On the Use of Vulnerable Code in Chromium, the Base of Google Chrome: A Case Study

Melissa Sarnowski, Keith Ecker, Zachary Blasczyk, Derrek Larson,
Saleh M. Alnaeli, and Mark Hall
University of Wisconsin-Colleges
Madison, Wisconsin 53715
{sarnm6825, eckek2561, blasz7012, larsd0542,
saleh.alnaeli, mark.hall}@uwc.edu

Abstract

Presented is a study that analyzes the use of known vulnerable code over a five-year history of Chromium, an open source system written in C/C++ comprising over 37 million lines of code, and provides foundational infrastructure to the popular web browser Google Chrome. The source code for each year's version of the system was converted into XML and parsed with a software tool that tabulated the calls to known unsafe functions and commands that have been banned by some companies (e.g., Microsoft). The results showed that *memcpy*, *strlen*, and *strcmp* were the most ubiquitous unsafe functions used and that the use of unsafe functions steadily increased over the five years studied. These findings can bring awareness to the developers of web browsers about the security of the systems that they write.

Introduction

The open source development approach of software systems is gaining popularity among research communities from both industry and academia, and a large number of software systems from a variety of domains have developed as open source systems. The developers that create theses systems come from diverse backgrounds and varying levels of experience and knowledge of software engineering practices. One of the factors used to determine the quality of a software system is the security of that system. There is concern from both individual users and companies about the security of the systems that they use on a daily basis—especially those systems that play a primary role in business transactions or deal with the personal information of clients.[1-4]

While there can be many sources of security vulnerabilities, one pervasive instance is the use of unsafe functions and commands known to cause innate susceptibilities. While safer replacement functions exist and the use of known unsafe functions can be avoided, the results of this study show that developers continue to use these unsafe functions. These threats could be the result of many factors such as bad programming habits or backgrounds lacking in the knowledge of secure programming practices. Nevertheless, vulnerabilities exist in systems like Google Chrome—the base for the popular web browser, Google Chrome (https://www.chromium.org/chromium-projects)—because developers continue to use functions known to be unsafe by the research community (e.g., *strlen*, *strcmp*, *memcpy*) [10]. Regardless of whether or not the utilization of these unsafe functions is a consequence of habitual practice or a product of insufficient backgrounds, it leads to vulnerable systems; mitigating quality, robustness, and reliability while simultaneously propagating the expenses required to maintain and fix the security threats therein.

Since several studies show that the majority of security concerns and threats come from problems at the source code level [3, 5-7], researchers have suggested a focus on the development of tools and methods that could be used to help developers find and remove the vulnerable code in their systems and minimize its usage [8]. Taking steps to eliminate and reduce the use of vulnerable code could play a major role in protecting systems from being targeted by attackers because of the existence of known unsafe functions at the source code level.

Helping developers understand the threat unsecure functions pose to their systems and teaching them good programming habits—that avoid the use of these unsafe functions—can save time, effort, and money resolving inevitable security issues. Besides creating secure programming standards for developers to follow, developers need to be evaluated and supervised to ensure that they follow those standards so that they release high quality, secure software systems.

C/C++ is preferred by many programmers because of the high level of performance, flexibility, and efficiency the languages offer. Even with their strengths, these languages also have security vulnerabilities (e.g., buffer overflow, integer vulnerability, string vulnerability) that need to be addressed from the beginning when writing source code, and need to be removed or replaced when a system is refactored.

For example, the standard C library includes the function *gets(),* which is often used to read strings entered by the user of the program. The *gets()* function receives a data type char pointer as a parameter and reads the user input, entering the first character in the location of the pointer it received, and the rest of the characters are placed in subsequent locations in the memory. The buffer is terminated with a null character once a newline is found. The security concern is that the length of the buffer passed to the *gets()* function is unknown. Because of this, an attacker can take advantage and write LENGTH + MORE bytes into the buffer for LENGTH bytes, and the attacker will always succeed if there were no newlines included [5, 8]. When a system is attacked in this way, locations adjacent to the buffer are at risk of being overwritten. If those adjacent locations held sensitive data, then it could be corrupted or modified by the attacker or the attacker could overflow the stack, which could lead to unpredictable results in the program.

Vulnerable code needs to be either removed completely or replaced by safer alternative functions. An example of replacing unsafe functions with safer alternatives would be to replace *gets()* with *fgets().* While *gets()* is unable to determine the length of the buffer passed to it, *fgets()* also receives an integer, n, as a parameter and will only read the input until n-1 characters have been read, or a newline or the end of a file is reached. This prevents attackers from being able to exploit a system in the same way they could when *gets()* is used, so sensitive data located in adjacent memory is safe from potential attackers looking to access it via this method. Cleaning a system to replace unsafe functions with safer replacements, like replacing *gets()* with *fgets(),* can increase a system's security, and therefore its quality. That is, avoiding the use of unsafe functions from the beginning when writing the code is far more effective, as time, effort, and expenses would not be needed to refactor the system.

This study examined the most recent 5-year period of the history of Chromium. The most recent version of each year was statistically analyzed, and counts for both unsafe functions and safe replacement functions generated. We were interested in finding the prevalence of known unsafe functions, the prevalence of safer replacement functions, and whether the use of unsafe functions and their safer replacements is increasing or decreasing over the 5-year period studied.

Year	LOC	Total Unsafe	Total Safe Replacements
2017	12264855	13331	1220
2016	12270095	13319	1219
2015	10904349	12418	1138
2014	1119097	361	26
2013	1058296	435	34

Table 1: Lines of Code and Total Unsafe and Safe Replacements over a 5-year period

Our results show that the developers continue to use functions that are known by the research community to be unsafe at an increasing rate, despite the existence of safer replacement functions. While the number of safer replacement functions also increased over the 5-year period, their total number remained much smaller than that of the total number of unsafe functions. The results in this work will help draw the attention of developers to a problem that plagues their systems and could increase the quality of the systems they develop. It will also show the need for researchers to focus on developing methods and tools that can detect and remove vulnerable code, and create secure programming standards for developers to follow.

Related Work

To the best of our knowledge, there are no other studies about the prevalence of unsafe functions in the source code of Chromium or historical trends about the usage of unsafe functions over the recent history of Chromium; however, the issue of software security is not limited to a single domain or a single system. In a previous study, the authors of this paper examined the prevalence of unsafe functions at the source code level, and the prevalence of safer replacement functions, of 15 open source mobile computing systems developed in the C/C++ programming languages and developed by reputable vendors (e.g., Microsoft, Google, Facebook). In this study, results showed that *strcmp*, *strlen*, and *memcpy* were the most pervasive unsafe functions among the 15 systems. The most recent 5-year history of 10 of the 15 systems was also examined in this study, and the results showed that the usage of unsafe functions was increasing over that timeframe [8].

Jovanovic et al. proposed a new approach for vulnerability prediction based on the CERT-C Secure Coding Standard. This approach was evaluated for prediction, accuracy, and cost. They showed that their new approach could predict any potential vulnerability with increased accuracy; however, the study did not show historical trends, the recent state of the systems they tested, or whether the use of vulnerable code in systems in the market was increasing or decreasing [9]. While literature is rich with ways to detect vulnerable source code, no study investigates the use of vulnerable code that causes security issues or the evolution of that use in open source web browsers.

Methodology

For this study, we first converted the source code for each year of Chromium into XML using the srcML toolkit. This toolkit is an efficient and robust way to transform source code written in C, C++, C#, and Java into XML, and it preserves all of the source code text, which allows complete access to the source code (http://www.srcml.org/). srcML also tells us the number of lines of code contained within that source code.

After the source code is transformed into XML, we used a tool developed by one of the authors, *CCPPVulnerabilityAnalyzer*, to parse the XML file and tabulate the number of known unsafe functions banned by some companies (e.g., Microsoft) [8, 10] and their safer alternatives in the code. We then use this data to look at the usage of unsafe functions in the source code of Chromium, and we repeat this process for the last updated version of each of the five years studied. With the results from all five years, we look for historical trends in the usage of both unsafe functions and their safer replacement functions.

Year	memcpy	strcmp	strlen	free	malloc	snprintf	fopen	strcpy	sprintf	strchr
2017	4051	2565	2524	1527	722	421	272	221	206	183
2016	4051	2563	2520	1528	720	420	271	218	206	183
2015	3639	2480	2254	1449	657	417	241	223	252	178
2014	190	37	76	15	16	11	1	0	1	1
2013	190	36	84	58	26	19	2	2	1	2

Table 2: Ten Most Prevalent Unsafe Functions Used in Chromium

Results

We now look at our results in the context of the following formal research questions:

RQ1: How many unsafe functions are called by the most recent version of Chromium?

RQ2: How many safer replacement functions are called by the most recent version of Chromium?

RQ3: What are the top ten most prevalent unsafe functions called?

RQ4: Are the calls to unsafe functions increasing or decreasing over the system's history?

To answer RQ1, we can look at the data presented earlier in Table 1, where the total unsafe functions and total safe replacement functions are displayed for each year. The total number of unsafe functions called by the 2017 version of Chromium is over 13 thousand calls. We can also use this data to answer RQ2, which shows that the total number of safer replacement functions called by the 2017 version of Chromium is 1.2 thousand calls. This data shows us that the prevalence of unsafe functions is far greater than the prevalence

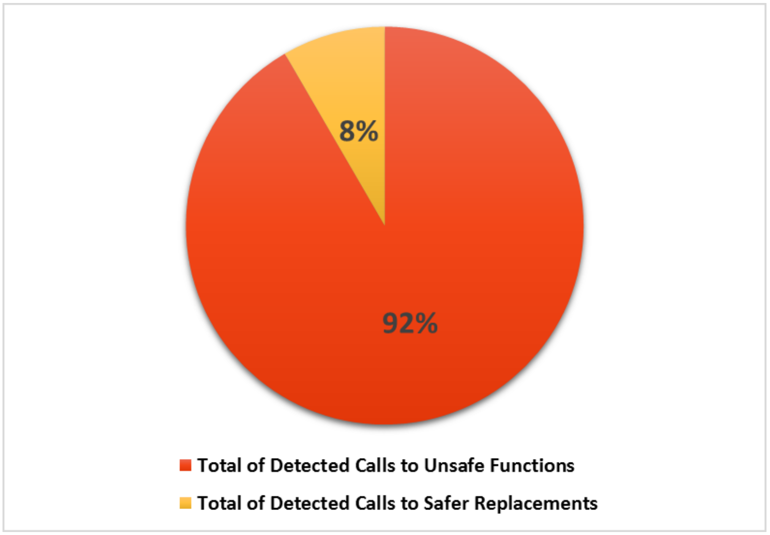

Figure 1: Chromium Comparison of Called unsafe functions vs. called safer replacements.

of safer replacement functions in the 2017 version of Chromium. While safer replacement functions exist, developers continue to use a far greater number of known unsafe functions.

Figure 1 shows the average percentage of called unsafe functions that have the potential to cause security vulnerabilities computed over the total number of detected function calls to known safer replacements. As can be seen, the overall average of detected calls to unsafe functions is 92%. That is, on average, most of the function calls to the relevant problem being studied in these systems can potentially cause significant challenges for software engineers and software vendors who are seeking to improve security and quality of their software systems.

We now address RQ3 by looking at the data presented earlier in Table 2. We see that the most prevalent unsafe function by far is *memcpy*, followed by *strncmp,* and *strlen*. These three functions combined make up over 9 thousand of the 13 thousand total unsafe functions called in the 2017 version of Chromium. Out of the top ten most prevalent unsafe functions called by Chromium, if the developers wanted to see the most benefit in their system's security when they refactor the code, we recommend that they start with the removal of *memcpy, strncmp,* and *strlen* since they represent a vast majority of the unsafe functions called, even compared to the other unsafe functions that make up the top ten most prevalent.

To address RQ4, we look at the data presented below in Figure 2. While the usage of unsafe functions saw a slight decrease between 2013 and 2014, that usage then had a significant increase between 2014 and 2015, increasing from 361 calls to over 12 thousand calls to unsafe functions. The usage of safer replacement functions also saw an increase between the years 2014 and 2015, increasing from 26 calls to over 1 thousand calls to safer replacement functions. After 2015, the usage of both unsafe functions and safer replacement functions continues to grow, but at a much slower rate than they increased between 2014 and 2015. Despite the slight increase in safer replacement functions, the number of unsafe functions increased at such a higher rate that the overall vulnerability of the system increased significantly.

Discussion

Our results show that the usage of unsafe functions in Chromium is increasing despite the availability of safer replacement functions. There could be many reasons for this, such as bad programming habits or a lack

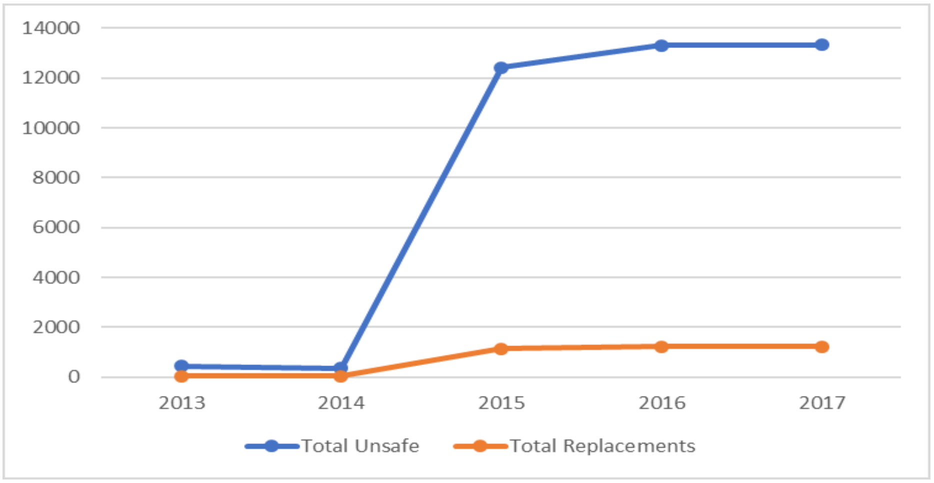

Figure 2: Change in the Usage of Unsafe Functions and Safe Replacements Over a 5-year Period

of knowledge about the negative effects the use of these functions can have on the quality and security of their systems. This study is meant to draw awareness to the security issues that arise from the use of these unsafe functions and to bring a focus to the need for the development of secure programming standards for developers to follow and the development of tools and methods that help developers detect and avoid the use of unsafe commands, as well as detect and remove or replace present unsafe functions. Furthermore, we recommend the evaluation, supervision, and education of developers to ensure that they follow secure coding practices and can release high-quality, secure systems while also saving the time, effort, and resources security issues would inevitably require.

One of the issues that may affect our results is that the tool used to tabulate the number of unsafe function calls and the number of safer replacement functions, *CCPPVulnerabilityAnalyzer*, does not differentiate between dead and active code. Because of this, it is possible that some of the unsafe functions and more secure replacements counted could be part of the dead code and never called by the system. In the future, we would like to refine this tool so that it counts only the functions that are part of the active code. We would also like to study a larger variety of systems in the web browser domain and in other domains to see if the trends found in Chromium are similar to those of other systems and if any generalizations can be made about the use of unsafe functions in web browsers.

Conclusion

In this study, we examined the usage of known unsafe functions in the most recent 5-year history of Chromium, the open source project that is the base for Google Chrome. We found that the use of known unsafe functions is increasing over this timeframe. While the number of safer replacement functions is also increasing, they are still greatly outnumbered by the use of unsafe functions. This could be due to bad programming habits or a lack of knowledge about the effect that using unsafe functions has on the security and quality of a software system. This work is meant to draw attention to the prevalence of the use of unsafe functions and the need for the erudition for developers on secure coding standards. It is also intended to draw attention to the need for the development of tools and methods that can help developers detect and remove or replace unsafe functions in their code, although avoiding the use of unsafe functions from the beginning when writing the code is a more effective approach.

Since this study examined only one web browser, we are unable to make generalizations about the use of unsafe functions in the web browser domain as a whole, although we would like to explore this topic in future work. We did, however, see that the top three most prevalent unsafe functions, *memcpy, strlen,* and *strcmp*, matched the top three most prevalent unsafe functions found in mobile computing systems in a previous study. This could point towards the existence of similar trends in the usage of unsafe functions across multiple software domains, but more studies need to be conducted before generalizations can be made.

We recommend that developers refactor their code to replace or remove unsafe functions in order to improve the security, quality, and robustness of their systems. For the greatest benefit, we recommend that developers start by eliminating *memcpy, strlen,* and *strcmp* first, as our results showed that they made up the majority of unsafe functions called not only in Chromium but a previous study conducted on 15 open source mobile computing systems as well. Although refactoring to remove unsafe functions is beneficial to the security of software systems, it is much more effective and efficient to avoid using them when first writing the code.

Acknowledgments

This work was funded in part by a grant from the University of Wisconsin-Colleges.

References

[1] J. Ryoo, B. Malone, P. A. Laplante, and P. Anand, "The Use of Security Tactics in Open Source Software Projects," *IEEE Transactions on Reliability,* vol. PP, pp. 1-10, 2015.

[2] S. Greiner, B. Boskovic, J. Brest, and V. Zumer, "Security issues in information systems based on open-source technologies," in *EUROCON 2003. Computer as a Tool. The IEEE Region 8*, 2003, pp. 12-15 vol.2.

[3] S. M. Alnaeli, M. Sarnowski, M. S. Aman, K. Yelamarthi, A. Abdelgawad, and H. Jiang, "On the evolution of mobile computing software systems and C/C&#x002B;&#x002B; vulnerable code: Empirical investigation," in *2016 IEEE 7th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*, 2016, pp. 1-7.

[4] S. M. Alnaeli, M. Sarnowski, M. S. Aman, A. Abdelgawad, and K. Yelamarthi, "Vulnerable C/C&#x002B;&#x002B; code usage in IoT software systems," in *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, 2016, pp. 348-352.

[5] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw, "ITS4: a static vulnerability scanner for C and C++ code," in *Computer Security Applications, 2000. ACSAC '00. 16th Annual Conference*, 2000, pp. 257-267.

[6] S. Rawat and L. Mounier, "An Evolutionary Computing Approach for Hunting Buffer Overflow Vulnerabilities: A Case of Aiming in Dim Light," in *Computer Network Defense (EC2ND), 2010 European Conference on*, 2010, pp. 37-45.

[7] 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.

[8] M. S. Saleh M. Alnaeli, Md Sayedul Aman, Ahmed Abdelgawad, Kumar Yelamarthi, "On the Evolution of Mobile Computing Software Systems and C/C++ Vulnerable Code," presented at the The 7th IEEE Annual Ubiquitous Computing, Electronics & Mobile Communication Conference, New York, USA, 2016.

[9] Y. Joonseok, R. Duksan, and B. Jongmoon, "Improving vulnerability prediction accuracy with Secure Coding Standard violation measures," in *2016 International Conference on Big Data and Smart Computing (BigComp)*, 2016, pp. 115-122.

[10] M. Howard. Security Development Lifecycle (SDL) Banned Function Calls [Online]. Available: https://msdn.microsoft.com/en-us/library/bb288454.aspx