EDITOR: Irena Bojanova, irena.bojanova@computer.org

DEPARTMENT: CYBERSECURITY

Critical Software Security Weaknesses

Assane Gueye, *Carnegie Mellon University Africa, Rwanda*
Carlos Eduardo C. Galhardo, *INMETRO, Duque de Caxias, RJ, 25250-020, Brazil*
Irena Bojanova, *NIST, Gaithersburg, MD, 20899, USA*

Abstract—In this work, we extend our historical study on the most significant software security weaknesses, re-evaluate our findings, and look closely at the Injection and Memory Corruption/Disclosure weaknesses through the NIST Bugs Framework (BF) lenses. Our goal is to continue raising awareness about the patterns of reoccurring software security vulnerabilities that enable malicious activity.
Keywords: Bug, Fault, Exploitable Error, Weakness, Vulnerability, Security Failure.

Raising awareness about the most critical software security weaknesses would urge programmers, software developers, and security experts concentrate their efforts on preventing them. Ultimately, this would reduce the number and severity of new vulnerabilities discovered over time.

A *software security vulnerability*, as defined by the National Institute of Standards and Technology (NIST) Bugs Framework (BF) [1], is "a chain of weaknesses linked by causality[; it] starts with a bug and ends with a final error, which, if exploited leads to a security failure" [2].

In the year 2022, there were a staggering 25,000+ software vulnerabilities documented in the Common Vulnerabilities and Exposures (CVE) [3] repository. Although a huge number of these vulnerabilities has been detected, they can be traced back to a considerably small set of underlying weaknesses.

A *significant weakness*, as we define it in [4], is one that is both commonly found among publicly documented vulnerabilities and leads to severe security issues (such are those that are easily exploitable and have a high impact).

In this article, we take a fresh look at the most significant software security weaknesses. We apply our Most Significant Security Weaknesses (MSSW) equation [4], which is designed to as evenly as possible factor together frequency and severity. We complement our historical study with new data from the last three years, 2021-2023. Then we re-evaluate our previous research [5] and confirm that injection and memory corruption/disclosure continue to reappear as the most dangerous software security weaknesses. We also look closely at injection and memory corruption/disclosure through the lens of the NIST Bugs Framework (BF) [6] [7] to continue raising awareness about the patterns of reoccurring software security vulnerabilities that enable malicious activity.

XXXX-XXX © 2023 IEEE
Digital Object Identifier 10.1109/XXX.0000.0000000
Date of current version xx July 2023.

Identifying Critical Weaknesses

We identify significant weaknesses by applying our previously developed Most Significant Security Weaknesses (MSSW) equation [4] to the Common Weaknesses Enumeration (CWE) [8] View-1003 [9] entries considering the CWE model levels of abstraction.

The CWE is an enumeration of 933 types of software weaknesses with descriptions and references. Each CWE entry is assigned a CWE-*X* ID, where *X* is an integer [8]. CWE View-1003 [9] is a subset of 130 CWEs selected specifically for the NIST National Vulnerabilities Database (NVD) [10] effort of labeling

Disclaimer: Certain equipment, instruments, software, or materials, commercial or non-commercial, are identified in this paper in order to specify the experimental procedure adequately. Such identification is not intended to imply recommendation or endorsement of any product or service by NIST, nor is it intended to imply that the materials or equipment identified are necessarily the best available for the purpose.

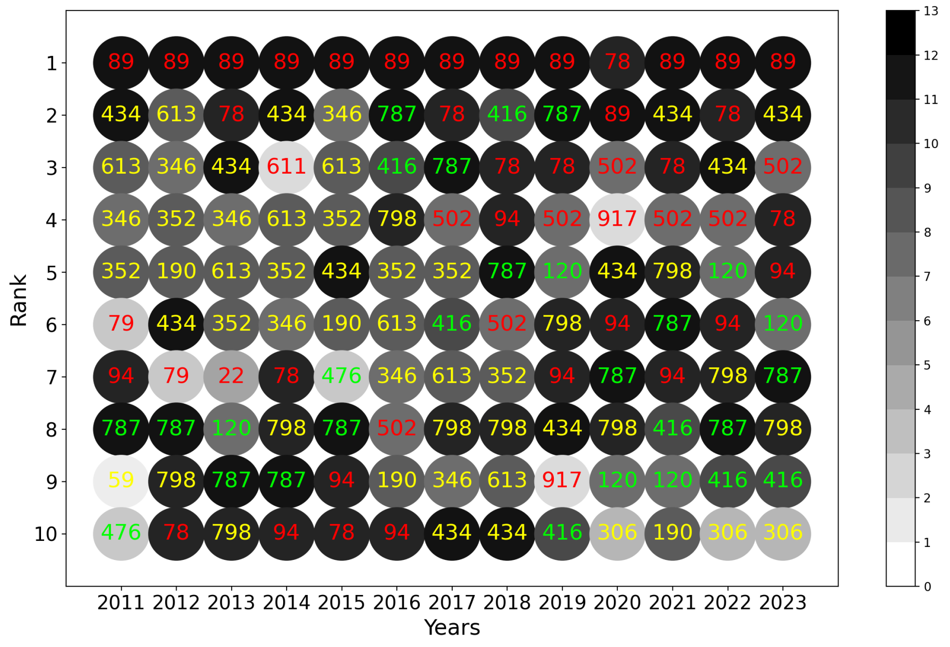

FIGURE 1: The MSSW Top 10 Base/Variant/Compound (BVC) CWEs for the last 12.5 years. Injection CWE IDs are in red, memory corruption/disclosure CWE IDs are in green, all other CWE IDs are in yellow. The most frequent CWEs are in the darkest ovals.

CVEs with CWEs. The CWE model comprises four layers of abstraction: Pillar, Class, Base, and Variant; and can use a Compound to associate two or more interacting or co-occurring CWEs. The abstractions reflect five dimensions: behavior, property, technology, language, and resource. Variant CWEs are the most specific; they describe at least three dimensions. Base CWEs are more abstract than variants and more specific than classes; they describe two to three dimensions. Class CWEs are very abstract; they describe one to two dimensions, typically not specific about any language or technology. Pillar CWEs are the highest level of abstraction. [11]

We assess the CWE frequency and severity using the MSSW equation [4] on NVD as a data source. The NVD assigns for each CVE a Common Vulnerability Scoring System (CVSS) score [12] and a relevant CWE from View-1003 [9] as the weakness allowing the vulnerability. The CVSS score provides a numerical assessment of the severity of a vulnerability by capturing its primary characteristics.

For each year, we identify the 10 CWEs with the highest MSSW value and rank them in descending order. Then, we plot the top 10 CWEs for each year to visualize and analyze how the most dangerous software security weaknesses have evolved through a period of years.

Historical Analysis

We complement our previous historical analysis over the CWEs from View-1003 for the 2010-2020 period [5] with new data for the last three years, 2021-2023.

Figure 1 shows the Top 10 list of CWEs for each year for the Base, Variant, and Compound (BVC) layer. Figure 2 shows the same for the Pillar and Class (PC) layer. Each oval with a number represents a CWE with its ID. The darkness of an oval indicates the number of times that particular CWE is used in a CVE over the years. Darker ovals correspond to the most frequent CWEs and lighter ovals correspond to the less frequent CWEs in the Top 10 lists.

Compared to the preceding decade, the software security weaknesses landscape has not evolved in the last three years. Both Figure 1 and Figure 2 are rather dark at the 2021-2023 columns, indicating that the Top 10 weaknesses found in the last three years are very frequent. Moreover, these are the same weaknesses that are seen in the previous years and that keep

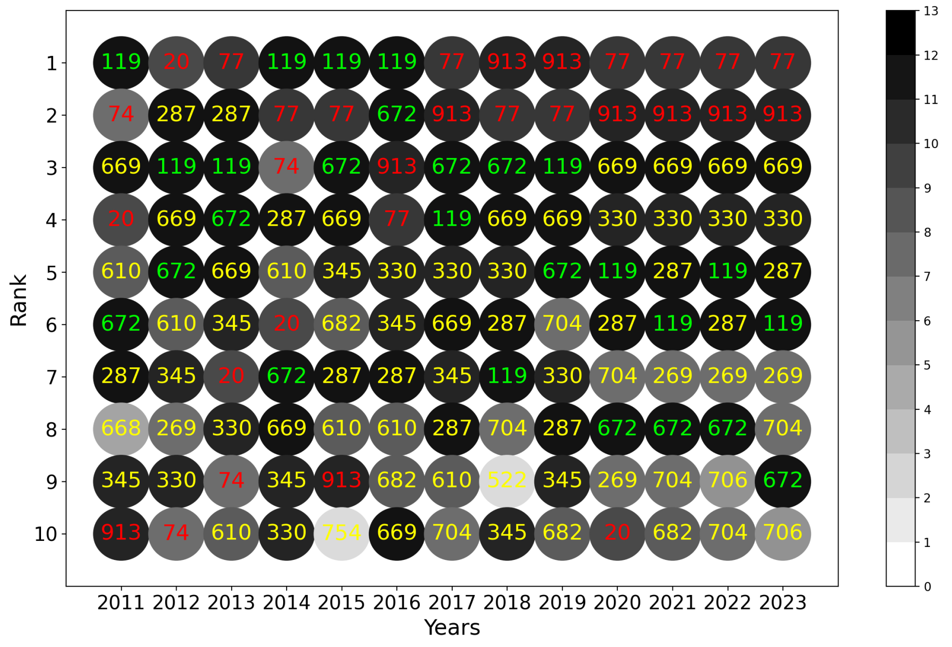

FIGURE 2: The Top 10 Pillar/Class (PC) CWEs for the last 12.5 years. Injection CWE IDs are in red, memory corruption/disclosure CWE IDs are in green, all other CWE IDs are in yellow. The most frequent CWEs are in the darkest ovals.

reoccurring. For the BVC layer, all the CWEs seen in the Top 10 weaknesses, except for one (CWE-306), in the last three years were already present in the top 10 CWEs of the preceding decade (see Figure 1). Among the 88 possible BVC CWEs, only 20 are present in the last 12.5 years. Similarly, for the PC layer, only CWE-706 was not present in the Top 10 lists of the preceding 10 years (Figure 2). Additionally, for the PC layer, the Top 4 lists have been unchanged in the last three years; and among the 39 possible PC CWEs, only 18 appear in the last 12.5 years.

These results show that a minority subset of CWEs have dominated the Top 10 lists for the last three years and the decade preceding them; from this vantage point the software weaknesses landscape is practically not changing. Instead of seeing a diversity of CWEs entering the Top 10 lists, the same kinds of weaknesses reappear year after year.

The two groups of weaknesses dominating the Top 10 lists are injection and memory corruption/disclosure. This is illustrated by Figure 3, which shows how the MSSW score in our BVC Top 10 lists evolves over the years. The blue line presents the sum of the MSSW score of all CWEs in the BVC Top 10 list of each year. The red line shows the sum for injection CWEs, while the green line shows the sum for memory corruption/disclosure CWEs. The yellow line shows all 'other CWEs', which are neither injection nor memory corruption/disclosure; these include CWEs related to file management, integer arithmetic, authentication, authorization, cryptographic authentication, and cryptographic verification. The three groups are also shown in Figure 1 and Figure 2 by the color of the CWE ID inside each oval.

One can observe in Figure 3 a consistent increase in the sum of the MSSW scores of all Top 10 BVC CWEs during the last 12.5 years. This represents a shift towards a subset of CWEs that increasingly become both the most frequent and the most impactful. Note that this is not due simply to an increase in the number of vulnerabilities discovered, because both frequency and impact are normalized within MSSW. One explanation for this trend could be that attackers are increasingly leveraging CWEs that give them the greatest influence on the targeted software systems.

Injection and memory corruption/disclosure CWEs dominate the Top 10 lists and follow this trend of increasing MSSW scores. Analyzing Figure 1, we can

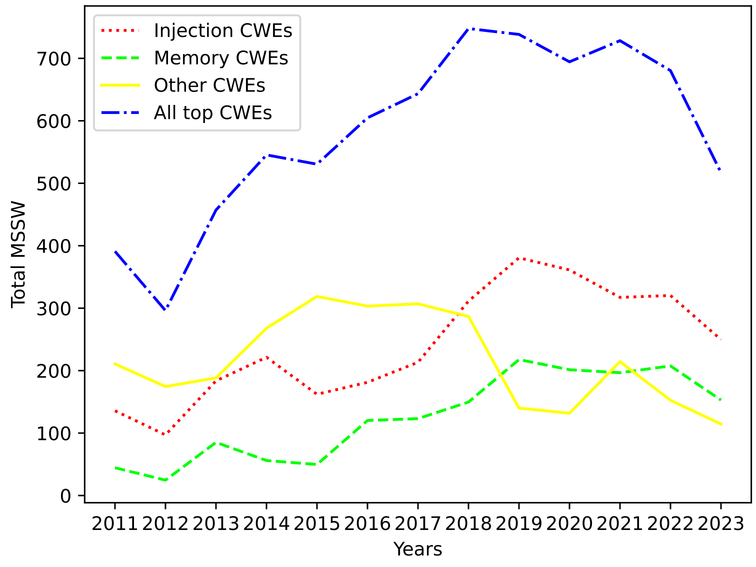

FIGURE 3: The sum of the MSSW score of all CWEs in the BVC Top 10 list of each year.

conclude that after 2017, all the five most dangerous CWEs are consistently either injection or memory corruption/disclosure. After 2019, only two CWEs are outside of those groups in the BVC Top 10 lists. This explains the increase of the MSSW score sum for injection and memory corruption/disclosure CWEs and the decrease of the MSSW score sum for other CWEs.

Injection: The Most Dangerous Software Security Weakness

The NIST Bugs Framework (BF) defines injection as "an undefined or exploitable system behavior caused by 'code separation' data validation bugs" [1]. Injection occurs because of violation of the data separation principle in modern application models. Data and code have the same internal representation and there is no formal way to distinguish them at that level. However, looking through the lens of BF, we can clearly see that code is an operation that the program executes – a behavior; and data are the operands used by the operation to produce an output. This allows us to distinguish them and reason about them on a higher level. When code (operations) and data (operands) are mixed, unexpected behaviors may arise from unpredictable data values. Therefore, programmers must be aware of the values of the input data to operations to ensure the application behaves appropriately.

We identify the most dangerous injection weaknesses (see Figure 1, CWEs with red IDs) applying the MSSW equation. From 2018 to 2023, injection is represented by CWE-89 (SQL Injection), CWE-78 (OS Command Injection), CWE-502 (Deserialization of Untrusted Data), CWE-94 (Code Injection), and CWE-917 (Expression Language Injection).

The BF taxonomy [6] groups injections into five exploitable errors: Query Injection (e.g., CWE-89), Command Injection (e.g., CWE-78), Source Code Injection (e.g., CWE-94, CWE-502, and CWE-917), Parameter Injection, and File Injection. They all are caused by improper input data validation or sanitization and allow malicious insertions: "Query Injection allows malicious insertions of condition parts or entire commands into an input used to construct a database query"; "Command Injection - of new commands into the input to a command that is sent to an operating system (OS) or a server"; "Source Code Injection – of new code into input used as part of an executing application code"; "Parameter Injection – of data into input used as parameter/argument in other parts of code"; and "File Injection – of data into input used to access/modify files or as file content". [1]

The full 12.5 years plot shows that 'SQL Injection' is consistently the number one weakness in every Top 10 BVC list, followed by 'OS Command Injection' (see Figure 1). For the last three years, some interesting observations are that 'Code Injection' first went down, but then started climbing up again; and 'Source Code Injection' (CWE-94 and CWE-502) is consistently climbing up, although the rare CWE-917 has dropped off.

SQL Injection is by far the most dangerous weakness, according to our analysis. OS Command Injection is the second most dangerous injection weakness. It is also a contributor to Class CWE-77 (Improper Neutralization of Special Elements used in a Command ('Command Injection')) (see Figure 2). Code Injection, CWE-94, also plays a significant role in the top 10 lists and is a contributor to Class CWE-913 (Improper Control of Dynamically-Managed Code Resources) (see Figure 2).

Deserialization of untrusted data (CWE-502) is a considerably new injection weakness. It appears for the first time in the 2016 BVC Top 10 list. The reason being, the exploitation of deserialization bugs increased after November 2015, when Foxglove Security published their exploits for the Java deserialization weakness [13]. From there on, currently CWE-502 is the second most dangerous injection weakness.

Looking through the lens of BF we can observe that three of the BF injection errors are solidly covered by the top 10 lists in this analysis (see Figure 1 and Figure 2): CWE-89 corresponds to Query Injection; CWE-78 – to Command Injection; CWE-502, CWE-94, and CWE-917 – to Source Code Injection.

The causes for such exploitable errors identified by BF can be missing or erroneous code in a validate or sanitize operation, an under/over-restrictive policy, corrupted/tampered data, or corrupted/tampered policy data. If programmers learn to always properly check any input data, they should be able to avoid injection errors. The BF Data Validation (DVL) class is a good start to learn about bugs, faults, operations, errors and final (exploitable) errors related to injection [1].

Memory Corruption/Disclosure: The Second Most Dangerous Software Security Weaknesses

The NIST Bugs Framework (BF) defines memory corruption/ disclosure as "an undefined or exploitable system behavior caused by memory addressing, allocation/ deallocation, or use bugs" [1]. Memory corruption/disclosure happens when data stored in memory are unintentionally modified or revealed via writing into or reading from an improper object, respectively. NULL pointer dereferencing is also related to both of them. An object is improper if its address (e.g., the associated pointer is over bounds) or size (e.g., not enough memory is available to allocate an object of that size) is improper; or if the data for its address (e.g., hardcoded address) or the data for its used size (e.g., not matching the actual size of the object) are improper; or if its type (its pointer/index type) is improper (e.g., casted pointer). Therefore, programmers must be aware of the values and the types of the pointers associated with the used object addresses and the values of the used sizes, to ensure the application behaves appropriately.

We identify the most dangerous memory corruption/disclosure weaknesses (see Figure 1, CWEs with green IDs) applying the MSSW equation. From 2018 to 2023, memory corruption/disclosure is represented by: CWE-787 (Out-of-bounds Write), CWE-120 (Classic Buffer Overflow), CWE-416 (Use After Free), and CWE-476 (NULL Pointer Dereference).

The BF taxonomy groups memory corruption/disclosure errors into 11 exploitable errors [6]. To discuss Figure 1, we focus here on the four exploitable errors associated with the most dangerous CWEs. BF defines them as follows: "Buffer Overflow is writes above the upper bound of an object". "Buffer Underflow is writes below the lower bound of an object". Use After Free is an attempt to read/write a deallocated object". "NULL Pointer Dereference is an attempt to access an object for reading or writing via a NULL pointer." [1]

The full 12.5 yeas plot shows that 'Out-of-bounds Write' appears in every Top 10 BVC list, while 'Classic Buffer Overflow' appears in 2013 and 2019 and after (see Figure 1). 'Use After Free' is present in the 2016 top 10 and after, except for 2020.

Class CWE-119, which encompasses the general memory corruption/disclosure weakness 'Use After Free', is also an area of concern. All memory CWEs on the Top 10 lists contribute to this class, except for CWE-476, which contributes to Class CWE-672. Given the broad scope of Class CWE-672, it is also the parent class of CWE-613. Addressing these memory corruption/disclosure weaknesses should be a top priority to improve memory safety.

Looking through the lens of BF, we can observe that these four BF memory corruption/disclosure errors are solidly covered by the top 10 lists in this analysis (see Figure 1 and Figure 2). CWE-787 and CWE-120 correspond to Buffer Overflow; CWE-787 – to Buffer Underflow; CWE-416 – to Use After Free; CWE-476 – to NULL Pointer Dereference.

The causes for such exploitable errors can be missing or erroneous code in a read, a write, or a dereference operation; or there is an improper address, size, address data, size data, or pointer type. Data are improper if a hardcoded (wrong specific) or forbidden (OS protected or non-existing) address or a wrong (not matching the actual object) size is used. Type is improper if a casted pointer is used. Address is improper if the associated pointer is over/under bounds (of its object), wild (arbitrary – e.g., uninitialized), untrusted (improperly checked), or dangling (of a deallocated object). Size is improper if not enough memory is available. It is improper also to use a NULL (zero address) pointer when dereferencing an object. Therefore, programmers must be aware to properly maintain pointers and types and to check size and available memory to ensure the application behaves appropriately. If programmers learn to always properly define and use pointers and objects they should be able to minimize memory corruption/disclosure errors. The BF Memory Use (MUS) and Memory Management (MMN) classes are a good start to learn about bugs, faults, operations, errors, and final (exploitable) errors related to memory corruption/disclosure [1].

It is worth noting that we have no choice but to deal with memory corruption/disclosure bugs in C legacy or embedded systems. However, we urge software developers to use programming languages (e.g., Rust) that provide memory safe alternatives such as smart pointers and other language features that prevent common memory safety issues [14].

CONCLUSION

The NIST National Vulnerabilities Database (NVD) [10] uses the carefully selected 130 CWEs from VIEW-1003 [9] to label CVEs with CWEs. From those 130 weakness types, only a limited number of CWEs (20) appears in our BVC analysis over a time span of almost 12.5 years. Looking at the 2018-2023 period, we can see the picture is getting worse, while just a few CWEs change from year to year (there are only 14 different CWEs for that period). This indicates that year after year, security researchers keep finding similar vulnerability patterns in software. In other words, no new dangerous types of weaknesses are entering the Top 10 lists, except for the newly appeared in 2015 deserialization of untrusted data.

We are not a lone voice in the wilderness calling attention to this issue. The danger of software security weaknesses is widely publicized by successful projects, such as Open Worldwide Application Security Project (OWASP) Top 10 [15] and MITRE Top 25 [16]. Still, these weaknesses keep reappearing in software security vulnerabilities. As Information Technology (IT) professionals, we should make an extra effort to spread the word for prioritizing education in secure coding. Mitigation techniques are important, but programmers, software developers, and security experts should concentrate their efforts on preventing the possible causes for software security weaknesses. Let's start with the most dangerous ones – injection and memory corruption/disclosure – and ultimately, reduce the number and severity of new vulnerabilities discovered over time. We look forward to writing this very same paper, finding that, as a community, we are making progress in changing the software security vulnerability landscape for good.

Assane Gueye, is an Associate Teaching Professor at Carnegie Mellon University Africa (CMU-Africa), co-Director of the Upanzi Network and CyLab-Africa Initiatives, and a Guest Researcher at NIST, USA. His research interest includes cybersecurity, security and resilience of large-scale systems, and Information and Communication Technologies for Development (ICT4D). Contact him at assaneg@andrew.cmu.edu.

Carlos E. C. Galhardo, is a researcher at Inmetro, Brazil. His research interests include information science, cybersecurity, and mathematical modeling in interdisciplinary applications. Contact him at cegalhardo@inmetro.gov.br.

Irena Bojanova, is a computer scientist at NIST, USA. She is the primary investigator and lead of the NIST Bugs Framework (BF) project. Her current research interests include cybersecurity and formal methods. She is a Senior member of the IEEE Computer Society. Contact her at irena.bojanova@nist.gov.

References

[1] NIST, I. Bojanova, *The Bugs Framework (BF)*, Accessed: 2023-02-14, 2023. [Online]. Available: https://samate.nist.gov/BF/.

[2] I. Bojanova and C. E. Galhardo, "Bug, fault, error, or weakness: Demystifying software security vulnerabilities," *IT Professional*, vol. 25, no. 1, pp. 7–12, Jan. 2023. DOI: 10.1109/MITP.2023.3238631.

[3] MITRE, *Metrics*, Accessed: 2023-07-07, 2023. [Online]. Available: https://www.cve.org/About/Metrics.

[4] C. E. Galhardo, P. Mell, I. Bojanova, and A. Gueye, "Measurements of the most significant software security weaknesses," in *2020 Annual Computer Security Applications Conference (ACSAC)*, 2020, pp. 154–164.

[5] A. Gueye, C. E. Galhardo, I. Bojanova, and P. Mell, "A decade of reoccurring software weaknesses," *Security & Privacy*, vol. 19, no. 6, pp. 74–82, Nov. 2021. DOI: 10.1109/MSEC.2021.3082757.

[6] I. Bojanova, C. E. Galhardo, and S. Moshtari, “Input/output check bugs taxonomy: Injection errors in spotlight,” in *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2021, pp. 111–120. DOI: 10.1109/ISSREW53611.2021.00052.

[7] I. Bojanova and C. E. Galhardo, “Classifying Memory Bugs Using Bugs Framework Approach,” in *2021 IEEE 45nd Annu. Computer, Software, and Applications Conf. (COMPSAC)*, 2021, in press.

[8] MITRE, *Common weakness enumeration (CWE)*, Accessed: 2023-03-02, 2023. [Online]. Available: https://cwe.mitre.org.

[9] MITRE, *CWE VIEW: Weaknesses for Simplified Mapping of Published Vulnerabilities*, Accessed: 2023-07-07, 2015. [Online]. Available: https://cwe.mitre.org/data/definitions/1003.html.

[10] NVD, *National Vulnerability Database (NVD)*, Accessed: 2023-03-02, 2023. [Online]. Available: https://nvd.nist.gov.

[11] MITRE, *Cwe glossary*, Accessed: 2020-05-11, 2020. [Online]. Available: https://cwe.mitre.org/documents/glossary/.

[12] FIRST, *Common vulnerability scoring system special interest group*, Accessed: 2023-07-07, 2023. [Online]. Available: https://www.first.org/cvss.

[13] L. Raghavan, *Lessons learned from the java deserialization bug*, Accessed: 2023-07-07, 2016. [Online]. Available: https://medium.com/paypal-engineering/%20lessons-learned-from-the-java-deserialization-bug-cb859e9c8d24.

[14] Internet Security Research Group (ISRG), *Memory safety for the Internet’s most critical infrastructure*, Accessed: 2023-07-07, 2023. [Online]. Available: https://www.memorysafety.org/.

[15] OWASP, *OWASP Top Ten*, Accessed: 2023-07-07, 2021. [Online]. Available: https://owasp.org/www-project-top-ten/.

[16] MITRE, *CWE Top 25 Most Dangerous Software Weaknesses*, Accessed: 2023-07-07, 2023. [Online]. Available: https://cwe.mitre.org/top25/index.html.