Dissecting Buffer Overflow Defenses: Evaluating Effectiveness and Reducing Risks

November 18, 2024

Purple Team
Team Leader: Feizal Mjidho (w10001167)
Daniel Ainsworth (w10142713)
Dylan Burnside (w10094548)
Khadichabonu (Khadi) Valieva (w10118633)
Kendall Jasper (w10055702)
Rhishika Guragain (w10170428)

Table of Contents

Executive Summary

Our project was centered on the evaluation of the buffer overflow defenses as implemented by Blue Team 3. The goal of the evaluation was to understand their defenses, determine their effectiveness and provide a report of any remaining unaddressed risks and the strategies to mitigate them. At this time, the status of our project is complete, the objectives and goals that were set forth have been successfully accomplished. In the course of this project, we were able to conduct in depth research into buffer overflows, and its known vulnerabilities and defenses. Additionally we were able to perform a thorough analysis on the blue team's code, and identified their multilayered defenses as well as some areas of improvement.

Project Overview

Purpose: Our project will assess how effective Blue Team's defense mechanisms are against buffer overflow vulnerabilities in a C++ program by reviewing, testing, and analyzing the defenses in their code.

Goals: Our team's goal was to understand most common buffer overflow vulnerabilities, how they occur and the risks they have to software security. Additionally to understand the protective and preventive measures against overflow vulnerabilities. Lastly to identify both mitigated vulnerabilities in the defense strategies, and the unaddressed risks resulting in an improvement report for Blue Team 3.

Objectives: We had three objectives for this project. The first was to conduct research on buffer overflows and its associated vulnerabilities and mitigation strategies. Our research was primarily focused on C++ as that would be the language that Blue Team was coding in. After our research, our second objective was to perform the code review through a mixture of static analysis and dynamic analysis. Our expected outcome was to be able to understand exactly what the code did, which sections mitigated which vulnerabilities and identify unaddressed risks. Lastly our third objective was to combine our results from the review into a comprehensive list of potential risks and the strategies for handling them.

Work Accomplished

Objective I: Research

In order to complete our goal and do a proper, thorough and in depth code analysis, we first started with researching the topic of buffer overflows. The Open Worldwide Application Security Project (OWASP) defines a buffer overflow as a condition that exists when a program attempts to put more data in a buffer than it can hold or when a program attempts to put data in a memory area past a buffer [1]. Writing outside the bounds of a buffer (a block of allocated memory) can corrupt data through overwriting pre-existing data and also lead to the program crashing or being stuck in an infinite loop. Another consequence of buffer overflows is with access control, where an attacker can load malicious instructions in the overflow that then becomes executed to bypass aspects of the program's security.

After gaining an understanding on buffer overflows, we then proceeded to research known vulnerabilities and the mitigation strategies to handle them. There are plenty of other vulnerabilities, like heap overflow that corrupts the program data by injecting malicious code into the dynamic memory. Or 'line of stack smashing' that overwrites adjacent memory, including the return address to corrupt the control flow and crash the entire program. (Other examples include Remote Code Execution (RCE), Memory Leaks, Cross-Site Scripting, Privilege Escalation, Man-in-the-Middle (MITM) Attacks, Race Conditions and so on.) While there can be many vulnerabilities as we create and execute software, there are many strategies to mitigate and prevent data breaches, loss, privacy violation etc. by keeping software vulnerabilities at bay.

Some defensive and mitigative measures we found include input and size validation and sanitation (validates user input to prevent malicious payloads and also helps prevent any overflows), parameterized queries and prepared statements (separates query logic from user provided input to prevent the query structure to change), use of memory safe languages like java, memory management defenses that detect and prevent exploitation of memory vulnerabilities, logging and monitoring to detect changes to prevent attacks like salami attacks, and patching and updating to fix bugs and vulnerabilities regularly.

Lastly our research also focused on tools and methodology for analyzing the Blue Team's C++ code. We determined that the best way to perform our evaluation was through code reviews, more specifically through a combination of static and dynamic analysis. In our research of static analysis, we attempted to find static code analysis tools through the use of OWASP[2]. Through OWASP, we discovered that there are a lot of tools but there are different advantages and disadvantages to utilizing them[3]. Additionally we discovered some important distinctions with the tools.The two most impactful to our project were the specific platforms for each tool and whether they were open source or commercial. We were able to come up with a list of tools, such as Flawfinder and Cppcheck but ultimately decided against them due to the limitations and the scope of the code we are analyzing. Due to the small scope of code, we determined that it would

be best for our team to personally analyze it to ensure a more thorough understanding for ourselves.

Table 5: Advantages To Using Source Code Scanners

Advantage	Explanation
Reduction in manual efforts	The type of patterns to be scanned for remains common across applications, computers are better at such scans than humans. In this scenario, scanners play a big role is automating the process of searching the vulnerabilities through large codebases.
Find all the instances of the vulnerabilities	Scanners are very effective in identifying all the instances of a particular vulnerability with their exact location. This is helpful for larger code base where tracing for flaws in all the files is difficult.
Source to sink analysis	Some analyzers can trace the code and identify the vulnerabilities through source to sink analysis. They identify possible inputs to the application and trace them thoroughly throughout the code until they find them to be associated with any insecure code pattern. Such a source to sink analysis helps the developers in understanding the flaws better as they get a complete root cause analysis of the flaw
Elaborate reporting format	Scanners provide a detailed report on the observed vulnerabilities with exact code snippets, risk rating and complete description of the vulnerabilities. This helps the development teams to easily understand the flaws and implement necessary controls

Table 6: Disadvantages To Using Source Code Scanners

Limitation	Explanation
Business logic flaws remain untouched	The flaws that are related to application's business logic, transactions, and sensitive data remain untouched by the scanners. The security controls that need to be implemented in the application specific to its features and design are often not pointed by the scanners. This is considered as the biggest limitation of static code analyzers.
Limited scope	Static code analyzers are often designed for specific frameworks or languages and within that scope they can search for a certain set of vulnerable patterns. Outside of this scope they fail to address the issues not covered in their search pattern repository.
Design flaws	Design flaws are not specific to the code structure and static code analyzers focus on the code. A scanner/analyzer will not spot a design issue when looking at the code, whilst a human can often identify design issues when looking at their implementation.
False positives	Not all of the issues flagged by static code analyzers are truly issues, and thus the results from these tools need to be understood and triaged by an experienced programmer who understands secure coding. Therefore anyone hoping that secure code checking can be automated and run at the end of the build will be disappointed, and there is still a deal of manual intervention required with analyzers.

Objective II: Code Evaluation

Our code evaluation was split into two different parts, static analysis and dynamic analysis. In our static analysis, we performed a code review on the Blue Team's code and dissected each function and line of code. In our dissection, we learned that the 4 different functions (secureFunction, secureStringConcat, secureDynamicMemory, and stimulateStackCanary) focused on mitigating an individual buffer vulnerability. Additionally, the blue team also incorporated safer and more secure functions such as strncpy and strncat instead of the more dangerous strcpy and strcat. Through our static analysis we were able to get a better understanding of the implemented defenses and thus were able to better prepare for the dynamic analysis.

In our dynamic analysis for the C++ code given by Blue Team 3 using a method that should prevent buffer overflow security incidents by avoiding insecure methods of managing memory and also in string manipulation. They used different functions developed to handle strings in a safe manner. While the code is well-designed and thoroughly through, there are some areas that require improvement for robustness and maintainability. First of all, when we ran the code, it crashed because of improper implementation of _TRANCATE. We had to modify the code a little in order to get the actual output and test the use cases. Something also to add, which is good about the rules, is the application of a threshold test. For example, in the secureFunction, there's a check that the length of the input string is greater than the size of the destination buffer before attempting any copy operations. This approach is necessary to prevent buffer overflows - a common security vulnerability. Also, the code consists of standard functions like strncpy_s and strncat_s that are designed to behave exactly as these functions do on systems where they are not supported. This has not only improved portability but at the same time maintained a level of security that is usually compromised during a normal string manipulation. One thing to add, it might be a good practice to add a couple functions to check the edge cases for these functions in order to avoid certain problems later on. Another strong point is in memory management. The code checks if malloc was successful in allocating memory, which will allow the program to handle situations when memory is not available. It is good practice in C++ programming since it can help prevent crashes due to null pointer dereferences.

Despite the above-mentioned strengths, there are several areas in which the code could be further improved. Among the most serious concerns is that of input handling. The version at its current stand makes use of std:cin for reading in user input. In such a case, buffer overflow may occur if the input provided goes beyond the buffered length. This may be better avoided through the use of std::getline.

```
std::string input;
std::cout << "Enter a string: ";
std::getline(std::cin, input);
```

Another area for improvement is the null termination of strings. Ensuring that the destination buffer is always null-terminated after copying is crucial, especially when strings get truncated. Also, the return value handling in the custom functions could be designed more robustly, instead of just printing an error message, the program could take corrective actions or exit more neatly providing a better user experience. Last but not least, the use of numbers like buffer sizes can detract from code readability. Defining the sizes as constants or using so-called constexpr would improve clarity.

```
constexpr size_t BUFFER_SIZE = 10;
```

Finally the code could be tested for corner cases such as for an empty string or a very large, long input (when the buffer size is not defined). This can definitely provide confidence in how the program behaves and lowers the probability of it crashing.

Overall, although code shows a very good job regarding buffer overflow vulnerabilities, the above-mentioned analysis shows some issues as well as areas of improvements that could be done on the code provided by Blue Team 3. The code can become even more secure and user-friendly by improving safer inputs, managing errors more effectively and following best practices in coding standards.

Objective III: Evaluation Report

Thanks to our analysis, we discovered the list of mitigation strategies that Blue Team implemented. There was the utilization of safer library functions as discussed earlier but also the 4 void functions mitigated against several vulnerabilities. The first function, the secureFunction, incorporated boundary checking of the user input against the buffer size. The second one, the secureStringConcat mitigated against concatenation vulnerabilities. Thirdly, the secureDynamicMemory function protects against dynamic memory allocation vulnerabilities. Lastly the stimulateStackCanary function protects against the stack smashing attack through utilizing a canary value to detect possible attacks.

While Blue Team did a great job layering their defenses from simple methods like using safer functions to more complex techniques like stack canaries, there was still an area of risks that was unaddressed. The main source of improvement would be with further fortifying the memory addresses. Though there was some memory based mitigation already implemented with the stack canary function, it could be further secured with techniques and tools like Address Space Layout Randomization (ASLR) and address sanitization. Further details on the risks and mitigation strategies can be found in the risk management section of this report.

Future Work (If any)

As we have completed all of our objectives and accomplished all our goals, we have no future work to expand on. Our evaluation and risk mitigation report will be shared with the blue team and thus concluding our work.

References

[1] “Buffer overflow,” Buffer Overflow | OWASP Foundation, https://owasp.org/www-community/vulnerabilities/Buffer_Overflow#:~:text=At%20the%20code%20level%2C%20buffer,the%20buffers%20they%20operate%20upon. (accessed Nov. 17, 2024).

[2] “Source code analysis tools,” Source Code Analysis Tools | OWASP Foundation, https://owasp.org/www-community/Source_Code_Analysis_Tools (accessed Nov. 17, 2024).

[3] “Code review guide,” owasp.org, https://owasp.org/www-project-code-review-guide/assets/OWASP_Code_Review_Guide_v2.pdf (accessed Nov. 17, 2024).

[4] “Address space layout randomization,” IBM, https://www.ibm.com/docs/en/zos/2.4.0?topic=overview-address-space-layout-randomization (accessed Nov. 17, 2024).

[5] S. S. R. Team, “Buffer overflow attacks in C++: A hands-on guide,” Snyk, https://snyk.io/blog/buffer-overflow-attacks-in-c/ (accessed Nov. 17, 2024).

[6] D. Klemba and D. Czarnota, “Understanding addresssanitizer: Better Memory Safety for Your Code,” Trail of Bits Blog, https://blog.trailofbits.com/2024/05/16/understanding-addresssanitizer-better-memory-safety-for-your-code/ (accessed Nov. 17, 2024).

Risk Management

As mentioned in our evaluation report, within the Works Accomplished section of the report, we were able to identify a potential risk within the memory address. At its core, buffer overflows are a vulnerability related to overflowing the allocated size of memory, which leads to overwriting data and corrupting existing information or worse. Though there were a lot of well layered defenses, there are still some risks remaining that could be fixed with ASLR and address sanitization.

First let's discuss ASLR, how it works and how it provides added security. ASLR is a mitigation technique that randomizes memory addresses in the operating system to make it harder for attackers to exploit the addresses [4]. While there are simple cases of buffer overflow where the user enters input that is way beyond the size of the buffer, there are more complex attacks where users utilize knowledge of the program’s memory addresses to exploit the program in several malicious ways. However this sophisticated attack can be mitigated with ASLR, which is used to increase the difficulty by randomizing the addresses making it harder for the attackers to effectively exploit.

Unlike with ASLR, address sanitization is more of a preventative measure than an active defense. Address sanitization refers to using an address sanitization tool like AddressSanitizer (ASan) to detect buffer overflows within a program [5]. ASan is among the most popular tools that is also supported by the major compilers such as Visual Studio and GCC. It works by inserting checks around memory accesses during compile time and then stops the program when it detects any improper memory access [6]. In its encounter with improper memory access, ASan displays diagnostic information about where the error happened as well as an estimation of the type of error. Though this technique is not useful in mitigating an attack as it’s being attempted, it is immensely helpful in securing the program during development. It allows the

developer to diagnose any remaining memory access vulnerabilities and resolve them thus leading to a more secure program/application.

Team Member Roles and Responsibilities

Team Leader

- Feizal Mjidho: 30% of work completed
    - Responsibilities: Overseen the project, ensured deadlines were met, facilitated communication with Blue Team 3, assisted with code based research, performed the static analysis and wrote code evaluation report

Team members:

- Rhishika Guragain: 20% of work completed
    - Responsibilities: Meeting notes, proper formatting of reports and research on other existing vulnerabilities and defense strategies.
- Khadi Valieva: 20% of work completed
    - Responsibilities: Conducted code based research, performed the dynamic analysis and wrote the code evaluation report
- Dylan Burnside: 10% of work completed
    - Responsibilities: Researched strategies and defenses for the code
- Daniel Ainsworth: 10% of work completed
    - Responsibilities: Researched known buffer overflow vulnerabilities and attacks
- Kendall Jasper: 10% of work completed
    - Responsibilities: Researched known buffer overflow defenses and mitigation strategies

Meeting Documentation

Meeting Record:

- These are all the meetings we had regarding the project. In addition there was frequent communication through the team's discord and groupMe, to update on progress, distribute tasks and further resolve any minor issues.
- Introductory Meeting on Sept. 5th (2:30 pm to 3:20 pm) All members were in attendance, team leader was selected and project topics were discussed.
- Project Selection Meeting on Oct. 15th (2:45 pm to 3:22 pm). All members were in attendance, project selection was discussed and narrowed down along with potential concerns.
- Project Abstract Meeting on Oct. 20th (3:30 pm to 4:15 pm). All members were in attendance, finalized project selection, team contract was agreed upon and tentative project structure was created.

- Project Planning and Joint Project Meeting (with Blue Team 3 and Instructor) on Oct. 24th (2:30pm to 3:35pm). All members except for Rhishika were in attendance (absent due to illness but informed the team beforehand). Contact and communication with the Blue Team was established, project concerns and expectations were established. Additionally research topics were selected and distributed.
- Research Progress Report on Nov 1st (8:00 pm to 8:30pm). All members were in attendance. Progress was reported for every member's research tasks. Every member presented their research and their findings as it related to the overall project. All members were encouraged to research C++ in preparation for the code analysis.
- Project Write Up Meeting on Nov. 15th (6:00 pm to 6:30pm). All members were in attendance. Tasks for the report write up and presentation were distributed. Progress on Blue Team code analysis was reported on. Questions and concerns about the project were addressed and solved. Plan for presentation and presenting sequence was established.