Cloud-Based Software Tool to Help Open-Source Computing Community with The Detection of Source Code Vulnerabilities and Insecure Patterns

Matthew Block, Andrew Nimmo, Benjamin Barcaskey, Saleh Alnaeli

Dept. of Mathematics, Statistics, and Computer Science
University of Wisconsin-Stout
Menomonie, Wisconsin, 54751
{blockm1048,nimmoa5845,barcaskeyb2473}@my.uwstout.edu, alnaelis@uwstout.edu

Ian Gilbert, Zaid Altahat

Dept. of Computer Science

University of Wisconsin-Parkside

Kenosha, Wisconsin, 53141

gilbe016@rangers.uwp.edu,

altahat@uwp.edu

Abstract

The open-source model has been widely accepted in the game development domain. When it comes to the source code, the security and the vulnerabilities of the developed systems are still some of the community's biggest concerns. Software vulnerabilities are introduced to the systems in a variety of different ways. The most prevalent way is how programmers write their source code. This paper presents an empirical study on the presence and distribution of some known vulnerable patterns in the open-source game engine Godot. This system is statically analyzed with a cloud-based tool that was developed by the authors. The goal is to help software developers identify vulnerabilities in their systems as these vulnerabilities pose potential risks. Additionally, a historical study over a five-year period was conducted on Godot. The results show that the vulnerabilities per line of code has steadily decreased, but the types of vulnerabilities have changed significantly over time.

1. Introduction

The prevalence of open-source systems has been increasing throughout the years in the game development computing domains [7-11]. With open-source systems becoming more common, the importance of maintaining a high level of security on those systems steadily rises. The question commonly proposed by both end-users and professionals is whether it is safe to assume that open source systems written and managed by well-known computing companies and institutions are free of security vulnerabilities. In this work, the authors try to answer the aforementioned question. Due to the nature of open-source systems within the game development domains, many developers from a variety of backgrounds are able to contribute to these systems [7-15]. This can lead to security oversights due to the lack of experience with C/C++ secure coding practices

The usage of open-source systems has become commonplace in everyday programs. This includes a wide array of programs, such as web browsers, operating systems, deep learning libraries, game engines, medical software, and others. With this high usage, there is an increased risk for vulnerabilities, such as the unauthorized access of user's data. The security of open-source systems would thus greatly benefit from a statistical analysis of open-source system's source code and from vulnerability reports that allow the developers to find and correct vulnerabilities before they become security breaches. Preventing security breaches is a high priority in the current state of the world as breaches are costing businesses more and more.

Detecting source code security vulnerabilities through static analysis has the potential to reduce security threats significantly [5, 6]. The security vulnerability detection tool that will be presented below promotes safe coding practices by detecting issues and reporting them back to the user. This tool detects security flaws that are well-known within the software development community and presents them as an array of graphs that reveal the most prevalent areas of vulnerability.

In the following study, a variety of vulnerabilities will be used to present a historical study. The vulnerabilities are often located by searching for specific patterns, called vulnerability patterns. Possible trends from the historical study will be explained. Three research questions will be answered:

RQ1: What is the distribution of the studied vulnerable patterns found in the open-source game engine Godot?
RQ2: What is the most prevalent pattern amongst the studied vulnerabilities?
RQ3: How does a popular game engine such as Godot evolve overtime in terms of vulnerabilities at the source code level?

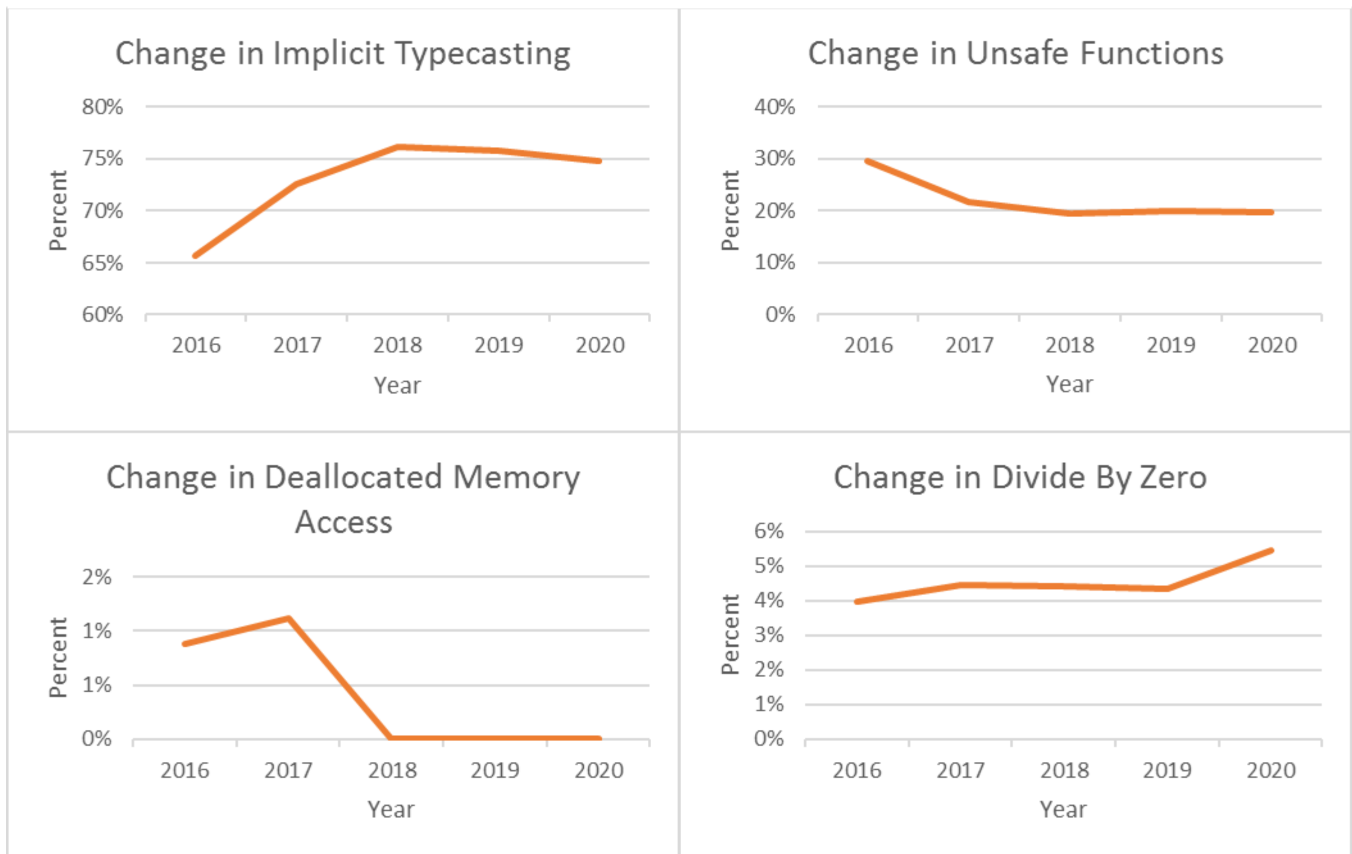


Figure 1. Change in vulnerability percentages by year

The analysis is done completely on the source code level. That is, the studied system is checked out and the source code is transformed into a more parsable format and results are tabulated and used to derive interactive reports to give a practical idea about the status of the system.

The remainder of this paper is organized as follows. Section 2 presents the related work. Section 3 describes the methodology and the techniques used in this study to detect the source code vulnerabilities, and how the analysis is performed. Information about the studied system is presented as well. Section 4 describes the scalable cloud-based tool used in this study and its features and limitations. Section 5 presents the findings of our study of Godot, an open-source game engine. It includes a discussion of the findings, and the historical trends revealed by the study, followed by threats to validity and challenges in section 6, and potential improvements and future work, and conclusions in sections 7 and 8.

2. Background and Related Work

The tool mentioned above relies heavily on source code to XML transformation. The tool srcML is used to convert code to XML rapidly, even faster than a compiler at a rate of

over 25 KLOC/sec [1]. srcML is able to markup source code with XML which allows for intelligent searching of known unsafe patterns in codebases. For example, srcML wraps all function calls with a <call> tag, thus allowing the quick and efficient gathering of every function call in a codebase. Those calls can then be quickly analyzed and compared to a list of known unsafe functions. Some powerful features of srcML include the lossless conversion of code. Any code converted to srcML can be transformed back into the base code perfectly. Additionally, the srcML transformed code can be queried efficiently using XPath and transformed using XSLT for source code modification. The srcML tool has also been used in a variety of contexts, such as reverse engineering class and method stereotypes, syntactic differencing, and the analysis of large systems [1].

A previous study of over one million lines of code was conducted, only detecting unsafe function calls. Dr. Saleh Alnaeli and his team developed a tool that checked for safe and unsafe functions calls, functions with probable side effects, and a variety of control flow statistics. In addition, the number of unsafe functions tends to increase over the years which suggests a degrading of safe security practices [2]. Using the foundation provided by this previously mentioned tool, a new tool has been created for this study that improves upon that foundation. The new tool utilizes a multi-threaded environment and, by doing so, can parse through the code significantly faster. The tool will be publicly available on the cloud [4].

3. Methodology for Detecting Vulnerabilities

The new tool is used to run static analysis on Godot by initially transforming the source code into XML using the open-source srcML tool [1]. Godot is an open-source system comprising more than 2 million lines of source code, as shown in TABLE I. By transforming the code into XML, various XML tree parsing techniques can then be used to extract information about the structure of the source code in a straightforward and flexible manner. After the source code is transformed into XML, the vulnerability analysis tool is run on the XML file. Utilizing a multi-threaded architecture, the tool rapidly evaluates the XML output, compiles a list of vulnerabilities, and summarizes them into easily readable charts and graphs.

A major security concern is race conditions [3]. The vulnerability analysis tool checks for several race conditions under sections of code that run in parallel. This includes OpenMP parallelized for-loops, std::thread parallelized functions, and p-thread parallelized functions. All variable declarations in the parallelized block are collected. Then all variable usages are gathered. Each variable usage is assessed to determine if a modifying operation is being applied to the variable. For example, the statement, “x += 2,” would count as a modifying operation. After a list of modified variables is complete, each variable is compared to the list of declared variables in the parallelized block to determine if any were declared outside of the block. Finally, any modified variables confirmed to be declared outside the block are marked as potential race conditions. False positives due to reduction, atomic, locks, critical, and other thread safety patterns, are ruled out.

System	Domain	Lines of Code	Files
Godot 2020	Game Engine	2337653	5010

Table 1. Characteristics of the studied systems used in this study, names, purpose, number files, and loc

Several more prevalent and highly exploitable security concerns are buffer overflows, unsafe functions, division by zero, deallocated memory access, and implicit type casting [3]. For instance, the tool detects a specific case of buffer overflow in which a stream insertion operator “<<” is used to direct the contents of a stream into a fixed-size array without any checks on the size of the stream contents. Functions that are considered unsafe by the tool are functions that have been deemed unsafe by the research community [2]. Division by zero can lead to unexpected exceptions, which can be considered a security vulnerability. Deallocated memory access can allow unauthorized access of data. Implicit type casting is detected when data is lost due to type casting. These patterns are detected by the tool because of the vulnerabilities they can pose to a system.

4. The Cloud-Based Tool Used in This Study

The tool introduced in this paper is a cloud-based software tool that was developed by the authors and scheduled to be launched as an open-source project in the near future. That is, a publicly available web application has been developed to encourage the software-development community to perform vulnerability checks more frequently. The tool is scalable and provides very interactive reports. These reports output the location of issues to allow developers to fix issues as they arise.

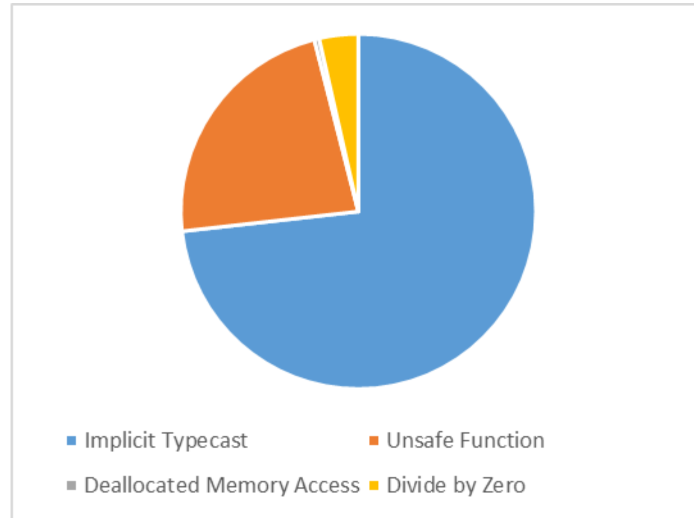
The general use case will be for a developer to frequently upload their tool to the web app. They then receive the output formatted into various summary graphs for a quick explanation as to the current state of the system. The developer will be able to download the full report, which lists the location and type of each vulnerability found so that the developer can begin verifying that the results are not false positives and can attempt to fix them.

5. Results and Discussion

The following are the results of a historical analysis of the Godot open-source game engine that was conducted by the tool utilizing the extremely fast computing power of the cloud. This analysis involved the cloud-based tool processing the progress of this open-sourced system over the course of five years in one-year increments. Within an elapsed 4.5 minutes of runtime, the tool analyzed over 9.2 million lines of code resulting in 659

implicit typecasts, 205 unsafe functions found, 0 race conditions detected, 4 deallocated memory accesses, 0 buffer overflows, and 41 potential divisions by zero.

Figure 2. Distribution of vulnerable patterns



To answer RQ1, as depicted in Figure 2, unsafe functions and implicit type casting makes up the majority of the distribution with division by zero and deallocated memory accesses making up a smaller percentage of the vulnerabilities. Buffer overflows and race conditions were not included in the distribution of Figure 2 because none were found. The subcase of buffer overflows detected were expected to be hard to find as manual investigations only found cases very sparingly.

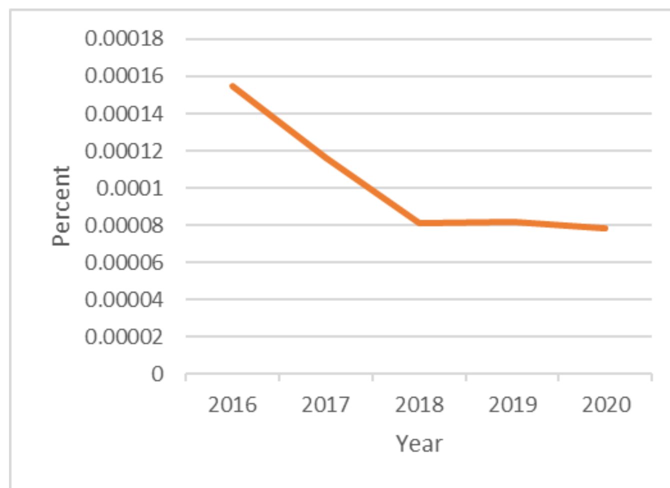


Figure 3. Ratio of Vulnerabilities per Line of Code Change

The number of race conditions found was expected to be extremely low because of the system-crashing errors that generally arise from them. Catching any undetected race conditions is extremely useful for large systems where maximum uptime is critical.

To answer RQ2, the most prevalent pattern is implicit type casting, as depicted in Figure 2. This was expected as some developers consider it a feature.

To answer RQ3, Godot has seen a steady decline in vulnerabilities per LOC over time, as seen in Figure 3. Implicit type casting errors made up 66% of the vulnerabilities in 2016, and rose to 76% by 2018, later dropping slightly to 75%. Divide by zeros errors have also increased from 4% to 5%. Unsafe functions and deallocated memory accesses both declined from 2016 to 2020.

Figure 1 demonstrates the ratio of a specific type of vulnerability relative to the overall number of vulnerabilities per year. The figure highlights that the types of vulnerabilities do not vary significantly from year to year. Generally, the percentage of implicit type casting increased while the percentage of unsafe functions decreased.

6. Threats to Validity

For most of the static analysis security tools and source code analyzers in the market and computing industry, the rate of false-negative (missed vulnerabilities) and false-positive cases have always been the most challenging part [1]. The security vulnerability detection tool is not an exception. That is, the main threat of the tool is undetected vulnerabilities. While the tool is constantly being improved and developed, there still is a chance that it may not catch a certain case or simply not check for a case at all. On the other hand, the tool could also return false positives, giving inaccurate results to the user and depreciating the credibility of the tool.

One major threat to the validity of the tool is that srcML only parses C, C++, C#, Objective C, and Java code. While a significant amount of security-sensitive code like enterprise applications, authentication servers, and cryptography software is coded in C/C++ and Java, this limits the possibilities of the tool to expand to languages such as Python, Typescript/JavaScript, and/or PHP. While this is not an issue for the near future given that the goal is to detect as many vulnerabilities in C and C++ code, this does limit the extensibility of the tool.

The tool also picks up vulnerabilities in source code that would never be executed. While analyzing dead code is still important, it can skew the end results prioritizing non-impactful issues over other more important issues. While the tool in its current state does not detect dead code, there are plans to implement this feature in the near future to implement this feature.

7. Future Work

The tool will be further refined and expanded to allow for better detection of currently covered vulnerabilities, detection of new types of vulnerabilities, and the ability to run

the tool on larger codebases as quickly as possible by better-utilizing parallelization. In addition, refining the current detection capabilities to reduce the number of false positives is a major objective in future development. Due to the extensive number of edge cases and the cost of computing power, the tool will never be able to achieve complete accuracy. However, there is a point where the tool will be able to guide users to be more cautious about how they write future code.

The tool can also be used to access more open source systems in other fields to see the correlation between an industry sector and the type of secure programming standards being performed. This can help other research groups and academia know common points of vulnerability in code overall, improving the way that code is written. Additionally, results from analyzed systems will be stored long-term so that developers can keep track of the prevalence of vulnerabilities in the projects. This can be used to analyze historical trends within a development cycle. When used within a company, it can be used to teach developers better development processes to avoid the usage of vulnerable code within systems.

Another plan for the tool is to return the portions of a repository which are never accessed, also known as dead code. Since dead code is never used and does more harm than good for source code security, it would be useful to find this code so developers can patch the codebase and reduce the size and vulnerabilities of the source code. This is extremely useful for repositories where hundreds of people are working on a single large repository for many years, especially if the project has gone through lots of large-scale refactoring operations.

8. Conclusion

In this empirical investigation, the open-source game engine Godot was examined and checked for the presence of known vulnerabilities patterns at the source code level. The patterns used in this study are unsafe functions, race conditions, buffer overflows, deallocated memory access, divide-by-zero errors, and implicit type-casting. The study also introduced a new release of a cloud-based tool that was designed and used in prior investigations. The tool has great potential to help our computing communities detect some common source code vulnerable patterns early on. The tool was proven to be very scalable via an empirical study that was described above. Historical trends on the prevalence of the vulnerabilities listed above were presented that highlighted the changes in Godot. The majority of results show the continued usage of vulnerable patterns that are already known to be harmful by the community from both academia and industry.

Our tool can help improve the quality, robustness, security, and reliability of software systems that are used on a daily basis. The tool is hosted on the cloud and can be used to observe how the systems evolve over time with respect to the source code vulnerabilities. This tool will be deployed and launched as an open-source project in the near future. Our goal is to decrease the number of false positives and negatives commonly present within

source code, which is the most challenging issue in most of the security computing and scanning markets regarding source code analysis level [6].

Acknowledgment

This work was funded by The Board of Regents of the University of Wisconsin System, the Department of Mathematics, Statistics, and Computer Science at the University of Wisconsin-Stout.

References

[1] M. L. Collard, M. J. Decker, And J. I. Maletic, “Srcml: An Infrastructure for The Exploration, Analysis, And Manipulation of Source Code: A Tool Demonstration,” 2013 IEEE International Conference on Software Maintenance, 2013.

[2] S. M. Alnaeli, M. Sarnowski, M. S. Aman, A. Abdelgawad, and K. Yelamarthi, “Vulnerable C/C code usage in IoT software systems,” 2016 IEEE 3rd World Forum on the Internet of Things (WF-IoT), 2016.

[3] C.C. Michael. S. Lavenhar, “Source Code Analysis Tools - Overview,” CISA Cyber Infrastructure, 2013.

[4] Amazon Web Services, Inc. (2018). AWS Lambda – Serverless Compute - Amazon Web Services. [online] Available at: https://aws.amazon.com/lambda/ [Accessed 1 Mar. 2018]

[5] R. K. McLean, "Comparing Static Security Analysis Tools Using Open Source Software," in Software Security and Reliability Companion (SERE-C), 2012 IEEE Sixth International Conference on, 2012, pp. 68-74.

[6] C.C. Michael Steven Lavenhar, “Source Code Analysis Tools - Overview.” (2013). - Official website of the Department of Homeland Security. [online 2013] Available at: https://www.us-cert.gov/bsi/articles/tools/source-code-analysis/source-code-analysis-tools---overview

[7] L. Pascarella, F. Palomba, M. Di Penta and A. Bacchelli, "How Is Video Game Development Different from Software Development in Open Source?," 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR), Gothenburg, 2018, pp. 392-402.

[8] R. Fahy and L. Krewer, "Using open source libraries in cross platform games development," 2012 IEEE International Games Innovation Conference, Rochester, NY, 2012, pp. 1-5.

[9] W. Scacchi, "Free and open source development practices in the game community," in IEEE Software, vol. 21, no. 1, pp. 59-66, Jan.-Feb. 2004.

[10] N. Sabahat, A. A. Malik and F. Azam, "Size estimation of open source board-based software games," 2015 International Conference on Open Source Systems & Technologies (ICOSST), Lahore, 2015, pp. 126-131.

[11] W. Scacchi, "Practices and Technologies in Computer Game Software Engineering," in IEEE Software, vol. 34, no. 1, pp. 110-116, Jan.-Feb. 2017.

[12] Y. Ban and Y. Zhao, "The Security Research of Massively Multiplayer Online Role Playing Games," 2012 International Conference on Computer Science and Service System, Nanjing, 2012, pp. 1900-1903.

[13] D. Callele, E. Neufeld and K. Schneider, "Balancing Security Requirements and Emotional Requirements in Video Games," 2008 16th IEEE International Requirements Engineering Conference, Catalunya, 2008, pp. 319-320.

[14] R. J. Robles, S. Yeo, Y. Moon, G. Park and S. Kim, "Online Games and Security Issues," 2008 Second International Conference on Future Generation Communication and Networking, Hainan Island, 2008, pp. 145-148.

[15] M. McMahon and M. Schukat, "A low-Cost, Open-Source, BCI- VR Game Control Development Environment Prototype for Game Based Neurorehabilitation," 2018 IEEE Games, Entertainment, Media Conference (GEM), Galway, 2018, pp. 1-9.