Analyzing Code Security: Approaches and Tools for Effective Review and Analysis

Alin-Marius Stanciu
Department of Computer and Information Technology
University Politehnica Timisoara
Timisoara, Romania
alin.stanciu@cs.upt.ro

Horia Ciocârlie
Department of Computer and Information Technology
University Politehnica Timisoara
Timisoara, Romania
horia.ciocarlie@cs.upt.ro

2023 International Conference on Electrical, Computer and Energy Technologies (ICECET) | 979-8-3503-2781-6/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICECET58911.2023.10389326

***Abstract*— Code security is a critical aspect of software development, as vulnerabilities in code can expose systems to various security threats. This paper aims to provide an in-depth analysis of different approaches and tools used for code security review and analysis. The objective is to explore effective methods and techniques that help identify and mitigate potential security risks in software code. Several approaches for code security review are examined, including manual code review, automated static analysis, dynamic analysis, and hybrid approaches. The strengths and limitations of each approach are discussed, along with their applicability in different development scenarios.**

Keywords—Code analysis, Code review, Code security, Dynamic analysis, Security tools, Static analysis, Vulnerability assessment

I. Introduction

Customers have a legitimate expectation for high-quality products that incorporate securely designed software. Secure software development entails the creation of code that can effectively withstand malicious attacks. Unlike writing code with no regard for security, coding securely demands additional knowledge and effort. Developers are required to implement essential safeguards to guarantee that the code they create and oversee remains devoid of potential vulnerabilities for exploitation. Secure software development entails embracing a proactive coding approach, diligently working to forestall errors that might lead to exploitable weaknesses or inadvertently assist malicious actors in any way.

In addition, the code must undergo a process of "hardening." An apt analogy for security hardening is the act of eliminating undesirable features from a castle to make it difficult for attackers to breach its defenses. In this analogy, secure coding corresponds to ensuring that the castle's construction materials and features are robust and of superior quality. Just as a castle made of subpar bricks or weak materials would be easily infiltrated by an attacker, the strength and integrity of the code must be fortified to withstand potential breaches [1].

Secure code demonstrates robustness, enabling it to effectively manage unforeseen scenarios and unusual inputs without encountering problems. Among the most detrimental security flaws a product can exhibit is its susceptibility to crashes when confronted with unexpected protocol inputs. An eminent real-world instance of such a weakness is the infamous "Ping of Death" Denial of Service vulnerability that emerged in the 1990s. In this situation, TCP/IP implementations would experience sudden crashes when presented with a "ping" message surpassing the designated size limit. [1].

Developers creating secure code need to be always paranoid. They will respect Murphy's laws and know that: "Anything can go wrong, will go wrong". As an example, one should not even expect that internal interfaces like function calls are used as intended and documented. The next colleague adding code to the software could just ignore all recommendations, so it always must be verified that received function parameters are within expected limits.

Common coding errors in terms of security can result in significant software weaknesses, which in turn can give rise to severe vulnerabilities. These weaknesses are often readily identifiable and exploitable, posing a significant risk. The danger lies in the fact that these vulnerabilities can enable attackers to gain complete control over the execution of the software, exfiltrate sensitive data, or disrupt the intended functionality of the software.

II. General Pitfalls in Coding

A number of frequently encountered vulnerabilities include:

Injection Vulnerabilities (SQL, OS commands, etc.)

Code injection refers to the manipulation of a computer vulnerability triggered by the processing of invalid data. Attackers employ injection techniques to insert code into a susceptible computer program, thereby altering the program's normal execution flow. [2].

Buffer overflow or underflow

An anomaly where a program while writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory locations or over-reads the buffer's boundary and retrieves potentially sensitive data from memory [3].

Missing or weak authentication or authorization

Weak authentication or authorization may allow a user to access an application feature or resource without having first authenticated and also been assigned the authority to use it [4].

Lacking input validation and reliance on untrusted inputs

User input and input from other systems may always be malicious and if it's not validated and/or is used to make security decisions, it can lead to many of the other vulnerabilities included here, such as injection vulnerabilities [5].

Incorporating external code without conducting integrity validation checks.

If part of the code is retrieved during runtime from another source and its integrity is not verified, this code may introduce vulnerabilities, backdoors, and other malicious actions into the product [6].

Utilizing flawed, vulnerable, or custom-designed algorithms.

Cryptographic algorithms that have been broken or that have not gone through sufficient scrutiny from the general community are generally vulnerable to attacks.

Hard-coded secrets

Passwords, API keys, etc. hard-coded in the application source code, which means that an attacker can identify and misuse them, and it might be difficult to change them afterwards [42].

Lack of proper error handling

Without proper error handling, an attacker may be able to find more information about the system under attack. Using that information, the attacker can use it to exploit the code by maybe providing unexpected input, or by using other information gained from the system output in case of an error.

Path traversal

A directory traversal (or path traversal) consists of exploiting insufficient security validation/sanitization of user-supplied input file names, such that characters representing "traverse to parent directory" are passed through the file APIs [43].

Web application vulnerabilities (XSS, CSRF, etc.)

Web applications have a specific set of typical vulnerabilities including Cross-Site Scripting, which enables attackers to inject client-side scripts into web pages viewed by other users [44].

So, potentially, a small simple vulnerability can cause many problems and significant damage.

III. Cause and Effect

In the following, some examples of security mistakes in coding from the real world are presented.

Equifax - CVE-2017-5638 bug in Apache Struts

CVE-2017-5638 bug in Apache Struts was publicly disclosed on March 6, 2017. The vulnerability enabled the attacker to execute remote code with malicious Content-Type using Apache Commons Object Graph Navigation Library (OGNL) language injection [7-9].

The vulnerability was fixed in Apache Struts versions 2.3.32 and 2.5.10.1 already on the same day. Equifax failed to detect that and did not patch the issue even for two months after it was detected and fixed. Between May and July 2017, attackers targeted their systems using OGNL injection and stole 143 million social security numbers, birthdays, driver's license numbers and, addresses [10].

This issue could have been easily avoided if Equifax screened properly their 3rd party software for vulnerabilities and promptly applied the patch when the issue was detected and corrected.

Panama Papers

A working exploit for the Slider Revolution plugin for WordPress was published in September 2014. It used unrestricted file upload vulnerability. Without proper authentication, in the "admin-ajax.php" module, the attacker only needed to send the malicious payload to the proper URL and gain access to the system over a web shell implementation [11, 12].

Mossack Fonseca law firm's website was vulnerable until March 2017. Attackers found their way into the system using the exploit and stole 11.5 million documents and 2.6 Terabytes of data as they accessed the company's mail server, downloading all the emails [13, 14].

This could have been avoided by simply updating the WordPress plugin that was used to gain access to the system.

TalkTalk – SQL Injection Attack

The subsequent incident prompted an inquiry by the UK's Information Commissioner's Office(ICO), leading to a penalty for TalkTalk, a telecommunications firm based in the UK.

The attacker used plain SQL injection to gain access to the customer data through the company's website. This enabled the attacker to steal 156.959 customer personal data records including their names, addresses, dates of birth, phone numbers, and email addresses. In 15.656 cases, the attacker also had access to bank account details and sort codes [15].

As a result of the subsequent investigation by the ICO, TalkTalk was fined £400,000 due to security deficiencies that made it relatively easy for a cyber attacker to access customer data. [16].

As we can see, what was seemingly a small flaw, led to a huge security problem. This is a good example that demonstrates why input validation is important and why we should always write code that does it properly.

IV. Basic Principles

Even a minor and straightforward vulnerability can lead to extensive issues and substantial harm. Here are some fundamental principles and secure coding guidelines that should be adhered to diligently.

Reliance on concealed security measures is not a viable strategy. It is essential to assume that the source code is accessible to the public at all times. [36].

Frequent security challenges – Acquaint yourself with the typical security issues stemming from the unique attributes of the programming language in use and steer clear of them. [37].

Documented vulnerabilities – Ascertain that employed constructs, functions, and libraries do not have documented security vulnerabilities. [1].

Uninspected input from the outside – Inspect all input from external origins to filter out data that doesn't meet the system's predefined criteria. [38].

Harness the entire suite of security tools available within the programming language and frameworks of choice.

Safely managing failures – Handling failures securely means avoiding the disclosure of data that wouldn't typically be exposed, even in the event of unexpected runtime errors. [39].

Least possible privileges – Ensure that only the minimal necessary privileges required to accomplish the intended task are granted. [40].

Expecting the unexpected – Make sure to consistently check for security-critical events, even if they are highly improbable. [1].

Readable code – Develop code with a consistent style across the entire project, accompanied by adequate comments, to ensure readability, clarity, and ease of maintenance. [41].

Conduct routine security reviews, quality assurance checks, and testing at various stages of development to guarantee the final product's security. This ensures that the end result is genuinely secure. [1].

Secure Coding should be promoted through education. Also, secure coding guidelines are available, collecting the most important secure coding rules. Regular code security reviews verify that the code stays secure while it is further developed and maintained.

V. Code Security Review Methods

A comprehensive examination of an application's source code, known as a secure core review, is a specialized undertaking aimed at detecting security-related weaknesses or flaws within the code. It is crucial to employ code security review methods consistently throughout the development lifecycle rather than solely at its conclusion [17].

The security of products is significantly strengthened through Source Code Security Analysis (SCSA), which uncovers vulnerabilities throughout the development process. SCSA encompasses two main methods: manual code examination and the use of automated **Static Application Security Testing (SAST)** tools. The synergy between manual code review and SAST tools expedites the identification of software defects during the early development phases, highlighting programming errors, hardening weaknesses, and suboptimal coding practices. Among the merits of SAST tools are wide-ranging code coverage, the amalgamation of software and vulnerability know-how, and a resilient set of code quality metrics. [18].

SAST tools are used for analyzing source code for possible issues. They are called “static” in contrast to “dynamic” testing as the software is not executed. This has the advantage that the analysis results do not depend on what external input was supplied for testing. However, SAST tools are also not a silver bullet as not all types of mistakes can be caught that way. SAST tools are particularly suited to find and localize “hard to test corner cases”, like security flaws, inadvertently causing the system to fail or to open doorways for attackers (vulnerabilities, enabling exploits). SAST tools provide automated means to control code quality and its security level [18, 19].

An example of a widely known simple SAST tool is “Lint”, the original simple static code analyzer for the C programming language. Lint originates from the late 1970s and generated the compiler warnings every C programmer knows [20].

Today, more sophisticated tools can provide inter-procedural, control, and data flow analysis across all source code included in a build. They can simulate potential runtime behavior with an automated view of every possible execution path. This takes any possible external input into account [21].

The types of bugs that be found in this way could lead to severe security and quality issues if not rectified:[18, 19]

- Buffer overflows, which are often used by hackers to inject their own code.
- Memory leaks, which could cause Denial of Service issues even without an active attacker involved.
- Usage of uninitialized data, which might not lead to any issue during testing but could eventually surface in the field by chance or triggered by someone on purpose to inject certain data.
- Critical use of data coming from unverified input. This tainted data could be outside the expected limits and is therefore not suited to be used for critical operations where it could trigger a wide variety of vulnerabilities.
- Operating system-specific issues that could lead to an opportunity for malicious user privilege escalation.
- Environment-specific problems like in cross-site scripting vulnerabilities on websites and command injections into database queries.

There are things that the SAST tool just cannot do. As the code is not executed, there are limitations on what can be checked automatically, so there can be missed issues that can be found during dynamic testing.

It is not possible to verify with static analysis that the required functionality was implemented correctly. It is not that hard to craft code that is completely useless but fully accepted by the analyzer. At max, there are indications that logic flaws exist. Code that isn’t reachable under any circumstances during execution, is such an indication. When it is not reachable, why was it then created in the first place? Usually investigating such issues reveals mistakes in the logic of the code [46].

In reasonably complex programs, SAST cannot be used to prove that problems do not exist. It can only give an indication that relatively obvious issues have been taken care of. Review by humans cannot be replaced. Also, all reported issues must be evaluated by a developer who understands the code. Effective SAST tools distinguish themselves from less effective ones by reporting fewer false positives while maintaining a low level of missed existing issues. In the end, conducting manual code review is still imperative, but, when utilizing automated tools for static code analysis, it will be faster, and developers can concentrate on more sophisticated issues [46].

As automated Static Application Security Testing tools can never identify all the vulnerabilities in an application's code, manual code reviews are a core part of ensuring code security. **Manual code reviews** can often identify vulnerabilities that automatic tools are not able to flag.

In a manual review, an analyst reviews code line by line, looking for defects and security-related flaws. An automated review uses a tool to scan the code and report potential flaws. Manual review is time-consuming and requires significant domain expertise to be done correctly.

Robustness or fuzz testing, bombards a system with malformed inputs to find misuse cases that trigger unknown vulnerabilities. Robustness testing can identify exceptions such as crashes, failing built-in code assertions, or potential memory leaks, that could be very difficult to identify with any other method, including source code review [22, 23].

Compared to static application security testing techniques, robustness testing is dynamic and tests a running instance of the code. A very important step, to ensure the high quality of any product, is robustness testing as it verifies that unexpected input cannot affect the system in an unpredictable way. This basically means that the system is not crashing, not showing denial of service characteristics or too extreme degradation of service, or any other abnormal behavior [24].

In addition to causing the product or component to crash, such vulnerabilities may also give attackers an opportunity to compromise security in other manners. The main objective of robustness testing is to assess the system's protocol implementations that can be interacted with externally.. It is important to test not just the application layer protocols, but the whole protocol stack.

During the testing, automatically generated, intentionally malformed protocol messages are sent to the target. Each of these messages contains some anomalies when compared to valid parameters. These anomalies can be omitting mandatory parameters, exceeding the parameter field size, using invalid characters, and so forth. For protocol sequences, also out of context messages are al submitted to test the robustness in case of incomplete or reordered sequences. After each test, the state of the system needs to be observed to notice robustness failures [25].

Even though a huge number of test cases have to be run for robustness testing, one should remember that only one single malformed message from an attacker can crash a live system. So, there should be put a lot of effort into finding those during the testing. How much effort should be put, it depends from case to case based on the identified risks.

The integration of **Dynamic Application Security Testing (DAST)**, or penetration testing, with SAST is a widely adopted approach to enrich the comprehensive security evaluation of an application. Through DAST, the application is intentionally exposed to established attack techniques across diverse interfaces, and the ensuing output is scrutinized in detail for any indications of potential vulnerabilities. This testing method can uncover previously undetected vulnerabilities and may also result in false positives. Due to the black-box nature of DAST, it is unable to provide specific source code line references that require attention. Instead, it focuses on the analysis of input and output messages. Dynamic application security testing tools are characterized by their automation, which allows them to perform security assessments spanning a multitude of real-world threats. These tools are versatile and can be seamlessly integrated into either the security testing phase or the continuous integration pipeline. Conversely, manual penetration testing employs more advanced techniques toward the same objective. [26].

Interactive Application Security Testing (IAST) stands as a cutting-edge technology that harmonizes SAST and penetration testing to leverage their combined potential. This approach yields a comprehensive overview of vulnerabilities, minimizing the occurrence of time-consuming false positives. Notably, IAST goes beyond identifying vulnerabilities and provides precise source code line locations where fixes should be implemented to address specific vulnerabilities. By offering accurate guidance on code fixes, IAST enhances the efficiency and effectiveness of security testing processes [27, 28].

Dynamic approaches, although valuable, often lack visibility into the source code, limiting their ability to comprehensively evaluate security. However, by utilizing advanced tools like Interactive Application Security Testing (IAST) and adopting a combined approach involving both penetration testing and source code review, it becomes possible to harness the combined benefits of static and dynamic approaches. This integrated approach enables a deeper understanding of the application's security posture by leveraging the strengths of each method, ultimately enhancing the overall effectiveness of security assessments.

VI. Security Of External Components

In many products, external libraries or components are used. Every library or component, whether open or closed source, could potentially create a security risk to the product that is being used within. This is not to say that these should not be utilized, but rather that it should be ensured that the best is done to understand and manage those risks.

The SVM (Software Vulnerability Management) process refers to the gathering of information on attacks, vulnerabilities, and available correction patches from various information sources. The process assesses the relevance of this information to the products and issues corresponding security recommendations for further action. The process should be continuous and concurrent with the product life cycle process. However, while the SVM process aims to support in making decisions based on the current state of security in components used in the products, the use of these components also needs to be considered during the development process. [45]

External libraries, components, and modules should always be vetted from a security perspective. The security of the technical component itself should be considered but in addition, because security is not a static state but rather a process, it should be considered how the security of the particular component may evolve in the future.

This may be affected for example by what party is maintaining it, how well security has been considered in development, and how quickly can vulnerabilities expected to be resolved. In summary, the effect of using externally developed code needs to be evaluated.

Software Composition Analysis (SCA) is an automated process that identifies the open-source software in a codebase. Effective SCA tools can help achieve a comprehensive overview of what open-source components are utilized in a particular product's code and create an accurate bill of materials, as well as understand how these open-source components affect the security of the product. SCA tools do not analyze proprietary code [29, 30].

Other Secure Coding Resources are:

SEI CERT Secure Coding Standard

One of the most used publicly accessible Secure Coding Rules is the CERT Secure Coding Standards. They are available for the C, C++, Java, and Perl programming languages [31].

CWE – Common Weakness Enumeration

The Common Weakness Enumeration project provides a free online catalog of software weaknesses and vulnerabilities for all kinds of languages. It is a valuable resource for secure coding [32].

MISRA – Motor Industry Software Reliability Association

The Motor Industry Software Reliability Association provides well-respected guidelines for coding in C and C++ programming languages to paying customers. As the name suggests, the work had initially been started for critical systems in the motor industry. Since then, the guidelines evolved into a widely accepted model for best coding practices for all kinds of industries creating software [33, 34].

OWASP – Open Web Application Security Project

The Open Web Application Security Project is a non-profit foundation that works to improve the security of software. Through community-led open-source software projects, hundreds of local chapters worldwide, tens of thousands of members, and leading educational and training conferences, the OWASP Foundation is the source for developers and technologies to secure the web [35].

VII. Conclusions

Highlighting the significance of thorough security evaluations during development is crucial. Employing a systematic strategy that integrates manual code review with automated tools like SAST is essential for effective vulnerability detection. Additionally, integrating dynamic approaches such as penetration testing can provide a more holistic understanding of an application's security posture. Analyzing code security is a critical aspect of software development to ensure the creation of secure and robust applications.

It is essential to embrace a systematic method that combines manual code review and automated tools like Static Application Security Testing (SAST) to enhance vulnerability identification.. Additionally, integrating dynamic approaches such as penetration testing can provide a more holistic understanding of an application's security posture.

Analyzing code security and implementing effective review and analysis practices contribute to developing secure and robust software. It helps prevent unauthorized access, data breaches, and the disruption of software functionality.

Continuous evaluation and improvement of code security should be prioritized throughout the software development life cycle. Regular security assessments, code reviews, and the adoption of secure coding practices are essential to address emerging threats and vulnerabilities.

In conclusion, by employing a systematic approach to code security analysis and leveraging appropriate tools, developers can enhance the security posture of their applications, protect against potential threats, and deliver high-quality software that meets the expectations of customers and users.

VIII. References

[1] A.M. Stanciu, (2023). Theoretical Study of Security for a Software Product. In: Nagar, A.K., Singh Jat, D., Mishra, D.K., Joshi, A. (eds) Intelligent Sustainable Systems. Lecture Notes in Networks and Systems, vol 578. Springer, Singapore. https://doi.org/10.1007/978-981-19-7660-5_20

[2] D. Ray, & J. Ligatti, (2012). Defining code-injection attacks. Acm Sigplan Notices, 47(1), 179-190.

[3] S. Sidiroglou, G. Giovanidis, & A. D. Keromytis, (2005). A dynamic mechanism for recovering from buffer overflow attacks. In Information Security: 8th International Conference, ISC 2005, Singapore, September 20-23, 2005. Proceedings 8 (pp. 1-15). Springer Berlin Heidelberg.

[4] B. Grobauer, T. Walloschek, & E. Stocker, (2010). Understanding cloud computing vulnerabilities. IEEE Security & privacy, 9(2), 50-57.

[5] W. Xu, S. Bhatkar, & R. Sekar (2005). Practical dynamic taint analysis for countering input validation attacks on web applications. Dept. Comput. Sci., Stony Brook Univ., Stony Brook, NY, USA, Tech. Rep. SECLAB-05-04.

[6] G. Wurster, P.C Van Oorschot, & A. Somayaji (2005, May). A generic attack on checksumming-based software tamper resistance. In 2005 IEEE Symposium on Security and Privacy (S&P'05) (pp. 127-138). IEEE.

[7] C. Cimpanu. [Online]. Available: https://www.bleepingcomputer.com/news/security/equifax-confirms-hackers-usedapache-struts-vulnerability-to-breach-its-servers/. [Accessed 2023 05 24].

[8] "G. Turcsányi," [Online]. Available: https://avatao.com/blog-deep-dive-into-theequifax-breach-and-the-apache-struts-vulnerability/ [Accessed 2023 04 06].

[9] [Online]. Available: https://www.cvedetails.com/cve/CVE-2017-5638/ [Accessed 2023 04 08].

[10] [Online]. Available: https://medium.com/@briskinfosec/command-execution-attackson-apache-struts-server-cve-2017-5638-e8f1cb52385 . [Accessed 2023 05 01].

[11] [Online]. Available: https://www.cvedetails.com/cve/CVE-2014-9734/ [Accessed 2023 05 01].

[12] [Online]. Available: https://blog.sucuri.net/2014/09/slider-revolution-plugin-criticalvulnerability-being-exploited.html [Accessed 2023 05 01]

[13] [Online]. Available: https://www.icij.org/investigations/panamapapers/pages/panama-papers-about-the-investigation/ [Accessed 2023 05 01].

[14] [Online]. Available: https://www.occrp.org/en/panamapapers/overview/intro/ [Accessed 2023 05 01].

[15] [Online]. Available: https://ico.org.uk/about-the-ico/media-centre/talktalk-cyber-attack-how-the-ico-investigation-unfolded/#:~:text=ICO%20investigation%20findings,of%20the%20Data%20Protection%20Act.. [Accessed 2023 06 02].

[16] [Online]. Available: https://www.theguardian.com/business/2016/oct/05/talktalk-hitwith-record-400k-fine-over-cyber-attack [Accessed 2023 06 03].

[17] A. Edmundson, et al. "An empirical study on the effectiveness of security code review." Engineering Secure Software and Systems: 5th International Symposium, ESSoS 2013, Paris, France, February 27-March 1, 2013. Proceedings 5. Springer Berlin Heidelberg, 2013.

[18] [Online]. Available: https://owasp.org/wwwcommunity/Source_Code_Analysis_Tools [Accessed 2023 06 05].

[19] [Online]. Available: https://owasp.org/www-community/Vulnerability_Scanning_Tools [Accessed 2023 06 04].

[20] S. C. Johnson, (1977). Lint, a C program checker. Murray Hill: Bell Telephone Laboratories.

[21] [Online]. Available: https://www.cs.odu.edu/~zeil/cs350/latest/Public/analysis/index.html. [Accessed 2023 06 04].

[22] [Online]. Available: https://www.synopsys.com/glossary/what-is-fuzz-testing.html [Accessed 2023 06 05].

[23] K. Wolter, A. Avritzer, M. Vieira and A. van Moorsel, "Robustness Testing Techniques and Tools," in Resilience Assessment and Evaluation of Computing Systems, 2012.

[24] C. Hutchison, M. Zizyte, P. E. Lanigan, D. Guttendorf, M. Wagner, C. L. Goues, & P. Koopman (2018, May). Robustness testing of autonomy software. In Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice (pp. 276-285)..

[25] [Online]. Available: https://tec.gov.in/pdf/Studypaper/fuzz%20testing%20study%20paper.pdf [Accessed 2023 06 05]

[26] [Online]. Available: https://u-tor.com/topic/black-box-penetration-testing [Accessed 2023 06 05].

[27] [Online]. Available: Interactive Application Security Testing . https://snyk.io/learn/application-security/iast-interactive-application-security-testing/ [Accessed 2023 06 05]

[28] J. Williams. [Online]. Available: https://www.contrastsecurity.com/securityinfluencers/why-the-difference-between-sast-dast-and-iast-matter . [Accessed 2023 06 05].

[29] [Online]. Available: https://www.synopsys.com/glossary/what-is-softwarecomposition-analysis.html [Accessed 2023 06 05].

[30] [Online]. Available: https://www.whitesourcesoftware.com/resources/blog/sast-vssca/. [Accessed 2023 06 04].

[31] [Online]. Available: https://wiki.sei.cmu.edu/confluence/display/seccode [Accessed 2023 06 02

[32] [Online]. Available: https://cwe.mitre.org/about/index.html [Accessed 2023 06 05].

[33] [Online]. Available: https://www.misra.org.uk/ [Accessed 2023 06 02].

[34] [Online]. Available: https://www.synopsys.com/automotive/what-is-misra.html [Accessed 2023 06 02].

[35] [Online]. Available: https://owasp.org . [Accessed 2023 06 03].

[36] J. C. Smith, "Effective Security by Obscurity." arXiv preprint arXiv:2205.01547 (2022).

[37] A. Shostack, (2014). Threat modeling: Designing for security. John Wiley & Sons.

[38] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, & G. Vigna(2008, May). Saner: Composing static and dynamic analysis to validate sanitization in web applications. In 2008 IEEE Symposium on Security and Privacy (sp 2008) (pp. 387-401). IEEE.

[39] L. A. Hunt, P. McGee, R. Gutteridge & M. Hughes (2016). Failing securely: The processes and support which underpin English nurse mentors' assessment decisions regarding under-performing students. Nurse Education Today, 39, 79-86.

[40] Y. Wu, J. Sun, Y. Liu, & J. S. Dong, (2013, November). Automatically partition software into least privilege components using dynamic data dependency analysis. In 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE) (pp. 323-333). IEEE..

[41] M. Allamanis, E. T. Barr, C. Bird & C. Sutton (2014, November). Learning natural coding conventions. In Proceedings of the 22nd acm sigsoft international symposium on foundations of software engineering (pp. 281-293).

[42] S. Shuai, D. Guowei, G. Tao, Y. Tianchang & S. Chenjie (2014, August). Modelling analysis and auto-detection of cryptographic misuse in android applications. In 2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing (pp. 75-80). IEEE.

[43] M. Flanders (2019). A simple and intuitive algorithm for preventing directory traversal attacks. arXiv preprint arXiv:1908.04502.

[44] S. Nagpure & S. Kurkure (2017, August). Vulnerability assessment and penetration testing of web application. In 2017 International Conference on Computing, Communication, Control and Automation (ICCUBEA) (pp. 1-6). IEEE.

[45] R. Syed (2020). Cybersecurity vulnerability management: A conceptual ontology and cyber intelligence alert system. Information & Management, 57(6), 103334.

[46] A. Moser, C. Kruegel, & E. Kirda (2007, December). Limits of static analysis for malware detection. In Twenty-third annual computer security applications conference (ACSAC 2007) (pp. 421-430). IEEE..