Security Analysis of Contiki IoT Operating System

Jack McBride
School of Computing
University of Kent
jlgm2@kent.ac.uk

Budi Arief
School of Computing
University of Kent
b.arief@kent.ac.uk

Julio Hernandez-Castro
School of Computing
University of Kent
jch27@kent.ac.uk

Abstract

The Internet of Things (IoT) has introduced a myriad of ways in which devices can interact with each other. The IoT concept provides opportunities for novel and useful applications but at the same time, concerns have been raised over potential security issues caused by buggy IoT software. It is therefore imperative to detect and fix these bugs in order to minimise the risk of IoT devices becoming the target or source of attacks. In this paper, we focus our investigation on the underlying IoT operating system (OS), which is critical for the overall security of IoT devices. We picked Contiki as our case study since it is a very popular IoT OS and we have access to part of the development team, allowing us to discuss potential vulnerabilities with them so that fixes can be implemented quickly. Using static program analysis tools and techniques, we are able to scan the source code of the Contiki OS systematically in order to identify, analyse and patch vulnerabilities. Our main contribution is a holistic and systematic analysis of Contiki, starting with an exploration of its metrics, fundamental architecture, and finally some of its vulnerabilities. Our analysis produced relevant data on the number of unsafe functions in use, as well as the bug density; both of which provide an indication of the overall security of the inspected system. Our effort led to the finding of two major issues, described in two Common Vulnerabilities and Exposures (CVE) reports.

Categories and Subject Descriptors

I.6 [**Systems security**]: Operating systems security – Mobile platform security; I.9 [**Software and application security**]: Software security engineering

General Terms

Security, operating systems, static analysis

Keywords

Security analysis, Contiki, Internet of Things

International Conference on Embedded Wireless Systems and Networks (EWSN) 2018
14–16 February, Madrid, Spain

ISBN: 978-0-9949886-2-1

1 Introduction

Many Internet of Things (IoT) devices are designed to maximise convenience and performance, but these and other characteristics such as time-to-market frequently come with an associated cost in terms of security. As IoT devices are becoming more integrated into our daily lives – for example, in smart home and e-health scenarios – insecure IoT decives can pose a very serious threat. One of the most recent and consequential attacks came from the Mirai botnet in late 2016 [11]. This botnet was created by malware which amassed the collective power of several thousand compromised IoT devices (mostly smart cameras) to launch a Distributed Denial of Service (DDoS) attack against several critical network infrastructures, notably the Dyn DNS service provider. More security risks are anticipated, as according to the 2018 market forecast by Forbes [10], the IoT market is expected to expand to around USD 29.02 Billion by 2022, with security and privacy remaining its biggest challenge.

According to the IoT developer survey of 2017 [12], amongst the most popular operating systems in use are IoT based variations of Linux and Windows, with open source platforms FreeRTOS and Contiki set to experience "steady growth" over the years to come. It is then paramount that we turn our attention to the security of popular up-and-coming systems ahead of time, before they are deployed to billions of devices; particularly those which have small development teams, and lesser exposure. Fortunately, many of these systems are open-source, so analysis can be conducted directly on their source code. Projects such as Contiki, TinyOS and RIOT feature open, publicly accessible repositories. This paves the way for the usage of security techniques such as *static analysis*, which can assist in determining the existence of bugs before they are released into the wild.

Static analysis is a method of program debugging by examining the source code without execution. This is done to assist a developer in understanding the structure of their code, and find potential bugs emerging from it. Static analysis tools are usually capable of additionally calculating some valuable metrics of a code base, from which an estimate of "bug density" of a system can be inferred. For example, the "cleanroom development" technique is a formal method of software engineering capable of achieving uniformly low failure rates in delivered systems: 3 defects per 1000 lines of code during in-house testing and 0.1 defects per 1000 lines of code in the released product [6]. However, there is not cur-

rently a standard measure for determining what may constitute a bug. By investigating this in more detail, it is possible to determine the stability of Contiki in relation to other similar operating systems, and compare their progression over time as the overall size and complexity of the source code increases. On a larger scale, this also gives us valuable insights into the developing landscape of IoT security, and helps determining whether operating systems are becoming safer or more unstable over time.

This paper explores the effectiveness of deploying static analysis on the code bases of open source operating systems. The main focus is Contiki: a popular IoT operating system designed to support networks of low-power devices. We believe that due to its growing popularity, Contiki will soon feature in a large number of IoT applications, and so would benefit from a security analysis to identify bugs at an early stage. We do so by deploying a range of static analysis tools and combining their results. Besides our main aim of improving the security of Contiki, we also intend to explore and evaluate the capabilities of the tools.

The rest of the paper is organised as follows. Section 2 provides an overview of the Contiki operating system, which serves as the case study of our approach. Section 3 describes our methodology in addressing the challenges in securing the Contiki IoT operating system through a static analysis approach. Section 4 reports the results and analysis we performed on Contiki, culminating in two Common Vulnerabilities and Exposures disclosures discussed in Section 5. Section 6 outlines related work, while Section 7 summarises our paper and provides several ideas for future work.

2 The Contiki Operating System

Contiki OS is an IoT operating system designed to support networked, resource-constrained devices. Implemented in C, Contiki prioritises lightweight memory management and power efficiency, with typical configurations being deployed using as little as 2 kilobytes of RAM and 60 kilobytes of ROM running at 1 MHz [7]. According to the IoT developer survey of 2017 [12], Contiki is used in roughly 13.4% of devices, and it is expected to grow steadily.

Designed to connect small, battery-powered devices to the internet, Contiki provides lightweight implementations for a variety of popular communication standards, including IEEE 802.15.4, 6LoWPAN, CoAP, MQTT, TSCH and RPL. Additionally, Contiki features a hardware-independent software infrastructure, with minimalistic abstraction being provided by the core system. Given the increasingly application-driven nature of sensor devices, this facilitates system portability, as additional platform support can be implemented in libraries and services on top of Contiki's fluid architecture.

Based on its support for a wide range of platforms and architectures, Contiki is considered an effective prototyping platform. Contiki has been integral to several recent IoT projects, in which it has provided a basis for consumer technology and hobbyist projects alike, such as the LiFX smart light bulb[1], the Thingsquare cloud platform[2], as well as general purpose sensor mote development such as the Zolertia ReMote[3]. Another popular choice is the Sensortag platform by Texas Instruments. One of its main products, the CC2650 LaunchPad[4], encompasses an array of built-in sensors such as a humidity sensor, gyroscope and accelerometer, ambient light sensors and an infrared temperature sensor into a board priced at $29 (£23) per unit. The Contiki source code also contains several examples of demo programs, which can be swiftly loaded onto devices and booted into a fully functioning Contiki system. In addition to its support for the TI MSP430 and Atmel AVR platforms, Contiki is also used across Redwire Econotags, Zolertia z1 motes, and ST Microelectronics development kits; and more recently, high end processor architectures such as the ARM Cortex.

As an open-source project, Contiki is maintained by a team of core developers alongside the technical community. The official github repository is open to the public to engage and contribute[5]. The latest stable release of the Contiki operating system, version 3.0, was published on 26 Aug 2015.

3 Methodology

The first step was to grasp the scope of the system at hand. With Contiki, this meant determining the size of the project, as well as investigating how it has changed over the course of its releases. We began by measuring the total number of lines of code, expressed in *Source Lines of Code (SLOC)* over the entire code base of the project, as well as for each directory. This allowed us to establish granularity in the system as a whole, and then in terms of its constituent components.

Once we had obtained a basic understanding of Contiki through its global metrics, the next stage was to investigate the deployment of static analysis tools on its source code. Static program analysis is a security technique involving the use of automated software tools on the source code of a software system. The analysis is, in principle, "static", in the sense that the code being analysed is not executed; but instead interpreted for errors, making it possible for the most elusive bugs to be detected. Such bugs may otherwise be completely imperceptible during execution time, leading to unexpected and sometimes dangerous behaviour. We deploy this analysis technique upon the Contiki source code to determine the existence of bugs and logic flaws.

The aim of using static analysis in this project is to measure the overall "bug density" of the Contiki operating system, as well as to locate any critical bugs. The bug density provides a measure of security for the scope of the system, usually by determining how many bugs are likely to be present in a given number of lines of code. From here, we then perform a closer analysis upon the main areas where the results show evidence of vulnerabilities. For example, we consider the code in the 'core' directory to be of higher priority than that in 'examples', on the basis that the former exists in *all* devices running Contiki, while the latter is deployed in particular platforms only.

As with any security analysis, providing a holistic and complete overview of a system is paramount. In previous

[1] https://www.lifx.com/collections/featured-products

[2] http://www.thingsquare.com/

[3] http://zolertia.io/product/hardware/re-mote

[4] http://www.ti.com/tool/launchxl-cc2650

[5] http://www.contiki-os.org/community.html

research, we had learned that on the market of static analysis tools, there are considerable gaps in terms of sensitivity, speed, usability and applicability. As such, the potential of our work would not be realised with a single tool. Our analysis, thus, starts with a range of eight tools of which we fully utilised six: two are commercial (CodeSonar[6] and Understand[7]), the other four being open source (Cppcheck[8], Clang[9], Flawfinder[10] and RATS[11]). Our aim here is to expand upon the coverage of the source code, for there is a high likelihood that some tools miss bugs which are spotted by others. It is known that certain tools specialise at finding particular bugs, so the aim is to gather different ones to increase the potential for vulnerabilities to be found.

4 Results and Analysis

Contiki is a large piece of software containing many directories and providing support for multiple programming languages. Furthermore, the software has evolved through many versions (at least ten stable versions since its inception). As such, there are three main types of analysis we carried out: by version, by programming language and by directory. However, due to space constraints, we present the results and analysis by version only. There are four main factors we consider: size of code base, number of errors, bug density, and unsafe function usage.

4.1 Code Base Size

We gathered source code metrics in order to understand the scale of the Contiki code base. This first step was fundamental to understanding the system's bug density, which we calculated by determining the average number of bugs present per 1,000 source lines of code (KSLOC). We determined how the size of Contiki had evolved over the past 10 years. This was used to investigate the system from a historical perspective. We believe that this provides assistance in estimating how the future bug density could progress.

4.2 Errors

Having obtained a set of software metrics, our next direction was to use the tools to briefly estimate the number of errors in the Contiki source code. By "error", we mean a mistake, misconception, or misunderstanding that a software developer made when writing a program. The presence of errors may create security vulnerabilities and increase the likelihood of bugs in the developed software system. The aim here was to reveal information regarding the current areas which are most heavily affected by potential bugs. Furthermore, we can observe the error rate over time between consecutive releases, and draw conclusions and estimations about its future evolution with regard to historical data. The results can be seen in Table 1.

According to our findings, the number of potential errors in Contiki has steadily increased over time, with the latest release exhibiting upwards of 4,000 errors. This is perhaps to be expected though, as over time the size of the code base of any project grows leading to higher complexity levels and therefore increasing the likelihood that bugs are introduced into the system. That said, over time studies have increasingly highlighted the dangers of unsafe programming practices in ANSI C. These findings are also generally reflected in the outputs of each tool; each of which shows a gradual rise in issues detected between versions.

Table 1. Errors detected by tool, for each Contiki version.

Version	SLOC	Flawfinder	RATS	Cppcheck
2.0	69165	1326	204	9
2.1	80029	1480	236	19
2.2	90217	1692	263	34
2.3	116648	2058	304	33
2.4	147042	2283	332	41
2.5	191187	2660	393	50
2.6	217308	2963	417	60
2.7	235456	3044	410	125
3.0	260346	3081	406	122
3.x	355913	3451	439	123

4.3 Bug Density

This section describes our findings regarding the error distribution of the Contiki source code, commonly referred to as the *bug density*.We postulate that there are several factors affecting the bug density metric, such as code complexity, the type of defects taken into account for the calculation, the time scale over bug density calculation, developer and tester skills, and sensitivity of tools used to locate bugs.

We argue that complexity of code is intrinsic to our case study of an operating system. We intend to target the next three points (type of defect, time taken, and developer/tester skills) by automating the process with our tools. As a result, time will not be an issue and neither will tester skills. However, the type of defects calculated presents an interesting challenge. This is applicable to the tools we used, in that each of them prioritise a different subset of bugs to detect. As such, our results presented a spectrum of sorts: Flawfinder, which reports argue is strong at minimising false positives, reports a significantly higher value for bug density. RATS is between the two extremes. We attempt to combat this by generating an average bug density across the open source tools we used, so as to scale the outputs appropriately. Table 2 provides the breakdown of the results.

We ran each tool to traverse all of the directories of the Contiki releases over the past 10 years. Using the code metrics we obtained from *SLOCCount*, we generated a measure of bug density in the Contiki operating system across its released versions.

Using this data, we then generated the total bug density and scaled it appropriately to determine the number of bugs per 1,000 lines of code. As there is no standard way of determining bug density, we calculated it first by taking the average number of bugs across the tools, as well as by generating a separate output for each individually, to allow us to compare the differences in tool output.

The bug density was initially measured by code base to establish a general overview of the bugs in Contiki. Whilst the tools are capable of reporting on specific cases of error prone behaviour in the source code, exploring those capabilities is further considered in Section 4.4. During this stage,

[6] https://www.grammatech.com/products/codesonar
[7] https://scitools.com/
[8] http://cppcheck.sourceforge.net/
[9] https://clang-analyzer.llvm.org/
[10] https://www.dwheeler.com/flawfinder/
[11] https://tinyurl.com/y94gyedm

the goal was to pinpoint the critical areas of Contiki, i.e. most vulnerable, most popularly used, and those hosting the highest concentration of potential bugs. This was used to pick the areas of the system on which to deploy the more accurate commercial tools. From our findings, we calculated the average bug density of Contiki in terms of its releases over time, and each specific directory of its latest release.

Table 2. Bug density per KLOC in Contiki, by version.

Version	SLOC	Avg # of errors	Bug Density (KLOC)
2.0	69165	513	5.6312
2.1	80029	578.3333	5.1644
2.2	90217	663	5.2181
2.3	116648	798.3333	4.8881
2.4	147042	885.3333	4.4587
2.5	191187	1034.3333	4.1939
2.6	217308	1146.6666	3.8071
2.7	235456	1193	3.9609
3.0	260346	1203	3.6571
3.x	355913	1338.3333	3.1441

It appears that whilst the number of bugs is increasing between Contiki releases, on average the bug density is decreasing. This is reflected both in terms of the average measure taken across all three tools, as well as in the individual cases besides that of Cppcheck, for which the bug density appears to peak twice: at Contiki versions 2.2 and 2.7, before settling. This may be attributed to Cppcheck's tendency to target a specific class of bug to minimise false positive results. However, it may also indicate a spike in numbers of a particular bug which Cppcheck is receptive to, prompting a more in-depth analysis.

Our investigation into the bug density per directory reveals a convergence of errors towards the *tools, apps* and *examples* directories (of 18.09, 12.07 and 6.54 respectively). This is not surprising, as these are the areas of the system which contain some of the more complex code. Across the majority of the analysis tools, the highest bug densities are found in the *tools* and *apps* directories. This is also unsurprising, as most of the code from these directories form the underlying functionality in Contiki. Once again, this result is shown clearly in the relationship between Flawfinder and RATS, but less so for Cppcheck: which interprets *tools* as having a lower bug density. Surprisingly, the *core* directory has one of the lowest bug density results at 3.743 per KLOC. This could be down to a number of reasons, e.g. the Contiki developers focus the majority of their time on maintaining the core functionality of the OS hence it is more polished.

It is difficult to reach a general consensus on bug density based in the reporting by these tools as, for example, RATS and Flawfinder show high levels, up to 29 potential bugs per 1,000 lines of code. However, as the definition of what constitutes a bug is down to interpretation, it may be that this particular result simply demonstrates high levels of false positives.

4.4 Unsafe Function Usage

For the tools that had the ability to report on unsafe function usage (which, in our study, are RATS and Flawfinder) we collected statistics of unsafe functions from data longitudinally over the past 10 years of Contiki releases, starting from version 2.0 and concluding with the current release.

This avenue of work was conducted mainly in response to previous research conducted by Alnaeli et al. [1] on Contiki and its competitors, and borrows their definition of "unsafe" in terms of functions written in C. Additionally, we researched the CVE database for information regarding bug types. Our primary consideration are the functions in the C standard library that have been deprecated and replaced with safer alternatives, such as `strcpy` and `strncpy`; the latter was introduced to combat buffer overflow vulnerabilities. Additionally, we drew upon the extensive knowledge base of professional C programmers, researchers, and documentation found on various sites online.

Our results show that one of the more verbose static analysis tools, Flawfinder, reports on a considerably higher amount of issues that RATS. Flawfinder has specific features for detecting the usage of unsafe functions including a built-in database. For illustration, the results from Flawfinder can be seen in Table 3.

Interestingly, there was a lack of consistency between the findings we produced and those reported in the research by Alnaeli et al.[1], in which the researchers postulated that there were 1,859 unsafe functions detected in Contiki using the *UnsafeFunsDetector* tool. This once again highlights the clear lack of consistency between the reporting of static analysis tools; possibly resulting in a skewed impression across a system's security. As their *UnsafeFunsDetector* tool was not made publicly available following the publication, it is impossible to verify their claims.

Flawfinder reports very high use of the dangerous functions `memcpy` and `strlen`. Using this, we can infer the circumstances under which the highest concentration of vulnerabilities in Contiki will arise. In this case, the computing of string lengths using `strlen` might lead to space issues for null terminating characters (which `strlen` does not compute). Further to this, `memcpy` is known to cause serious buffer issues; suggesting the possibility of buffer overflows in the system. Alternatively, had the programmers implemented a safer version of `memcpy` (such as `memmove`) this threat would be reduced.

5 CVE Disclosure

As a result of testing the Contiki operating system in our testbed environment, we discovered two major exploitable vulnerabilities, which we disclosed to the Common Vulnerabilities and Exposures List[12].

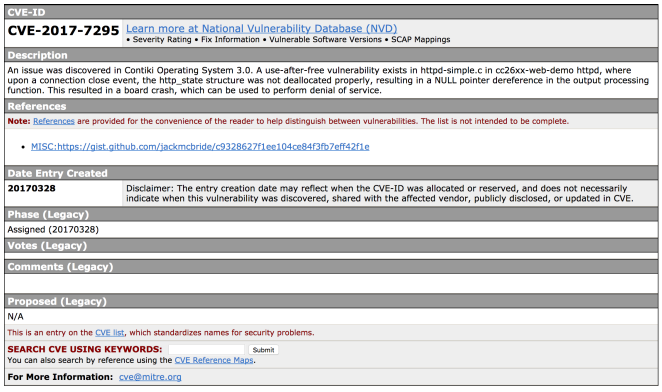
5.1 CVE-2017-7295

The first vulnerability resulted in a system crash for a Contiki device running in network node. We discovered this by tracing a use-after-free vulnerability in the `httpd-simple.c` file in the *cc26xx-web-demo*, detected as a result of our use of static analysis tools and debugging within our test bed environment. A typical use-after-free bug occurs when a previously released memory resource which has been deallocated following its use, is called upon again by the program. This can result in a program or system crash. An attacker could abuse this to cause issues in a network deployed in the real world, potentially using it as leverage to disrupt or damage network infrastructure.

[12]https://cve.mitre.org/

Table 3. Unsafe function usage by Contiki version reported by Flawfinder.

Version	2.0	2.1	2.2	2.3	2.4	2.5	2.6	2.7	3.0	3.x	Total
puts	0	0	0	0	0	0	0	0	0	0	0
gets	0	0	0	0	0	2	2	2	2	0	8
fgets	0	0	0	0	0	0	0	0	0	0	0
sprintf	34	44	52	60	77	101	101	102	97	90	758
strchr	0	0	0	0	0	0	0	0	0	0	0
strlen	143	149	176	189	199	184	236	284	318	375	2253
sscanf	0	0	0	0	0	2	2	2	2	2	10
scanf	0	0	0	1	1	6	6	6	6	6	32
strcmp	0	0	0	0	0	0	0	0	0	0	0
malloc	0	0	0	0	0	0	0	0	0	0	0
free	0	0	0	0	0	0	0	0	0	0	0
fopen	0	0	6	6	11	11	8	10	10	9	71
localtime	0	0	0	0	0	0	0	0	0	0	0
system	9	7	12	14	14	16	15	14	16	17	134
memcpy	142	168	215	330	381	465	585	604	658	743	4291
alloca	0	0	0	0	0	0	0	0	0	0	0
CopyMemory	2	1	2	2	2	4	4	4	4	4	29
vsprintf	1	2	2	2	2	2	1	1	1	1	15
snprintf	0	0	10	12	17	32	33	33	33	44	214
vsnprintf	3	2	3	5	7	11	14	14	16	20	95
wsprintf	0	0	0	0	0	0	0	0	0	0	0
strtok	0	0	0	0	0	0	0	0	0	0	0
Total	334	373	478	621	711	836	1007	1076	1163	1311	7910



CVE-ID

CVE-2017-7295 Learn more at National Vulnerability Database (NVD)
• Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings

Description

An issue was discovered in Contiki Operating System 3.0. A use-after-free vulnerability exists in httpd-simple.c in cc26xx-web-demo httpd, where upon a connection close event, the http_state structure was not deallocated properly, resulting in a NULL pointer dereference in the output processing function. This resulted in a board crash, which can be used to perform denial of service.

References

Note: References are provided for the convenience of the reader to help distinguish between vulnerabilities. The list is not intended to be complete.

- MISC:https://gist.github.com/jackmcbride/c9328627f1ee104ce84f3fb7eff42f1e

Date Entry Created

20170328 Disclaimer: The entry creation date may reflect when the CVE-ID was allocated or reserved, and does not necessarily indicate when this vulnerability was discovered, shared with the affected vendor, publicly disclosed, or updated in CVE.

Phase (Legacy)

Assigned (20170328)

Votes (Legacy)

Comments (Legacy)

Proposed (Legacy)

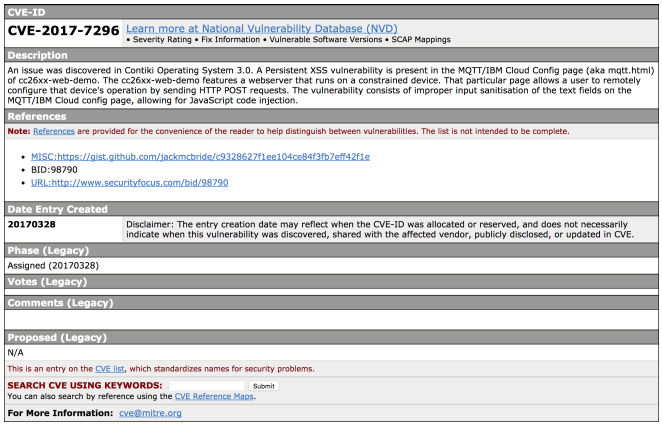
N/A

This is an entry on the CVE list, which standardizes names for security problems.

SEARCH CVE USING KEYWORDS: Submit
You can also search by reference using the CVE Reference Maps.

For More Information: cve@mitre.org

Figure 1. CVE-2017-7295 entry.



CVE-ID

CVE-2017-7296 Learn more at National Vulnerability Database (NVD)
• Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings

Description

An issue was discovered in Contiki Operating System 3.0. A Persistent XSS vulnerability is present in the MQTT/IBM Cloud Config page (aka mqtt.html) of cc26xx-web-demo. The cc26xx-web-demo features a webserver that runs on a constrained device. That particular page allows a user to remotely configure that device's operation by sending HTTP POST requests. The vulnerability consists of improper input sanitisation of the text fields on the MQTT/IBM Cloud config page, allowing for JavaScript code injection.

References

Note: References are provided for the convenience of the reader to help distinguish between vulnerabilities. The list is not intended to be complete.

- MISC:https://gist.github.com/jackmcbride/c9328627f1ee104ce84f3fb7eff42f1e
- BID:98790
- URL:http://www.securityfocus.com/bid/98790

Date Entry Created

20170328 Disclaimer: The entry creation date may reflect when the CVE-ID was allocated or reserved, and does not necessarily indicate when this vulnerability was discovered, shared with the affected vendor, publicly disclosed, or updated in CVE.

Phase (Legacy)

Assigned (20170328)

Votes (Legacy)

Comments (Legacy)

Proposed (Legacy)

N/A

This is an entry on the CVE list, which standardizes names for security problems.

SEARCH CVE USING KEYWORDS: Submit
You can also search by reference using the CVE Reference Maps.

For More Information: cve@mitre.org

Figure 2. CVE-2017-7296 entry.

5.2 CVE-2017-7296

The flaw we discovered presents the possibility of a persistent XSS attack on a Contiki device running as a native node. In the MQTT/IBM Cloud configuration page running on the node, a lack of input checking in the text input fields makes it possible for an attacker to submit malicious Javascript code – tested in our case using the "Type ID" field. When the devices MQTT configuration is updated with the malicious script input, the stored code may be executed in a users browser when they visit the configuration page to view or update settings.

We documented this vulnerability with a description of its reproducible steps, along with a video [13] in which we injected our own arbitrary Javascript code into a device's cloud configuration page. We concluded that due to this vulnerability, an attacker would be capable of remotely hacking into any vulnerable system running Contiki 3.0.

6 Related Work

Whilst there have been almost universal benefits following advances in the IoT, this technology introduces an increased risk to the privacy and integrity of our assets and lives.

The work of Zhang et al. [14] describes seven major priorities and challenges currently faced in IoT research. According to their research, some of the field's greatest challenges are rooted in security; namely in enforcing a suitable standard of measures for protecting IoT devices. Particularly, the authors acknowledge the emerging challenge of scalable cryptography, emphasising the need for lightweight and IoT driven solutions. The factor of software vulnerability in the development stages is also cited as critical.

A core use case of Contiki is for wireless sensor networks (WSN), where a series of connected devices, referred to as "nodes" or "motes" exchange data to provide a service [8]. Based on time and execution overhead, most WSN deployments do not possess robust protection measures. This presents a high security risk, as many WSN systems are featured in sensitive applications such as health monitoring, where there is an intrinsic need for confidentiality and availability. Data integrity must also be accounted for, as critical

[13] https://tinyurl.com/y8glecv9

decisions are often made assuming that the data collected has not been manipulated in transit. As such, security should not be overlooked for the sake of efficiency.

Borgohain et al. [3] examine the features of several operating systems, including Contiki, mbed, and TinyOS. They acknowledge the differences between the security of standard computer systems and those in IoT. In their work, they emphasise the importance of data encryption, citing its inclusion as "paramount" to the success of IoT. When discussing the specific security implementations of each system, the research postulates that end-to-end security such as TLS and DTLS is imperative for secure communications. Both Contiki and TinyOS implement this, with the former also featuring "ContikiSec" for additional network layer security [4]. The authors later emphasise the need for improving the general robustness of IoT systems against dictionary attacks, based on the possibility for weak, derivative passwords to be brute-forced [2]. This was showcased in recent hacking attempts on Virgin media routers, which in 2017 faced issues regarding the default password of the "Super Hub 2" network router. Based on the weakness of the password, attackers were able to remotely leverage unsolicited access to IoT devices on the home network, subsequently compromising over 800,000 Virgin customers[14]. As the IoT continues to expand, we propose that regular surveys would be a useful asset for monitoring security at a general level. Our research builds upon this idea, by running static analysis tools over the code bases of popular IoT systems, and generating a measure of bug density across consecutive software releases to provide historical coverage.

Security analysis at the code level has developed considerably since the times of tools such as ITS4 [13]. It is well understood that the majority of vulnerabilities discovered in a system are resolvable during the implementation stage of the development life cycle. However, due to industry-wide scaling of code bases as projects develop, manual code analysis becomes an inefficient, arduous and - in some cases - impossible task [5]. More so than ever, we rely upon the use of automated tools to provide error detection.

In recent years, static analysis tools have evolved way beyond simple pattern matching; incorporating developments such as abstract syntax trees (AST) for semantic evaluation and control flow graphs (CFG) to measure all possible execution paths of a program. The latter is a characteristic of modern state-of-the-art tools such as CodeSonar, which uses deep analysis techniques for scanning distributed systems. As these systems continue to grow, the appropriate scaling of static analysis tools will be critical. Researchers have proposed that the security coverage of static analysis tools will assist IoT developers to maintain their software more effectively, without creating hindrances in the system development life cycle [9].

7 Conclusions and Future Work

This paper evaluated several ways in which researchers can address the challenges of securing IoT devices. In particular, we used a number of quite different static analysis tools for finding potential vulnerabilities at the operating system level. We put our focus on assessing the security of a popular IoT operating system called Contiki, by carrying out static analysis on its code base, resulting in the identification of two major vulnerabilities and a number of other minor issues, that have since been reported and patched.

We have demonstrated the effectiveness of deploying static analysis tools to improve system stability, by locating and patching some critical flaws in Contiki, with the assistance of state of the art tools including CodeSonar, Flawfinder and Cppcheck. Further to this, we have determined a general measure of bug density of the operating system, as well as an overview of software metrics obtained through the use of *SLOCCount*.

This work served as a pilot and a feasibility study for our approach in leveraging static analysis techniques for finding security vulnerabilities in IoT software. The results from our preliminary study have so far been promising. We plan to investigate the effectiveness of our approach by running it on other IoT systems, or even on more generic operating systems such as Android. Lessons learned from this investigation will allow us to develop and refine a more robust framework to assist software developers to implement more secure IoT software.

[14] https://tinyurl.com/yd22ekvd

8 References

[1] S. M. Alnaeli, M. Sarnowski, M. S. Aman, A. Abdelgawad, and K. Yelamarthi. Vulnerable C/C++ code usage in IoT software systems. In *Internet of Things (WF-IoT), 2016 IEEE 3rd World Forum on*, pages 348–352. IEEE, 2016.

[2] M. B. Barcena and C. Wueest. Insecurity in the Internet of Things. *Security Response, Symantec*, 2015.

[3] T. Borgohain, U. Kumar, and S. Sanyal. Survey of Operating Systems for the IoT Environment. *arXiv preprint arXiv:1504.02517*, 2015.

[4] L. Casado and P. Tsigas. Contikisec: A secure network layer for wireless sensor networks under the contiki operating system. *Identity and Privacy in the Internet Age*, pages 133–147, 2009.

[5] B. Chess and G. McGraw. Static Analysis for Security. *IEEE Security & Privacy*, 2(6):76–79, 2004.

[6] R. H. Cobb and H. D. Mills. Engineering software under statistical quality control. *IEEE Software*, 7(6):45–54, 1990.

[7] A. Dunkels, B. Gronvall, and T. Voigt. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, pages 455–462. IEEE, 2004.

[8] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future generation computer systems*, 29(7):1645–1660, 2013.

[9] R. Huuck. IoT: The Internet of Threats and Static Program Analysis Defense. In *EmbeddedWorld 2015: Exibition & Conferences*, page 493, 2015.

[10] B. Joffe. Six Key Internet Of Things (IoT) Trends To Watch For In 2018. *https://www.forbes.com/sites/benjaminjoffe/2017/07/25/hardware-trends-2017-complete-slides-and-some-analysis/*, 2017.

[11] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas. DDoS in the IoT: Mirai and Other Botnets. *Computer*, 50(7):80–84, 2017.

[12] I. Skerrett. IoT Developer Survey 2017. *https://www.slideshare.net/IanSkerrett/iot-developer-survey-2017*.

[13] 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*, pages 257–267. IEEE, 2000.

[14] Z.-K. Zhang, M. C. Y. Cho, C.-W. Wang, C.-W. Hsu, C.-K. Chen, and S. Shieh. IoT security: ongoing challenges and research opportunities. In *Service-Oriented Computing and Applications (SOCA), 2014 IEEE 7th International Conference on*, pages 230–234. IEEE, 2014.