Failure Analysis: Buffer Overflow Vulnerabilities

University of Florida

Table of Contents

[Introduction 3](#_Toc148157221)

[Failure in Computer Science 3](#_Toc148157222)

[1.1 Common Failures 3](#_Toc148157223)

[1.2 Failure Mitigation and Analysis 4](#_Toc148157224)

[Case Study 5](#_Toc148157225)

[2.1 Case Description 5](#_Toc148157226)

[2.2 Case Investigation 6](#_Toc148157227)

[2.3 Recommendations 7](#_Toc148157228)

[References 10](#_Toc148157229)

# Introduction

Building Software as a product pertains to building programs to solve real-life problems using technology. Software engineering is intertwined with many other industries such as the medical field, banks, and digital media platforms. Software engineers follow pre-established procedures and optimal practices to ensure that the product satisfies the customer while being efficient, reliable, and maintainable in a competitive environment.

# Failure in Computer Science

The pipeline that goes into a Software is huge and has many sections such as requirements analysis, design, coding, testing, deployment etc. That includes strict deadlines, financial constraints, and catering to clients. There are bound to be some errors that pass through the pipeline without being caught for various reasons, starting from poor project management to negligence in testing.

## 1.1 Common Failures

Errors are an unavoidable part of coding, whether that be in a personal project a singular student is creating, or whether it be an industrially made application. There are different classifications of these errors and in the software space, they are usually called “bugs”. Not all bugs are equally damaging to the infrastructure. Bugs are categorized by severity, with low-severity bugs primarily affecting the user interface, medium-severity bugs influencing functionality, high-severity bugs causing significant software malfunctions, and critical bugs rendering further testing impossible [11]. An example of a common bug which can cause complicated vulnerabilities and chances of memory leaks is the Integer Overflow to Buffer Overflow vulnerability (IO2BO), which arises from integer overflows during memory allocation calculations. This results in inadequate memory allocation, triggering potential buffer overflow issues [7].

## 1.2 Failure Mitigation and Analysis

Testing is very crucial in Software Engineering and the quality of the product and liability of the owner of any software depends on how thoroughly the product is tested and validated to match all requirements of the customer. Most of these testing types are preventive tests focusing on functionality and usability. Unit testing is a critical phase of software development where developers test individual components or units, typically using test automation tools like NUnit or JUnit to identify defects early [5]. White box testing involves examining the internal code, while gorilla testing thoroughly tests specific modules. Integration testing assesses how modules work together, like testing the integration between an airline website and payment processing. Gray box testing combines knowledge of internal code with external testing [5].

 Forensic tests are a way of letting buffer overflow errors occur in order to analyze that data and information to find out exactly what is causing the vulnerabilities. This research can then be used to detect future vulnerabilities in other code bases with better accuracy [2]. Considering edge cases and hypothetical situations during development can also help identify potential faults [5]. Diverse testing environments, including local development, developer testing, client testing, and production environments, play a critical role in identifying and preventing issues early in the development process. This approach with several checkpoints allows for more extensive testing and failure detection before the product reaches its final stages, establishing clear accountability for bug resolution, including responsible parties, and timeframes for addressing bugs, all of which is crucial for efficient bug management [11].

Software Testing standard protocols encompass several aspects of software testing including defining test completion criteria to determine when testing is considered finished, specifying test documentation and reporting procedures, establishing the level of independence for testing teams, selecting test design techniques, outlining test data requirements, defining the test environment, collecting relevant metrics during testing, and providing guidelines for retesting practices, conditions, and activities at each test level or type. These protocols ensure the systematic and comprehensive planning and execution of software testing activities, aligning them with organizational objectives and quality standards [10].

# Case Study

Cybersecurity is a complex and evolving field, from securing third-party software modules to combating buffer overflow (BOF) vulnerabilities. A buffer overflow is a prevalent software vulnerability that arises when a program tries to store an excessive amount of data in a buffer, exceeding its capacity, or attempts to write data beyond the bounds of a buffer [4]. These vulnerabilities are particularly associated with the C and C++ programming languages, encompassing both legacy and contemporary code. C and C++ are widely utilized languages in the industry and, due to their power and versatility, are susceptible to such errors [8].  In the context of third-party software, vulnerabilities within these modules have led to severe security breaches, posing a significant challenge [1].

## 2.1 Case Description

BOFs can occur and be taken advantage of in various ways. Attackers take advantage of BOF vulnerabilities and try to corrupt the Function Stack Frame (FSF) and overwrite data with malicious code which then allows them control over the software. By understanding the stack's layout and replacing the current FSF return address with a location containing malicious code, attackers can exploit the Buffer Overflow (BOF) vulnerability [3]. In other cases, they can be induced by both functions and loops, which are categorized as fun-BoFs and loop-BoFs, respectively. Fun-BoFs may arise from functions that incorporate loops in their internal code, typically occurring in specific cases. Functions that induce BoFs encompass I/O functions, memory manipulation functions, string manipulation functions, and memory allocation functions that allocate insufficient memory and loop-BoFs can result from loops containing data movement operations involving pointers or arrays that exceed the boundaries of allocated buffers [2]. Buffer Overflow can also be a result of integer overflows, specifically when an integer overflow occurs during memory allocation calculations, leading to insufficient memory allocation. This can trigger a subsequent buffer overflow when the inadequate memory A diagram of a problem

Description automatically generatedblock is used [7].

Figure 1: Buffer overflow causes and attributes [6]

2.2 Case Investigation

Before releasing a product, it is usually tested extensively for security and Buffer Overflow vulnerabilities, as failing to do so might result in a costly data breach. Common Testing Methods used in the industry to induce and test the extents of these vulnerabilities are Security Testing, including Penetration Testing, which is an authorized cyberattack performed to find the weak parts of a program [5]. To prevent buffer overflow (BOF) vulnerabilities, many measures can be taken, including optimizing compilers and multicore processors to reduce performance concerns, allowing developers to focus on algorithmic enhancements rather than extensive BOF checks. Various internal techniques, both passive and active prevention, exist for mitigation of BOF risks, including bounds checking and memory violation interruptions [6].

Negligence during testing could lead to oversight of vulnerabilities if the procedure outlined in the standard is not followed, which is, defining testing conditions, going through peer review, analyze residual risks, all throughout test completion [10]. However, for buffer overflow vulnerabilities, sometimes it is difficult for programmers to detect them. Such as in the case of Integer Overflow, of which detecting and fixing is not a straightforward task for programmers, and it requires a deep understanding of Integer Overflow to Buffer Overflow (IO2BO). Even when programmers are aware of this vulnerability, safeguarding programs from integer overflows remains a complex and error-prone process. Even for diligent programmers, their chosen method to check integer overflow may work in some cases but fail in others, particularly as changes in program logic may render their efforts useless [7]. Overall, since buffer overflows can be induced from various parts of code as seen above, including loops, functions, integers, and memory manipulations, it is extremely easy for some vulnerabilities to remain in any code no matter the precautions [1].

2.3 Recommendations

Buffer overflow (BOF) is a persistent and profoundly damaging security vulnerability in today's complex computing systems. The increasing complexity of code often results in unnoticed vulnerabilities, leading to significant damage. Even the slightest vulnerability can result in substantial financial, intellectual, or data loss. Notable examples include the SWIFT code hack that cost US $81 million and the Home Depot data breach that exposed 56 million credit and debit card details. The Equifax data breach in 2017 affected approximately 147 million individuals. BOF, despite being a thirty-year-old vulnerability, remains one of the most common and dangerous security breaches. It was the leading cause of security breaches in the decade following its discovery, earning it the title of the "vulnerability of the decade." The Common Vulnerabilities and Exposures (CVE) list in 2019 reported over 400 vulnerabilities related to buffer overflows, and as of May 2021, the number has exceeded 13,700 [3].

A graph of number of years

Description automatically generated with medium confidence

Figure 2: Buffer Overflow Statistics [3].

There are several mitigation strategies against Buffer Overflow used by programmers, and some are better than others. Empirical studies reveal that a considerable portion of real-world BOFs is induced by loops, inspiring the development of DBloop, a static detection method. DBloop analyzes loop-induced data-movement lengths and employs constraint solving to accurately pinpoint BOFs, outperforming existing tools significantly [2]. A prominent technique that is easy to deploy is called NX, which prevents malicious code from running in certain parts of a program's memory. NX can be set to 0 or 1 with designated tasks so that it does not let change come from outside. One of the notable advantages of the NX bit is that it doesn't mandate the recompilation of software programs. Instead, the operating system harnesses the NX bit's power to label specific memory areas, such as the stack and heap, as non-executable. This proactive measure blocks code injection attacks. This technology does not require anything special from the user’s side so can be used by anyone seeking to protect their technology [3]. Another technique, ASLR, confuses attackers by randomly placing program elements in memory. Additional programs like PIE, RELRO, and StackGuard provide extra layers of security to make it even more challenging for attackers [3]. Some other general solutions deal with BOF prevention involves bounds checking, considering control and data dependencies in vulnerable statements, implementing various features to handle overflows, and performing adequate input validation, with static code attributes and data mining used to provide probabilistic examinations when dynamic analysis is inconclusive [9]. However, despite all this, BOFs endure as a persistent cybersecurity challenge, underscoring the ongoing necessity for robust protection measures and innovative approaches in this dynamic landscape.

# 

# References

[1] S. Ali, P. Anantharaman, and S. W. Smith, “Armor within: Defending against vulnerabilities in third-party libraries,” 2020 IEEE Security and Privacy Workshops (SPW), 2019. doi:10.1109/spw50608.2020.00063

[2] P. Luo et al., “Static detection of real-world buffer overflow induced by Loop,” Computers &amp; Security, vol. 89, p. 101616, 2020. doi:10.1016/j.cose.2019.101616

[3] M. A. Butt, Z. Ajmal, Z. I. Khan, M. Idrees, and Y. Javed, “An in-depth survey of bypassing buffer overflow mitigation techniques,” Applied Sciences, vol. 12, no. 13, p. 6702, 2022. doi:10.3390/app12136702

[4] “Buffer overflow,” Buffer Overflow | OWASP Foundation, https://owasp.org/www-community/vulnerabilities/Buffer\_Overflow (accessed Oct. 6, 2023).

[5] Vijay, “Types of software testing: Different testing types with details,” Software Testing Help, https://www.softwaretestinghelp.com/types-of-software-testing/ (accessed Oct. 6, 2023).

[6] P. E. Black and I. Bojanova, “Defeating buffer overflow: A trivial but dangerous bug,” *IT Professional*, vol. 18, no. 6, pp. 58–61, 2016. doi:10.1109/mitp.2016.117

[7] C. Zhang, W. Zou, T. Wang, Y. Chen, and T. Wei, “Using type analysis in compiler to mitigate integer-overflow-to-buffer-overflow threat,” *Journal of Computer Security*, vol. 19, no. 6, pp. 1083–1107, Dec. 2011, doi: <https://doi.org/10.3233/jcs-2011-0434>.

[8] H. Shahriar, H. M. Haddad, and I. Vaidya, “Buffer overflow patching for C and C++ programs,” *ACM SIGAPP Applied Computing Review*, vol. 13, no. 2, pp. 8–19, Jun. 2013, doi: <https://doi.org/10.1145/2505420.2505421>.

[9] B. M. Padmanabhuni and H. B. K. Tan, “Auditing buffer overflow vulnerabilities using hybrid static–dynamic analysis,” *IET Software*, vol. 10, no. 2, pp. 54–61, Apr. 2016, doi: <https://doi.org/10.1049/iet-sen.2014.0185>.

[10] “IEEE/ISO/IEC International Standard for Software and systems engineering–Software testing–Part 3:Test documentation,” *ISO/IEC/IEEE 29119-3:2021(E)*, pp. 1–98, Oct. 2021, doi: https://doi.org/10.1109/IEEESTD.2021.9591577.

‌[11]“16 Types Of Software Bugs In Software Testing[Updated 2023]-ThinkSys Inc,” *thinksys.com*, Mar. 14, 2022. https://thinksys.com/qa-testing/types-software-testing-bugs/

‌

‌

‌

‌