k-$25k	$25k-$100k

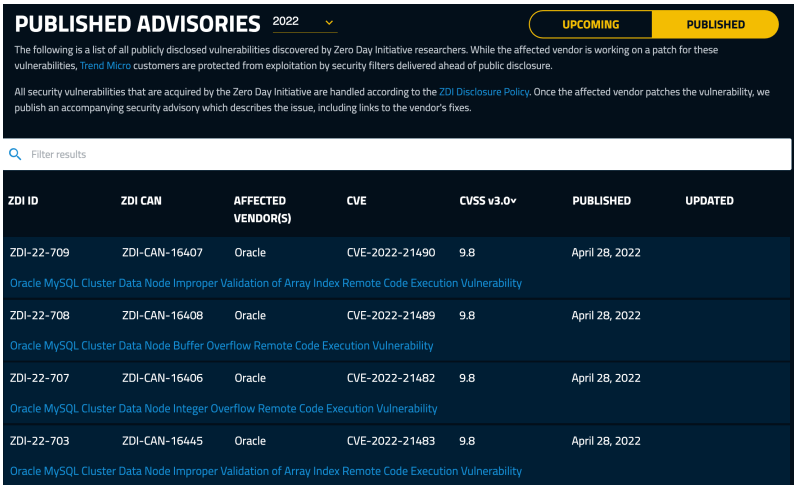
(https://vuldb.com/?exploits.top)



Once available at: https://zerodium.com/program.html

Vulnerability Disclosure

- Vulnerabilities are disclosure to the community (compiled in publicly accessible databases)
  - Of course, when a vulnerability is made public, it becomes useable to attack systems.
    - Would it make sense to outlaw the publication of vulnerabilities?
    - What is the ethical way to publish a vulnerability?



PUBLISHED ADVISORIES 2022 — UPCOMING | PUBLISHED

The following is a list of all publicly disclosed vulnerabilities discovered by Zero Day Initiative researchers. While the affected vendor is working on a patch for these vulnerabilities, Trend Micro customers are protected from exploitation by security filters delivered ahead of public disclosure.

All security vulnerabilities that are acquired by the Zero Day Initiative are handled according to the ZDI Disclosure Policy. Once the affected vendor patches the vulnerability, we publish an accompanying security advisory which describes the issue, including links to the vendor's fixes.

Filter results

ZDI ID	ZDI CAN	AFFECTED VENDOR(S)	CVE	CVSS v3.0	PUBLISHED	UPDATED
ZDI-22-709	ZDI-CAN-16407	Oracle	CVE-2022-21490	9.8	April 28, 2022	
Oracle MySQL Cluster Data Node Improper Validation of Array Index Remote Code Execution Vulnerability						
ZDI-22-708	ZDI-CAN-16408	Oracle	CVE-2022-21489	9.8	April 28, 2022	
Oracle MySQL Cluster Data Node Buffer Overflow Remote Code Execution Vulnerability						
ZDI-22-707	ZDI-CAN-16406	Oracle	CVE-2022-21482	9.8	April 28, 2022	
Oracle MySQL Cluster Data Node Integer Overflow Remote Code Execution Vulnerability						
ZDI-22-703	ZDI-CAN-16445	Oracle	CVE-2022-21483	9.8	April 28, 2022	
Oracle MySQL Cluster Data Node Improper Validation of Array Index Remote Code Execution Vulnerability						

https://www.zerodayinitiative.com/advisories/published/

Vulnerability

ZDI-22-709
ZDI-CAN-16407

CVE ID	CVE-2022-21490
CVSS SCORE	9.8, (AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H)
AFFECTED VENDORS	Oracle
AFFECTED PRODUCTS	MySQL Cluster
VULNERABILITY DETAILS	This vulnerability allows remote attackers to execute arbitrary code on affected installations of Oracle MySQL Cluster. Authentication is not required to exploit this vulnerability. The specific flaw exists within the processing of Data Node jobs. The issue results from the lack of proper validation of user-supplied data, which can result in a write past the end of an array. An attacker can leverage this vulnerability to execute code in the context of the service account.
ADDITIONAL DETAILS	Oracle has issued an update to correct this vulnerability. More details can be found at: https://www.oracle.com/security-alerts/cpuapr2022.html
DISCLOSURE TIMELINE	2022-03-02 - Vulnerability reported to vendor 2022-04-28 - Coordinated public release of advisory
CREDIT	Anonymous

(https://www.zerodayinitiative.com/advisories/ZDI-22-709/)

Vulnerability Catalogs

- **Several sources of information often cross-referenced**…
- MITRE
  - **Common Weakness Enumeration (CWE)** – Vulnerability class classification, in which each vulnerability class is assigned an identifier with the CWE-NNNN format, with NNNN being the number assigned to the class.
  - **Common Vulnerabilities and Exposures (CVE)** – Catalogue of vulnerabilities (design and coding) existing in commercial or open software, with identifier with CVE-YYYY-NNNN format, being YYYY the year and NNNN its number.
  - **Common Attack Pattern Enumeration and Classification (CAPEC)** — common attack patterns that helps users understand how adversaries exploit weaknesses in applications.

Vulnerability Catalogs (…)

- OWASP:
  - OWASP Top-10 — a ranking of the most common vulnerability categories
- NIST:
  - **Security Content Automation Protocol (SCAP)** [NIST SP 800-126]— synthesis of interoperable specifications (for formats, nomenclature, etc.) derived from community ideas.
  - **National Vulnerability Database (NVD)** — U.S. government repository of standards-based vulnerability management data represented using the Security Content Automation Protocol (SCAP).
  - National Checklist Program (NCP) and Common Configuration Enumeration (CCE) — security checklists (or benchmarks) that provide detailed low-level guidance on setting the security configuration of operating systems and applications.
- Other initiatives:
  - **Common Vulnerability Scoring System (CVSS)** - provides a way to capture the principal characteristics of a vulnerability and produce a numerical score reflecting its severity
  - **A distributed vulnerability database for Open Source (OSV)** — a distributed approach to producing and consuming vulnerability information for open source.

…and Exploits databases

- When a vulnerability is made public, the path to construct an exploit is open.
- An attack from a known exploit is always a simple task accessible even to non-specialist hackers.
- Exploit catalogs:
  - **Exploit Database** - public exploit archive, identifying the CVE of the vulnerability and/or the targeted software. It is intended to be used by vulnerability investigators and penetration testers (but…).
  - Google Hacking Database - Google Dorks file (search queries that return sensitive information) that, although a kind of exploit, can also be used to create new exploits.
    - Used in tools such as Metasploit and similars.

Secure Software Development

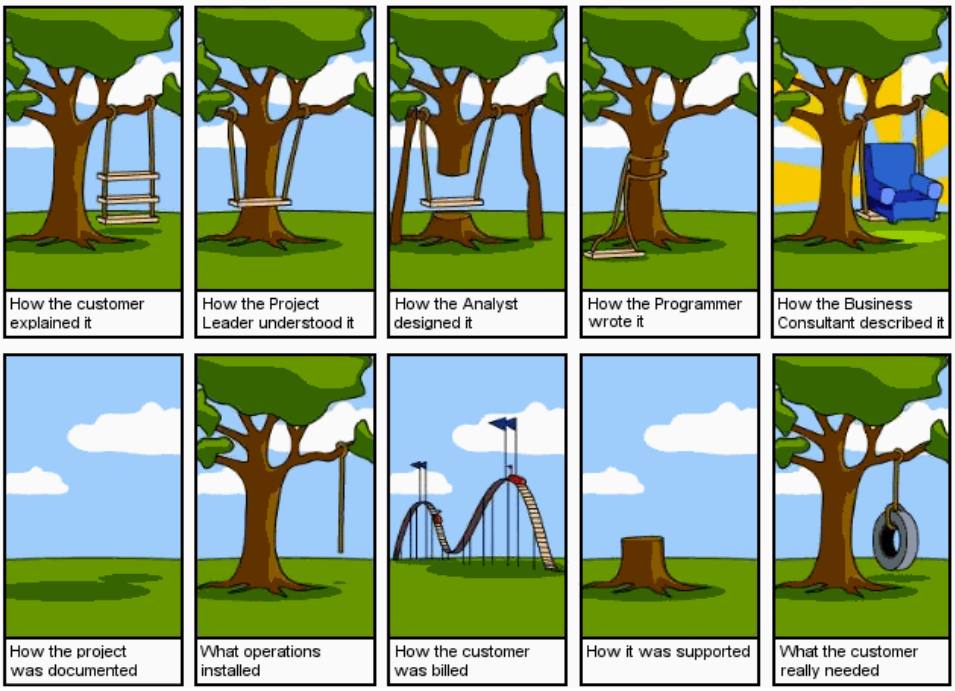
Software Security

Secure Software Development Lifecycle (S-SDLC)

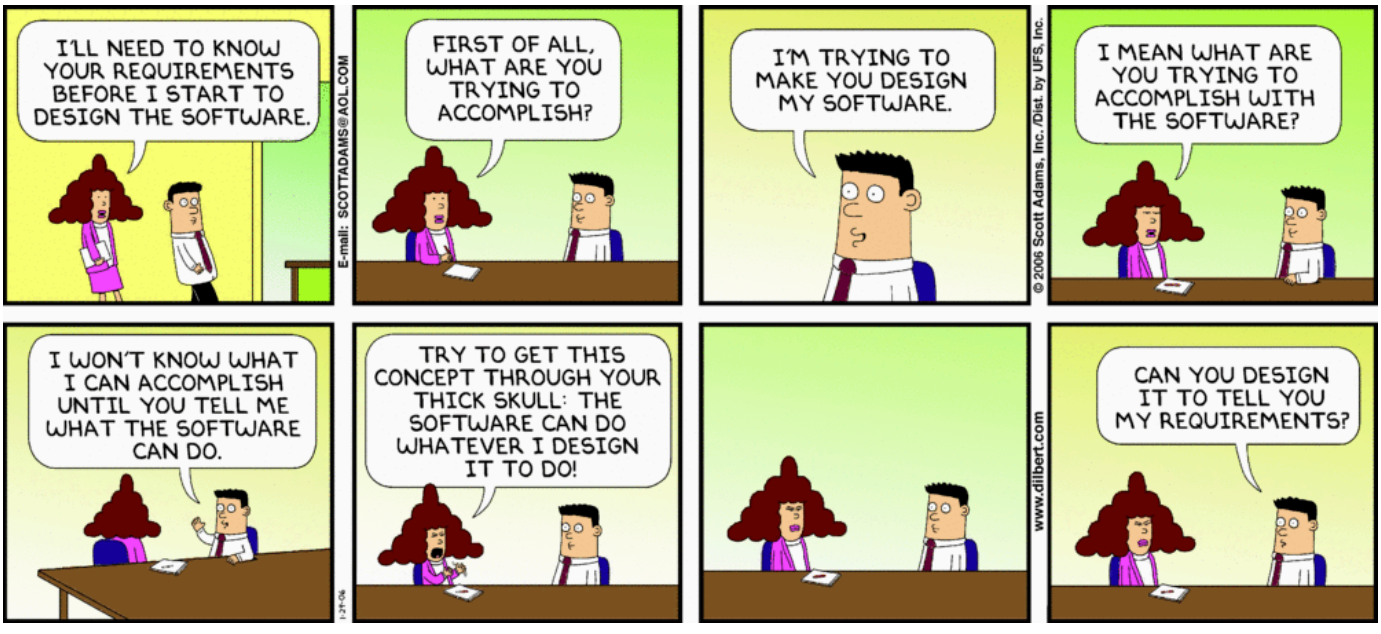
Common Code Vulnerabilities

based on the slides of the 23/24 edition of the course (author: jose.miranda@devisefutures.com)

Secure Software Development Lifecycle (S-SDLC)



Secure Software Development Lifecycle (S-SDLC)



Secure Software Development Lifecycle (S-SDLC)

Software Life Cycle Process

- The process used by the software industry to acquire, supply, develop, operate, and maintain software;
- Its **objective** is to produce high-quality software that meets or exceeds the customer's expectations and is completed within the estimated *time and cost*;
- ISO/IEC/IEEE 12207:2017 (*Systems and software engineering -- Software life cycle processes*) defines Software Engineering processes, activities, and tasks associated with the software life cycle from its conception to the withdrawal/discontinuation of the software, to be adapted according to each software project.
  - **Fundamental processes**: Acquisition, Supply, Development, Operation, Maintenance.
  - **Supporting processes**: Documentation, Configuration management, Quality assurance, Verification, Validation, Joint review, Auditing, Problem resolution, Usability, Change request management, Product evaluation.
  - **Organizational processes**: Management, Infrastructure, Improvement, Human Resources.
  - **Software reuse processes**: Asset management, Reuse program management, Maintenance engineering.
  - **Adaptation processes** within the scope of the Project, Organization, Culture, Life cycle model, methods and techniques, and languages.

Fundamental processes (ISO/IEC/IEEE 12207:2017)

- **Acquisition** – all activities to obtain the product/service that meets the needs of the contracting company;
- **Supply** – development of the project management plan, with the different stages/milestones, to be used in the development process;
- **Development** – design, creation and testing of the software product/system, resulting in a software product/system capable of being delivered to the customer;
- **Operation** – activities necessary to use the software product/system;
- **Maintenance** – activities necessary to keep the product/system working.

Acquisition (ISO/IEC/IEEE 12207:2017)

- **Beginning and scope**: Description of the need to acquire, develop, or improve a software product/system; Definition and approval of software product/system requirements; Evaluation of alternatives (purchase of an off-the-shelf product, specific development or improvement of an existing product); Development of acquisition plan; Definition of acceptance criteria for the software product/system;
- **Preparation of the request for proposal**: Contains product/software system requirements and technical restrictions (interoperability with other systems, the environment in which it will be used, ...), delivery stages/milestones, and acceptance criteria;
- **Contract preparation**: Development of supplier selection procedure; Selection of suppliers, based on the selection procedure; Initial version of the contract;
- **Negotiation of changes**: Negotiation with selected suppliers;
- **Contract update**: The contract is updated as a result of negotiations from the previous activity;
- **Supplier monitoring**: Monitoring supplier activity according to the contracted stages/milestones (if necessary, work directly with suppliers to ensure timely deliveries);
- **Acceptance and finalization**: Development of tests and acceptance procedures; Carrying out acceptance tests; Carrying out product/system configuration management.

Development (ISO/IEC/IEEE 12207:2017)

Requirements Analysis

- **Requirements gathering** – obtain and define the customer's requirements and requests through their direct request or through the requested proposal. It is essential to understand customer expectations and ensure that both the customer and the supplier understand the requirements in the same way. It is important to manage all changes made to customer requirements in relation to the reference scenario ensuring that the outcome of technological changes and customer needs are identified and the impacts of introducing these changes are assessed.
- **Requirements specification** – Definition and documentation of the requirements of the software product/system (system functions and capabilities - functional requirements; Business, organizational, and user requirements; Security, ergonomics, interface, operation, and maintenance requirements; Project restrictions and certification/accreditation requirements);

Design

- **High-level architecture** – Design of the basic organization of the product/system, i.e., identification of the different modules (hardware, software, and manual operations) and how they communicate with each other (without great detail of the modules);
- **Solution design** – The different modules are designed separately, with as much detail as possible;

Implementation

- **Coding/Implementation** – Creation of the code according to the high-level architecture and design of the modules; Execution of change control; Documentation and resolution of non-conformities and problems found; Selection, adaptation, and use of standards, methods, tools, and programming languages.

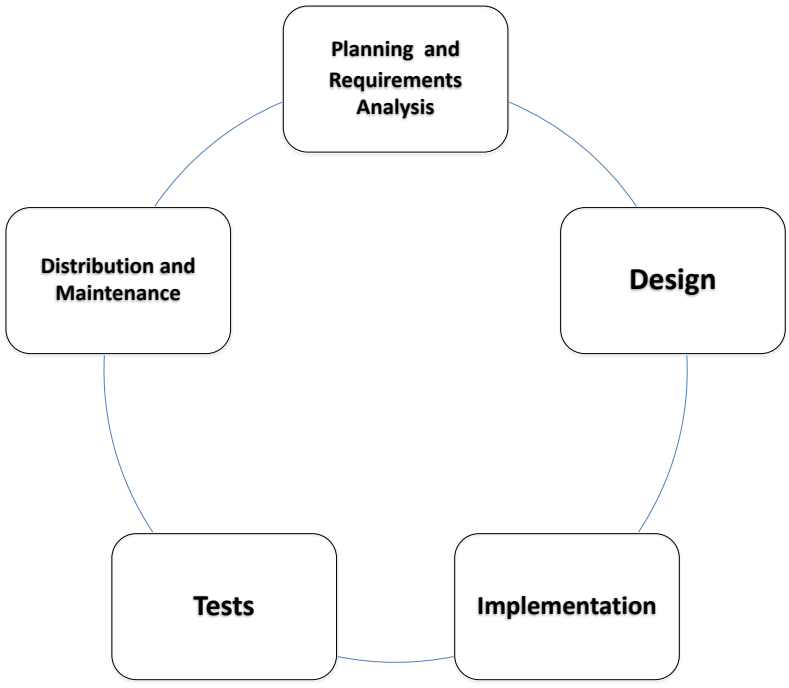
Testing

- **Functional testing** – Testing the correct functioning of each of the different modules;
- **Integration test** – Testing the correct functioning of communication between modules;
- **System testing** – Testing that all functional requirements are present in the software product/system. In this case, the product is finished and can be delivered to the customer;

Distribution/ Maintenance

- **Installation/deployment of the software product/system** – Installation of the software product/system in the target environment; Configuration and initialization of the software product/system and other necessary systems (e.g. database);
- **Support for software product/system acceptance** – Support for customer acceptance of the software, including customer testing.

Typical phases of the Software Development Lifecycle (SDLC)



Security in the Software Development Lifecycle

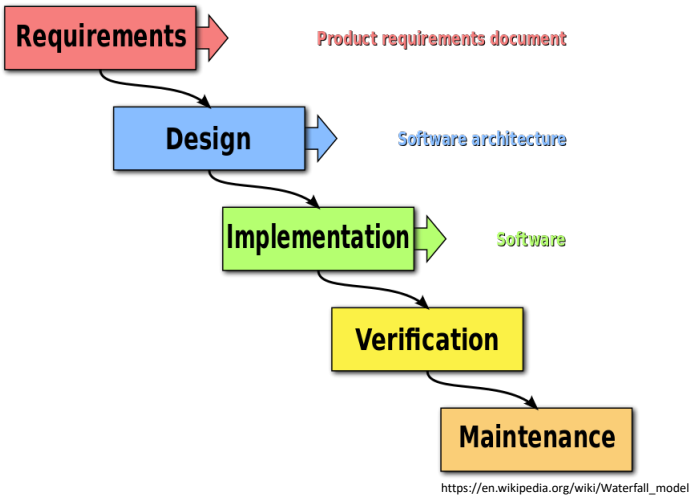
- **Security flaws** can be introduced at any stage of the development cycle:
  - At the very beginning, by not identifying security needs
  - When creating conceptual architectures that have logic errors
  - The use of bad programming practices that introduce technical vulnerabilities
  - Implementing errors or inappropriate coding policy
  - Introducing faults during maintenance or software upgrade
- **Software vulnerabilities** can result from a much broader scope than the software product itself. Among others:
  - The third-party APIs used by the software
  - The application servers (execution environment) used
  - The operating system of the associated servers
  - The backend database (storage)
  - Other applications in a shared environment
  - The system users
  - Other software with which the user interacts

Security in the Software Development Lifecycle

- Security in SDLC:
  - **Adding security in conventional software development models;**
  - Secure software development models;
  - Assessment and improvement – Maturity Model.

Adding security into SDLC

- One way to introduce security into the SDLC is to consider (?introduce?) security into conventional software development models.
  - e.g. Waterfall model



E.g. Waterfall model

- Security must be considered in the different phases of the model
  - **Requirements Phase**
    - Security requirements must be based on current legislation (e.g., Sarbanes-Oxley Act, Health Insurance Portability and Accountability Act (HIPAA), GDPR, ...) and international recommendations and standards (ISO 27000 family; industry standards; etc.), as applicable, and must be translated into specific requirements for the software to be developed.
  - **Design Phase**
    - The security requirements defined in the previous phase are translated into mechanisms that implement them (authentication mechanisms, protection of confidentiality and data integrity, ...).

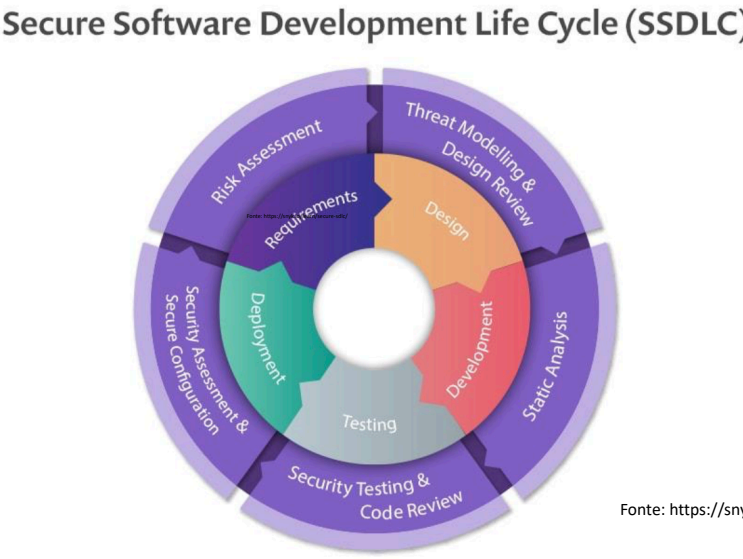
Project Vulnerabilities

  - **Implementation Phase**
    - Secure programming requires knowledge of good coding practices, tool support to detect possible vulnerabilities, adopt secure code standards, etc.
  - **Testing Phase**
    - Perform validations and security tests.

Program Vulnerabilities

  - **Maintenance Phase**
    - Security measures in the distribution of patches, product updates/upgrades, ...

- But in fact, security aspects end up constituting a new layer that comprises all phases of the model.



Fonte: https://snyk.io/learn/secure-sdlc/

Security in the Software Development Lifecycle

- Security in SDLC:
  - Adding security in conventional software development models;
  - **Secure software development models;**
  - Assessment and improvement – Maturity Model.

Microsoft Secure Software Development Lifecycle (S-SDLC)

- (Pre)**Training Phase**
  - Preparation of software development team members to deal with security aspects.
- **Requirements Phase**
  - Identify the people responsible for coordinating project security issues;
  - Identify bug tracking tool suitable for vulnerability management;
  - Define minimum security requirements in a similar way to what was defined for the requirements phase of the waterfall model.
- **Design Phase**
  - Design the security mechanisms to be implemented, following recommendations and best practices, and avoiding design vulnerabilities;
  - Carry out security risk analysis in order to identify the most critical risks from a security and privacy point of view.
- **Coding Phase**
  - Establish and follow good coding practices;
  - Document how users should configure software securely.
- **Verification Phase**
  - The objective is to ensure the software has the desired security properties through manual code review, testing, and static code analysis.
- **Publication/Maintenance Phase**
  - Plan the actions to be taken when vulnerabilities are discovered;
  - A final, independent security review must be carried out before publication.
- (Post)**Response Phase**
  - Collecting information about security incidents and creating reports on vulnerabilities and patches.

Security Requirements

- Functional vs. Security requirements
  - functional requirement: what the software must do;
  - security requirement: security expectations when running/using the software.
- Two main types of security requirements:
  - Security-related goals (**security policy**):
    - **Confidentiality** (and Privacy and Anonymity)
      - Example Policy: Bank account balance is known only to its legitimate owner
    - **Integrity**
      - Example Policy: operations on the account must be authorized by the holder
    - **Availability**
      - Example Policy: the user can always access their account balance

Security Requirements

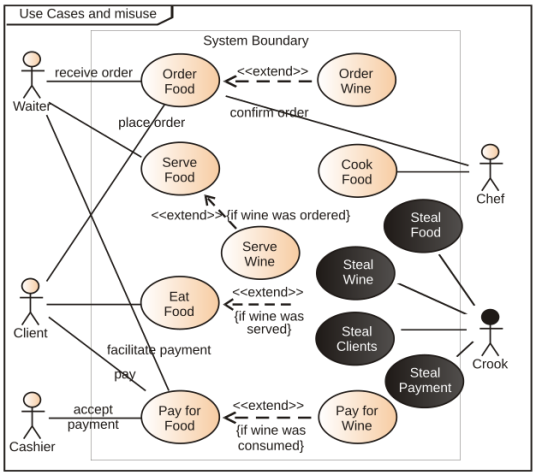
- Functional vs. Security requirements
  - functional requirement: what the software must do;
  - security requirement: security expectations when running/using the software.
- Two main types of security requirements:
  - Security-related goals (**security policy**):
  - **Mechanisms** to enforce requirements:
    - **Authentication**
      - What the user knows/is/has (e.g., password/fingerprint/citizen card),
    - **Authorization**
      - According to policies that define actions that can be authorized (access control, ...)
    - **Auditability**
      - Store enough information to determine the circumstances of a given action (usually in logs, protected from tampering)
- Example: https://github.com/eu-digital-identity-wallet/eudi-doc-architecture-and-reference-framework/blob/main/docs/annexes/annex-2/annex-2-high-level-requirements.md

Security Requirements

- Common sources of security requirements:
  - Based on legislation applicable to the business;
  - Based on standards;
  - Based on company values and/or policies;
  - Based on misuse cases or attack trees;
  - Based on risk assessment and threat modeling.

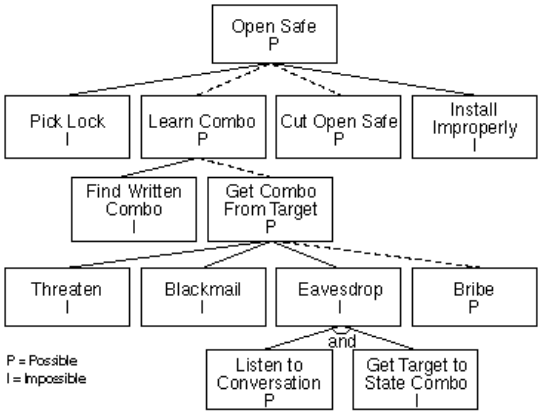
Misuse cases

- Variation/Extension of use-case scenarios to capture “negative behavior.”
- Describe what **should not happen** and/or be allowed;
- Build abuse cases in which an attacker can violate security requirements from attack patterns and likely scenarios.
  - Example: an attacker steals the password file and obtains all user passwords;
  - Another example: an attacker repeating a money transfer message.



Attack-Trees

- Model security threats by focusing on the attackers and the different ways they may try to attack systems
  - Root node represents the attack goal.
  - Branches in the tree represent the different paths an attacker can follow to achieve his or her goal.
    - OR-nodes represent alternatives,
    - while AND-nodes represent subgoals where all must be fulfilled in order for the attack to be successful.
  - Trees can be shown graphically or be written in outline form.



https://www.schneier.com/academic/archives/1999/12/attack_trees.html

Risk Assessment vs. Threat Modeling

Aspect	Risk Assessment	Threat Modeling
Focus	Evaluating potential risks and their impact on the organization as a whole.	Identifying and mitigating specific threats and vulnerabilities.
Scope	Broad – Considers the organization's overall risks and objectives.	Narrow – Focused on a specific system, application, or component.
Time horizon	Ongoing process that considers both current and future risks.	Typically conducted during the development or change phase of a system.
Implementation	Guides the selection and implementation of strategies to manage risks.	Helps design and implement specific countermeasures to mitigate threats.
Goal	Assessing potential risks and their impact for effective risk assessment	Identifying and prioritizing specific threats for proactive mitigation.

Source: https://www.practical-devsecops.com/threat-modeling-vs-risk-assessment/

Threat Modeling

Threat Modeling

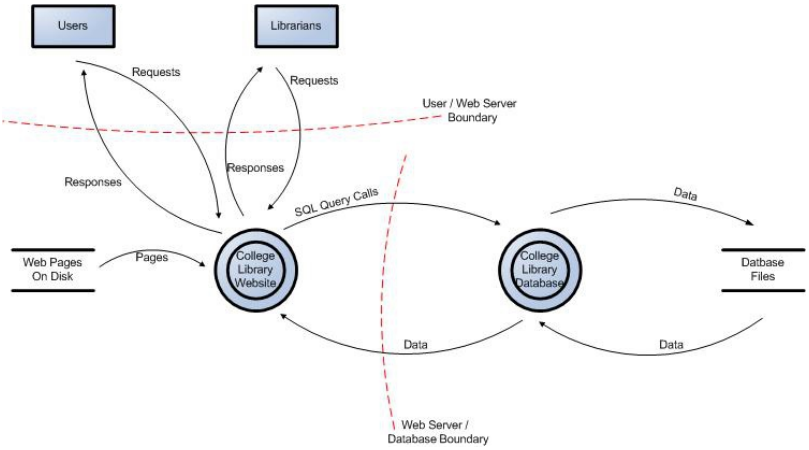
- Threat modeling analyzes system representations to highlight concerns about security and privacy characteristics.
- It examines the system from an attacker's perspective.
- Shostack's Four Question Framework:
    - **What are we working on?**
    - **What can go wrong?**
    - **What are we going to do about it?**
    - **Did we do a good enough job?**
- Principles (Threat Modeling Manifesto):
    - The best use of threat modeling is to improve the security and privacy of a system through early and frequent analysis.
    - Threat modeling must align with an organization's development practices and follow design changes in iterations that are each scoped to manageable portions of the system.
    - The outcomes of threat modeling are meaningful when they are of value to stakeholders.
    - Dialog is key to establishing the common understandings that lead to value, while documents record those understandings and enable measurement.

Threat Modeling

- Benefits:
    - Identify potential flaws at the start of the SLDC
        - …with benefits to the whole SDLC;
    - Create secure-by-design software
- It can be deployed at different levels (e.g., application vs. operational threat modeling)
- Steps:
    1. **Scope of the work** — gaining an understanding of what you're working on;
    2. **Identify Assets & Data Flows** — document key assets, such as data stores, user credentials, etc.;
    3. **Threat Finding** — identifying potential security risks that could exploit vulnerabilities in a system;
    4. **Analyse Vulnerabilities & Risk Assessment** — prioritize risks based on likelihood and impact
    5. **Countermeasures and Mitigation** — evaluate the impact and likelihood of threats; accept/eliminate/mitigate/transfer risks;
    6. **Assessment** and documentation

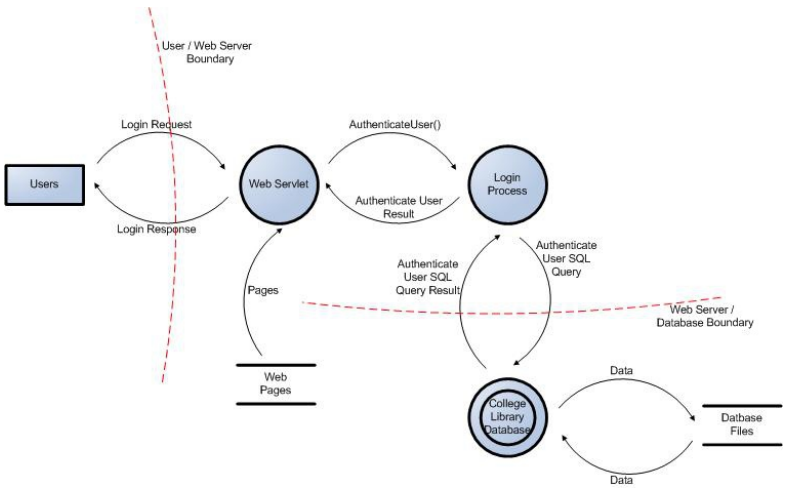
Data Flow Diagrams

- Data flow diagrams (DFDs) are frequently used to present a high-level view of the system.
- Provide a visual representation of how the application processes data.
- Show data flow pathways, highlighting the privilege or trust boundaries.



Data Flow Diagrams

- DFDs can be used to decompose the application into subsystems and lower-level subsystems.
  - The high-level DFD clarifies the scope of the application being modeled.
  - The lower-level iterations allow focusing on the specific processes involved when processing specific data



Threat Modeling frameworks

- Each methodology is unique and provides varied benefits; it is not uncommon to mix (ingredients from) distinct methodologies.
- They usually put emphasis on different aspects (e.g., organizational risk, concrete programming vulnerabilities, privacy, etc.).
- Pretty much all depend/benefit from a comprehensive knowledge base of common vulnerabilities.
- Some examples:
  - STRIDE
    - Focused on the identification of threats and vulnerabilities based on simple categorization;
    - Simple to setup and running, making it ideal for less experienced developers
    - Works well with other security frameworks like DREAD, PASTA, and NIST.
  - PASTA (Process for Attack Simulation and Threat Analysis)
    - A risk-focused approach integrating business impact.
  - LINDDUN
    - Focused on privacy threats, such as data-leaks, profiling, surveillance, or compliance with legal privacy requirements (e.g. RGPD).
  - Attack-Trees
    - Visualize attack paths to find weaknesses.

STRIDE

Type	Description	Security Control
Spoofing	Threat action aimed at accessing and use of another user's credentials, such as username and password.	**Authentication**
Tampering	Threat action intending to maliciously change or modify persistent data, such as records in a database, and the alteration of data in transit between two computers over an open network, such as the Internet.	**Integrity**
Repudiation	Threat action aimed at performing prohibited operations in a system that lacks the ability to trace the operations.	**Non-Repudiation**
Information Disclosure	Threat action intending to read a file that one was not granted access to, or to read data in transit.	**Confidentiality**
Denial of Service	Threat action attempting to deny access to valid users, such as by making a web server temporarily unavailable or unusable.	**Availability**
Elevation of Privilege	Threat action intending to gain privileged access to resources in order to gain unauthorized access to information or to compromise a system.	**Authorization**

Analyse Vulnerabilities & Risk Assessment

- Map vulnerabilities to known attack patterns (CAPEC, OWASP top-10, etc.)
- Threats should be prioritized according to their risk:
  - Focus on high-impact, high-likelihood threats first.
  - Or rely on some risk metric (e.g., CVSS)

Risk-factor	High	Medium	Low
Likelihod	Easy to exploit, widely known	Requires effort, some knowledge needed	Hard to exploit, rare occurrence
Impact	Severe system/data loss	Limited impact, minor disruptions	Minimal damage, easily recoverable

Mitigation & Countermeasures

- For each high-priority threat, implement appropriate security controls.
  - **Preventive** — such as the use of cryptography authentication, and authorization mechanisms, etc;
  - **Detective** — such as logging, monitoring, etc.
  - **Corrective** — incidence response, backups, etc.
- Example (SQL injection):
  - input validation;
  - use parametric procedures;
  - restrict database permissions.

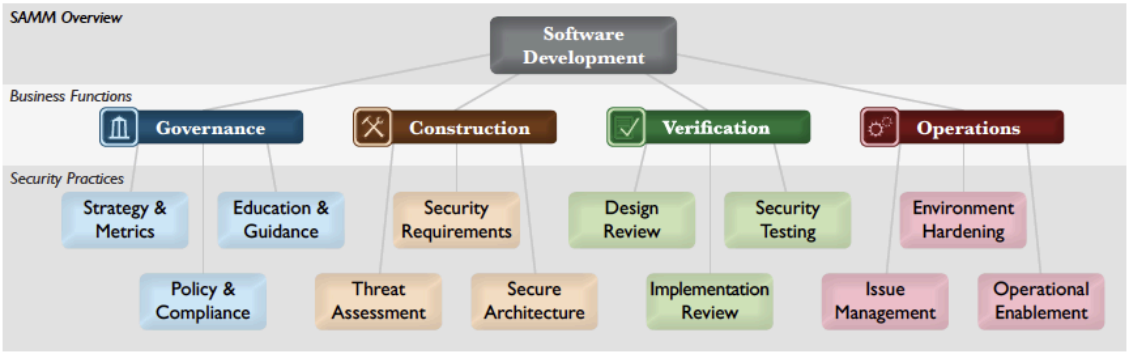
Maturity Models

Maturity Models

- We can use a **Maturity Model** to evaluate and improve secure software development practices in an organization.
  - The behavior of organizations changes slowly, so changes must be iterative, according to long-term objectives (secure software development);
  - There is no recipe that works for all organizations, so the organization must be able to choose/prioritize risks;
  - It becomes necessary to evaluate the existing software security practices in the organization and create a software security program that allows you to achieve your goals in well-defined iterations;
  - The guide to security activities to be considered in software development should provide sufficient details for people outside the security area;
  - In general, the model to be used for the development of secure software should be simple, well-defined, and measurable, as well as integrated into the organization's SDLC.
- Some Examples:
  - Microsoft SDL Optimization Model
  - Building Security In Maturity Model (BSIMM)
  - OWASP Software Assurance Maturity Model (SAMM) - https://owasp.org/www-project-samm/ -

Software Assurance Maturity Model (SAMM)

- The goal is to help organizations (whether developing, outsourcing, or acquiring software) evaluate, formulate, and implement a strategy for software security, which can be integrated into the SDLC used.
- SAMM Description:
  - Based on 12 security practices, which are grouped into 4 business functions;
  - Each security practice contains a set of activities structured in three maturity levels;
  - Activities at a lower maturity level are more easily executable and require less formalization than activities at a higher level of maturity.



Software Assurance Maturity Model (SAMM)

- The value of SAMM is to provide a mechanism to assess where the organization is on its way toward software security and to allow an understanding of what is recommended to move to a higher level of maturity.
- SAMM does not insist that all organizations reach the highest maturity level (3) in all security practices. Each organization determines the level of maturity of each "Security Practice" that best suits its needs.
- The configuration and cycle of the SAMM maturity model are made to allow:
  - The evaluation of current software security practices,
  - The definition of the objectives that best suit the organization,
  - The formulation of a path to achieve the targeted objectives (iteratively), and
  - The execution of the specific activities of security practices in the appropriate maturity for the company, following descriptive measures.

SAMM - Maturity Level (example)

- The three maturity levels of the "Education and Guidance" security practice (included in the "Governance" business function)

	EG 1	EG 2	EG 3
Objective	**Offer development staff access to resources around the topics of secure programming and deployment.**	**Educate all personnel in the software life-cycle with role-specific guidance on secure development.**	**Mandate comprehensive security training and certify personnel for baseline knowledge.**
Activities	A. Conduct technical security awareness training B. Build and maintain technical guidelines	A. Conduct role-specific application security training B. Utilize security coaches to enhance project teams	A. Create formal application security support portal B. Establish role-based examination/certification
Assessment	✦ Have developers been given high-level security awareness training? ✦ Does each project team understand where to find secure development best-practices and guidance?	✦ Are those involved in the development process given role-specific security training and guidance? ✦ Are stakeholders able to pull in security coaches for use on projects?	✦ Is security-related guidance centrally controlled and consistently distributed throughout the organization? ✦ Are developers tested to ensure a baseline skill-set for secure development practices?
Results	✦ Increased developer awareness on the most common problems at the code level ✦ Maintain software with rudimentary security best-practices in place ✦ Set baseline for security know-how among technical staff ✦ Enable qualitative security checks for baseline security knowledge	✦ End-to-end awareness of the issues that leads to security vulnerabilities at the product, design, and code levels ✦ Build plans to remediate vulnerabilities and design flaws in ongoing projects ✦ Enable qualitative security checkpoints at requirements, design, and development stages ✦ Deeper understanding of security issues encourages more proactive security planning	✦ Efficient remediation of vulnerabilities in both ongoing and legacy code bases ✦ Quickly understand and mitigate against new attacks and threats ✦ Measure the amount of security knowledge of the staff and measure against a common standard ✦ Establish fair incentives toward security awareness

SAMM cycle

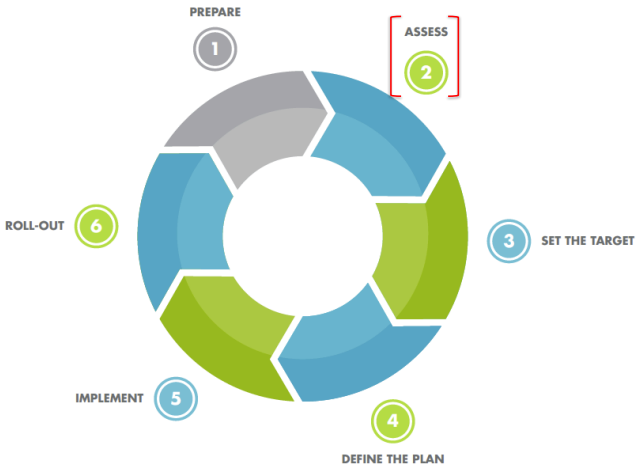
- SAMM cycle, suitable for continuous improvement (run continuously in periods of 3 to 12 months)

- **Goal**:
  - Ensure the involvement of the people necessary to determine the current level of maturity of the organization and identify the level of maturity in which it intends to be;

- **Activities**:
  - Define the scope;
  - Identify stakeholders and ensure their support for the project;
  - Inform people about the initiative and provide the information that allows them to understand what will be done.

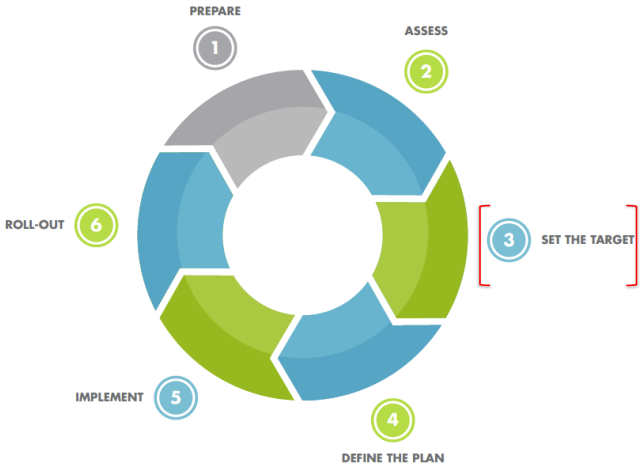
- **Goal**:
  - Identify and understand the maturity of the organization in each of the 12 security practices;

- **Activities**:
  - Evaluate current practices (using the SAMM toolbox);
  - Determine the maturity level (using the SAMM toolbox).



- **Goal**:
  - Identify a target score for each of the 12 security practices, which will serve as a guide to act on the most important activities;

- **Activities**:
  - Set the goal (using the SAMM toolbox);
  - Estimate the impact of the objective on the organization (in financial terms, if possible).

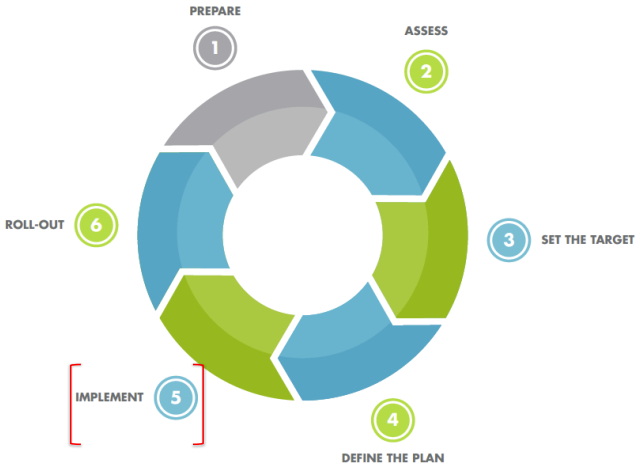


- **Goal**:
  - Develop the plan to achieve the desired maturity level (identified in phase 3);

- **Activities**:
  - Define a calendar in terms of the number and duration of phases (usually 4 to 6 phases, for 3 to 12 months);
  - Develop a plan considering the necessary effort and the possible dependence between activities.

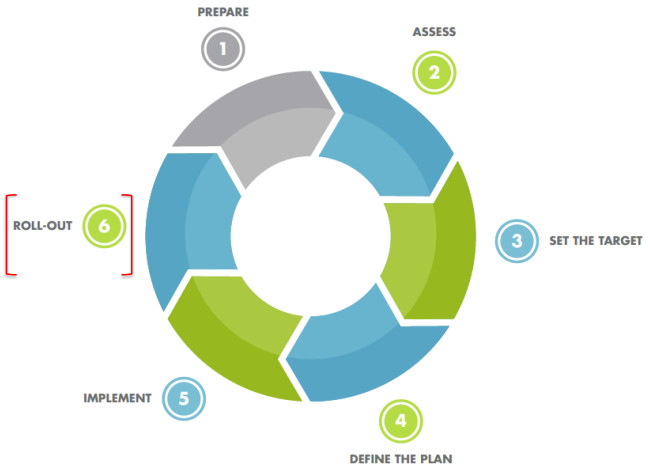
- **Goal**:
  - Implementation of the plan to achieve the desired maturity level (defined in phase 4);

- **Activities**:
  - Plan implementation activities, considering their impact on processes, people, knowledge, and tools.



- **Goal**:
  - Ensure that improvements in the software security maturity model are effective and are being followed/used in the organization;

- **Activities**:
  - Give visibility through training sessions and communication with stakeholders.
  - Measure the adoption and effectiveness of the implemented improvements by analyzing their use and impact.



Secure Software Development

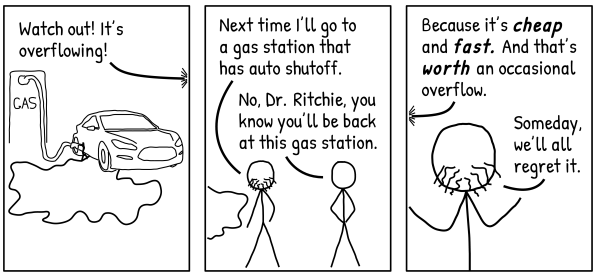
Secure Software Development Lifecycle (S-SDLC)

Software Security

Common Code Vulnerabilities

based on the slides of the 23/24 edition of the course (author: jose.miranda@devisefutures.com)

Buffer Overflow



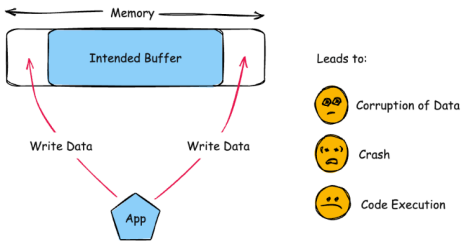
CWE-787: Out-of-bounds Write

Weakness ID: 787
Vulnerability Mapping: ALLOWED
Abstraction: Base

View customized information: **Conceptual** | Operational | Mapping Friendly | Complete | Custom

Description

The product writes data past the end, or before the beginning, of the intended buffer.



Alternate Terms

Memory Corruption: Often used to describe the consequences of writing to memory outside the bounds of a buffer, or to memory that is otherwise invalid.

Common Consequences

Relationships

Memberships

Vulnerability Mapping Notes

Taxonomy Mappings

Content History

https://cwe.mitre.org/data/definitions/787

Common Consequences

Scope	Impact	Likelihood
Integrity	**Technical Impact:** *Modify Memory; Execute Unauthorized Code or Commands* Write operations could cause memory corruption. In some cases, an adversary can modify control data such as return addresses in order to execute unexpected code.	
Availability	**Technical Impact:** *DoS: Crash, Exit, or Restart* Attempting to access out-of-range, invalid, or unauthorized memory could cause the product to crash.	
Other	**Technical Impact:** *Unexpected State* Subsequent write operations can produce undefined or unexpected results.	

This table specifies different individual consequences associated with the weakness. The Scope identifies the application security area that is violated, while the Impact describes the negative technical impact that arises if an adversary succeeds in exploiting this weakness. The Likelihood provides information about how likely the specific consequence is expected to be seen relative to the other consequences in the list. For example, there may be high likelihood that a weakness will be exploited to achieve a certain impact, but a low likelihood that it will be exploited to achieve a different impact.

Relationships

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name
ChildOf	B	119	Improper Restriction of Operations within the Bounds of a Memory Buffer
ParentOf	V	121	Stack-based Buffer Overflow
ParentOf	V	122	Heap-based Buffer Overflow
ParentOf	B	123	Write-what-where Condition
ParentOf	B	124	Buffer Underwrite ('Buffer Underflow')
CanFollow	B	822	Untrusted Pointer Dereference
CanFollow	B	823	Use of Out-of-range Pointer Offset
CanFollow	B	824	Access of Uninitialized Pointer
CanFollow	B	825	Expired Pointer Dereference

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name
MemberOf	C	1218	Memory Buffer Errors

Relevant to the view "Weaknesses for Simplified Mapping of Published Vulnerabilities" (CWE-1003)

Nature	Type	ID	Name
ChildOf	B	119	Improper Restriction of Operations within the Bounds of a Memory Buffer

Relevant to the view "CISQ Quality Measures (2020)" (CWE-1305)

Nature	Type	ID	Name
ChildOf	B	119	Improper Restriction of Operations within the Bounds of a Memory Buffer

Relevant to the view "CISQ Data Protection Measures" (CWE-1340)

Nature	Type	ID	Name
ChildOf	B	119	Improper Restriction of Operations within the Bounds of a Memory Buffer

This table shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOf and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that the user may want to explore.

Memberships

Nature	Type	ID	Name
MemberOf	V	1200	Weaknesses in the 2019 CWE Top 25 Most Dangerous Software Errors
MemberOf	V	1337	Weaknesses in the 2021 CWE Top 25 Most Dangerous Software Weaknesses
MemberOf	V	1350	Weaknesses in the 2020 CWE Top 25 Most Dangerous Software Weaknesses
MemberOf	C	1366	ICS Communications: Frail Security in Protocols
MemberOf	V	1387	Weaknesses in the 2022 CWE Top 25 Most Dangerous Software Weaknesses
MemberOf	C	1399	Comprehensive Categorization: Memory Safety
MemberOf	V	1425	Weaknesses in the 2023 CWE Top 25 Most Dangerous Software Weaknesses
MemberOf	V	1430	Weaknesses in the 2024 CWE Top 25 Most Dangerous Software Weaknesses

This MemberOf Relationships table shows additional CWE Categories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources

Vulnerability Mapping Notes

Usage: ALLOWED *(this CWE ID may be used to map to real-world vulnerabilities)*

Reason: Acceptable-Use

Rationale:
This CWE entry is at the Base level of abstraction, which is a preferred level of abstraction for mapping to the root causes of vulnerabilities.

Comments:
Carefully read both the name and description to ensure that this mapping is an appropriate fit. Do not try to 'force' a mapping to a lower-level Base/Variant simply to comply with this preferred level of abstraction.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
ISA/IEC 62443	Part 3-3		Req SR 3.5
ISA/IEC 62443	Part 4-1		Req SI-1
ISA/IEC 62443	Part 4-1		Req SI-2
ISA/IEC 62443	Part 4-1		Req SVV-1
ISA/IEC 62443	Part 4-1		Req SVV-3
ISA/IEC 62443	Part 4-2		Req CR 3.5

SR 3.5 - Input Validation

The control system should validate any input used as a process input or any input that directly impacts the action of the control system. Consider setting up the following:

- Set up input validation on all values that can be externally modified. Note that input is not only process data values. It can also be scripts, database queries, (potentially malformed) packets and other material that via tampering can change the working of the control-system.
- Set up reporting of anomalies to the SIEM system, as they may indicate system tampering and may assist in detecting a security breach.
- Consider encrypting or otherwise securing I/O Server and other external communication to add a security barrier against tampering.
- Consider using the **OWASP Code Review Guide**.

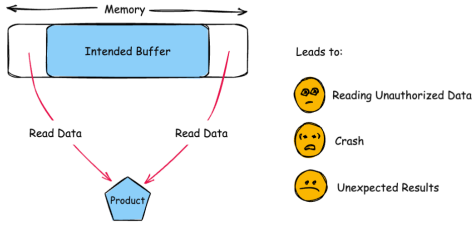
CWE-125: Out-of-bounds Read

Weakness ID: 125
Vulnerability Mapping: ALLOWED
Abstraction: Base

View customized information: Conceptual | Operational | Mapping Friendly | Complete | Custom

Description

The product reads data past the end, or before the beginning, of the intended buffer.



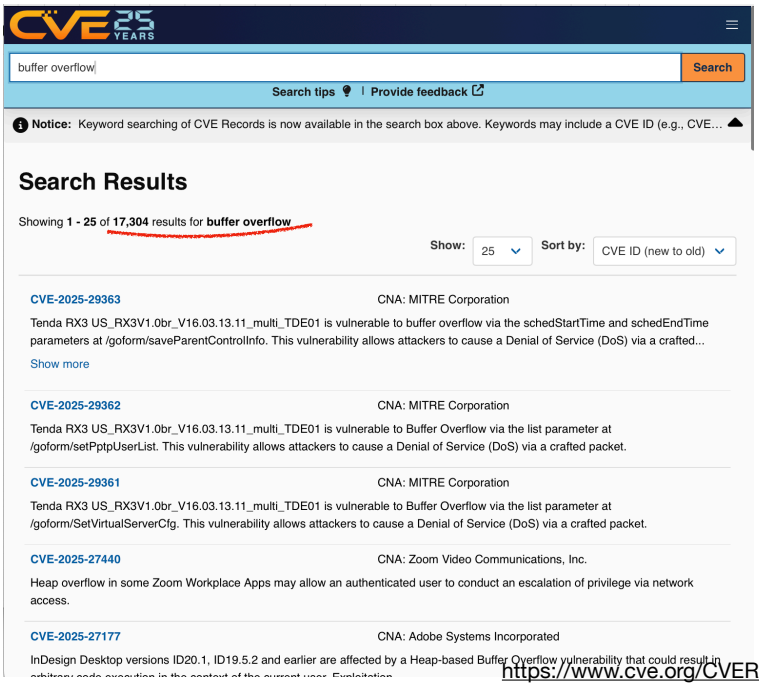
Alternate Terms

Common Consequences

Scope	Impact	Likelihood
Confidentiality	**Technical Impact:** *Read Memory* An attacker could get secret values such as cryptographic keys, PII, memory addresses, or other information that could be used in additional attacks.	
Confidentiality	**Technical Impact:** *Bypass Protection Mechanism* Out-of-bounds memory could contain memory addresses or other information that can be used to bypass ASLR and other protection mechanisms in order to improve the reliability of exploiting a separate weakness for code execution.	
Availability	**Technical Impact:** *DoS: Crash, Exit, or Restart* An attacker could cause a segmentation fault or crash by causing memory to be read outside of the bounds of the buffer. This is especially likely when the code reads a variable amount of data and assumes that a sentinel exists to stop the read operation, such as a NUL in a string.	
Other	**Technical Impact:** *Varies by Context* The read operation could produce other undefined or unexpected results.	

https://cwe.mitre.org/data/definitions/125

...and if we look at known vulnerabilities...



https://www.cve.org/CVERecord/SearchResults?query=buffer+overflow

...or attack strategies...

Search the CAPEC Web Site

Search

To search the CAPEC Web site, enter a keyword by typing in a specific term or multiple keywords separated by a space, and click the Google Search button or press return.

buffer overflow

About 87 results (0.11 seconds)

CAPEC-100: **Overflow Buffers** (Version 3.9) - Mitre
capec.mitre.org › data › definitions
CAPEC-100: **Overflow Buffers** · Find injection vector: The adversary identifies an injection vector to deliver the excessive content to the targeted application's ...

CAPEC-8: **Buffer Overflow** in an API Call (Version 3.9) - Mitre
capec.mitre.org › data › definitions
CAPEC-8: **Buffer Overflow** in an API Call ... This attack targets libraries or shared code modules which are vulnerable to **buffer overflow** attacks. An adversary who ...

CAPEC-47: **Buffer Overflow** via Parameter Expansion (Version 3.9)
capec.mitre.org › data › definitions
This attack relies on the target software failing to anticipate that the expanded data may exceed some internal limit, thereby creating a **buffer overflow**.

CAPEC-45: **Buffer Overflow** via Symbolic Links (Version 3.9) - Mitre
capec.mitre.org › data › definitions
CAPEC-45: **Buffer Overflow** via Symbolic Links ... This type of attack leverages the use of symbolic links to cause **buffer overflows**. An adversary can try to create ...

CAPEC-9: **Buffer Overflow** in Local Command-Line Utilities - Mitre
capec.mitre.org › data › definitions
CAPEC-9: **Buffer Overflow** in Local Command-Line Utilities · Identify target system: The adversary first finds a target system that they want to gain elevated ...

CAPEC-10: **Buffer Overflow** via Environment Variables (Version 3.9)
capec.mitre.org › data › definitions
CAPEC-10: **Buffer Overflow** via Environment Variables ... This attack pattern involves causing a **buffer overflow** through manipulation of environment variables. Once ...

CAPEC-24: Filter Failure through **Buffer Overflow** (Version 3.9)

https://capec.mitre.org/find/index.html

CAPEC-8: Buffer Overflow in an API Call

Attack Pattern ID: 8
Abstraction: Detailed

View customized information: Conceptual | Operational | Mapping-Friendly | **Complete**

Description

This attack targets libraries or shared code modules which are vulnerable to buffer overflow attacks. An adversary who has knowledge of known vulnerable libraries or shared code can easily target software that makes use of these libraries. All clients that make use of the code library thus become vulnerable by association. This has a very broad effect on security across a system, usually affecting more than one software process.

Likelihood Of Attack

High

Typical Severity

High

Relationships

Nature	Type	ID	Name
ChildOf	S	100	Overflow Buffers
PeerOf	D	46	Overflow Variables and Tags
CanFollow	S	69	Target Programs with Elevated Privileges

View Name	Top Level Categories
Domains of Attack	Software
Mechanisms of Attack	Manipulate Data Structures

Execution Flow

Explore

Identify target application: The adversary, with knowledge of vulnerable libraries or shared code modules, identifies a target application or program that makes use of these.

Experiment

1. **Find injection vector:** The adversary attempts to use the API, and if they can they send a large amount of data to see if the buffer overflow attack really does work.

Techniques
Provide large input to a program or application and observe the behavior. If there is a crash, this means that a buffer overflow attack is possible.

2. **Craft overflow content:** The adversary crafts the content to be injected based on their knowledge of the vulnerability and their desired outcome. If the intent is to simply cause the software to crash, the content need only consist of an excessive quantity of random data. If the intent is to leverage the overflow for execution of arbitrary code, the adversary will craft a set of content that not only overflows the targeted buffer but does so in such a way that the overwritten return address is replaced with one of the adversaries' choosing which points to code injected by the adversary.

Techniques
Create malicious shellcode that will execute when the program execution is returned to it.
Use a NOP-sled in the overflow content to more easily "slide" into the malicious code. This is done so that the exact return address need not be correct, only in the range of all of the NOPs

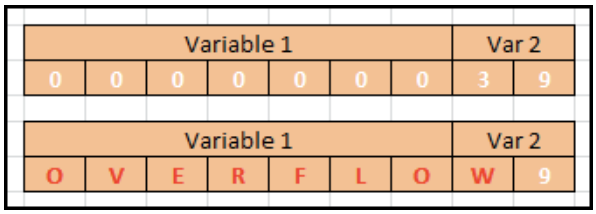
Exploit

Overflow the buffer: Using the API as the injection vector, the adversary injects the crafted overflow content into the buffer.

https://capec.mitre.org/data/definitions/8.html

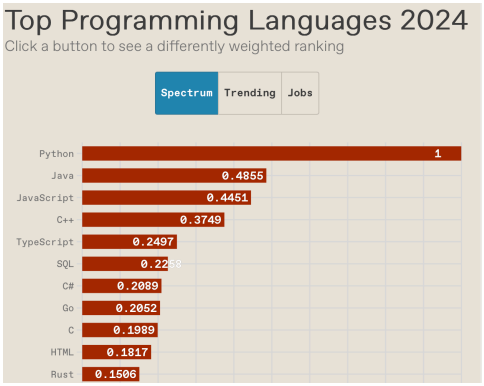
Buffer Overflow

- *Buffer* - temporary memory space intended to store data during program execution;
- *Buffer overflow*: occurs when the data written to a buffer is larger than the buffer size, and due to insufficient checks, it is written to adjacent memory locations, which can cause a program failure or the creation of a vulnerability that attackers can exploit.
- Rewrites/overwrites adjacent memory positions, including other buffers, variables, and program code (can change the program flow).
- Considered the "atomic bomb" of the software industry, buffer overflow is one of the most persistent software vulnerabilities and where attack attempts are most frequent.



Programming Languages & Memory-safety

- C and C++ are particularly vulnerable to buffer overflow and buffer over-read.
  - Still, they are two of the most used and most requested programming languages by employers (for a matter of performance) and in which critical programs are written (OSs, IIS, shell, Apache httpd, MySQL, MS SQL server, industrial control systems, sw automobiles, IoT, ...) ;
- Modern languages (since Java) provide much better memory-safety guarantees.
  - By design, avoid buffer Overflow, check the limits of the Buffer, and prevent access beyond this limit.



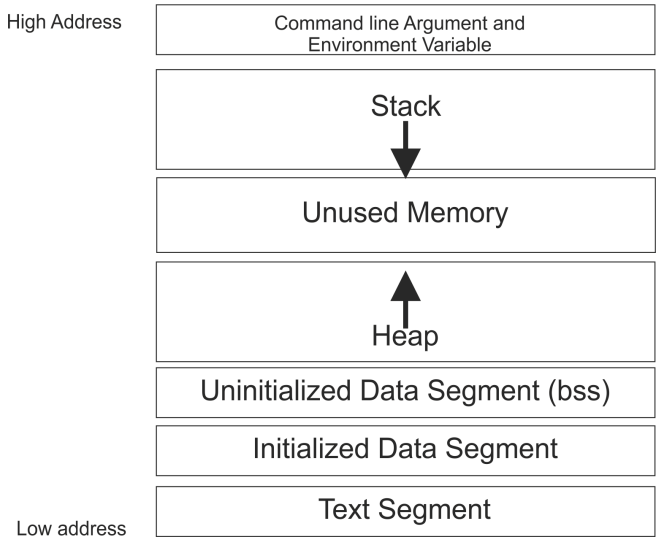
https://spectrum.ieee.org/top-programming-languages-2024

Buffer Overflow / Buffer Over-read

- **Accidental buffer overflow**:
  - Manifestations:
    - The program becomes unstable;
    - The program causes an exception (ends abruptly);
    - The program works normally;
  - The likelihood of “observable” manifestations can be very small.
  - Side effects depend on:
    - The amount of data that is written after the end of the buffer;
    - What data (if any) are superimposed/overwritten;
    - If the program tries to read the overwritten data;
    - What data replaces the memory contents.
  - Debugging it is usually difficult:
    - Effects can manifest themselves much later.

- **Security problems**:
  - It can be exploited intentionally to allow the attacker to produce the effects that best suit him.
  - Buffer Overflow:
    - The goal is usually to execute code with administrator privileges
      - What is "simple" if the server is running with administrator privileges;
      - Or it can be done after exploiting the buffer overflow, through another attack allowing to perform elevation of privileges.
    - Reference: Aleph One, "Smashing the Stack for Fun and Profit", Phrack 49 #14, 1996
  - Buffer Over-read:
    - Typically exploited to perform an unauthorized access of secret data (information flow).

Memory Layout



Text segment - code segment where the executable instructions of the program are located (in machine/assembly code). Usually, this segment is:

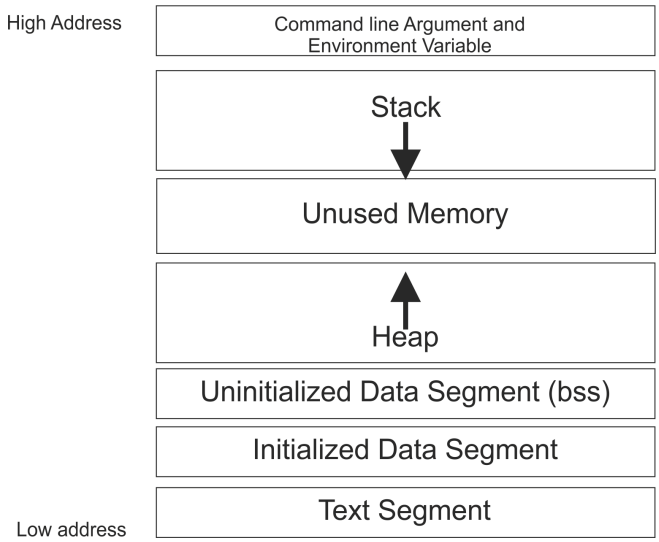
- Shared, so that there is only a copy of the code in memory,
- Read-only, in order to prevent the accidental modification of the instructions;

Initialised **Data Segment** - contains the global and static variables initialized by the programmer (e.g., `char s[] = "hello world"; int i = 10;`);

Uninitialized Data Segment (BSS) - contains the global and static variables not initialized or initialized to 0 (e.g., int i).

Note: All these segments (Text, Data and BSS) are known at the compile time.

Memory Layout



Stack - is a LIFO structure that contains the program stack and typically grows (x86 architecture) from the highest memory addresses to the lowest. It contains the local variables, as well as other information that is saved each time a function is called. The "stack pointer" always points to the top of the stack;

Heap - segment for dynamic memory allocation (malloc, realloc, free). All shared libraries and dynamic modules of a process share this segment;

Buffer overflow on the Heap

```
int main(int argc, char **argv) {
    char *dummy = (char *) malloc (sizeof(char) * 10);
    char *readonly = (char *) malloc (sizeof(char) * 10);

    strcpy(readonly, "laranjas");
    strcpy(dummy, argv[1]);
    printf("%s\n", readonly);
}
```

```
user@CSI:~/Aulas/Aula11.a/codigofonte$ ./a.out BOOMMMM...
laranjas
```

- Is the `readonly` variable out of the control of the user?

Buffer overflow on the Heap

```
int main(int argc, char **argv) {
    char *dummy = (char *) malloc (sizeof(char) * 10);
    char *readonly = (char *) malloc (sizeof(char) * 10);

    printf("Endereço da variavel dummy: %p\n", dummy);
    printf("Endereço da variavel readonly: %p\n", readonly);

    strcpy(readonly, "laranjas");
    strcpy(dummy, argv[1]);
    printf("%s\n", readonly);
}
```

```
user@CSI:~$ ./a.out BOOMMMM...
Endereço da variavel dummy: 0x564dc9cf7010
Endereço da variavel readonly: 0x564dc9cf7030
laranjas
```

Buffer overflow on the Heap

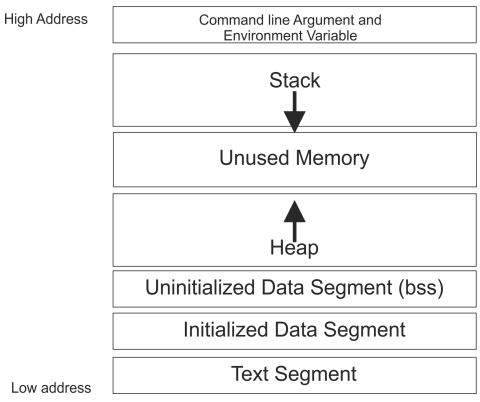
```
int main(int argc, char **argv) {
    char *dummy = (char *) malloc (sizeof(char) * 10);
    char *readonly = (char *) malloc (sizeof(char) * 10);

    printf("Endereço da variavel dummy: %p\n", dummy);
    printf("Endereço da variavel readonly: %p\n", readonly);

    strcpy(readonly, "laranjas");
    strcpy(dummy, argv[1]);
    printf("%s\n", readonly);
}
```

```
user@CSI:~$ ./a.out `for i in {1..32}; do echo -n 1; done`BOOMMMM...
Endereço da variavel dummy: 0x55c3eeba2010
Endereço da variavel readonly: 0x55c3eeba2030
BOOMMMM...
```

(Stack) Buffer overflow



Stack and function-calls (x86/64 bit architecture)

call-site:

1. Push the arguments to the stack
2. Pushes the return address (i.e., the address of the instruction to be executed immediately after returning);
3. Jump to the address of the function;

called function:

4. Pushes the pointer of the old frame (%rbp) (relative to the stack of the source function);
5. The frame pointer now has the value of the stack pointer (%rbp = %rsp);
6. Pushes local variables to the stack;

on returning:

1. Obtains the point to the frame of the source function (content of the address %rbp);
2. Execution passes to the return address, i.e., %rip = 8(%rbp)

Buffer overflow na Stack

```
void debug() {
   printf("Palavras-chave:\n");
   printf("root: ola123\n");
   printf("admin: 3eLdf75\n");
}

void store(char *valor) {
   char buf[10];
   strcpy(buf, valor);
}

int main(int argc, char **argv) {
   store(argv[1]);
}
```

Stack – This LIFO structure contains the program stack and typically (x86) grows from the highest memory addresses to the lowest. It stores local variables, as well as other information that is saved each time a function is called (including the return address). The "stack pointer" always points to the top of the stack.

Can the program user call the function *debug()*?

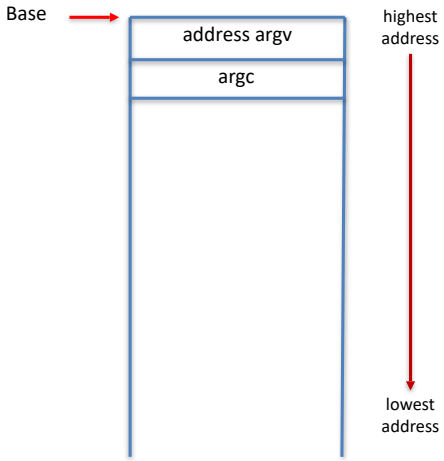
Buffer overflow na Stack

Vamos ver como se comporta a *stack*.

```
void debug() {
   printf("Palavras-chave:\n");
   printf("root: ola123\n");
   printf("admin: 3eLdf75\n");
}

void store(char *valor) {
   char buf[10];
   strcpy(buf, valor);
}

int main(int argc, char **argv) {
   store(argv[1]);
}
```



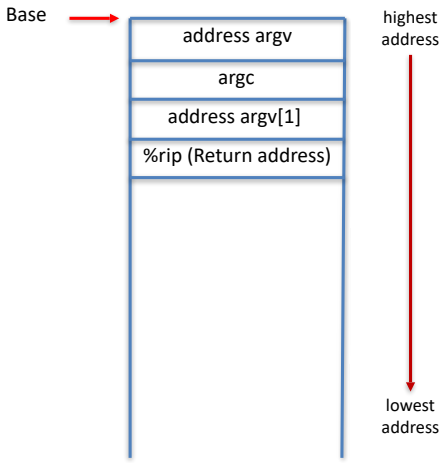
Buffer overflow na Stack

Vamos ver como se comporta a *stack*.

```
void debug() {
   printf("Palavras-chave:\n");
   printf("root: ola123\n");
   printf("admin: 3eLdf75\n");
}

void store(char *valor) {
   char buf[10];
   strcpy(buf, valor);
}

int main(int argc, char **argv) {
   store(argv[1]);
}
```



Note: The "Return Address" is stored in "%rip" (instruction pointer).

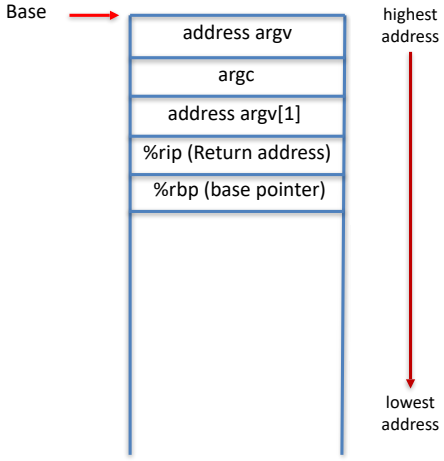
Buffer overflow na Stack

Vamos ver como se comporta a *stack*.

```
void debug() {
   printf("Palavras-chave:\n");
   printf("root: ola123\n");
   printf("admin: 3eLdf75\n");
}

void store(char *valor) {
   char buf[10];
   strcpy(buf, valor);
}

int main(int argc, char **argv) {
   store(argv[1]);
}
```



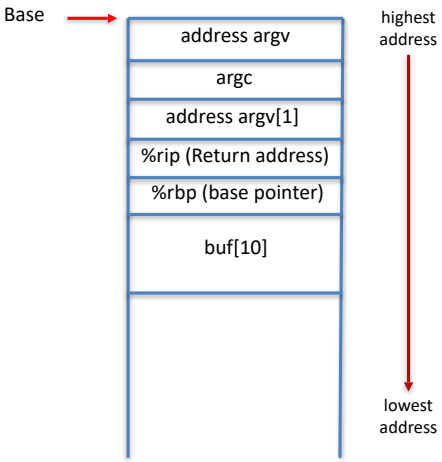
Buffer overflow na Stack

Vamos ver como se comporta a *stack*.

```
void debug() {
   printf("Palavras-chave:\n");
   printf("root: ola123\n");
   printf("admin: 3eLdf75\n");
}

void store(char *valor) {
   char buf[10];
   strcpy(buf, valor);
}

int main(int argc, char **argv) {
   store(argv[1]);
}
```



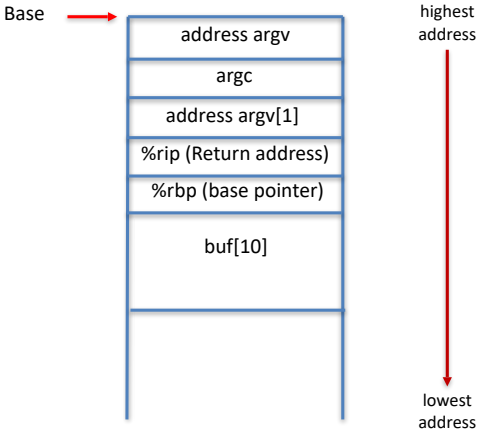
Before the execution of the `strcpy` function, the stack looked like this: What shall be changed so that the user can call the `debug`() function?

Buffer overflow na Stack

```
void debug() {
   printf("Palavras-chave:\n");
   printf("root: ola123\n");
   printf("admin: 3eLdf75\n");
}

void store(char *valor) {
   char buf[10];
   strcpy(buf, valor);
}

int main(int argc, char **argv) {
   store(argv[1]);
}
```



Before the execution of the `strcpy` function, the stack looked like this: What shall be changed so that the user can call the `debug`() function?

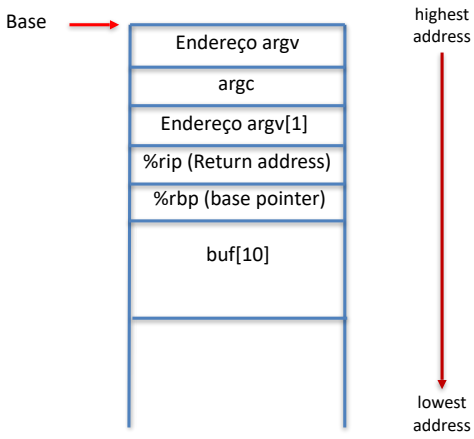
- If the "Return address" points to the `debug`() function, the program execution will be directed to the `debug`() function in the return of the `store`() function;

Buffer overflow na Stack

```
void debug() {
   printf("Palavras-chave:\n");
   printf("root: ola123\n");
   printf("admin: 3eLdf75\n");
}

void store(char *valor) {
   char buf[10];
   strcpy(buf, valor);
}

int main(int argc, char **argv) {
   store(argv[1]);
}
```



Before the execution of the `strcpy` function, the stack looked like this: What shall be changed so that the user can call the `debug`() function?

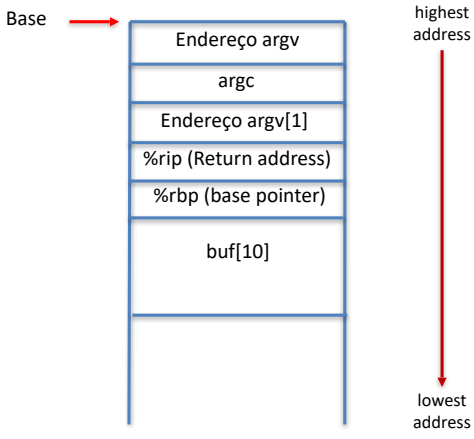
- If the "Return address" points to the `debug`() function, the program execution will be directed to the `debug`() function in the return of the `store`() function;
- To rewrite the return address, we have to put it in `buf` (10 + 8 + 8 bytes), where the last 8 bytes must correspond to the address of the `debug`() function.

Buffer overflow na Stack

```
void debug() {
   printf("Palavras-chave:\n");
   printf("root: ola123\n");
   printf("admin: 3eLdf75\n");
}

void store(char *valor) {
   char buf[10];
   strcpy(buf, valor);
}

int main(int argc, char **argv) {
   store(argv[1]);
}
```



Before the execution of the `strcpy` function, the stack looked like this: What shall be changed so that the user can call the `debug`() function?

- If the "Return address" points to the `debug`() function, the program execution will be directed to the `debug`() function in the return of the `store`() function;
- To rewrite the return address, we have to put it in `buf` (10 + 8 + 8 bytes), where the last 8 bytes must correspond to the address of the `debug`() function.
- One can quickly obtain the address of the `debug`() function (e.g. resorting to a debugger)

Buffer overflow na Stack

```
void debug() {
   printf("Palavras-chave:\n");
   printf("root: ola123\n");
   printf("admin: 3eLdf75\n");
}

void store(char *valor) {
   char buf[10];
   strcpy(buf, valor);
}

int main(int argc, char **argv) {
   printf("Endereco da funcao debug: %p\n", &debug);
   store(argv[1]);
}
```

```
user@CSI:~/Aulas/Aula12$ ./a.out teste
Endereco da funcao debug: 0x555555554740
```

We have all the data needed to run the `debug`() function.

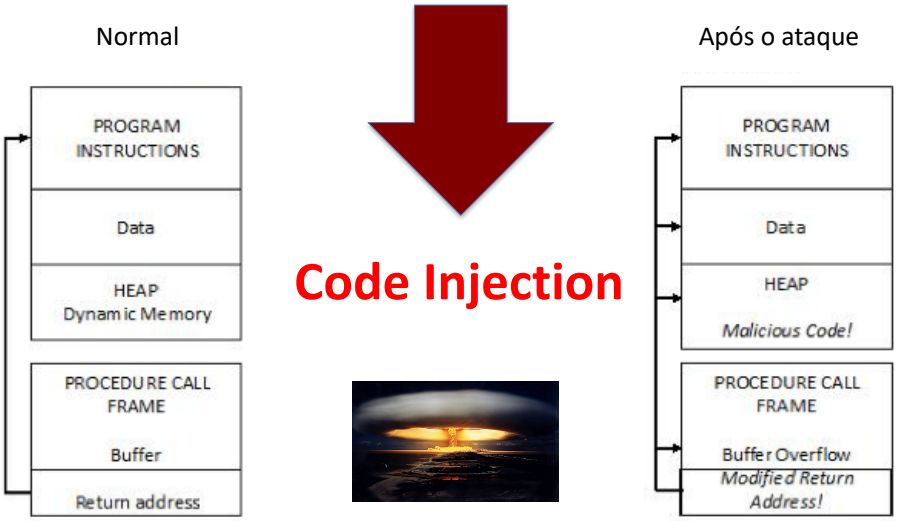
Note: this has been executed in a little-endian architecture, in which the least significant byte is placed in the lowest memory address.

```
user@CSI:~/Aulas/Aula12$ ./a.out `python -c 'print "X"*18 + "\x40\x47\x55\x55\x55\x55"'`
Endereco da funcao debug: 0x555555554740
Palavras-chave:
root: ola123
admin: 3eLdf75
Segmentation fault
```

Buffer Overflow

- An important factor in attacking buffer overflow vulnerabilities is access to the source code.
  - If not available, one can opt for a trial-to-error strategy (it is not very efficient) or apply reverse engineering techniques (decompilation).
- Another crucial factor was the address of the `debug`() function not changing between successive executions of the program.
  - Recent compilers often randomise the addressing space of an application, making it harder for the attacker to cause the execution of the `debug`() function.
  - However, quite improbable does not mean impossible, as you can see in the book "*Hacking – The art of exploitation (2nd edition), Jon Erickson*" (But outside the scope of this course).
- In the shown example, the consequences of the attack amount to the execution of an existing function. It is possible, however, to execute self-defined code (e.g. opening a shell) or install a remote administration tool — also called *code injection*.

(Stack/Heap) Buffer overflow



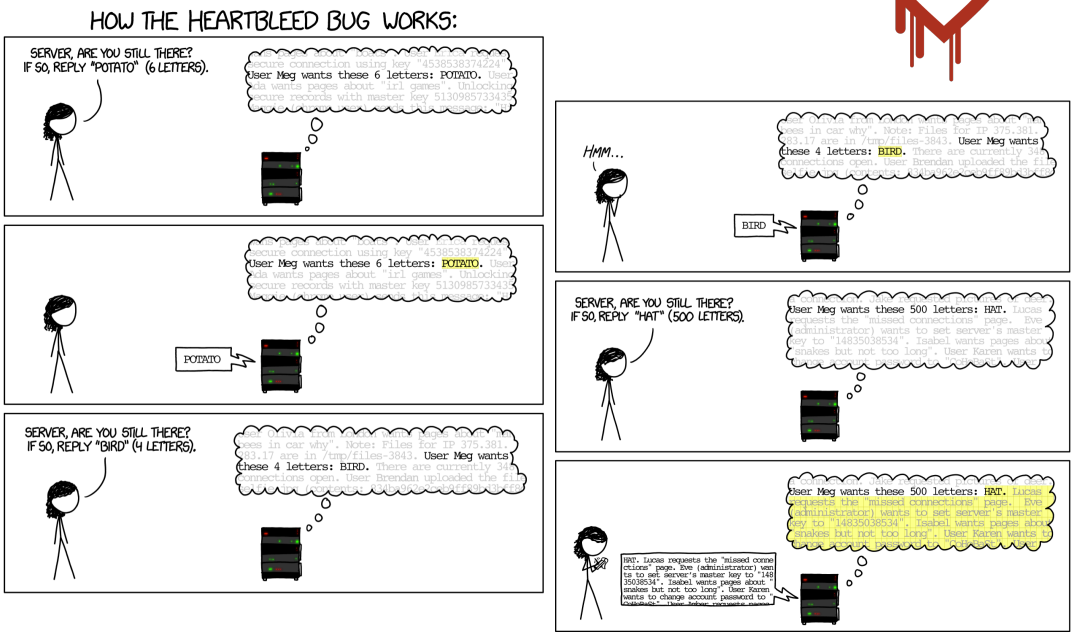
The attacker, through a (stack/heap) buffer overflow, corrupts the return address. Instead of returning to the calling function, the return address returns the control to the malicious code, located somewhere in the process memory.

Read Overflow

- Example: Heartbleed bug (http://heartbleed.com/)
  - SSL/TLS is the protocol for secure communication on the Web
  - **Heartbleed** is a bug that existed in *OpenSSL*– versions 1.0.1 to 1.0.1f
  - The bug was discovered in March 2014 but has been available since March 2012!!!
  - It exploits an obscure "heartbeat" feature in which the server listens to a message and echoes back;
  - The “heartbeat” message specifies the size of the echoed message, but the *OpenSSL* implementation didn’t validate it;
  - In this way, the attacker could request a larger size and read beyond the content of the message, which allowed access to server memory (including confidential data, such as passwords, keys, ID information, …).
  - The attack leaves no trace!!!

Read Overflow

- Example: Heartbleed bug (https://xkcd.com/1354/



Mitigations of Buffer overflow vulnerabilities

- Defensive programming:
  - Validate indexes: Checking that the index values are integers and are within the array addressing limits. This validation is mandatory for values provided by the user or by another unreliable input source (e.g., information read from a file or obtained through a network connection).
    - Be aware of loops (for, while, ...) !
    - Speculative execution adds to the problem!!!
  - Allocated space: Before copying the data, make sure that the destination variable has enough space to store this data. If you do not have enough space, do not copy more data than the available space.
  - Array size: Programming languages have functions that return the size allocated to an array. Use them!
    - Se utilizar um array como argumento de uma função, utilize outro argumento para enviar também o tamanho do array. Esse valor pode ser utilizado como limite máximo do índice do array.
  - Alternative data structures: Buffer overflow vulnerabilities can be reduced if you use alternative data structures, such as vectors and iterators. Use them!
  - Allocate memory: Whenever possible, allocate memory only after knowing how much you need.
  - Avoid risk functions: When using functions to read, copy data or allocate/free memory, use libraries that provide secure versions of the outdated C standard library functions;
  - Use the tools: enable compiler warnings for potential buffer overflows. Use static analysis tools to analyse the source code or rely on dynamic checks to exclude problematic memory accesses.
  - Recovery: If the program cannot continue, an adequate recovery must be guaranteed. Note: handle exceptions with care!

Integer Overflow

Integer Overflow Vulnerability

CWE Common Weakness Enumeration
A community-developed list of SW & HW weaknesses that can become vulnerabilities

New to CWE? Start here!

Home | About | Learn | Access Content | Community | Search

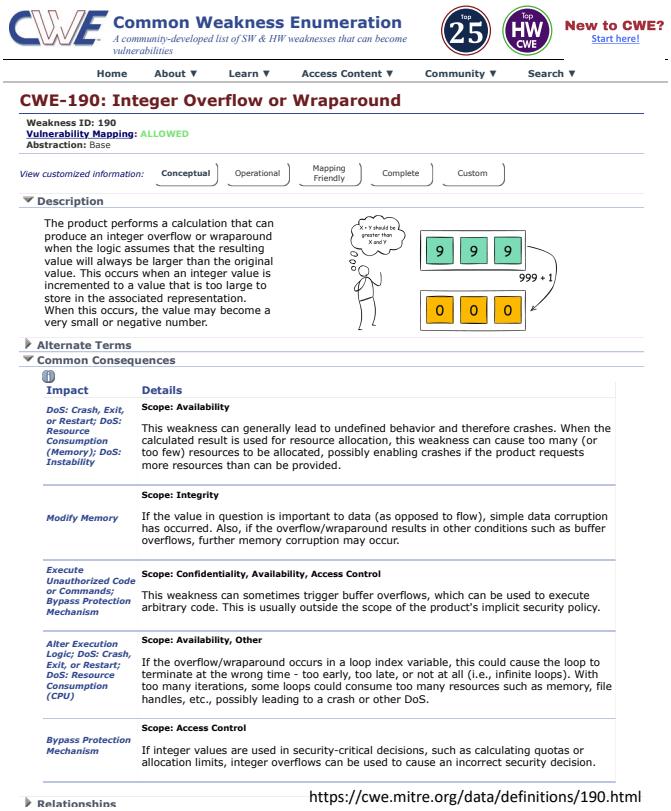
2024 CWE Top 25 Most Dangerous Software Weaknesses

Top 25 Home | Share via: | View in table format | Key Insights | Methodology

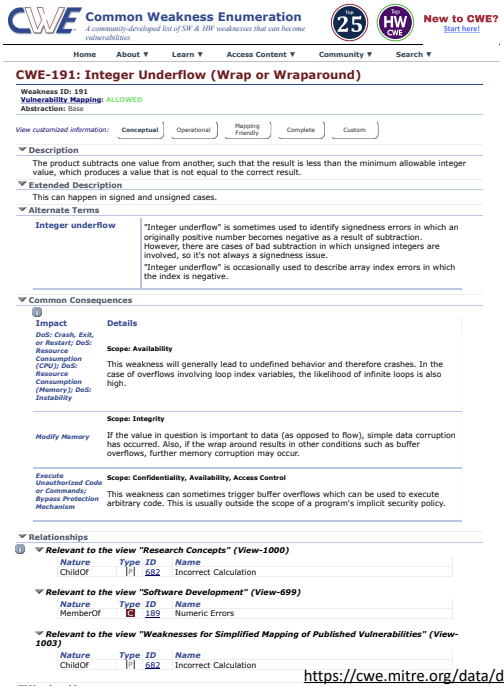
1	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	CWE-79 | CVEs in KEV: 3 | Rank Last Year: 2 (up 1) ▲
2	Out-of-bounds Write	CWE-787 | CVEs in KEV: 18 | Rank Last Year: 1 (down 1) ▼
3	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	CWE-89 | CVEs in KEV: 4 | Rank Last Year: 3
4	Cross-Site Request Forgery (CSRF)	CWE-352 | CVEs in KEV: 0 | Rank Last Year: 9 (up 5) ▲
5	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	CWE-22 | CVEs in KEV: 4 | Rank Last Year: 8 (up 3) ▲
6	Out-of-bounds Read	CWE-125 | CVEs in KEV: 3 | Rank Last Year: 7 (up 1) ▲
7	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	CWE-78 | CVEs in KEV: 5 | Rank Last Year: 5 (down 2) ▼
8	Use After Free	CWE-416 | CVEs in KEV: 5 | Rank Last Year: 4 (down 4) ▼
9	Missing Authorization	CWE-862 | CVEs in KEV: 0 | Rank Last Year: 11 (up 2) ▲
10	Unrestricted Upload of File with Dangerous Type	CWE-434 | CVEs in KEV: 0 | Rank Last Year: 10
11	Improper Control of Generation of Code ('Code Injection')	CWE-94 | CVEs in KEV: 7 | Rank Last Year: 23 (up 12) ▲
12	Improper Input Validation	CWE-20 | CVEs in KEV: 1 | Rank Last Year: 6 (down 6) ▼
13	Improper Neutralization of Special Elements used in a Command ('Command Injection')	CWE-77 | CVEs in KEV: 4 | Rank Last Year: 16 (up 3) ▲
14	Improper Authentication	CWE-287 | CVEs in KEV: 4 | Rank Last Year: 13 (down 1) ▼
15	Improper Privilege Management	CWE-269 | CVEs in KEV: 0 | Rank Last Year: 22 (up 7) ▲
16	Deserialization of Untrusted Data	CWE-502 | CVEs in KEV: 5 | Rank Last Year: 15 (down 1) ▼
17	Exposure of Sensitive Information to an Unauthorized Actor	CWE-200 | CVEs in KEV: 0 | Rank Last Year: 30 (up 13) ▲
18	Incorrect Authorization	CWE-863 | CVEs in KEV: 2 | Rank Last Year: 24 (up 6) ▲
19	Server-Side Request Forgery (SSRF)	CWE-918 | CVEs in KEV: 2 | Rank Last Year: 19
20	Improper Restriction of Operations within the Bounds of a Memory Buffer	CWE-119 | CVEs in KEV: 2 | Rank Last Year: 17 (down 3) ▼
21	NULL Pointer Dereference	CWE-476 | CVEs in KEV: 0 | Rank Last Year: 12 (down 9) ▼
22	Use of Hard-coded Credentials	CWE-798 | CVEs in KEV: 2 | Rank Last Year: 18 (down 4) ▼
23	Integer Overflow or Wraparound	CWE-190 | CVEs in KEV: 3 | Rank Last Year: 14 (down 9) ▼
24	Uncontrolled Resource Consumption	CWE-400 | CVEs in KEV: 0 | Rank Last Year: 37 (up 13) ▲
25	Missing Authentication for Critical Function	CWE-306 | CVEs in KEV: 5 | Rank Last Year: 20 (down 5) ▼

https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html

Integer Overflow/underflow



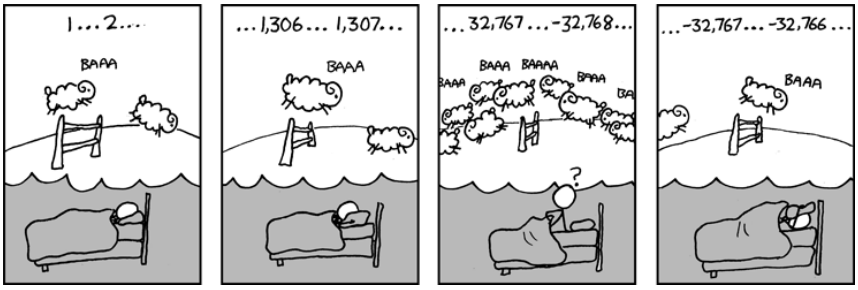
https://cwe.mitre.org/data/definitions/190.html



https://cwe.mitre.org/data/definitions/191.html

Integer overflow

- Very large or very small values of integers can fall out of the range of the data type, leading to an indefinite behaviour that can reduce the robustness of the code, as well as give rise to security vulnerabilities.
- For example, a 32-bit **int** may contain values between -2^{31} and $2^{31}-1$.
- An Integers error can lead to unexpected behaviour or can be exploited to cause a program to crash, corrupt data, lead to incorrect behaviour or allow the execution of malicious software.



Integer overflow/underflow Vulnerabilities

Notice: Keyword searching of CVE Records is now available in the search box above…

Search Results

Showing **1 - 25** of **3,702** results for **integer overflow**

Show: 25 **Sort by:** CVE ID (new to old)

CVE-2025-3408 CNA: VulDB

A vulnerability was found in Nothings stb up to f056911. It has been rated as critical. Affected by this issue is the function stb_dupreplace. The manipulation leads to integer overflow....

Show more

CVE-2025-3407 CNA: VulDB

A vulnerability was found in Nothings stb up to f056911. It has been declared as critical. Affected by this vulnerability is the function stbhw_build_tileset_from_image. The manipulation of the argument h_count/v_count...

Show more

CVE-2025-3360 CNA: Red Hat, Inc.

A flaw was found in GLib. An integer overflow and buffer under-read occur when parsing a long invalid ISO 8601 timestamp with the g_date_time_new_from_iso8601() function.

CVE-2025-32364 CNA: MITRE Corporation

A floating-point exception in the PSStack::roll function of Poppler before 25.04.0 can cause an application to crash when handling malformed inputs associated with INT_MIN.

Required CVE Record Information

CNA: GitHub (maintainer security advisories)

Published: 2025-03-17 **Updated:** 2025-03-17
Title: CryptoLib's Crypto_TC_Prep_AAD Has Buffer Overflow Due To Integer Underflow

Description

CryptoLib provides a software-only solution using the CCSDS Space Data Link Security Protocol - Extended Procedures (SDLS-EP) to secure communications between a spacecraft running the core Flight System (cFS) and a ground station. A critical heap buffer overflow vulnerability was identified in the `Crypto_TC_Prep_AAD` function of CryptoLib versions 1.3.3 and prior. This vulnerability allows an attacker to trigger a Denial of Service (DoS) or potentially execute arbitrary code (RCE) by providing a maliciously crafted telecommand (TC) frame that causes an unsigned integer underflow. The vulnerability lies in the function `Crypto_TC_Prep_AAD`, specifically during the computation of `tc_mac_start_index`. The affected code incorrectly calculates the MAC start index without ensuring it remains within the bounds of the `ingest` buffer. When `tc_mac_start_index` underflows due to an incorrect length calculation, the function attempts to access an out-of-bounds memory location, leading to a segmentation fault. The vulnerability is still present in the repository as of commit `d3cc420ace96d02a5b7e83d88cbd2e48010d5723`.

CWE 2 Total
Learn more

- CWE-125: CWE-125: Out-of-bounds Read
- CWE-191: CWE-191: Integer Underflow (Wrap or Wraparound)

CVSS 1 Total
Learn more

Score	Severity	Version	Vector String
8.9	HIGH	4.0	CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:H/VA:H/SC:N/SI:N/SA:N/E:P

Product Status
Learn more

Integer overflow

- Most Unix and embedded systems (automotive, military, medical, aviation, ...) store the date/time in a 32-bit int variable - the date/time is saved as the number of seconds since 00:00 on 1/Jan/1970. On 19/Jan/2038, the overflow of this variable occurs, making the date/time negative. More information at http://en.wikipedia.org/wiki/Year_2038_problem.

Binary : 01111111 11111111 11111111 11110000
Decimal : 2147483632
Date : 2038-01-19 03:13:52 (UTC)
Date : 2038-01-19 03:13:52 (UTC)

- On Facebook there is a group that affirms the Following:
  - Why do they say that?
  - What would be the potential problems if this occurred?

Integer overflow

- YouTube couldn't stand Gangnam Style

YouTube não aguentou o *Gangnam Style*

PÚBLICO 03/12/2014 - 16:38

É o próprio site a confirmar que teve problemas. "*Gangnam Style* foi visto tantas vezes que tivemos que fazer um *upgrade*."

É o próprio *site* a confirmar que teve problemas. "*Gangnam Style* foi visto tantas vezes que tivemos que fazer um *upgrade*", escreve o YouTube na sua página no Google+.

Se passarmos o cursor sobre o contador que se pode ver na página do vídeo este não pára de rodar e isso porque o YouTube nunca pensou que um tal número pudesse vir a ser atingido. "Nunca pensámos que um vídeo pudesse ser visto em números mais do que um número inteiro de 32-bit (=2.147.483.647 visualizações), mas isso foi antes de termos conhecido Psy", admite o YouTube.

Integer overflow

- On December 25, 2004, the airline Comair Airlines was forced to maintain 1,100 flights on the ground after the crew scheduling software collapsed. The software used a 16-bit integer (maximum 32,767) to number crew changes for a month, and this number was exceeded that month due to bad weather that led to numerous crew changes.

FEATURE

Comair's Christmas Disaster: Bound To Fail

The 2004 crash of a critical legacy system at Comair is a classic risk management mistake that cost the airline $20 million and badly damaged its reputation.

By Stephanie Overby

CIO | MAY 1, 2005 8:00 AM PT

ars TECHNICA BIZ & IT TECH SCIENCE POLICY CARS GAMING & CULT

UNCATEGORIZED —

Comair/Delta airline debacle caused by the overflow of 16-bit pointer

One of the most nightmarish Christmas travel foul-ups in recent memory was ...

CLINT ECKER - 12/30/2004, 7:24 PM

Integer overflow – truncation problem

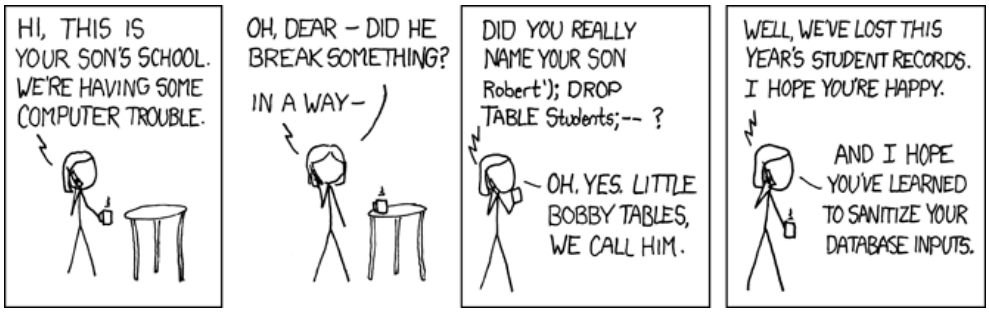
- On June 4, 1996, the Ariane 5 unmanned rocket exploded 40 seconds after launch. The rocket made its first trip after a decade of development, with costs in the order of $7 billion, with the rocket destroyed and its cargo valued at $500 million.
- The cause was a software error in the inertia reference system. More specifically, a 64-bit float number related to the horizontal speed of the rocket was converted into a 16-bit integer (signed int). The number to convert was greater than 32,767 (the largest integer that can be saved in a signed int), so the conversion failed.

https://www.youtube.com/embed/A1gGGDG580E

Integer overflow

- Risk
  - Declaring a variable with a certain type allocates a fixed memory space. Most languages allow you to declare several types of integers (short, int, long, etc.). For example, a 32-bit int can save values between -231 (-2 147 483 648) and 231-1 (2 147 483 647).
  - Often, the data types' size depends on the machine and compiler...
- "Responsible" coding
  - Know the limits: As the size of the data type is dependent on the machine and compiler, it is a good idea to familiarise yourself with the limits on the machine where the program will run;
  - Datatypes: Choose the most appropriate datatype to hold values handled by your program;
  - Validate the input: (more details later…);
  - Check for possible overflows/underflows on integer operations;
  - Compiler options and tools: activate relevant warnings and take advantage of static analysis tools;
  - Use safe libraries: e.g., class `SafeInt` in C++.

Input Validation



Common Weakness Enumeration
A community-developed list of SW & HW weaknesses that can become vulnerabilities

New to CWE? Start here!

Home | About | Learn | Access Content | Community | Search

2024 CWE Top 25 Most Dangerous Software Weaknesses

Top 25 Home | Share via: | View in table format | Key Insights | Methodology

1. Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting') — CWE-79 | CVEs in KEV: 3 | Rank Last Year: 2 (up 1)
2. Out-of-bounds Write — CWE-787 | CVEs in KEV: 18 | Rank Last Year: 1 (down 1)
3. Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') — CWE-89 | CVEs in KEV: 4 | Rank Last Year: 3
4. Cross-Site Request Forgery (CSRF) — CWE-352 | CVEs in KEV: 0 | Rank Last Year: 9 (up 5)
5. Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal') — CWE-22 | CVEs in KEV: 4 | Rank Last Year: 8 (up 3)
6. Out-of-bounds Read — CWE-125 | CVEs in KEV: 3 | Rank Last Year: 7 (up 1)
7. Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection') — CWE-78 | CVEs in KEV: 5 | Rank Last Year: 5 (down 2)
8. Use After Free — CWE-416 | CVEs in KEV: 5 | Rank Last Year: 4 (down 4)
9. Missing Authorization — CWE-862 | CVEs in KEV: 0 | Rank Last Year: 11 (up 2)
10. Unrestricted Upload of File with Dangerous Type — CWE-434 | CVEs in KEV: 0 | Rank Last Year: 10
11. Improper Control of Generation of Code ('Code Injection') — CWE-94 | CVEs in KEV: 7 | Rank Last Year: 23 (up 12)
12. Improper Input Validation — CWE-20 | CVEs in KEV: 1 | Rank Last Year: 6 (down 6)
13. Improper Neutralization of Special Elements used in a Command ('Command Injection') — CWE-77 | CVEs in KEV: 4 | Rank Last Year: 16 (up 3)
14. Improper Authentication — CWE-287 | CVEs in KEV: 4 | Rank Last Year: 13 (down 1)
15. Improper Privilege Management — CWE-269 | CVEs in KEV: 0 | Rank Last Year: 22 (up 7)
16. Deserialization of Untrusted Data — CWE-502 | CVEs in KEV: 5 | Rank Last Year: 15 (down 1)
17. Exposure of Sensitive Information to an Unauthorized Actor — CWE-200 | CVEs in KEV: 0 | Rank Last Year: 30 (up 13)
18. Incorrect Authorization — CWE-863 | CVEs in KEV: 2 | Rank Last Year: 24 (up 6)
19. Server-Side Request Forgery (SSRF) — CWE-918 | CVEs in KEV: 2 | Rank Last Year: 19
20. Improper Restriction of Operations within the Bounds of a Memory Buffer — CWE-119 | CVEs in KEV: 2 | Rank Last Year: 17 (down 3)
21. NULL Pointer Dereference — CWE-476 | CVEs in KEV: 0 | Rank Last Year: 12 (down 9)
22. Use of Hard-coded Credentials — CWE-798 | CVEs in KEV: 2 | Rank Last Year: 18 (down 4)
23. Integer Overflow or Wraparound — CWE-190 | CVEs in KEV: 3 | Rank Last Year: 14 (down 9)
24. Uncontrolled Resource Consumption — CWE-400 | CVEs in KEV: 0 | Rank Last Year: 37 (up 13)
25. Missing Authentication for Critical Function — CWE-306 | CVEs in KEV: 5 | Rank Last Year: 20 (down 5)

Input Validation Vulnerabibilites

https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html

Input Validation

- All input for a program (keyboard, network, files, external devices, environment variables, web services, ...) can be the source of bugs and security vulnerabilities;
- Any program that **processes input data without proper validation** is susceptible to security vulnerabilities;
- An attacker can pass **malformed arguments** to a program;
- All input should be treated as potentially dangerous;
- These issues are particularly relevant in programs with *setuid* root permissions (i.e., users run the program as if they were root) or that run in privileged mode.

Input Validation

CVE ID	Vulnerability type	Publish Date	CVSS Score	Description
CVE-2020-3161	Improper Input Validation	04/15/2020	9.8	A vulnerability in the **web server for Cisco IP Phones** could allow an unauthenticated, remote attacker to execute code with root privileges or cause a reload of an affected IP phone, resulting in a denial of service (DoS) condition. The vulnerability is due to a lack of proper input validation of HTTP requests.
CVE-2020-8147	Improper Input Validation	04/03/2020	9.8	Flaw in input validation in **npm package utils-extend** version 1.0.8 and earlier may allow prototype pollution attack that may result in remote code execution or denial of service of applications using utils-extend.
CVE-2020-7947	Improper Neutralization of Special Elements	04/01/2020	9.8	An issue was discovered in the Login by **Auth0 plugin** before 4.0.0 for **WordPress**. It has numerous fields that can contain data that is pulled from different sources. One issue with this is that the data isn't sanitized, and no input validation is performed, before the exporting of the user data. This can lead to (at least) CSV injection if a crafted Excel document is uploaded.
CVE-2020-8132	Improper Input Validation	02/28/2020	9.8	Lack of input validation in **pdf-image npm package** version <= 2.0.0 may allow an attacker to run arbitrary code if PDF file path is constructed based on untrusted user input.
CVE-2019-0370	XML Injection	10/08/2019	6.5	Due to missing input validation, **SAP Financial Consolidation**, before versions 10.0 and 10.1, enables an attacker to use crafted input to interfere with the structure of the surrounding query leading to XPath Injection.
CVE-2019-9117	OS Command Injections	03/07/2019	9.8	An issue was discovered on **Motorola** C1 and M2 devices with firmware 1.01 and 1.07 respectively. This issue is a Command Injection allowing a remote attacker to execute arbitrary code, and get a root shell. A command Injection vulnerability allows attackers to execute arbitrary OS commands via a crafted /HNAP1 POST request
CVE-2019-6318	Input Validation	04/11/2019	9.8	**HP LaserJet** Enterprise printers, HP PageWide Enterprise printers, HP LaserJet Managed printers, HP Officejet Enterprise printers have an insufficient solution bundle signature validation that potentially allows execution of arbitrary code.

Input Validation

- The **attack surface** of an application consists of the set of interfaces through which inputs from outside are passed:
  - Remote communication mechanisms (e.g., sockets, web services, ...);
  - Communication mechanisms between processes (e.g., signals, semaphores, ...);
  - Application Programmatic Interface (API);
  - Files used by the application;
  - User Interface (e.g., application arguments, user input, ...);
  - Operating system (e.g., environment variables, ...).

- The golden rule in building safe software is **never to trust the input**.

Input Validation from the parent process

- In Unix family operating systems, applications are typically launched from a shell and executed as a child process of that shell.
- An attacker with access to this shell, can launch the application, providing inputs that exploits some vulnerability:
  - Arguments with undue size (may cause buffer overflow);
  - Signal handling routines (callbacks) with malicious code;
  - Set the default permissions for files created by the application (through the `umask` command);
  - Environment variables with erroneous values.
    - `PATH` stores where the executable programs may be found (when the execution of a program is requested without specifying its absolute path, the operating system goes through the directories in the order in which they appear in this variable, until it finds the program and executes it).

Input Validation from the parent process

Example:

- The standard library functions `system(command)` and `popen(command, type)` execute the `program`/`command` passed as an argument, with the environment variables of the parent process → **Avoid both!**
  - Suppose a program includes the `system("ls")` instruction to list the contents of the current directory (recall that `ls` is in the `/bin` directory)
  - What would happen if an attacker:
    - Changes the `PATH` to "`.:/usr/local/bin:/usr/bin:/bin`", and
    - Performs the bash command "`cp ./malicious_program ./ls`" ?
  - And what would happen if the program had `setuid` root permissions?

Input Validation from the parent process

- What other environment variables are used by your program or the libraries/APIs you use?
- If the libraries/API you use do not adequately control the environment variables:
  - It is up to you to carry out this control, or
  - Set the variable yourself.

Input Validation: Metacharacters

- Metacharacters are a source of special concern, since they are responsible for many vulnerabilities, typically in applications that deal with strings.
- The solution for this type of vulnerability is simple: validate the inputs received, controlling the accepted (meta)characters.
  - Opt for **whitelisting** techniques, where a list includes all the valid characters the application accepts.
  - Do not opt for **black listing**, i.e., for the list of characters the application should not accept. Why?
    - For example, if UTF-8 encoding is used, the character '.' (dot) can also be written as '%2e' and '%c0%ae', among others.

Input Validation: Metacharacters

- The most common attacks based on Metacharacters:

1. **Embedded delimiters**
   - When the input for the application includes different types of information separated by delimiters.
   - Suppose an application that stores usernames with their passwords in a file in which each line has the format `user:password\n`
   - What can happen if Alice, when changing the password, chooses `potato\nhacker:ola123`?

Input Validation: Metacharacters

- The most common attacks based on Metacharacters:
  1. Embedded delimiters
  2. **Injection of the character `\0`**
     - Dangerous because it is not always interpreted as a string terminator (for example, in Perl and Java, unlike C/C++)
     - Consider a Web application built in C that allows you to open text files terminated with the .txt extension.
     - …what happens if Alice provides the string `/etc/passwd\0.txt` as a file name?

Input Validation: Metacharacters

- The most common attacks based on Metacharacters:
  1. Embedded delimiters
  2. Injection of the character `\0`
  3. **Injection of separators**
     - Injection of command separators, which can allow the execution of arbitrary commands. In Unix, using the metacharacter '`;`'.
     - Injecting folder separators, usually called **path traversal attack**, can allow the reading and/or writing of arbitrary files.
       - Consider a Web application that gives a user prints statistics using `system("cat", "/var/stats/$username");`
       - How can Alice use this vulnerability to print any system file (e.g. `/etc/passwd`)?

Input Validation: Format String Vulnerability

- Class of vulnerabilities in which:
  - The lack of input validation allows an attacker to control the execution of an application;
  - The validation of entries necessary to avoid vulnerability is straightforward.
- Most prevalent and dangerous in C and C++, although other languages (e.g., Java, Perl, PHP, OCaml, Python and Ruby) also allow format strings with related vulnerabilities.

Input Validation: Format String Vulnerability

- Simple (and classic) example of the format string vulnerability:

```c
#include <stdio.h>

int main(int argc, char **argv) {
        char buf[1024];

        if(argc > 1) {
                strncpy(buf, argv[1], 1023);
                buf[1023] = '\0';
                printf(buf);
        }
}
```

- The `printf` function's first argument is under the user's control and can specify the format of different data types(e.g., `%d` indicates an integer variable, `%s` a string, ...).

```
$ ./a.out  "string - %s | apontador -  %p | inteiro -  %d"
string - (null) | apontador -  0xffffffffffffffff | inteiro -  19
```

Input Validation: Format String Vulnerability

- Simple (and classic) example of the format string vulnerability:

```c
#include <stdio.h>

int main(int argc, char **argv) {
        char buf[1024];

        if(argc > 1) {
                strncpy(buf, argv[1], 1023);
                buf[1023] = '\0';
                printf(buf);
        }
}
```

- What is the biggest problem with this vulnerability?
  - Allows you to read/write stack values through the appropriate use of string formatters.
  - E.g.: (the value `0x41` corresponds to the letter '`A`').

```
$ ./a.out AAAAAAAA%p%p%p%p%p%p%p%p%p
AAAAAAAA0x1d0x7ffeea8db9400x1d0x00xffffffff000000000x00x7ffeea8db3700x7ffeea8db7a80x20x4141414141414141
```

Input Validation: Format String Vulnerability

- Simple (and classic) example of the format string vulnerability:

```c
#include <stdio.h>

int main(int argc, char **argv) {
        char buf[1024];

        if(argc > 1) {
                strncpy(buf, argv[1], 1023);
                buf[1023] = '\0';
                printf(buf);
        }
}
```

- Whenever an attacker can control the format string, we face a vulnerability.
  - Occurs in all function families with format strings as an argument (e.g., `printf`, `err`, `syslog`).

Input Validation: Format String Vulnerability

- Simple (and classic) example of the format string vulnerability:

```
1   #include <stdio.h>
2
3   int main(int argc, char **argv) {
4           char buf[1024];
5
6           if(argc > 1) {
7                   strncpy(buf, argv[1], 1023);
8                   buf[1023] = '\0';
9                   printf(buf);
10          }
11  }
```

- How do you solve this vulnerability?
  - Replacing on line 9 `printf(buf)` with `printf("%s", buf)`.

```
$ ./a.out  "string - %s | apontador -  %p | inteiro -  %d"
string - %s | apontador -  %p | inteiro -  %d
```

Input Validation

- Risk
  - **If input data is not validated** to ensure that it contains the correct type, size and structure, **problems can (and likely will) happen**;
  - Input validation errors **can lead to buffer overflows** if the data is used as array indexes, or used in a **SQL injection**, allowing access/change/delete private data in a Database;
  - **Attackers can use carefully chosen inputs** to cause arbitrary code execution, access/erase data, propagate worms, obtain confidential information, etc.

Input Validation

- "Responsible" validation of all input
  - Type: Validate that the input has the expected data type, for example, the age is an integer. Many programs handle input data assuming that it is a string, then checking that this string contains the appropriate characters, and converting it to the desired data type;
  - Size: Validate that the input has the expected size (for example, the phone number has 9 digits);
  - Range: Validate that the input is within the expected range (for example, the month is between 1 and 12);
  - Reasonability: Validate that the input has a reasonable value (for example, the name does not contain punctuation or non-alphanumeric characters);
  - Division by Zero: Validate the input so as not to accept values that may cause problems later in the program, such as division by zero;
  - Format: Ensure that the input data is in the appropriate format (for example, the date is in the format DD/MM/YYYY);
  - Mandatory data: Ensure that the user enters the mandatory data;
  - *Checksums*: Many identification numbers have check digits (additional digits inserted at the end of the number for validation). See the check digit of Citizen Card, Passport, credit cards, Luhn's algorithm.

Input Validation

- Use of programming language tools
  - Languages such as C and C++ read (by default) the input into a character buffer, without validating the buffer limit, causing buffer overflow and input validation problems. However, there are specific reading libraries available, more robust, designed with safety in mind;
  - Typed languages, such as Java and C++, require that the type of data stored in a variable be known a priori (which leads to type incompatibilities when, for example, a string is inserted in response to an integer request);
  - Untyped languages, such as PHP or Python, do not have these requirements - any variable can store any value. This fact does not eliminate validation problems (test the input of a string to be used as an index of an array);
- Appropriate recovery
  - A robust program should treat an invalid input appropriately, correctly and safely, repeating the input request or continuing with predefined values (trunking or reformatting data to adjust it to the intended format should be avoided).