Vulnerable C/C++ Code Usage in IoT Software Systems

Saleh M. Alnaeli, Melissa Sarnowski
Department of Computer Science
University of Wisconsin-Colleges
University of Wisconsin-Fox Valley
Wisconsin, USA
{Saleh.alnaeli, sarnm6825}@uwc.edu

Md Sayedul Aman, Ahmed Abdelgawad, Kumar Yelamarthi
College of Science and Engineering
Central Michigan University
Mt Pleasant, MI, USA
{aman1m, abdel1a, yelam1k}@cmich.edu

***Abstract*— An empirical study that examines the usage of known vulnerable statements in software systems developed in C/C++ and used for IoT is presented. The study is conducted on 3 open source systems comprising more than one million lines of code and containing almost 5K files. Static analysis methods are applied to each system to determine the number of unsafe commands known among research communities to cause potential risks and security concerns, thereby decreasing a system's robustness and quality (i.e., strcpy, strcmp, and strlen). Some of those statements are banned by some companies (e.g., Microsoft). These commands are not supposed to be used in new code and should be removed from legacy code over time as recommended by new C/C++ language standards. Additionally, each system is analyzed and the distribution of the known unsafe commands is presented. Historical trends in the usage of the unsafe commands are presented to show how the studied systems evolved over time with respect to the vulnerable code. The results show that the most prevalent unsafe command used across all systems is memcpy, followed by strlen.**

***Index Terms*— unsafe commands, vulnerable software, scientific, security, static analysis.**

I. Introduction

With the expected growth of the IoT, probably by tens of billion devices that run millions of software systems (open source and proprietary), security becomes a major concern for most of the expected IoT users (individuals and organizations) in both academia and industry. Each software system and device will potentially be a target or an access point to hackers, or lawbreakers [1-3]. Currently, most of the software and embedded systems used for IoT applications are available to the community as open source systems. Most of those systems are developed by programmers from different disciplines and backgrounds. Many of those programmers have little to no background on the security challenges imposed by the usage of vulnerable source code in some programming languages (e.g., C/C++) caused by unsafe commands (e.g., *strycpy*, *strcmp*, *strcat* etc.) well known to the research community.

According to [4, 5], most of the detected security threats are due to vulnerabilities in the code. Thus, minimizing usage of insecure commands can play a big role in protecting the software systems from any potential attacks. Additionally, making sure that developers are aware of those unsafe commands and following good programming practices can minimize the time and effort spent on finding and fixing them in later stages. For example, C/C++ programming languages are preferred when it comes to performance, flexibility, and efficiency. However, security vulnerabilities, such as integer vulnerability, Buffer overflow, and String vulnerability, need to be addressed and avoided from the beginning when writing the source code. Programmers also need to make sure that they do not use commands that are known to cause security concerns.

For instance, the standard C library includes a function called *gets*() that is usually used for reading strings from the user. This function takes a pointer from data type char as a parameter and reads a string of characters from the standard input, placing the first character in the location specified by that pointer and subsequent data consecutively in memory. The reading process continues until a newline is detected, at which point the buffer is terminated with a null character. The issue is that the developer cannot determine the size or length of the buffer passed to *gets*(). As a result, when the buffer is *size* bytes an attacker trying to write *size* + *extra* bytes into the buffer will always succeed if the data excludes newlines [4]. Consequently, memory locations that are adjacent to the buffer in the memory may be overwritten. If these locations hold sensitive data, then an attacker can modify the data. Additionally, an attacker could even overflow the stack leading the program to run into an arbitrary or unexpected status.

The process of protecting software systems from those vulnerability issues is done by either completely removing known unsafe commands and functions, or replacing them with what some call safe replacements (e.g., *strncpy*, *strncmp*, *strncat* etc.). Here we have undertaken an empirical examination of some of the open source software systems that are used for IoT in order to better understand how well an IoT developer can do when it comes to vulnerable code. We are particularly interested in determining the most prevalent unsafe statements that occur in a wide variety of IoT software applications and if there are general trends. While this work does not directly address the problem of removing them, it does serve as a foundation for understanding the problem requirements in the context of a broad set of applications. Moreover, the focus of this research is on unsafe statement detection and identifying their distribution over time.

978-1-5090-4130-5/16/$31.00 ©2016 IEEE

In this study, several software systems used for IoT and developed in C/C++ are analyzed and evaluated with respect to the usage of vulnerable commands that can affect the quality of IoT systems. The chosen systems are all written in C/C++. For each system, the number of unsafe functions is counted. A count of each of the unsafe functions is tabulated and this data is then presented to compare the different systems, uncover trends, and make other observations. Furthermore, the history of each system is examined and the number of detected unsafe functions is calculated for each release.

This work focuses on addressing the following questions: How many unsafe functions are used in these systems, which of those are the most frequent, and what are their respective distributions? Lastly, are the numbers or distributions of unsafe functions changing over the history/versions of a software system?

This work contributes in several ways. First, it is one of the only studies on the usage of vulnerable functions in IoT software applications. Our findings show that the functions *memcpy* and *strlen* represent the vast majority of unsafe functions occurring in these systems. Moreover, the findings show that, for the most part, the systems reviewed have increased their potential to be a target to vulnerabilities due to the increase in a number of the unsafe functions used overtime. This knowledge will assist researchers in formulating and directing their work to address this problem.

The remainder of this paper is organized as follows. Section 2 presents a definition of the problem and related work on the topic of source code vulnerabilities and security of IoT and its challenges. Section 3 describes the methodology we used in the study with how we performed the analysis to identify each unsafe function. Section 4 presents the findings of our study of 3 open source software systems used for IoT. There is a discussion of results in the same section as well. Historical trends found are explained in section 5, followed by threats to validity in 6 and conclusions in section 7.

II. Background and Related Work

The majority of previous research has focused on identifying security concerns related to the communication processes and authentication methods used with IoT. Some other studies have been conducted on the data privacy within different levels. However, to the best of our knowledge, no study has addressed the usage or distribution of vulnerable code in IoT systems that are developed in C/C++.

A study has been conducted by VERACODE [6] to investigate a selection of always-on consumer IoT devices to understand the security posture of each product. They found that product manufacturers weren't focused enough on security and privacy, as a design priority, putting consumers at risk for an attack or physical intrusion. Their team performed a set of uniform tests across all devices and organized the findings into four different domains: user-facing cloud services, back-end cloud services, mobile application interface, and device debugging interfaces. The results showed that the majority of the tested devices exhibited vulnerabilities across most categories. The findings prove the need to perform security reviews of device architecture and accompanying applications to minimize the risk to users.

Hui Suo, Jiafu Wan, Caifeng Zou, and Liu in [1] reviewed security in the IoT, and analyzed security characteristics and requirements from four layers, including perceptual layer, network layer, support layer and application layer. They discussed the research status in this field from encryption mechanism, communication security, protecting sensor data, and encryption algorithm. They confirmed the need to develop customized technologies and methodologies that fit the IoT needs and nature to meet the higher requirements in terms of security and privacy. However, they have not discussed the security concerns that can be caused specifically by programming standards or the usage of the vulnerable code.

A research group from George Mason University and National Institute of Standards and Technology in [2] presented a set of use cases that leverage commercial off-the-shelf services and products in order to raise awareness of security challenges in current practices and prove the need for IoT security standards, as well as their possible implications. They recommended experts must begin formulating suitable guidance and identifying the right security and privacy primitives for more secure and reliable IoT products. However, the study has not discussed any security issues related to the source code level and possible vulnerabilities that might be caused by the usage of unsafe statements.

Although literature is rich with studies that privilege the methods and tools that are used for detecting vulnerable source code [4], no studies have been conducted specifically in the domain of IoT that evaluates the usage and the distribution of vulnerable source code in the IoT software systems and applications.

The work presented here differs from previous work on IoT security in that we conduct an empirical study of unsafe functions on the source code level. We empirically examine a number of systems to determine what unsafe functions and codes exist in order to develop better software systems for IoT and show how these systems evolve over time in terms of secure programming standards based on the usage of unsafe versus safe functions.

TABLE I. Properties of Software Systems Evaluated

System	Language	LOC	Files
openWSN	C	284,094	853
Contiki	C	552,415	2,256
TinyOS	C/C++	164,472	1,037
TOTAL		**1,000,981**	**4,146**

III. METHODOLOGY FOR DETECTING UNSAFE FUNCTIONS

A function or a command is considered unsafe if it is one of the functions known to the research community and industry as a security vulnerability causer. Some of those unsafe functions and commands are already banned by some of the compiler producers (e.g., Microsoft). Literature is rich with lists of the unsafe C/C++ functions and commands. We used a tool, *UnsafeFunsDetector*, developed by one of the main authors, to analyze source code files and determine if they contain any unsafe function calls. First, we collected all files with C/C++ source-code extensions (i.e., c, cc, cpp, cxx, h, and hpp). Then, we used the srcML (www.srcML.org) toolkit [7] to parse and analyze each file. The srcML format wraps the statements and structures of the source-code syntax with XML elements, allowing tools, such as *UnsafeFunsDetector*, to use XML APIs to locate such things as unsafe functions and to analyze expressions. Once in the srcML format, *UnsafeFunsDetector* iteratively parses every single source code unit to find each call of unsafe functions and adjusts the counters. That is, a count of each unsafe function was recorded. Finally, all calls of unsafe functions were counted and their distributions determined.

The systems that were chosen in this study were carefully selected to represent a variety of open source systems developed in C/C++ that are used for IoT. These are well-known software systems to both academia and research IoT communities. Findings are discussed later in this paper along with limitations of our approach.

IV. FINDINGS, RESULTS AND DISCUSSION

We now study the usage of unsafe functions and commands in the studied systems along with their distributions.

TABLE I. presents the list of software systems used for IoT examined along with number of files, and LOCs for each system.

OpenWSN is an open source project that provide open-source implementations of a complete protocol stack based on Internet of Things standards, on a variety of software and hardware platforms. This implementation can then help academia and industry verify the applicability of these standards to the Internet of Things, for those networks to become truly ubiquitous [8].

The *TinyOS* is an open source, operating system designed for low-power wireless devices, such as those used in sensor networks, ubiquitous computing, smart buildings, and smart meters [9], while *Contiki* is a lightweight and flexible operating system for tiny networked sensors [10].

A. Design of the Empirical Study

This study focuses on three aspects regarding the security of software systems used for IoT in terms of unsafe function and vulnerable code usage. First, the number of calls to known unsafe functions. This gives a handle of how much of the system needs to be refactored to remove or replace the vulnerable code to increase its security and quality. Second, we examine which unsafe functions are the most prevalent. This can give IoT software engineers an idea about the most prevalent unsafe function to consider as a priority, should they plan for refactoring operations in aims to improve their systems in terms of security and quality. Lastly, we examine how the presence of unsafe commands (vs. some replacements) changes over the lifetime of a software system.

We propose the following research questions as a more formal definition of the study:

RQ1: What is the number of unsafe functions used per system?

RQ2: Which unsafe functions are the most prevalent?

RQ3: Over the history of a system, is the presence of unsafe functions increasing or decreasing?

We now examine our findings within the context of these research questions.

TABLE II. NUMBER OF UNSAFE FUNCTIONS FOR EACH SYSTEM FOUND USING *UNSAFEFUNSDETECTOR* TOOL.

Unsafe Function	TinyOS	openWSN	Contiki
puts	0	0	92
strcpy	11	0	91
strcat	10	0	13
fgets	1	0	3
sprintf	5	5	79
strchr	3	0	33
strlen	94	2	343
sscanf, scanf	3	0	40
strcmp	133	0	90
malloc	162	1	7
free	69	1	21
fopen	7	0	8
Localtime, system	5	0	22
memcpy	236	211	712
alloca, CopyMemory	0	0	11
Vsprintf, snprintf, vsnprintf, wsprintf	32	0	280
strtok	0	0	14
Total	772	220	1,859

B. Number of Detected Unsafe Function and their distribution

TABLE II. presents the results collected for the three studied systems. It shows a count of how many calls to unsafe functions and commands were found in each system. One item of interest in TABLE II. is that *TinyOS* and *contiki* show a much larger use of unsafe functions than openWSN. This shows promise for better planning for an effective refactoring process so that developers can start with those functions and replace them with safer alternatives.

As can be seen, *contiki* has the largest number of unsafe functions. In fact, it has some functions that are not used by the other two systems (e.g. *scanf* and *strtok*). However, in general, the number of the detected calls to memcpy was the highest for all the studied systems. Calls to *strlen* and *snprintf* are relatively high for *TinyOS* and *contiki*, but low for *openWSN*. Although, *snprintf* is considered as a replacement in

some compilers, it was banned by Microsoft and that is why it is considered an unsafe function in this study. As a total number of called unsafe functions, *contiki* has 1,859, followed by *TinyOS* with 772, and then *openWSN* with 220. This addresses RQ1.

We now address RQ2 and present the details of our findings on the distribution of detected unsafe functions. TABLE II. also presents the counts of each unsafe function that occur within each system. As can be seen, the *memcpy* function is by far the most prevalent across all systems. Except for *openWSN*, this is then followed by *strlen*, and then *strcmp* and *snprintf*, thus addressing RQ2. We lumped (*Vsprintf, snprintf, vsnprintf, wsprintf)* together, but we note that *snprintf* is much more prevalent than other similar functions. Note that some compilers consider *snprintf* as a replacement statement of function for *sprint*; however, this function was already banned by some companies (e.g. Microsoft), so it is considered unsafe in this study.

Clearly, a number of well-known unsafe functions still exist in the studied systems, thus complicating the security concerns when it comes to systems used for IoT. But no matter how we present the data, it is apparent that unsafe functions present one of the most serious security issues that need to be taken care of. Moreover, while there is much literature devoted to addressing the problems of how to get rid of those unsafe functions from software systems, it appears that software developers are underestimating the real threats imposed by those functions.

V. Historical Change of Unsafe Function Frequency

Each of the systems has been under development for many years. To address RQ3 we examined the most recent 5-year period of those three systems. Our goal is to uncover how each system evolves in the context of potential security threats caused by usage of unsafe functions. Here, we measure this by examining the change of used unsafe functions within every system. Our feeling is that this information could lead to recommendations for refactoring and efficiently removing those unsafe functions, thus enhancing the system's security and quality. Additionally, we are interested in discovering whether developers of IoT systems have put in enough effort to improve their systems within this context.

The change in the number of unsafe functions was computed for each version in the same manner as we described in the previous sections. These values were aggregated for each year so the systems could be compared on a yearly basis. The systems were updated to the last revision for each year. As before, all files with source code extensions (i.e., c, cc, cpp, cxx, h, and hpp) were examined and their calls to unsafe functions were then extracted.

Fig. 1. presents the change in the number of unsafe functions for each of the three systems. During the 5-year period all systems show a fairly flat to increasing trend. Almost all of the systems have an increase, and then two systems, *contiki* and *openWSN*, have a steep decline in 2014 and then are relatively increasing. *TinyOS* shows a relatively flat in proceeding years. Apparently, the systems that increased

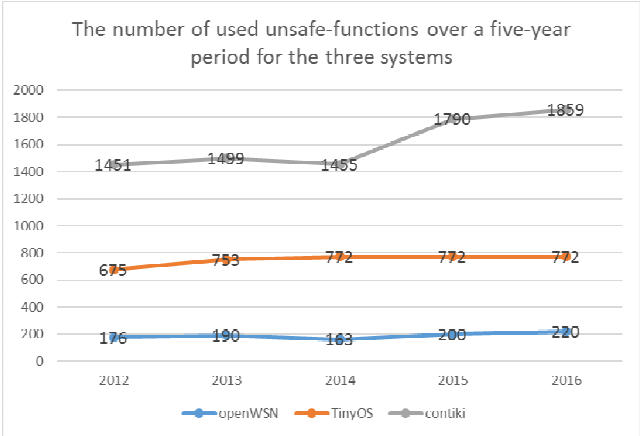

Fig. 1. The number of used unsafe-functions over a five-year period for the studied systems

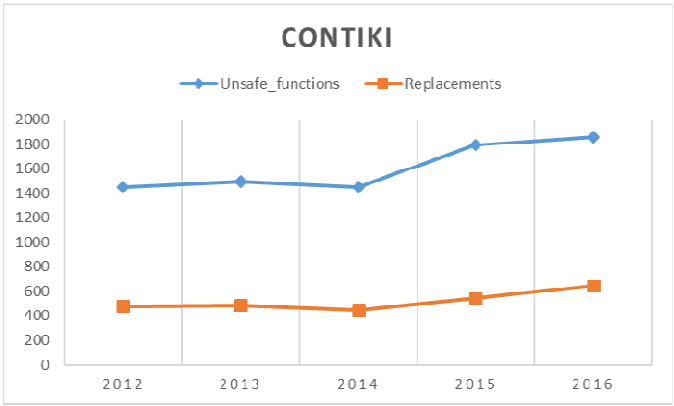

Fig. 2. **CONTIKI** evolution (total of used unsafe-functions vs. replacements)

in the number of unsafe functions are having a security issue that is getting worse overtime (e.g., *contiki*, *openWSN*).

The figure also shows that the total number of the detected unsafe functions is the same for *TinyOS* from 2014 to 2016. Although that could be a good sign, the system needs to go through a refactoring process to eliminate all of those unsafe functions, as recommended by the research communities, by either removing them or replacing them with safer replacements.

Fig. 2. presents the change in both unsafe functions and some of the replacements for those unsafe-function that have been used in only *contiki* during this time period. The figure shows that both of the function types have almost identical trends. That is, there is an increase in the number of unsafe functions and some of the replacements (e.g., *strncmp*, *strncpy*, *strncat* and *snprintf*). One item of interest in Fig. 2. is that *contiki* shows a much larger use of unsafe functions than the known replacements, and even though the developers are changing their way of handling some operations by using a safer function, they are still using unsafe functions. This gives us an idea about the programmers' background related to the unsafe functions. They prove that they are aware of the problem but they are not doing that much about solving it. That can be used

to convince authorities to develop some techniques that can be used to monitor the developers' behavior in this context.

Again, we are not sure if the usage of the replacement functions was introduced to the system as new portions or as part of refactoring processes conducted by the developers.

VI. THREADS TO VALIDITY

The tools we developed for this study only work with any language supported by srcML (C/C++). This has restricted us from using some existing projects written in languages such as Python and Java.

Upon examination of the unsafe functions used in the studied systems in the study, we found that some of them were part of dead code (i.e., code that would never be executed). As part of the static analysis, there was no distinction made in the study between calls of unsafe functions in dead code or active code which might affect the accuracy of the results we present in terms of the systems' security and vulnerability. In the future, we are planning to refine the results to only include active code and to include more systems developed using different programming languages.

VII. CONCLUSION

An empirical study that examines the usage of known vulnerable statements in software systems developed in C/C++ and used for IoT is presented. The study examined three open source systems comprising more than one million lines of code and containing almost 5000 files. Static analysis methods are applied to each system to determine the number of unsafe commands that cause potential risks and security concerns that decrease system robustness and quality.

The results show that usage of vulnerable functions is very common among the three systems where *memcpy*() is the most prevalent unsafe statement, followed by *strlen*().

The historical trend for the last five years is also shown with an interesting fact that, although the developers are working on replacing vulnerable statements, they are also using vulnerable statements at even higher rate. The majority of previous works on security concerns in IoT also focused on the security issues in the communication layer rather than in the vulnerability of the source codes.

The results presented show that software vulnerability is an issue that requires additional focus by the research community. Our suggestion would be to emphasize software vulnerability, reduce the usage of unsafe statements, especially the most prevalent statements to improve IoT platforms in terms of security, and thus enhance performance.

REFERENCES

[1] H. Suo, J. Wan, C. Zou, and J. Liu, "Security in the Internet of Things: A Review," in *Computer Science and Electronics Engineering (ICCSEE), 2012 International Conference on*, 2012, pp. 648-651.

[2] C. Kolias, A. Stavrou, J. Voas, I. Bojanova, and R. Kuhn, "Learning Internet-of-Things Security "Hands-On"," *IEEE Security & Privacy,* vol. 14, pp. 37-46, 2016.

[3] A. M. Gamundani, "An impact review on internet of things attacks," in *Emerging Trends in Networks and Computer Communications (ETNCC), 2015 International Conference on*, 2015, pp. 114-118.

[4] 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.

[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] Veracode. (2015). *The Internet of Things: Security Research Study*. Available: https://www.veracode.com/sites/default/files/Resources/Whitepapers/internet-of-things-whitepaper.pdf

[7] S. M. Alnaeli, A. A. Taha, and T. Timm, "On the Prevalence of Function Side Effects in General Purpose Open Source Software Systems," in *Software Engineering Research, Management and Applications*, R. Lee, Ed., ed Cham: Springer International Publishing, 2016, pp. 115-131.

[8] The University of California. (2016). *openWSN* https://openwsn.atlassian.net/wiki/pages/viewpage.action?pageId=688187

[9] The TinyOS WorkingGroup, (2013), TinyOS. http://www.tinyos.net/

[10] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors" in 29th Annual IEEE International Conference on Local Computer Networks, 2004.