Part 1. Software Security

1. Context

This report has been commissioned as part of an ongoing initiative to improve the security posture of ScottishGlen, an emerging company in the energy sector. The focus will be on providing a recommendation to the team of developers responsible for the company's software applications. With the knowledge that there is no single individual responsible for security within the developer or wider IT department, the recommendation will centre around a practice that can be adopted by the entire development team, with the aim of improving the security of the Kerberos application following threats that have recently been received by a hacktivist group.

1.1 Kerberos

Network authentication for ScottishGlen is provided by an installation of Kerberos, a protocol developed by the Massachusetts Institute of Technology (MIT) which uses secret-key cryptography to authenticate a client-server connection over a network, supporting encryption of the session to maintain privacy and ensure data integrity (MIT, 2024). A diagram of the authentication process can be found in Figure 1.

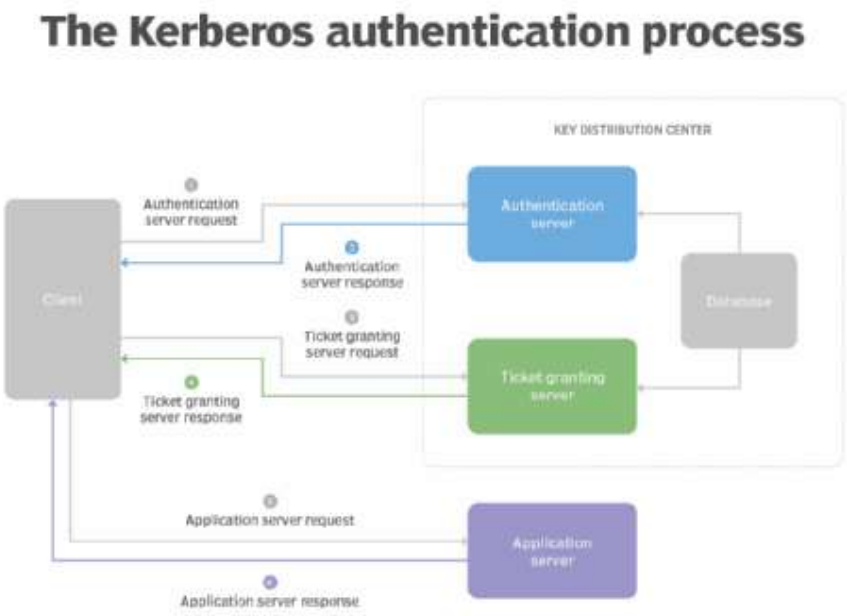

Figure 1: diagram of the Kerberos authentication process (TechTarget, 2021).

A free version of the Kerberos software is available to download from MIT for implementations outside of other commercial products, allowing developers to customise the software (if required). Being able to inspect the program's source code also allows developers to check for any potential software bugs. According to CVEdetails.com (n.d.), the documented vulnerabilities specific to Kerberos can largely be categorised as overflow or memory corruption vulnerabilities, both of which can be described as types of memory errors (Jiang and Wang, 2019).

1.2 Memory Errors

Any software written in the C programming language, such as Kerberos, has the potential to be susceptible to memory errors. It, along with its close relation C++, is considered to be a memory-unsafe language (Wikipedia, 2024). The term "memory errors" encompasses several more specifically

named bugs, each of which operates on the same basic principle - that a program's unprotected memory space can be accessed by an unauthorised entity. If exploited, this unauthorised access can cause software crashes, the corruption of data, or the integrity of data being threatened (Granot and Spektor, 2024). According to Granot and Spektor (2024), there are five common security flaws that fall under the category of memory errors that should be given consideration by software programmers:

1. Use after free vulnerabilities.
2. Out of bounds reads.
3. Out of bounds writes.
4. Null pointer dereferencing.
5. Integer flow or wraparound.

These memory errors (and others within the same category) have a range of causes, including unsafe practices and programming errors (Camacho, 2023). Whilst it is widely accepted that many of these can be detected with the practice of manual code review, it has been considered that static code analysis can perhaps offer an effective technique as a first step in identifying memory errors, when incorporated as part of the software development lifecycle (Gjomemo, et al., 2016).

2. Recommendation

Static code analysis is a practice that takes place during a wider code review process during the implementation phase of the software development lifecycle (SLDC). Its purpose is to find potential security flaws within a piece of source code that could lead to vulnerabilities being realised in the completed software if not rectified (OWASP, 2021). There is a wide range of tools available for conducting automated static analysis on source code, many of which can be incorporated into existing integrated development environments (IDEs). On a basic level, they function by reading source code and analysing the syntax using a set of underlying rules. Any programming errors found in accordance with those rules are then highlighted in a report and can be further investigated. The practice is referred to as "static" because the code is not executed during the analysis - the program simply scans the code looking for breaches of its ruleset.

According to a recent survey conducted by Incredibuild (2022), over three quarters of IT managers claimed to already make use of static analysis tools in their SLDC pipeline, second only to dynamic code analysis. The survey also showed that a further 11% of those surveyed were planning to implement it in the near future. Both of these results can be seen in Figure 2 below.

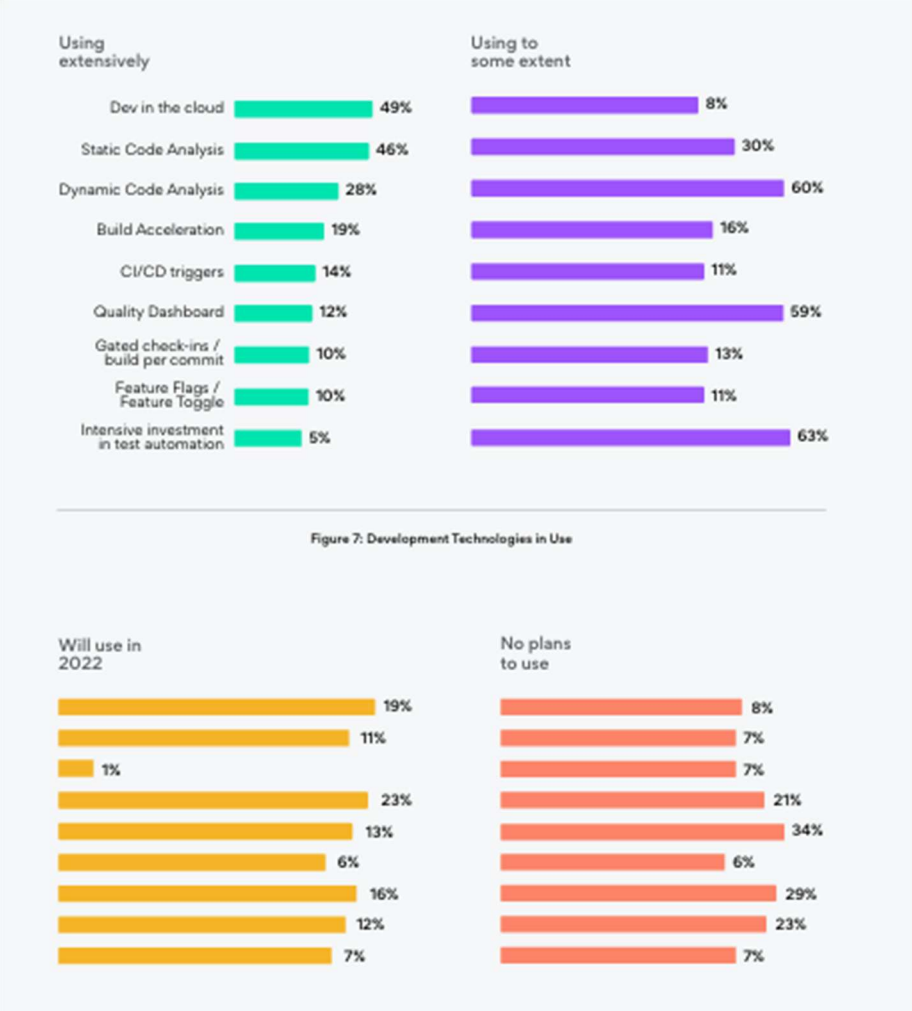

Figure 2: results showing static code analysis as the second most popular practice in SLDC (Incredibuild, 2022).

The practice of static code analysis has been found to be a highly efficient method for discovering source code errors at an early stage of the application's development, allowing developers to fix vulnerabilities before the software is released. Furthermore, the automation of this process has been determined to provide further cost- and time-efficiencies (Kaur and Nayyar, 2020). The popularity and efficiency of this practice, combined with the wide range of tools available (many of which are open source) suggest that this practice would be a practical choice for a development team in the early stages of adopting a secure SLDC. It can also be argued that automated static code analysis is an effective tool for enhancing the skills and knowledge of developers with regards to improving security standards (CodeCruze, 2024).

When considering the likely effectiveness of this practice, consideration can be given to the guidance provided by MITRE in their Common Weakness Enumeration (CWE) documentation. Automated static code analysis is given as a suggested detection method for each of the security flaws identified by Granot and Spektor (2024), and in each case is given an effectiveness rating of "high" (MITRE, 2024).

Despite the widespread adoption of static code analysis, its use can only be considered truly effective if implemented as part of a wider SLDC incorporating other practices, such as dynamic code analysis, runtime debuggers, and manual code review. This can be attributed to the high number of false positives that static code analysis can report and the inconsistency of accuracy between tools (Kaur and Nayyar, 2020), as well as the fact that some memory errors can only be realised during the runtime

of the application (Gjomemo, et al., 2016). It should also be noted that whilst static code analysis can be considered an effective practice for detecting memory errors, the technique requires further manual intervention from a developer/engineer to both validate and correct the fault.

In the case of out-of-bounds-read and use-after-free errors, CWE suggests that fuzzing can be a highly effective practice in finding the flaw (MITRE, 2024), with manual analysis and architecture/design reviews also being considered "high" in their effectiveness for detecting integer overflow/wraparound errors (MITRE, 2024). Dynamic analysis is suggested as a possibility for each of the memory errors mentioned in this report, though the CWE documentation does not always provide an evaluation of the likely effectiveness of the practice. It should be noted that whilst dynamic analysis can be particularly effective at finding subtle faults that involve the program's interaction with other application components (Codacy, 2023), it is typically conducted during later stages of the SLDC, thereby reducing some of the cost- and time efficiencies created when using automated static code analysis tools (Kaur and Nayyar, 2020).

Fundamentally, many memory errors could be avoided entirely by using memory safe languages (such as Rust and Python) instead of memory-unsafe language (such as C and C++) when writing software. This seemingly simple proposal comes rife with challenges including hardware support, regulatory requirements, lack of developer knowledge, and the inevitability of new bugs emerging when migrating to the new language (Help Net Security, 2024). A recent survey discovered that more than half of all open-source projects are written in a memory-unsafe language (CISA, 2024), with further findings suggesting that developers use memory-unsafe languages because they are unaware of unbothered by the risks connected to using them (TechRepublic, 2024).

As this report has demonstrated, automated static code analysis is just one of a number of practices that could be adopted to contribute to an improved security resilience at ScottishGlen, though it should not be considered a complete solution and would provide best results if used in conjunction with other SLDC practices (Codacy, 2023). It is the opinion of the author that, taking into account that the security posture of the company could be considered as having a low level of maturity, automated static code analysis would constitute a beneficial first step into the process of developing a more comprehensive SLDC in the future.

3. Implementation

Whilst MITRE's CWE documentation can be used to be provide guidance on broad issues common to all software, their Common Vulnerability Enumeration (CVE) database provides information on individual instances of reported vulnerabilities. It is possible to search the CVE database (via a third-party) to list all vulnerabilities specific to a product or vendor. Further information can then be garnered from the entry, which includes details about how the vulnerability can be exploited, which exact version(s) it affects, severity and likelihood of exploitation, corresponding CWEs, and useful references for research purposes.

CVE-2018-5710

CVE-2018-5710 is an example of a null pointer dereference vulnerability in Kerberos that can be found in the CVE database (CVEdetails.com, 2018). The vulnerability affects all versions of the software up to and including 5-1.16 and specifically concerns the ldap_principal2.c plugin of Kerberos. Null pointer dereferences can affect programs written in any language that uses pointers (such as C) and can occur when an attempt is made to access a memory address of an object that has been stored in a pointer that has a null value (Snyk, n.d.). A diagram showing the flow of a null pointer dereference can be seen in Figure 3 below.

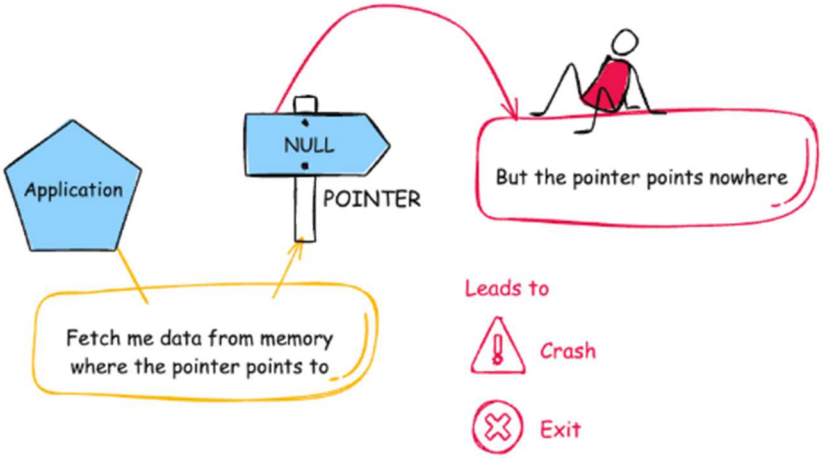

Figure 3: diagram showing the flow of a null pointer dereference (MITRE, 2024).

Null pointer dereferences can cause the program to display unpredictable behaviour or to crash entirely. In the case of CVE-2018-5710, there is a possibility that its existence could cause a denial of service to users attempting to authenticate (CVEdetails.com, 2018), breaking one of the three core pillars of information security: confidentiality, integrity, and availability (a denial of service would be considered as a loss of availability).

Prevention

Snyk is an open-source automated static code analysis tool that allows developers to discover errors, bugs, and vulnerabilities in source code. It provides guidance in its reports for prioritisation of discovered faults and extensive documentation on how to mitigate any vulnerabilities that it finds (Snyk, n.d.). It can be used to interrogate Github repositories (public or private) and to scan local software projects from the command line. When using the latter, the results can be output in a JSON format, which can then be converted (with the use of an additional Snyk tool) to an HTML report for easy reading.

With regards to CVE-2018-5710, Snyk's effectiveness at detecting the fault concerned can be demonstrated using the procedure described below, which assumes that Snyk and snyk-to-html are already installed on a Linux machine.

1. Download a copy of any of the vulnerable versions (<= 5-1.16) of Kerberos from the MIT website.
2. Open a terminal window and change directory to the location of the Kerberos source code (`cd <directoryPath>`).
3. Run the following command:
   `snyk code test --json | snyk-to-html -o snykresults.html`
4. Open snykresults.html in an internet browser window.

The results of the above procedure demonstrating Snyk's ability to detect the presence of CVE-2018-5710 can be seen in Figure 4.

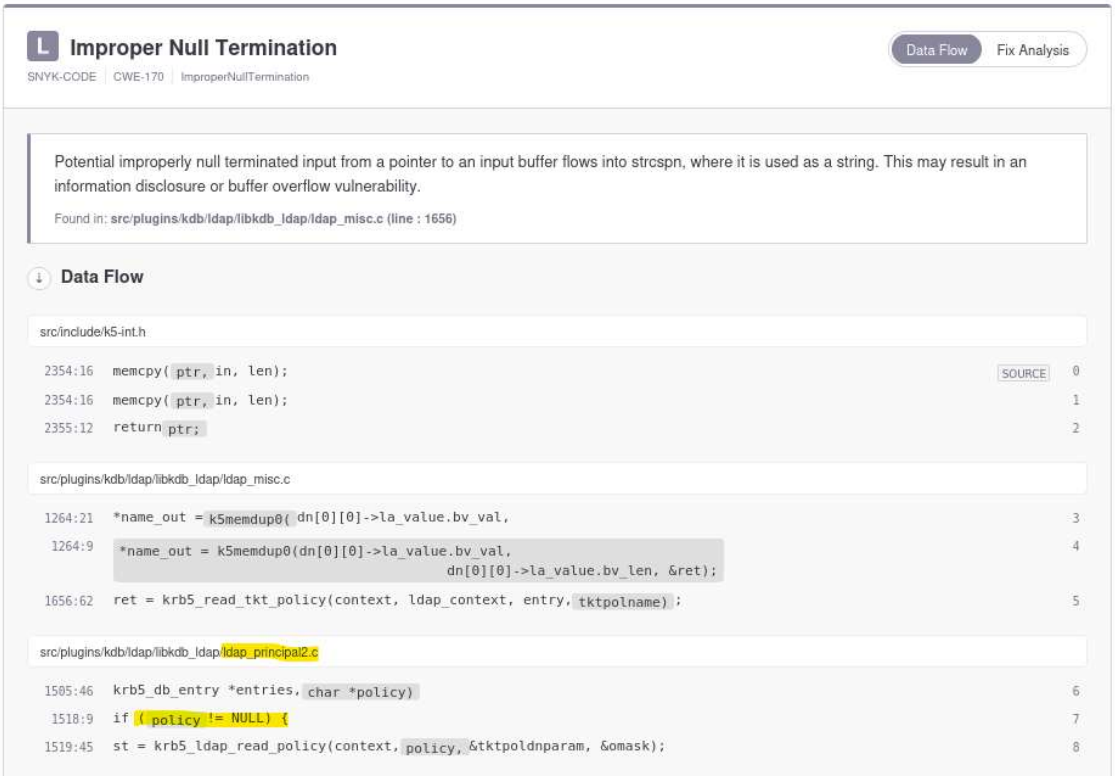

Figure 4: Snyk report showing the null pointer dereference documented in CVE-2018-5710.

The report also offers guidance on how to fix and prevent the error in the source code by use of the "Fix Analysis" link located in the detail for that particular finding. The recommendation offered for the discovery of the null pointer dereference detailed in CVE-2018-5710 can be seen in Figure 5.

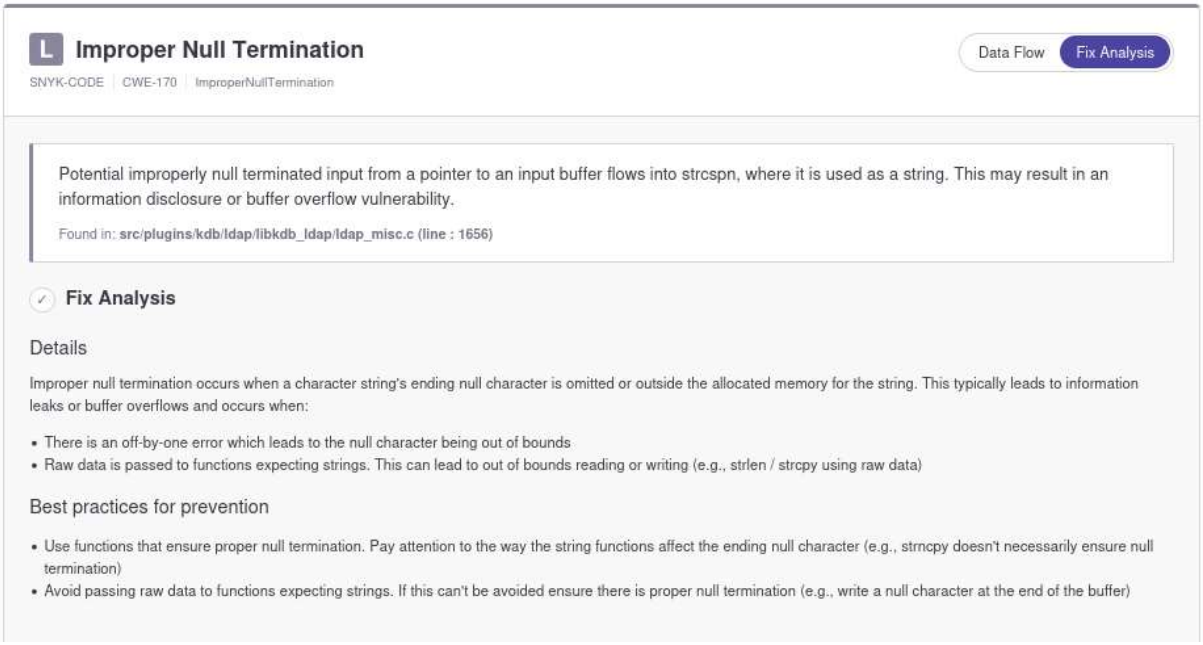

Figure 5: Snyk recommendation for fixing the null pointer dereference memory error.

Furthermore, there is a summary of all the findings that Snyk detected during the scan at the top of the report. In the case of Kerberos 5-1.16.1, Snyk found 1 high-, 404 medium-, and 118 low-rated issues, thereby demonstrating its effectiveness (and by association the effectiveness of automated static code analysis tools) at discovering sources of memory errors within supplied source code.

References

Camacho, R. (2023) *Detecting Memory Corruption in C and C++.* Available at: https://www.parasoft.com/blog/detecting-memory-corruption-in-c-and-c/ (Accessed: 2 February 2025).

CISA (2024) *Exploring Memory Safety in Critical Open Source Projects.* Available at: https://www.cisa.gov/sites/default/files/2024-06/joint-guidance-exploring-memory-safety-in-critical-open-source-projects-508c.pdf (Accessed: 2 February 2025).

Codacy (2023) *Static Code Analysis: Everything You Need to Know.* Available at: https://blog.codacy.com/static-code-analysis (Accessed: 3 February 2025).

CodeCruze (2024) *The Benefits of Static Code Analysis in Software Development.* Available at: https://www.linkedin.com/pulse/benefits-static-code-analysis-software-development-codecruze-wos2f/ (Accessed: 2 February 2025).

CVEdetails.com (2018) *Vulnerability Details : CVE-2018-5710.* Available at: https://www.cvedetails.com/cve/CVE-2018-5710/ (Accessed: 2 February 2025).

CVEdetails.com (n.d.) *MIT » Kerberos 5 : Product details, threats and statistics.* Available at: https://www.cvedetails.com/product/12666/MIT-Kerberos-5.html?vendor_id=42 (Accessed: 2 February 2025).

Gjomemo, R. et al. (2016) Leveraging Static Analysis Tools for Improving Usability of Memory Error Sanitization Compilers, *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, IEEE, Vienna, Austria, 01-03 August 2016. Available at: https://ieeexplore.ieee.org/document/7589812 (Accessed: 2 February 2025).

Granot, L. and Spektor, H. (2024) *Memory Safety: 5 Common Memory Bugs and How to Secure Your Systems.* Available at: https://sternumiot.com/iot-blog/memory-safety-5-common-memory-bugs-and-how-to-secure-your-system/ (Accessed: 2 February 2025).

Help Net Security (2024) *Transitioning to memory-safe languages: Challenges and considerations.* Available at: https://www.helpnetsecurity.com/2024/03/11/omkhar-arasaratnam-openssf-memory-safe-programming-languages/ (Accessed: 2 February 2025).

Incredibuild (2022) *Big Dev Build Times*. Available at: https://www.incredibuild.com/wp-content/uploads/2023/09/incredibuild_big_dev_build_times-1.pdf (Accessed: 2 February 2025)

Jiang, C. and Wang, Y. (2019) Survey on Memory Corruption Mitigation, *2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, IEEE, Chengdu, China, 15-17 March 2019. Available at: https://ieeexplore.ieee.org/abstract/document/8728974 (Accessed: February 2 2025).

Kaur, A. and Nayyar, R. (2020) 'A Comparative Study of Static Code Analysis tools for Vulnerability', *Procedia Computer Science,* 171, pp. 2023-2029. Available at: https://doi.org/10.1016/j.procs.2020.04.217

MIT (2024) *Kerberos: The Network Authentication Protocol.* Available at: https://web.mit.edu/kerberos/ (Accessed: 2 February 2024).

MITRE (2024) *CWE-125: Out-of-bounds Read (4.16).* Available at: https://cwe.mitre.org/data/definitions/125.html (Accessed: 2 February 2025).

MITRE (2024) *CWE-190: Integer Overflow or Wraparound (4.16).* Available at: https://cwe.mitre.org/data/definitions/190.html (Accessed: 2 February 2025).

MITRE (2024) *CWE-416: Use After Free (4.16).* Available at: https://cwe.mitre.org/data/definitions/416.html (Accessed: 2 February 2025).

MITRE (2024) *CWE-476: NULL Pointer Dereference (4.16).* Available at: https://cwe.mitre.org/data/definitions/476.html (Accessed: 2 February 2025).

MITRE (2024) *CWE-787: Out-of-bounds Write.* Available at: https://cwe.mitre.org/data/definitions/787.html (Accessed: 2 February 2025).

OWASP (2021) *Static Code Analysis.* Available at: https://owasp.org/www-community/controls/Static_Code_Analysis (Accessed: 2 February 2025).

Snyk (n.d) *Snyk Open Source.* Available at: https://snyk.io/product/open-source-security-management/ (Accessed: 2 February 2025).

Snyk (n.d) *What is a null dereference?.* Available at: https://learn.snyk.io/lesson/null-dereference/ (Accessed: 2 February 2025).

TechRepublic (2024) *CISA Report Finds Most Open-Source Projects Contain Memory-Unsafe Code.* Available at: https://www.techrepublic.com/article/open-source-projects-memory-unsafe-code-cisa/ (Accessed: 2 February 2025).

TechTarget (2021) *Kerberos.* Available at: https://www.techtarget.com/searchsecurity/definition/Kerberos (Accessed: 2 February 2025).

Wikipedia (2024) *Memory safety.* Available at: https://en.wikipedia.org/wiki/Memory_safety (Accessed: 2 February 2025).