

Security Analysis of RIOT Operating System

M.Sc. Computer Security

Dissertation

By: Kelechi Nnorom(ken87@kent.ac.uk)

Supervisor: Dr. Julio Cesar Hernandez-Castro

ABSTRACT

Lately, software security has become a major concern for organisations worldwide due to the growing threat of hackers and cyber criminals. A new form of technology developed in the software engineering field is the “Internet of Things”. It is based on a concept where machines and objects communicate or send data over the internet without the need for human intervention. This form of technology enables a wide range of possibilities because machines can perform different operations unaided while humans better utilise their time. Unfortunately, as technology in system engineering rapidly evolves, vulnerabilities arise and could be exploited.

This M.Sc. dissertation is an in-depth security analysis of RIOT, an ‘Internet of Things’ operating system. The operating system is new as well as the underlying technology. The goal of the project is to uncover vulnerabilities within the system through various testing methods.

ACKNOWLEDGEMENTS

I would like to express my sincerest gratitude to all those who have helped me throughout this year.

I would like to thank the Almighty God for the strength and determination to undertake and complete this project. I also want to thank God for the breakthrough at the end, when all hope seemed lost.

A special thank you goes to my supervisor, Dr. Julio Cesar Hernandez-Castro, for his enthusiasm, patience and guidance throughout this project. I also appreciate his constant supervision as well as providing necessary information and feedback throughout the project.

Finally I would like to thank my friends, my family, the University of Kent staff and all other students who have helped or encouraged me along the way. God bless you all.

TABLE OF CONTENTS

ABSTRACT	2
ACKNOWLEDGEMENTS	3
LIST OF FIGURES	6
CHAPTER 1: INTRODUCTION	8
1.1 PROBLEM DESCRIPTION	9
1.2 PROJECT OBJECTIVES	10
CHAPTER 2: BACKGROUND AND LITERATURE REVIEW	11
2.1 RELATED WORK	11
2.2 THE INTERNET OF THINGS	11
2.3 INTERNET OF THINGS (IoT) REQUIREMENTS	15
2.3.1 DEVICE	15
2.3.2 OPERATING SYSTEM	16
CHAPTER 3: INTRODUCTION TO RIOT	19
3.1 RIOT FEATURES	20
3.1.1 DEVELOPER-FRIENDLINESS	20
3.1.2 RESOURCE-FRIENDLINESS	21
3.1.3 IoT-FRIENDLILESS	22
3.2 RIOT ARCHITECTURE	22
3.2.1 HARDWARE DEPENDENT CODE	23
3.2.2 HARDWARE INDEPENDENT CODE	24
3.2.2 RIOT NETWORK STACK	24
CHAPTER 4: PROJECT ANALYSIS	27
4.1 PROJECT REQUIREMENTS	27
4.2 PROJECT PLAN	29
4.3 RISK ANALYSIS	30
CHAPTER 5: METHODOLOGY	31

5.1 INSTALLATION OF RIOT	31
5.2 SOURCE CODE ANALYSIS	32
5.2.1 RIOT SOURCE CODE ANALYSIS TOOLS	34
5.2.2 SOURCE CODE RESULT ANALYSIS	54
5.3 PENETRATION TESTING	55
5.3.1 NATIVE PORT	55
5.3.2 MBED NXP LPC1768	60
5.3.3 PENETRATION TEST RESULT ANALYSIS	66
CHAPTER 6: PROBLEMS ENCOUNTERED	67
CHAPTER 7: SUMMARY	68
7.1 CONCLUSION AND RECOMMENDATIONS	68
7.1.1 SOURCE CODE SYNTAX	68
7.1.2 MORE RIOT APPLICATIONS	70
7.2 PERSONAL EXPERIENCE	70
7.3 FUTURE WORK	71
REFERENCES	72
APPENDICES	77
APPENDIX A: Flawfinder Installation	77
APPENDIX B: Installation of RATS and Expats Library	77
APPENDIX C: Visual Code Grepper Analysis Output	78
APPENDIX D: ClamAV Installation	78

LIST OF FIGURES

Figure 1: 'Internet of Things' Model (Trifa, V. 2015)	13
Figure 2: IoT Devices in Communication	15
Figure 3: RIOT Network Stack	25
Figure 4: Project Plan	29
Figure 5: RIOT Installation	31
Figure 6: RIOT sub-directory	31
Figure 7: Executing Flawfinder against RIOT directory	35
Figure 8: Flawfinder Output	35
Figure 9: Bar chart of Potential Flaws per Directory	38
Figure 10: Pie chart of Error Density - Flawfinder	38
Figure 11: Fast RATS Analysis of RIOT OS	39
Figure 12: Bar chart of Vulnerabilities per Directory	42
Figure 13: Pie chart of Error Density - RATS	42
Figure 14: Yasca Analysis Process	43
Figure 15: "Bad Interpreter" Error	44
Figure 16: Failed Execution of Yasca with RATS plugin	45
Figure 17: Execution of Yasca with CPP plugin	45
Figure 18: Bar chart of Vulnerabilities per Directory	47
Figure 19: Pie chart of Error Density - Yasca	48
Figure 20: Loading RIOT OS files for Analysis	49
Figure 21: VCG Analysis of RIOT OS	50
Figure 22: Pie chart of RIOT OS Code Analysed	51
Figure 23: VCG Pie Chart of Vulnerability Severity	52
Figure 24: ClamAV execution on RIOT directory	53
Figure 25: ClamAV result for RIOT directory	53
Figure 26: Installation of openvpn on Virtual Machine	56
Figure 27: Compilation of 'gnrc_networking' project	56
Figure 28: Execution of 'gnrc_networking' project	57
Figure 29: Obtaining IPv6 address from ifconfig command output	57
Figure 30: Pinging IPv6 address	58
Figure 31: UDP server setup	58
Figure 32: Connecting Linux host to RIOT UDP server	58
Figure 33: Sending UDP packets from RIOT side	58
Figure 34: UDP packet received on UDP server on Linux side	59
Figure 35: Nmap output - 'top ports' command	59
Figure 36: Nmap output	60
Figure 37: My Mbed NXPLPC1768 Microcontroller	61
Figure 38: Compiling HTTPServerHelloWorld project for mbed microcontroller	62

Figure 39: Output from HTTPServerHelloWorld application in TeraTerm terminal	62
Figure 40: Compilation of original HelloWorld application	63
Figure 41: Uploading HelloWorld Application to mbed microcontroller	64
Figure 42: Editing 'gnrc_networking' Application Makefile	65
Figure 43: Compiling 'gnrc_networking' application for mbed board	65
Figure 44: Uploading 'gnrc_networking' application to mbed board	65
Figure 45: 'gnrc_networking' application executed on mbed board via Pyterm	66
Figure 46: Flawfinder Installation	77
Figure 47: RATS Installation	77
Figure 48: RATS 'Expat' Error	77
Figure 49: Part of VCG Analysis Output for RIOT OS	78
Figure 50: ClamAV Installation	78

CHAPTER 1: INTRODUCTION

Technology can be regarded as very unstable in terms of its rate of evolution and yet it affects millions of people worldwide. There have been various forms of technological advances in different fields but software engineering by far has undergone the most development in any given amount of time. The 'Internet of Things' or 'Internet of Everything' is a new field of technology that is rapidly expanding and projected to consist of billions of devices in the near future (Hesseldahl, 2015). It is a technology that will soon become part of everyday lives.

RIOT is an IoT operating system designed originally in Germany for wireless sensor networks (Mihai, 2015). The operating system's foundation and architecture was then improved upon a few years later and extended for compatibility with low-power consuming IoT devices.

This report discusses a security analysis of the RIOT operating system. It is organised into three main sections; a literature review providing background and technical information about the RIOT operating system, a project analysis chapter detailing various requirements needed for the execution of the project and a methodology section. The report ends with a segment to discuss ideas on how this project could be improved further. This

introduction continues with highlights on problems that could arise from an inadequate testing of a commercial software, as well as a reasonable plan of project objectives.

1.1 PROBLEM DESCRIPTION

In the world today, software developers are more concerned about how soon they can release a product rather than ensuring that the product is fully tested. This is usually in order to gain a market share edge over their competitors. This means less time is spent to fully develop and test products or software before deployment. With any new software technology however, it is important to check for security vulnerabilities that could be exploited in the system. Vulnerabilities are holes in a software or hardware that provides a route through which the system can be attacked. They could be as simple as weak passwords or poorly implemented program features, but also as complex as buffer overflows or SQL injection vulnerabilities (Kirsch, 2013). A key stage in any software development cycle is the “Testing” phase (DuPaul, 2014). This is done for various reasons, primarily to confirm that the system meets the set requirements, but also to ensure any vulnerabilities in the system do not prove to be costly later. This is because software systems currently face all forms of attacks daily, especially from malware. A compromised system could lead to multiple lawsuits and possible bankruptcy of the company.

RIOT is a new operating system in a new field of technology. It is also an open-source software. This means it benefits from continuous contributions

and improvements from the development community. It also means that the source code is free to download and can be analysed by anyone. This can make it susceptible to attacks because highly skilled hackers may find a vulnerability in the system, not report it, and exploit this to their gain later on. Vulnerabilities could be used to launch various forms of attacks such as smurf, buffer-overflow, denial-of-service(DoS), distributed denial-of-service(DDoS), malware infestation and so on. This could have a massive impact; especially if a large number of devices running a compromised operating system are deployed.

It is therefore vital that any IoT operating system is analysed and tested to reveal weaknesses that could expose it to attacks.

1.2 PROJECT OBJECTIVES

The first objective is to perform a source code analysis of RIOT operating system code to pinpoint potential vulnerabilities, bugs and errors in the source code.

The second goal will be to perform penetration testing on a device running RIOT operating system. The aim is to mimic a hacker's attempt to exploit vulnerabilities in the system. This will involve locating backdoors or open ports in the system through which the system can be attacked.

These tests will be executed using a number of open source and commercial testing tools. The third and final objective will be the examination of the results of the tests conducted. This will also involve suggestions for

improvement and sharing the test results with the RIOT operating system developers.

CHAPTER 2: BACKGROUND AND LITERATURE REVIEW

2.1 RELATED WORK

Security analysis of both software and hardware devices have been researched and implemented for many years. While this paper deals mainly with the RIOT operating system, the methods and techniques applied to detect vulnerabilities in a software system are very similar. Although diverse in terms of design features and components, many systems can be regarded as secure if they pass source code and penetration tests (Northcutt, S. et al., 2015).

The aim is to leverage as many analysis tools as possible during the tests and then compare and analyse results. Results from these tests reveal vulnerabilities such as errors, bugs in the source code and open ports that could be used to gain unauthorized access to the system. The combination of the software tools used for security analysis may lead to a variation in methods and results. This is because these tools use different algorithms for their analysis techniques. For instance, tools used for source code analysis have different algorithms that define rules that the source code must conform to or else an error is triggered (Kirkov et al., 2010).

2.2 THE INTERNET OF THINGS

The 'Internet of Things' is a new form of technology designed for low-power consumption devices with the aim of communication over the internet. The concept of standalone computers simply equipped with sensors and transceivers has been envisioned since the 1990s (Baccelli, E. et al., 2013). The goal is a system running various applications, which can communicate or send data over the internet. Over the last few years, The 'Internet of Things' technology has grown immensely and is projected to consist of billions of devices in the near future (Hesseldahl, 2015). This is no surprise, because with the constant development of technology in the IoT field, there are endless possibilities of applications for which new IoT devices would need to be created to service. The devices, however, need to be running a secure operating system, so as to prevent confidential information from being intercepted during transmission of data.

The limitations to this technology are mainly low power consumption and computing power for the devices. This is because for the IoT device to standalone and function, it needs to conserve power. This means low internal memory and an operating system that can manage itself efficiently. Also, the IoT technology is still relatively new. Therefore, there is no de facto standard to be used as a reference in terms of best practices for designing and implementing a secure and efficient IoT operating system (Hahm, 2014).

In truth, the 'Internet of Things' does not really exist as described. What exists is the 'Intranet of Things'; isolated islands of networks that cannot really speak and interact with each other (Trifa, V. 2015).

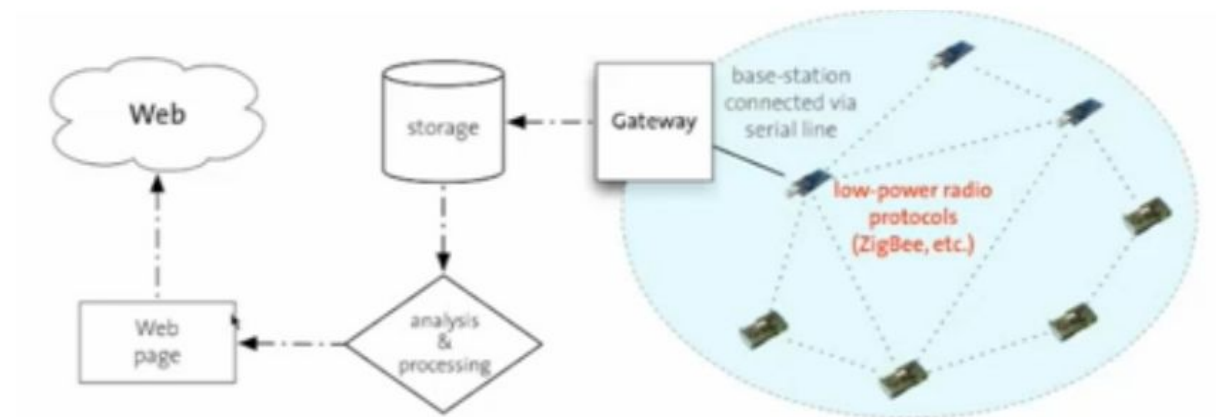


Figure 1: 'Internet of Things' Model (Trifa, V. 2015)

This involves storage devices connecting to a base station using low power radio transmission to collect data that has been extracted from nodes (see Figure 1). The data is then stored in a database and processed by applications for use for various functions. This could be for output to a user interface; where the user can read/write data to the device, or output to a debug application used by technicians to monitor the status of the device and fix problems. The 'Internet of Things' is more about how things connect to the internet rather than how applications connect with things. Devices containing these nodes are usually called 'Smart', due to their ability to have a permanent URL that can be accessed via Wi-Fi and an API for managing commands. For instance, in a home with a 'Smart Fridge' and a 'Smart TV', a user running an application connected to both devices can obtain information about the Fridge and display it on the TV, using the

HTTP request methods; GET and PUT (Trifa, V. 2015). Another example in a home could be an alarm clock communicating or sending a signal to a coffee machine at a certain time so that coffee is brewed and ready. The user only needs to set the alarm on a smartphone. This form of technology where the system is standalone and can send information periodically can also be extended for use in various fields such as aviation and medical industries. It could also be implemented in the automobile industry with roads equipped with sensors to collect data which is then stored in the cloud. Traffic congestions and road accidents could be easily monitored and reduced using this technology. The main benefits of the IoT technology to humans are better utilisation of time and increased productivity.

The problem however, is that for every application, each component for IoT needs to be re-invented from scratch. This is because there are many protocols, standards and programming models that are not compatible with each other. Also, in terms of functionalities, different applications require different hardware and requirements. This means IoT applications are usually hard-wired for a specific function and are difficult to evolve. Finally, IoT devices differ from other internet-compatible devices such as laptops, smartphones and tablets, in terms of internal memory. While laptops, smartphones and tablets have internal memory ranging from 512 Megabytes to 4 Gigabytes, IoT devices need to have very low internal memory (~kilobytes). This is to consume as little power as possible so as to run independently over a long period of time. It means a different kind of

operating system is required for IoT devices because standard operating systems such as Apple IOS, Windows and Android, are designed to be compatible with devices with higher internal memory; typically in the megabyte to gigabyte range.

2.3 INTERNET OF THINGS (IoT) REQUIREMENTS

2.3.1 DEVICE

IoT devices are typically cheap, tiny, wireless hardware used for various applications such as receiving and transmitting data over the internet. For instance, the devices can communicate data with each other or send information, such as meter readings, to a utility network base station (see Figure 2 below).

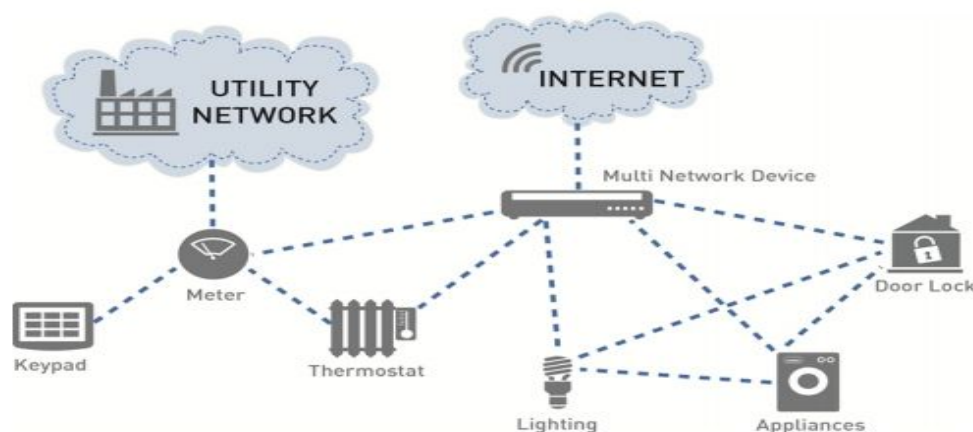


Figure 2: IoT Devices in Communication

Image source : <https://www.silabs.com/Support%20Documents/TechnicalDocs/bringing-the-internet-of-things-to-life.pdf>

IoT devices are of a heterogeneous nature, ranging from 8-bit microcontrollers to devices with more powerful processors (Emmanuel, B. et

al., 2015). The devices however are very constrained in the way they communicate due to the following technicalities:

- **Energy efficiency:** IoT devices are required to save as much energy as possible during operations in order to sustain battery life. This can be achieved using efficient wireless communication technologies rather than radio transmission during data transmission. As such, any operating system installed on the device would need to support the IEEE 802.15.4 MAC wireless standard; the internet standard for low-cost, low-speed ubiquitous communication.
- **Memory:** IoT devices also typically have very little internal memory for RAM and ROM. This is due to the constraint of the environment in which they are deployed. Although the devices require some memory to store data received from external sensors for example, no real complex operations are required of the device. Their main aim is to obtain data and forward the received information to a determined destination. With a small memory footprint, less power and energy is required to run the device which would extend the battery life.

2.3.2 OPERATING SYSTEM

There are various IoT operating systems currently available in the market today including tinyOS, Mantis, Contiki etc. These systems are designed for compatibility with low-power consuming devices. In order to fully support these devices, the operating systems are required to satisfy the following requirements:

- Scalability: IoT operating systems need to support heterogeneous hardware such as microcontrollers, hardware resources, radio technology devices and devices with memory constraints. For instance, Linux is not a good IoT operating system because it does not offer support 8 or 16-bit microcontrollers (Bovet and Cesati, 2001). A good IoT operating system should support these devices without the need for more advanced components such as Memory Management Unit(MMU) (Baccelli, E. et al., 2013).
- Efficiency: IoT operating systems should support low memory devices but also have memory efficient data structures. They should include power saving features; for example sleep cycles for when the device is idle.
- CPU: The bulk of operations running on the operating system need to be kept very low, so as to save energy. This can be achieved through efficient scheduling and multi-threading.
- Language: The operating system should be easily programmable. That is, it should be written in simple programming languages such as Java or C. This way it uses standard tools and libraries which are familiar to software developers thereby negating the need for a learning curve.
- Reliability: IoT devices are usually used in critical applications where constant physical access to the device may be difficult. As such, it is important that the operating system is developed and tested properly

for reliability before deployment, so as to prevent frequent maintenance.

- Network Adaptability: The operating system should be able to support new network technologies such as IPV6, as well as existing and more constrained networks e.g. 6LoWPAN (Canonical, 2015).

It can be deduced then, from the above requirements, that an IoT operating system should satisfy the requirements of heterogeneous devices with varying capacities and capabilities as well as operate at a very high efficiency. The system should also be portable and scalable so as to minimize redundant code development and reduce maintenance costs.

CHAPTER 3: INTRODUCTION TO RIOT

RIOT (Real-time Internet of Things Operating System) is an open-source operating system software for Internet of Things devices. The kernel was originally developed in 2008, as part of a research project, to monitor wireless sensor networks with the aim of achieving reliability and real-time guarantees. In the few years since it was deployed, it was improved with the goal of creating an IoT operating system. For instance, in 2010, it was developed to include support for various network protocols such as 6LoWPAN and RPL (routing protocol for low power networks). By 2013, RIOT was released to the public as a full-fledged IoT operating system (Hahm, 2014).

The operating system is based on a microkernel architecture and designed to support a range of devices from small 8-bit microcontrollers to high-capacity processors. The architecture of the kernel is a fundamental design aspect of the operating system because it influences the characteristics of the system in terms of modularity. Operating systems typically can either be designed using a monolithic, layered or microkernel approach. The monolithic method is very easy to implement and the components of the system are simply interwoven. The system generally will lack modularity and is one that would be complex and difficult to debug. The layered model segments the system in a hierarchical way, thereby introducing some modularity. The lower layers typically will provide functions and services to the upper layers. In the microkernel model,

however, the whole operating system is split up into small modules that perform specific functions. The benefit of this design includes memory protection for user applications and OS components. It also improves the reliability of the system as bugs in individual components can easily be identified and fixed (Baccelli, E. et al., 2013).

The RIOT operating system is targeted at systems which are too low in power to run Linux operating system. It supports drivers used for various external devices such as radio transceivers, environmental sensors, acceleration sensors, gyroscopes and light sensors. The system also provides various configuration sets for hardware platforms, thereby supporting different architectures and development boards such as Arduino Due or even Bluetooth enabled devices such as Nordic nRF51822. The operating system aims to satisfy all the requirements highlighted in Section 2.3.2 and provides a solution for devices not compatible with traditional operating systems. Although research has been done to adapt existing traditional operating systems for IoT, features such as maximum energy efficiency and real-time support, cannot simply be added to the system due to the complexity of implementing such features.

By offering full multi-threading, real-time capabilities, adaptability, modularity and energy efficiency, RIOT can be leveraged for IoT devices as it satisfies the fundamental requirements of an IoT operating system.

3.1 RIOT FEATURES

3.1.1 DEVELOPER-FRIENDLINESS

The RIOT source code is written in standard C++. This is of great benefit because standard C tools and libraries are already familiar to software developers. Unlike tinyOS, which is also open-source, but written in nesC (network embedded systems C), there is no learning curve needed for developing applications in RIOT. Developers are already familiar with the language, APIs (e.g. POSIX sockets), debugging tools and documentation process. They can also leverage standard development tools already familiar to them such as GNU Debugger, GNU Compiler Collection and Wireshark.

RIOT can be run as a process on Linux or MacOS and across multiple bit-platforms (8, 16 and 32-bit), using the ‘native’ port. This means no actual hardware is required to develop applications. The application code can be coded once, compiled and tested before deployment to any supported board. This allows the developer to test the functionality of the application, as well as analyse the code for bugs initially, without having to deal with problems when the application is already mounted on the board.

3.1.2 RESOURCE-FRIENDLINESS

Compared to other operating systems with similar memory capacity such as tinyOS and Contiki, RIOT provides full multi-threading as well as real-time support. A major requirement for IoT devices is energy-efficiency and RIOT accomplishes this through low threading overhead during multithreading and low interrupt latency during real-time support (Emmanuel et al, 2015).

The RIOT operating system also requires a very low internal RAM (~1.5kB) to function.

3.1.3 IoT-FRIENDLILESS

RIOT supports the IEEE 802.15.4MAC standard which is the internet standard for low power embedded devices. This standard supports very small packet sizes and can be susceptible to packet loss due to various factors such as congestion in the network or weak radio signals. The operating system also has multiple network stacks; this allows it to support various standard network protocols including TCP, UDP and IPv6.

3.2 RIOT ARCHITECTURE

The RIOT operating system is based on a microkernel architecture; a structure based on achieving high modularity, robustness and abstraction. The objective of this design is the ‘separation of duties’ across the software components, thereby ensuring the operating system can be adaptable for different hardware platforms. This allows the operating system to be portable between different systems including current and future platforms. This architectural design also ensures that minimal internal memory is required for operations, as there is less dependency between components or modules.

The operating system uses a “tickless scheduler” for scheduling the normal functions of the operating system. This allows the system to avoid the trade-off of a fixed interval when scheduling interrupts for different

operations (Baccelli, E. et al., 2013). For example, if the timer interrupt is set to tick at too low a frequency e.g. 5 milliseconds, then the need for the kernel time keeping is not really efficient. Too high a frequency and a lot of processor time is used up processing the interrupt. Although complex to implement, the advantage of using a tickless scheduler is low power consumption. In this method of scheduling, the kernel schedules interrupts relative to the current running process. This means if there is no pending tasks, the system can go into an idle or sleep mode, thereby minimising energy consumption.

The operating system repository consists of various components which can be grouped into hardware dependent code, hardware independent code, and a support directory. Abstraction is mainly achieved by ensuring that the hardware dependent code is separated from hardware-independent code (Hahm O, 2014).

3.2.1 HARDWARE DEPENDENT CODE

This consists of the code that is needed for the various hardware devices compatible with RIOT. The codes are reduced to a minimum and are separated from the Kernel. They include source codes for the CPU, as well as boards supported by the operating system.

- The CPU directory comprises source code implementations such as code for clock initialization, power management, task switching, stack

handling, start up and interrupt handling for the various hardware platforms compatible with RIOT.

- The boards directory contains the pin configuration and board and clock initialization codes for external devices such as sensors and radios.

3.2.2 HARDWARE INDEPENDENT CODE

This consists of code that is needed for communication between an application running on the RIOT OS and the various hardware platforms. It includes code for the microkernel, system, network stack and the various drivers compatible with RIOT.

- The microkernel directory contains code for the core functionality of the operating system. This includes code for interrupt handling, inter-process communication, the scheduler, the timer, power management and multithreading.
- The drivers directory contains code for the drivers for specific devices such as sensors or radio transceivers. These drivers are installed on the device as part of the operating system application.
- The systems directory contains code for all tools and libraries utilized by a typical operating system.

3.2.2 RIOT NETWORK STACK

The RIOT network stack is slightly different from a traditional IP network stack. Due to various constraints of IoT devices, all layers of the network stack are modified to support new protocols.

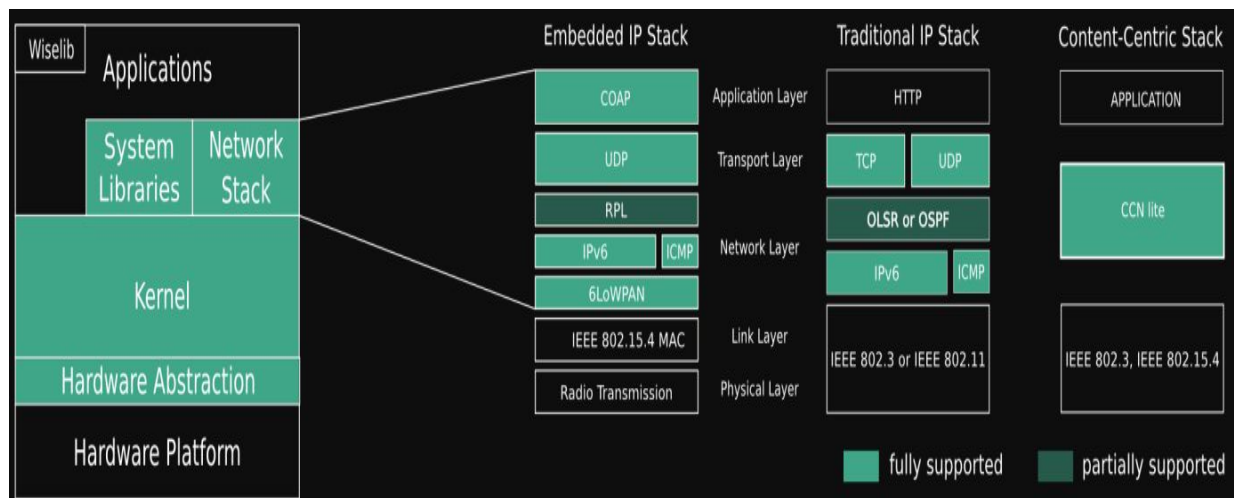


Figure 3: RIOT Network Stack

Image source : <http://wiki.eclipse.org/images/9/92/EclipseGrenoble2015-RIOT.pdf>

Firstly, in the application layer, while HTTP is used in a traditional IP stack for the internet, it is too complex for constrained IoT nodes. The Internet Engineering Task Force (IETF) is currently developing a Constrained Application Protocol (CoAP) to be used on constrained nodes in low power networks (Baccelli, E. et al., 2013). The RIOT network stack currently supports CoAP, Concise Binary Object Representation (CBOR) and Universal Binary JSON specification (UBJSON) at the application layer.

Secondly, the transport layer supports only User Datagram Protocol (UDP), due to the complexity of implementing a more reliable and connectionless protocol i.e. Transmission Control Protocol (TCP).

Thirdly, at the network layer, high level protocols such as Border Gateway Protocol (BGP) and Open Shortest Path First (OSPF) are not compatible with IoT devices because the devices do not have enough power to run such high-end protocols. Rather, the IETF has proposed and standardised RPL, which is a routing protocol for low-power networks (Kuryla, 2010). Also, due to the packet fragmentation constraint of the IEEE 802.15.4 standard, an adaptation layer(6LoWPAN) is required for header compression of the IoT datagrams. The RIOT network stack currently supports IPv6, 6LoWPAN, RPL and CCN-lite, for content-centric networking support, at the network layer.

Fourthly, typical internet protocols such as IEEE 802.11 Wi-Fi used at the link layer provide a much higher bandwidth of several gigabits/second compared to IEEE 802.15.4 used by IoT networks which only provide about 500 kilobits/second. The consequence of this is frequent packet loss due to very small packet sizes. The RIOT developers currently plan to include support for Bluetooth in the link layer later on in the future.

Finally, at the physical layer, IoT devices transmit data using radio technology that can only transmit packets of about 150 bytes. This is a problem as standard IPv4 requires a minimum payload of 576 bytes, while

IPv6 requires 1280 bytes. Thus, the standard TCP/IP stack must be adapted to support IoT data transmission.

CHAPTER 4: PROJECT ANALYSIS

This chapter covers the 'Requirement Analysis' phase of the project. It analyses the technical and business requirements needed to satisfy the scope of the project; which is to uncover vulnerabilities in the RIOT operating system. This chapter is organised into three main sections: a project requirement analysis to discuss project stakeholders and technical requirements needed to undertake the project, a project plan for a proposed timeline to planning and executing the project, as well as the risk analysis of the project.

4.1 PROJECT REQUIREMENTS

The aim of this project is to detect, uncover and report any vulnerabilities in the RIOT operating system. Aligned with this project are a few stakeholders who have different roles in order to achieve the scope of the project.

	Stakeholder	Role
Supervisor	Dr. Julio Cesar Hernandez-Castro	<ul style="list-style-type: none"> • Provide feedback on the project plan and implementation. • Supervise project and highlight concerns and opportunities to improve the project. • Encourage and maintain communication throughout the duration of the project.
Technical Developer	Kelechi Nnorom	<ul style="list-style-type: none"> • Schedule regular meetings with stakeholders to discuss the project. • Research the project thoroughly and report findings. • Document results for the different stages of the project.

Table 1: Project Stakeholders

The technical requirements needed to undertake the project are detailed below:

Requirement	Description
RIOT source code	This is needed to perform the security analysis RIOT. It was obtained from GitHub on the 16 th of June, 2015.
Computer / Virtual machine	RIOT was installed on a Linux environment virtual machine. This was done in order to perform the analysis of the OS.
Hardware device (MBED NXP LPC1768)	A board compatible with the RIOT operating system is needed for penetration testing and analysis.

Analysis tools	Both commercial and open source analysis tools are required for source code and penetration tests of the RIOT operating system.
Logbook	This is updated regularly with the project execution stages as well as any problems encountered throughout the duration of the project.

Table 2: Project Technical Requirements

The following section provides a guide for the lifecycle and execution of the project. It includes a proposed timeline for the completion of the project.

4.2 PROJECT PLAN

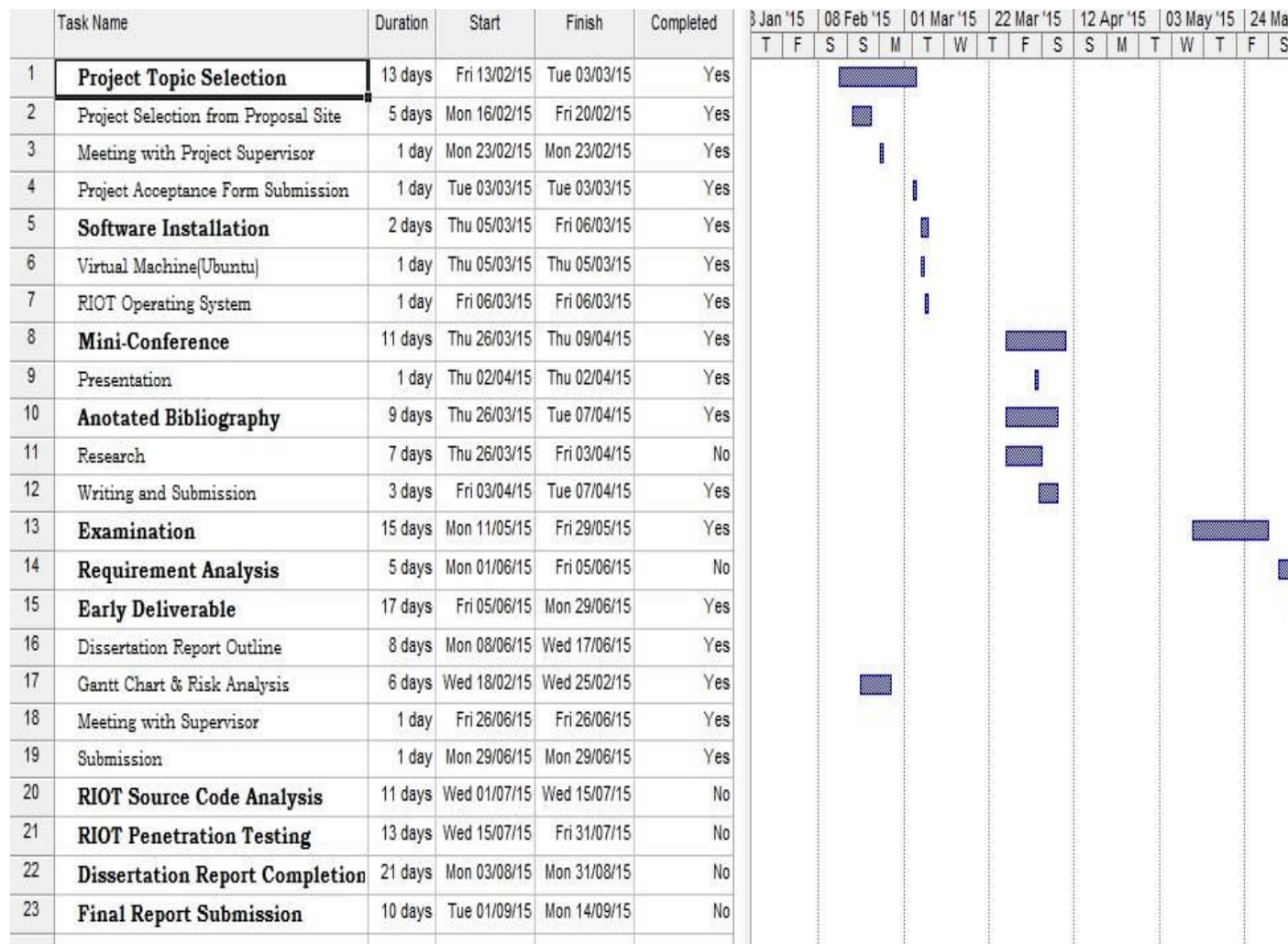


Figure 4: Project Plan

4.3 RISK ANALYSIS

This section describes the risks that could influence the timeline for the completion of the project. In this context, project risks includes risks that may affect the project schedule as well as health and safety risks.

ASSET	PROJECT RISK	SEVERITY	RISK EFFECT	RISK MITIGATION
Kelechi Nnorom	Medical complication	Low	Delay in project completion	Maintain healthy lifestyle
Hardware - Laptop - USB	Loss or damage	High	Delay in project completion	Backup of project files –Google Drive cloud storage
Mbed Lpc1768 device	Unavailable or out of stock	Medium	Research new compatible device	Native port as backup option
Virtual machine	Crash	High	Delay in project completion	Backup virtual machine using snapshot.

Analysis tools	Not trusted for accurate results	High	Biased test results	Use of only highly-rated testing tools
----------------	----------------------------------	------	---------------------	--

Table 3: Project Risk Assessment

CHAPTER 5: METHODOLOGY

This chapter discusses the methods and procedures for the security analysis of RIOT operating system, outlined in section 1.2. These include a source code analysis of the RIOT source code (obtained from GitHub) and a penetration test. The source code analysis identifies and reports any vulnerabilities in the source code, while a penetration test aims to exploit these vulnerabilities to determine whether unauthorised access to internal resources or any other form of malicious activity is possible.

5.1 INSTALLATION OF RIOT

The RIOT operating system repository was obtained from GitHub on the 16th of June 2015. It was downloaded by using the ‘git’ clone command.


```

kelz@kelz-VirtualBox:~$
kelz@kelz-VirtualBox:~$ git clone https://github.com/RIOT-OS/RIOT RIOT
Cloning into 'RIOT'...
remote: Counting objects: 62333, done.
remote: Compressing objects: 100% (39/39), done.
remote: Total 62333 (delta 14), reused 0 (delta 0), pack-reused 62294
Receiving objects: 100% (62333/62333), 19.57 MiB | 810.00 KiB/s, done.
Resolving deltas: 100% (40583/40583), done.
Checking connectivity... done.
Checking out files: 100% (2427/2427), done.
kelz@kelz-VirtualBox:~$

```

Figure 5: RIOT Installation

```

kelz@kelz-VirtualBox:~$ cd RIOT/
kelz@kelz-VirtualBox:~/RIOT$ ls
boards      dist        examples    Makefile.application  Makefile.defaultmodules  Makefile.modules      pkg          tests
CONTRIBUTING.md  doc        keyword_space.sed  Makefile.base         Makefile.dep            Makefile.pseudomodules  README.md
core          doc.txt    LICENSE       Makefile.builttests   Makefile.docker         Makefile.scan-build    release-notes.txt
cpu          drivers    Makefile      Makefile.cflags       Makefile.include        Makefile.vars          sys

```

Figure 6: RIOT sub-directory

5.2 SOURCE CODE ANALYSIS

Source code analysis is an automated test of the code of a program to uncover security issues and flaws within the code. It is a technique that involves checking for coding mistakes, as well as comparing the code against known problem signatures, usually stored in a database. Source code analysis can either be static or dynamic. Static source code analysis involves examining the code without executing the program while dynamic source code analysis is done on a running program or in real-time.

For the project, static source code analysis was done to reveal minor errors in the various components of the operating system. These errors could affect the performance especially when an application is developed and run on the operating system.

The RIOT operating system is written in standard C. Although the language is widely used and easy to learn, it is fundamentally an unsafe language to use to write code (Thien, L. 2002). This is because most of its standard library string functions are susceptible to buffer overflow and string format attacks if not used properly. Below are some of the major vulnerabilities that are flagged by source code analysis tools:

- Buffer overflow risks: This is code that contains the functions `strcpy()`, `strcat()`, `gets()`, `sprintf()` or `scanf()`. These functions typically use internal buffers for their operations. There is a possibility that buffer overflow could occur because these functions do not perform array boundary checks.
- String format: This is code that contains the functions `printf()`, `vprintf()`, `vsprintf()`, `fprintf` and `syslog()`. These functions are mainly used for standard output or output to a buffer. However, these can be exploited because the functions assume an infinite length for the buffer. A format string attack can be carried out by inserting format specifiers in the 'string' variable of the function to manipulate the memory stack so that an attacker can write any value of choice, including programs to be executed (Thien, L. 2002).
- Race conditions: This is code that contains the functions `access()`, `chgrp()`, `chmod()`, `tmpfile()`, `chown()`, `tmpnam()`, `tempnam()` and `mktemp()`. These are functions that are used to create files as well as modify existing file ownership and permissions. Some of these

commands are dependent on the timing of the execution of other commands which could lead to unpredictable results. For instance, if two processes are trying to access a single resource, they may both be locked out. Also, one might acquire the privileges of the other. If this is the case and an attacker's program acquires the privileges of a high-privileged (setuid) program, it can create a vulnerability as the attacker can exploit this and execute any program with the new elevated permission.

- Shell meta-character: This is code that includes the functions `popen()`, `system()` and most `exec()` functions. These functions are system calls used to execute external programs. They can be exploited by an attacker mainly because the `system()` function accepts an arbitrary buffer as the command line to execute a program.

For the tests, various source code analysis tools were used and the results were then compared. This was done to be as comprehensive as possible because the tools will not all have the same database composition of vulnerability signatures.

5.2.1 RIOT SOURCE CODE ANALYSIS TOOLS

5.2.1.1 Flawfinder

Flawfinder is a source code analysis tool developed by David Wheeler, a well-known security expert (Wheeler, 2015). According to the SANS Institute, a company that specializes in information security, Flawfinder is one of the best tools that can be leveraged for source code analysis (Thien,

L. 2002). The software can be used in both Unix and Windows environments. I also realised that it found more vulnerabilities than any other tool I used for source code analysis.

Flawfinder works by analysing code and outputting possible vulnerabilities sorted by risk level i.e. the higher the risk level number for a line of code, the higher the vulnerability. The software analyses the code for problems such as buffer overflow risks, wrong string format, race conditions, possible shell meta-character problems and wrong use of the random number generator library.

Flawfinder result analysis for RIOT operating system

In order to perform the analysis of the RIOT operating system source code, I was required to accomplish the following tasks:

Installation of Flawfinder

I installed Flawfinder version 1.31 by downloading the .tar file from <http://www.dwheeler.com/flawfinder/flawfinder-1.31.tar.gz>. Please refer to APPENDIX A for installation screenshots.

Execution of the program against the RIOT source code

This was done by navigating to the Flawfinder directory and executing the Flawfinder program against the RIOT operating system directory. The result was then redirected to an output file which I analysed.

```

knnorom1@knnorom1-VirtualBox:~/SourceCodeAnalysisTools/FlawFinder/Flawfinder-1.31$ ls
announcement      cwe.l             flawfinder.ps      junk.c             setup.cfg          test-results.html
ChangeLog          flawfinder         flawfinder.spec     makefile           setup.py           test-results.txt
COPYING            flawfinder.1       flawfinder.spec.orig MANIFEST.in        sloctest.c
correct-results.html flawfinder.1.gz    flawtest.c         no-ending-newline.c test2.c
correct-results.txt flawfinder.pdf     INSTALL.txt         README            test.c
knnorom1@knnorom1-VirtualBox:~/SourceCodeAnalysisTools/FlawFinder/Flawfinder-1.31$
knnorom1@knnorom1-VirtualBox:~/SourceCodeAnalysisTools/FlawFinder/Flawfinder-1.31$
knnorom1@knnorom1-VirtualBox:~/SourceCodeAnalysisTools/FlawFinder/Flawfinder-1.31$ flawfinder /home/knnorom1/RIOT > /home/knnorom1/output.txt
Warning: Skipping directory with initial dot /home/knnorom1/RIOT/.git
Warning: Skipping symbolic link file /home/knnorom1/RIOT/cpu/k60/include/system_MK60DZ10.h
knnorom1@knnorom1-VirtualBox:~/SourceCodeAnalysisTools/FlawFinder/Flawfinder-1.31$

```

Figure 7: Executing Flawfinder against RIOT directory

Result Analysis

```

Easily used incorrectly; doesn't always \0-terminate or check for invalid
pointers (CWE-120).

ANALYSIS SUMMARY:

Hits = 1306
Lines analyzed = 433138 in approximately 41.60 seconds (10413 lines/second)
Physical Source Lines of Code (SLOC) = 284542
Hits@level = [0] 0 [1] 260 [2] 985 [3] 8 [4] 53 [5] 0
Hits@level+ = [0+] 1306 [1+] 1306 [2+] 1046 [3+] 61 [4+] 53 [5+] 0
Hits/KSLOC@level+ = [0+] 4.58983 [1+] 4.58983 [2+] 3.67608 [3+] 0.21438 [4+] 0.186264 [5+] 0
Symlinks skipped = 1 (--allowlink overrides but see doc for security issue)
Dot directories skipped = 1 (--followdotdir overrides)
Minimum risk level = 1
Not every hit is necessarily a security vulnerability.
There may be other security vulnerabilities; review your code!
See 'Secure Programming for Linux and Unix HOWTO'
(http://www.dwheeler.com/secure-programs) for more information.
knnorom1@knnorom1-VirtualBox:~/SourceCodeAnalysisTools/FlawFinder/Flawfinder-1.31$

```

Figure 8: Flawfinder Output

From the Figure 8, a total of 433138 lines of code were analysed by the tool. The total number of 'Hits' or potential vulnerabilities was **1306** with the highest risk level of 4; which is moderate according to Flawfinder. A breakdown of the results of the test is shown below.

Directory	Vulnerability	Potential	# Files	Error
	Breakdown Count	Flaws	Analysed	Density
Boards	Buffer overflow – 100	123	207	0.59
	String format – 1			

	Integer format - 3			
	Shell character - 4			
	Exploitable 'usleep' command - 7			
	Exploitable 'fopen' command - 8			
CPU	Buffer overflow - 98	106	589	0.17
	String format - 4			
	Integer format - 2			
	Exploitable 'memalign' command - 2			
Core	Buffer overflow - 5	8	46	0.17
	String format - 3			
Dist	Buffer overflow - 63	84	3	28
	String format - 3			
	Integer format - 6			
	Shell character - 3			
	Exploitable 'open' - 6			
	Exploitable 'usleep' command - 3			
Drivers	Buffer overflow - 65	65	158	0.41
Examples	Buffer overflow - 24	30	17	1.76
	Integer format - 6			

Tests	Buffer overflow - 228	268	149	1.79
	String format - 1			
	Integer format - 38			
	Exploitable 'usleep' command - 1			
Sys	Buffer overflow - 558	622	425	1.46
	String format - 14			
	Integer format - 40			
	Exploitable 'usleep' command - 2			

Table 4: Breakdown of Flawfinder Output

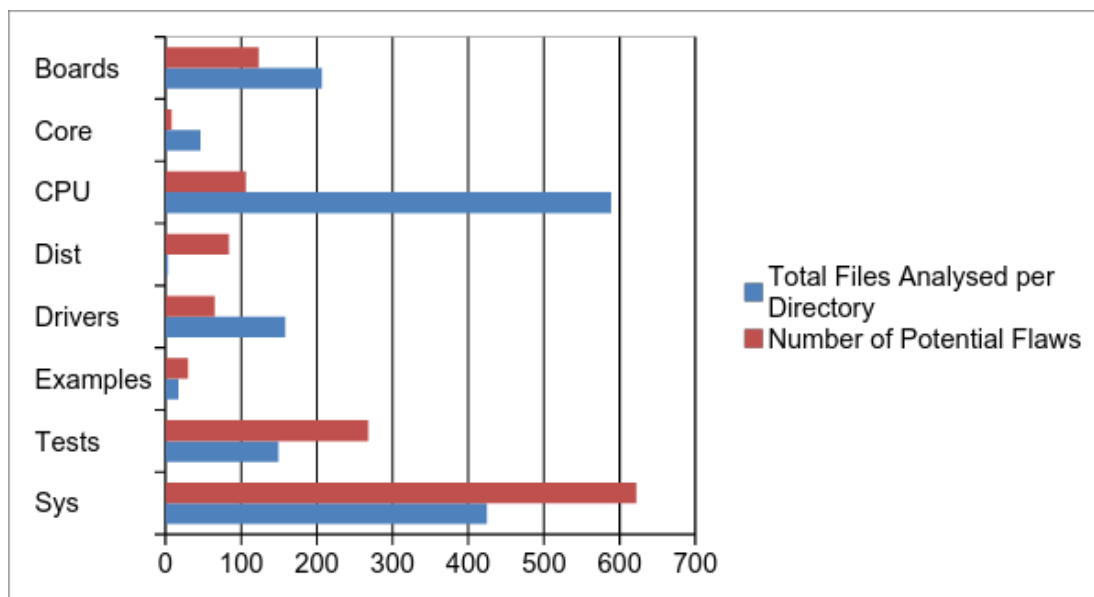


Figure 9: Bar chart of Potential Flaws per Directory

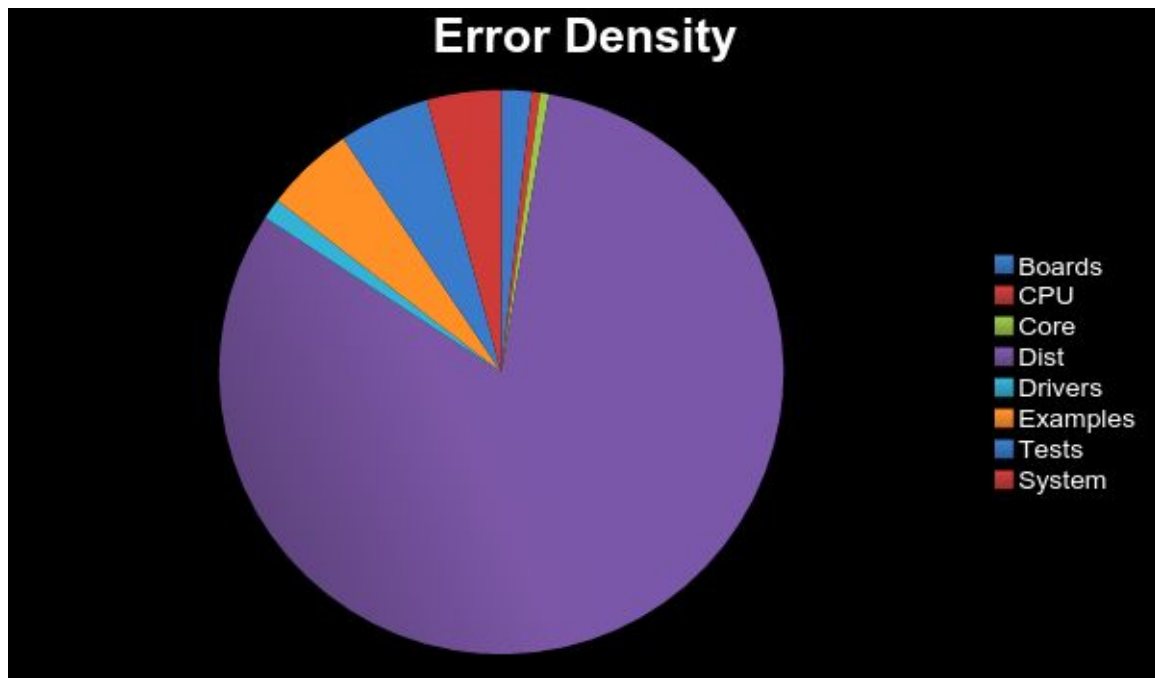


Figure 10: Pie chart of Error Density - Flawfinder

The total number of vulnerabilities for each risk level were:

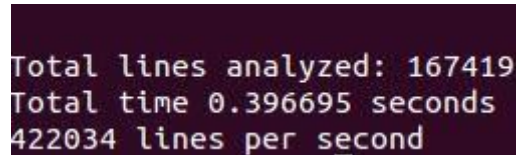
- Level 1- 260
- Level 2 - 985
- Level 3 - 8
- Level 4 - 53

I found the Flawfinder software very easy to use. After installing it, the directory containing the code only needed to be given as input in the command line. The program then searches through the sub-directories recursively for .C files and analyses the contents of the files.

5.2.1.2 RATS (Rough Auditing Tool for Security)

The Rough Auditing Tool for Security is a free open source code analysis tool developed originally by Secure Software Incorporated, a company

recently acquired by Fortify (CERN Computer Security, 2015). It can be used to analyse code written in Perl, C++, C, PHP and Python programming languages. It is also able to generate detailed output in both HTML and XML. It analyses code for buffer overflows, race conditions and TOCTOU (time-of-check, time-of-use). I also found it performed a quicker analysis of the source code, compared to any other tool I used for the test.



```
Total lines analyzed: 167419
Total time 0.396695 seconds
422034 lines per second
```

Figure 11: Fast RATS Analysis of RIOT OS

The only downside to this tool is that it does not perform an exhaustive search. For instance, it does not analyse header (.h) files.

RATS Analysis of RIOT Operating System

In order to analyse the RIOT source code, I undertook the following tasks:

Installation of RATS

This was pretty straightforward. I downloaded the source code and tried installing it. I ran into some problems when trying to build and run the program. This was because the tool's default output method is .xml and so it requires "Expat", an XML parser library, installed first. Please refer to APPENDIX B for installation screenshots.

Result Analysis

The output from the RATS tool provides a detailed assessment on possible vulnerabilities and suggestions on fixes. A total of 466 vulnerabilities in the RIOT source code were found by the tool and classified as High(414) and Medium(52) risk level. Below is a table of the breakdown of the output by the RATS tool:

Risk Classification	Vulnerability	Code	Count
High	Potential buffer Overflow	Fixed buffer size	227
		Strcpy	11
		Strcat	2
		Strncat	2
	String format	Printf	152
		vprintf	1
		sprintf	2
		getopt	4
		getopt_long	1
	Shell commands	compile	1
		system	4
		popen	3
	Environment variable	getenv	1
	Use of 'eval' command	eval	3
Medium	Potential buffer overflow	realloc	6
		read	24
		getchar	2
	Race condition	signal	14
	Shell command	open	1
		system	1

	Time check/Time use	of of	stat	2
	Use of 'srand'.		srand	2

Table 5: Breakdown of RATS Output

Directory	# Files Analyzed	Number of Errors		Error Density
		High	Medium	
Boards	99	30	16	0.46
Core	14	0	0	0
CPU	220	17	5	0.1
Dist	12	35	16	4.8
Drivers	58	34	0	0.58
Examples	14	18	0	1.2
Tests	128	121	7	1
Sys	215	159	8	0.77

Table 6: Error Density per Directory

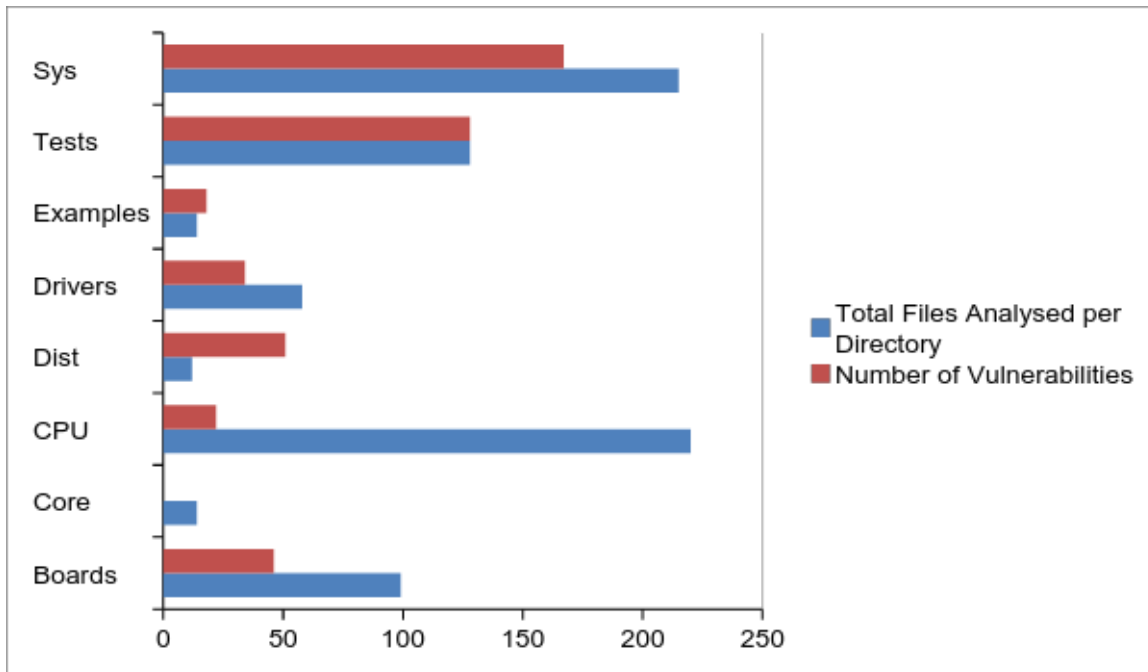


Figure 12: Bar chart of Vulnerabilities per Directory

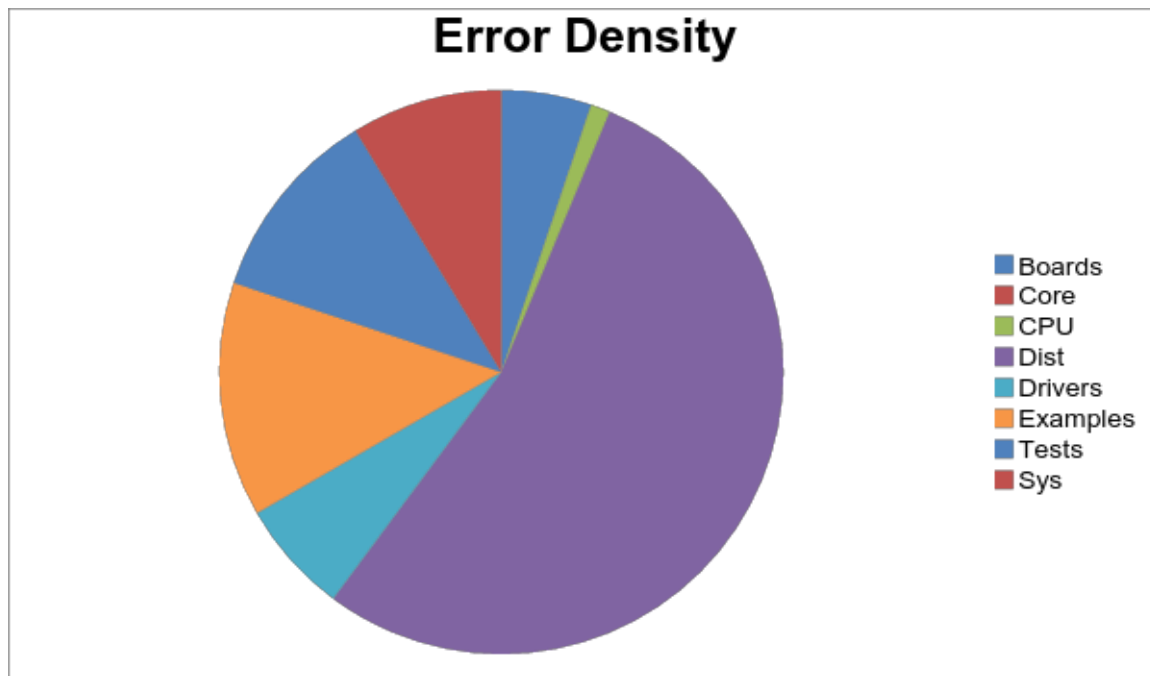


Figure 13: Pie chart of Error Density - RATS

I found the RATS tool quite simple to use. Like Flawfinder, the directory to be analysed was the only input required in the command line. RATS also allows the user to specify options to better streamline the output such as the vulnerability database to be loaded, length of output, programming language to analyse etc.

5.2.1.3 Yasca (Yet Another Source Code Analyser)

Yasca is an analysis tool developed by Michael Scovetta in 2007 for quality assurance testing and vulnerability testing. It can be run on both Windows and Unix platforms. The tool is typically used to analyse source code written in Java, C++, HTML, JavaScript, ColdFusion, PHP, COBOL, .NET, ASP etc. In order to accomplish this, Yasca leverages other external open source code analysis tools such as RATS and CppCheck via plugins

(Scovetta, 2015). The tool then generates output reports in HTML, CSV, XML and other formats depending on the user specification.

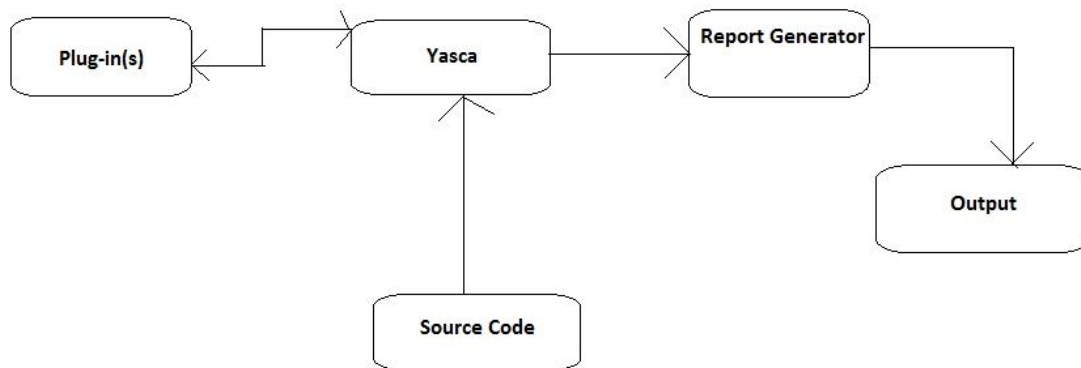


Figure 14: Yasca Analysis Process

Yasca Analysis of RIOT Operating System

Installation of Yasca

I installed Yasca by downloading the source code on a Linux virtual machine. I also had to install Java and PHP. Running the tool proved to be a little difficult at first. I kept running into an error: “bad interpreter: No such file or directory”.

```
student@csvm2C25:~/Desktop/Yasca$  
student@csvm2C25:~/Desktop/Yasca$  
student@csvm2C25:~/Desktop/Yasca$  
student@csvm2C25:~/Desktop/Yasca$ sudo yasca resources/test  
sudo: yasca: command not found  
student@csvm2C25:~/Desktop/Yasca$ ls -l yasca  
-rwxr-xr-x 1 student co876 80 Jul 14 2009 yasca  
student@csvm2C25:~/Desktop/Yasca$ ./yasca  
bash: ./yasca: /bin/sh^M: bad interpreter: No such file or directory  
student@csvm2C25:~/Desktop/Yasca$ ./yasca.exe
```

Figure 15: "Bad Interpreter" Error

After some research, I found out this error is usually caused when scripts are edited on a windows based system and transferred into Unix servers using FTP. I fixed this problem by running the following Perl command on the yasca script: “perl -i -pe’s/\r\$/;’ yasca”

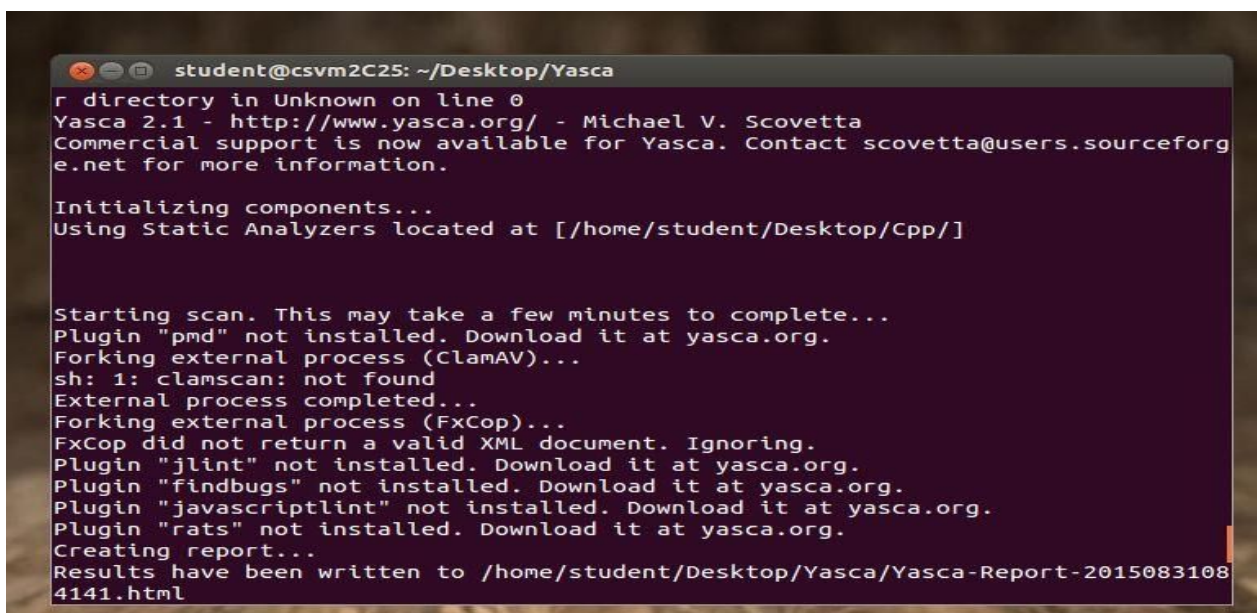
Also, because Yasca leverages external tools for analysis, I tried to run the program using the RATS and CppCheck tools as they were the only plugins available that analyse C files. I was able to get CppCheck to run but had problems with RATS (see Figures 16 and 17). There seems to be a problem with the PHP source code for the RATS plugin. I sent an email to the Yasca team about the bug.

```
e.net for more information.

Initializing components...
Using Static Analyzers located at [/home/student/Desktop/RATS/]

Starting scan. This may take a few minutes to complete...
Plugin "pmd" not installed. Download it at yasca.org.
Forking external process (ClamAV)...
sh: 1: clamscan: not found
External process completed...
Forking external process (FxCop)...
FxCop did not return a valid XML document. Ignoring.
Plugin "jlint" not installed. Download it at yasca.org.
Plugin "findbugs" not installed. Download it at yasca.org.
Plugin "javascriptlint" not installed. Download it at yasca.org.
Plugin "cppcheck" not installed. Download it at yasca.org.
Forking external process (RATS)...
RATS did not return a valid XML document. Ignoring.
```

Figure 16: Failed Execution of Yasca with RATS plugin



```
student@csvm2C25: ~/Desktop/Yasca
r directory in Unknown on line 0
Yasca 2.1 - http://www.yasca.org/ - Michael V. Scovetta
Commercial support is now available for Yasca. Contact scovetta@users.sourceforge
e.net for more information.

Initializing components...
Using Static Analyzers located at [/home/student/Desktop/Cpp/]

Starting scan. This may take a few minutes to complete...
Plugin "pmd" not installed. Download it at yasca.org.
Forking external process (ClamAV)...
sh: 1: clamscan: not found
External process completed...
Forking external process (FxCop)...
FxCop did not return a valid XML document. Ignoring.
Plugin "jlint" not installed. Download it at yasca.org.
Plugin "findbugs" not installed. Download it at yasca.org.
Plugin "javascriptlint" not installed. Download it at yasca.org.
Plugin "rats" not installed. Download it at yasca.org.
Creating report...
Results have been written to /home/student/Desktop/Yasca/Yasca-Report-2015083108
4141.html
```

Figure 17: Execution of Yasca with CPP plugin

Result Analysis

The result was outputted to a HTML file after about 1 minute. In total 1902 vulnerabilities were found in the RIOT operating system directory. However, 20 of them were for backup files that should not be part of the code distribution package while a massive 1389 vulnerabilities were false positives caused from comments. Therefore, only 493 vulnerabilities were observed in total. A total of 2611 files were analysed including source code, header and .git files.

Directory	Vulnerability Breakdown	# Vulnerabilities	# Files Analysed	Error Density
Boards	This was caused mainly due to commands such as fgets, strlen & sscanf.	33	429	0.07
CPU	This was due to commands such as strlen, native_write() etc.	41	724	0.05
Core	Vulnerabilities observed were mainly due to comments	13	48	0.27
Dist	This was due to functions such as strcat, strncpy and sprintf	24	85	0.28
Drivers	Vulnerabilities were caused by 'char' and strncpy.	27	200	0.14
Examples	Due to mostly commands such as puts(), strncpy & char	13	170	0.07

Tests	Vulnerabilities in this directory were highlighted due to the use of strncpy, strlen, and function calls banned by Microsoft.	42	324	0.13
Sys	Vulnerabilities were caused by buffer overflow risks as well as pointers to stack in the return statement.	290	518	0.56

Table 7: Breakdown of Yasca Output

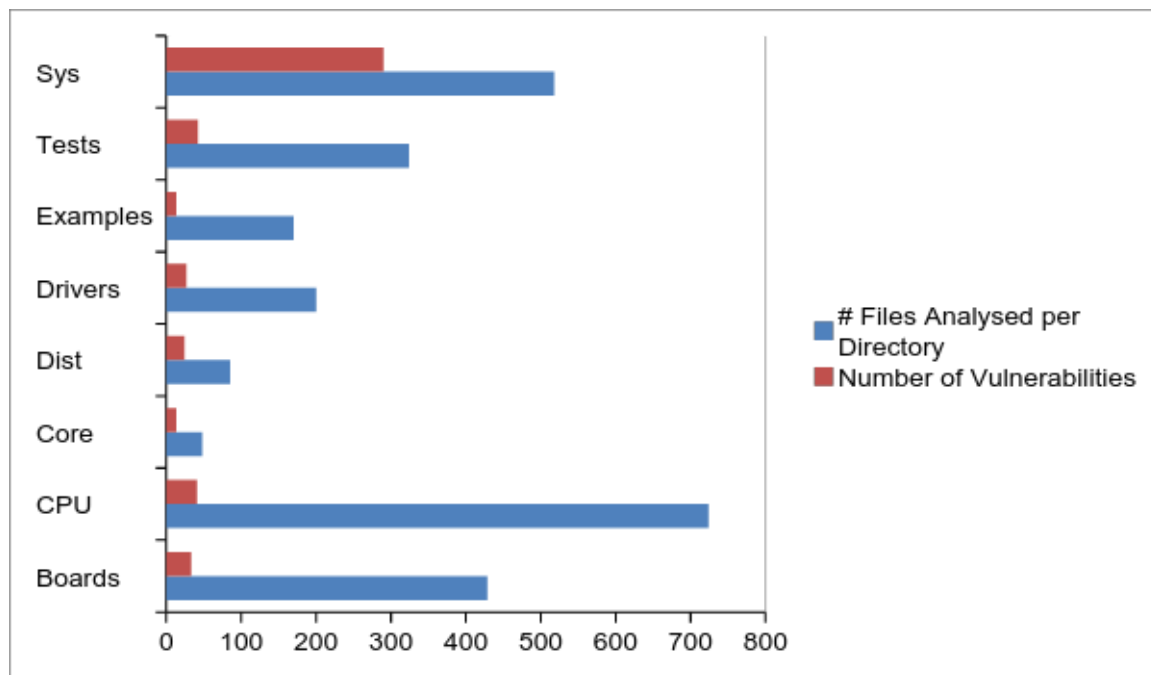


Figure 18: Bar chart of Vulnerabilities per Directory

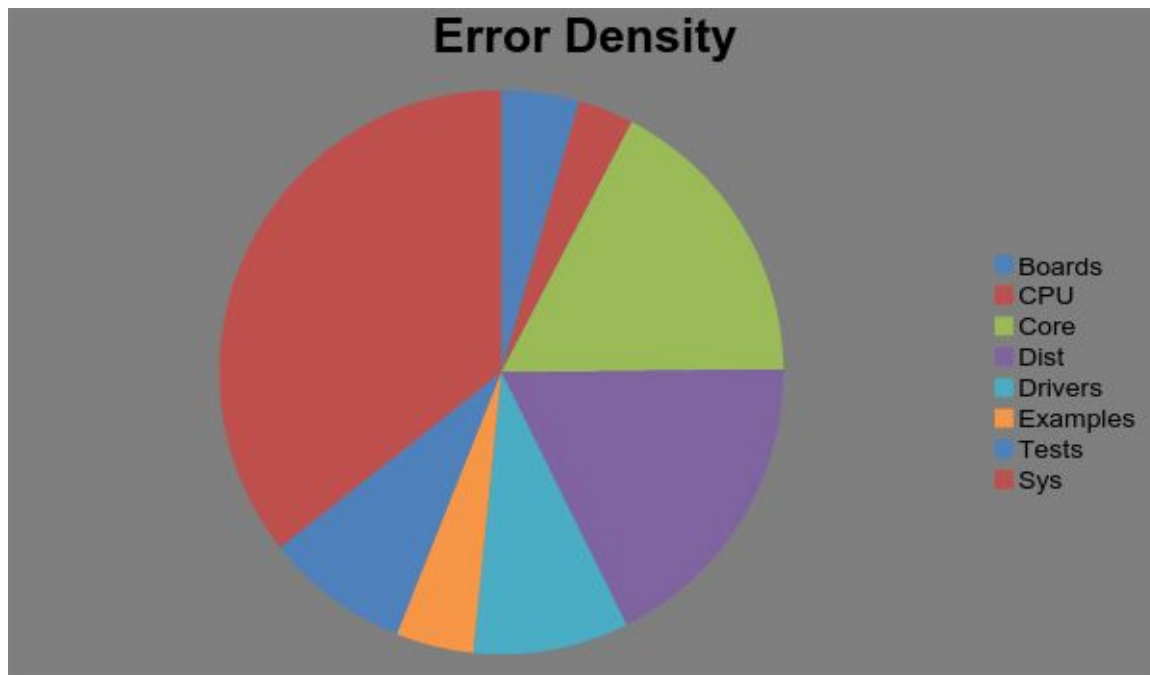


Figure 19: Pie chart of Error Density - Yasca

The Yasca tool presented the output in a decent and readable format. However, I noticed there were a lot of false positives due to comments in the source code being picked up as vulnerabilities by the tool. Also, it did not give a detailed explanation about the vulnerabilities that were detected. In most cases, vulnerability classification were quite vague and included unsafe function, bad code quality etc.

5.2.1.4 Visual Code Grepper (VCG)

Visual Code Grepper is an open source analysis tool that can be used to analyse code written in C++, C#, VisualBasic, PHP, Java and PL/SQL (Checkmarx, 2014). It can be run on both Linux and Windows environments. The tool requires the user to specify the programming

language being analysed and then categorises the vulnerabilities based on 6 levels with '1' being the highest or most critical level.

VCG Analysis of RIOT Operating System

Installation of VCG

I installed the tool on the windows environment. This was easy and very straightforward as the windows .msi installation file was downloaded from the Source Forge website (<http://sourceforge.net/projects/visualcodegrepp/>) and then the software was installed.

Result Analysis

The source code files were first loaded from the RIOT directory. In total 1595 files in the directory were analysed including .cpp, .c and .h files.

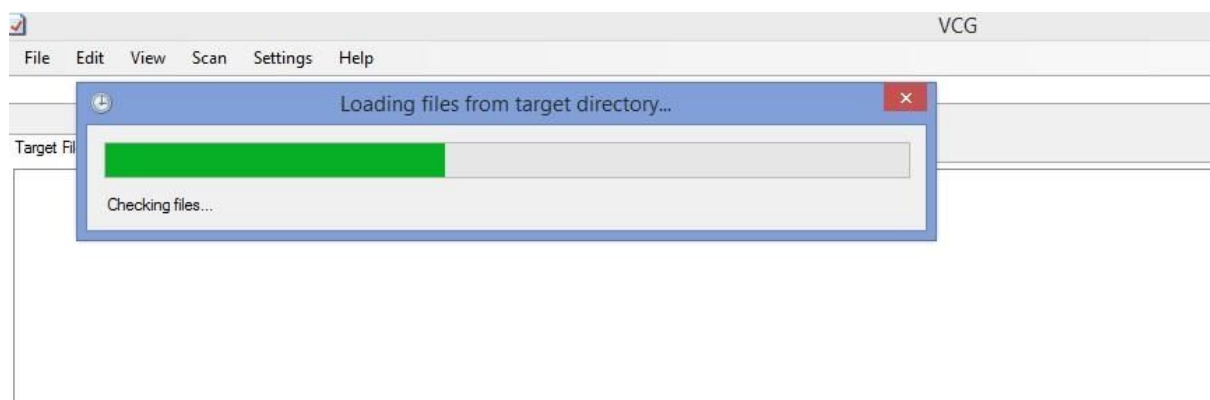


Figure 20: Loading RIOT OS files for Analysis

I selected a 'Full Scan' option and then the software began to analyse the files. It took about 10mins for the scan to be completed and the result was outputted.

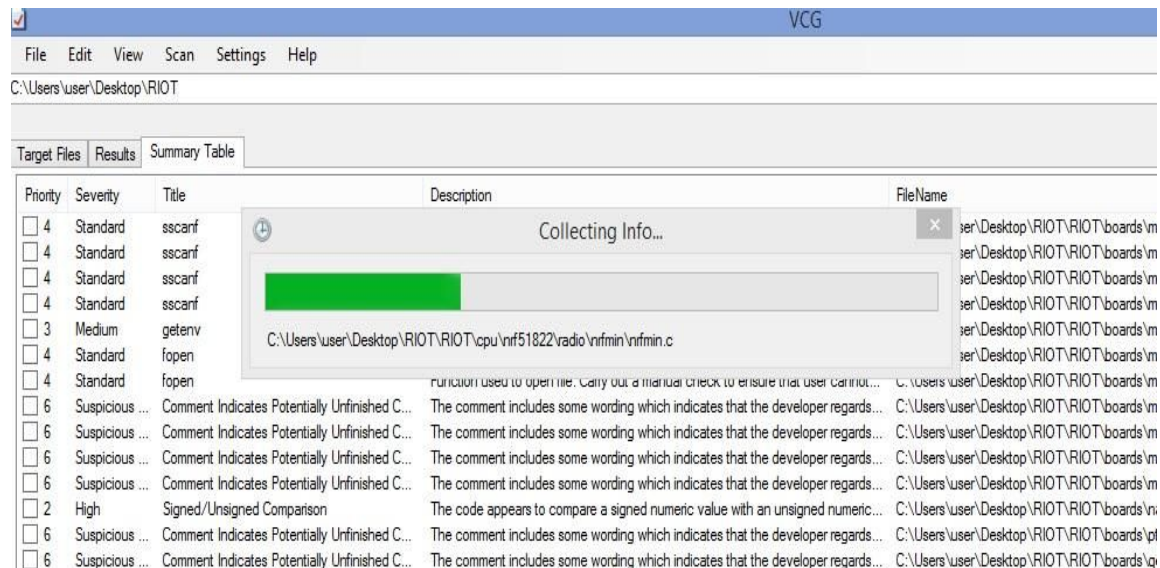


Figure 21: VCG Analysis of RIOT OS

By far the best analysis tool I've used, VCG outputs a detailed breakdown of vulnerabilities including description, filename, line number, vulnerability severity, priority etc. The result breakdown was as follows:

- Number of files analysed – 1595
- Lines of code analysed – 191779
- Number of comments – 177726
- Number of whitespace – 52758
- Potential broken flags – 293
- Potential dangerous code – 1188

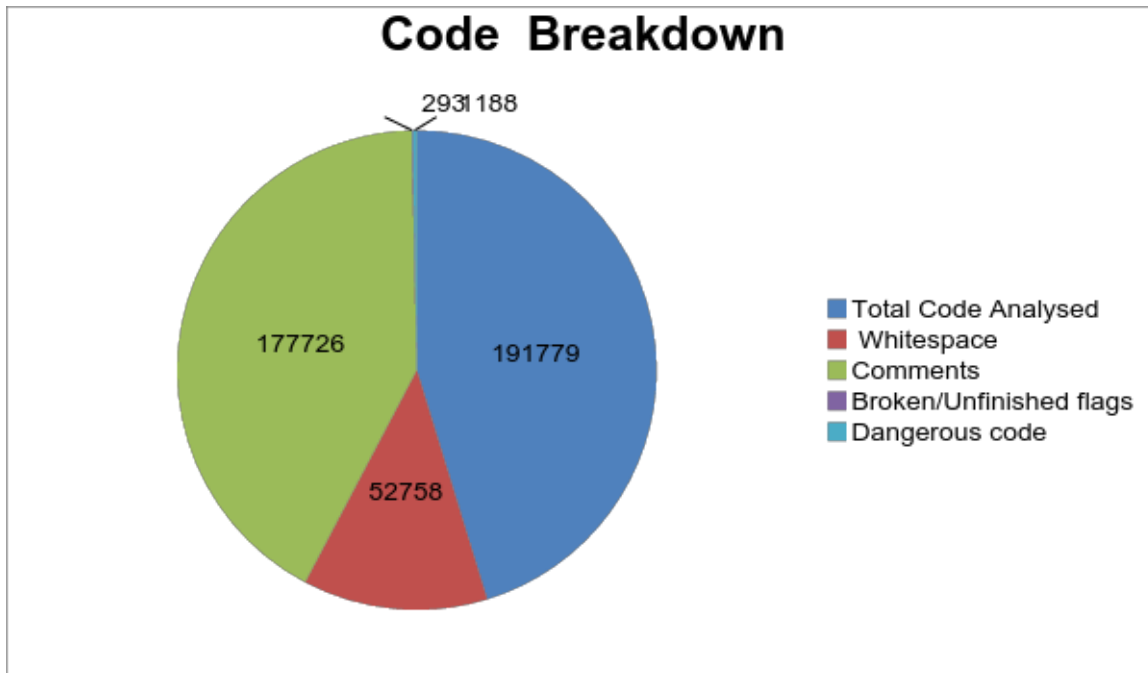


Figure 22: Pie chart of RIOT OS Code Analysed

Files	#	Potential Dangerous Code
..\sys\net\network_layer\sixlowpan\lowpan.c	57	
..\sys\net\ccn_lite\ccnl-ext-mgmt.c	50	
..\sys\od\od.c		
..\sys\net\network_layer\sixlowpan\icmp.c	44	
..\dist\tools\tunslip\tunslip.c	33	
..\sys\net\transport_layer\tcp\tcp_hc.c		
..\tests\unittests\netdev_dummy\netdev_dummy.c	30	
..\sys\cbor\cbor.c	29	
..\dist\tools\tunslip\tunslip6.c	24	

Table 8: Files containing the highest number of potential dangerous code

Priorit y	Severity	Count	Analysis
1	Critical	7	This is mainly due to potential buffer overflow problems from the use of strncat, strncpy, memcpy, strcpy, and strcat functions.
2	High	262	This was mainly generated from code that compares signed numeric values to unsigned numeric values.
3	Medium	596	Vulnerabilities in this category were mainly due to functions in Microsoft's banned function list. Such functions include memcpy, goto, strcat, sprintf etc.
4	Standard	323	This vulnerabilities were due to functions that when used could cause potential memory mis-management as well as some use of strtok, strlen and realloc.
5	Low	0	There were no vulnerabilities found for this category.
6	Suspicious	293	Vulnerabilities in this category were mainly due to comments that indicates potentially unfinished code such as "TODO:", "Assume", "Nothing to do..".

Table 9: Vulnerability Breakdown – VCG

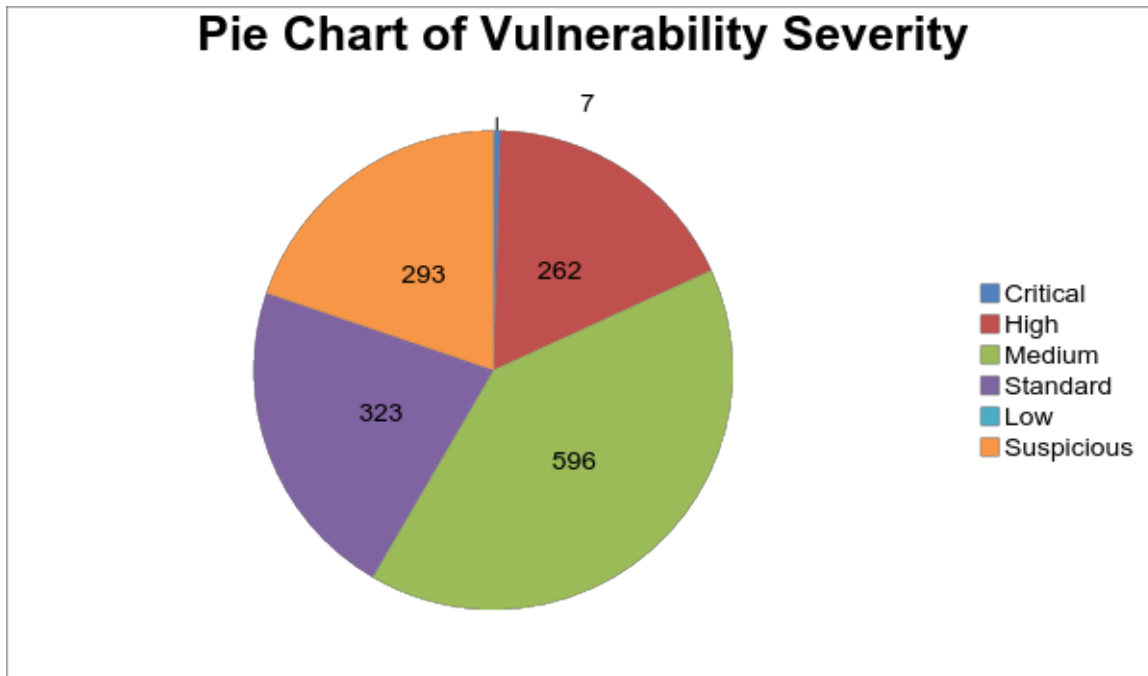


Figure 23: VCG Pie Chart of Vulnerability Severity

5.2.1.5 ClamAV

ClamAV is not a typical source code analysis tool. Rather, it is an open source (GPL) antivirus engine used to scan various services for malware signatures. The tool can detect viruses in files based on known malware signatures and can delete infected files automatically.

ClamAV Analysis of RIOT Operating System

The ClamAV package was installed using 'sudo apt-get install' (See APPENDIX D). A recursive 'clamscan' was then done for all sub-folders in the RIOT directory.

```
kelz@kelz-VirtualBox:~/RIOT$  
kelz@kelz-VirtualBox:~/RIOT$  
kelz@kelz-VirtualBox:~/RIOT$ clamscan -r /home/kelz/RIOT/
```

Figure 24: ClamAV execution on RIOT directory

Result Analysis

```
----- SCAN SUMMARY -----  
Known viruses: 3993693  
Engine version: 0.98.7  
Scanned directories: 703  
Scanned files: 2809  
Infected files: 0  
Data scanned: 67.35 MB  
Data read: 50.77 MB (ratio 1.33:1)  
Time: 106.162 sec (1 m 46 s)  
kelz@kelz-VirtualBox:~/RIOT$ █
```

Figure 25: ClamAV result for RIOT directory

From the image above (Figure 24), there were no malware signatures observed within the RIOT directory.

5.2.2 SOURCE CODE RESULT ANALYSIS

Error Density

According to the analysis tools and results, below are the directories with the highest average error density as well as files that contain the most errors:

- The RIOT OS sub-directories with the highest average error densities were: Dist(11), Examples(1.01), Tests(0.97) and Sys(0.93).
- Files in the sub-directories with the highest number of vulnerabilities include:
 - o ..\dist\tools\tunslip\tunslip6.c
 - o ..\dist\tools\tunslip\tunslip.c

- o `..\examples\ccn-lite-client\main.c`
- o `..\tests\unittests\tests-netdev_dummy\tests-netdev_dummy.c`
- o `..\sys\shell\commands\sc_transceiver.c`
- o `..\sys\net\network_layer\sixlowpan\lowpan.c`
- o `..\sys\net\ccn_lite\ccnl-ext-mgmt.c`
- o `..\sys\od\od.c`
- o `..\sys\net\network_layer\sixlowpan\icmp.c`
- o `..\sys\net\transport_layer\tcp\tcp_hc.c`
- o `..\sys\cbor\cbor.c`

The use of the source code analysis tools is a basic first step to uncovering vulnerabilities in the source code. However, it is important that a manual check is still done to reveal false positives and negatives highlighted by the tools.

During the source code analysis phase of the project, I had to analyse output from the tools which were usually very long (about 200 pages sometimes). In order to perform this I learnt how to use regular expressions (regex) to parse files. This skill will definitely be of benefit to me in the future.

5.3 PENETRATION TESTING

Penetration testing is a method of detecting vulnerabilities in a system by finding hosts or services in the system network that could be accessed illegally. It is performed on all core systems critical to a company such as web servers, firewalls, DNS, FTP etc. (SANS Institute, 2002). Penetration test is usually done by using an automated test tool to scan all the ports within the system to find open ports. Open ports can be used to gain authorised access to resources when it is compromised and used to sniff

passwords or other confidential information. An attacker might also attempt to crash the system by launching buffer overflow or smurf attacks.

The RIOT operating system can be run via an internal native port or installed on a hardware device. In order to perform a thorough penetration test on the operating system, I will attempt to execute penetration testing tools on both the native port and the Mbed NXP LPC1768 board IPv6 addresses.

5.3.1 NATIVE PORT

The RIOT repository includes an examples sub-directory which contains sample projects that could be compiled and run on the RIOT operating system. The 'gnrc_networking' project (Hauke, 2015) is one of these projects and can be used to establish communication between the RIOT native port and the Linux host. For the penetration test on the RIOT native port, I will be using the 'gnrc_networking' project to create a tap interface (to which RIOT will connect) and a bridge (to which Linux will connect). I will then verify the connection by pinging the IP address as well as sending UDP packets between the native port node and the Linux host node. Finally, I would execute the Nmap command on the RIOT node's IPv6 address (RIOT does not support IPv4).

In order to create the tap interface(tap0), I installed openvpn:

```

kelz@kelz-VirtualBox:~/RIOT/examples/gnrc_networking$
kelz@kelz-VirtualBox:~/RIOT/examples/gnrc_networking$
kelz@kelz-VirtualBox:~/RIOT/examples/gnrc_networking$ sudo apt-get install openvpn bridge-utils
Reading package lists... Done
Building dependency tree
Reading state information... Done
bridge-utils is already the newest version.
Suggested packages:
  easy-rsa
The following NEW packages will be installed:
  libpkcs11-helper1 openvpn
0 to upgrade, 2 to newly install, 0 to remove and 3 not to upgrade.
Need to get 432 kB of archives.
After this operation, 1,191 kB of additional disk space will be used.
Get:1 http://gb.archive.ubuntu.com/ubuntu/ trusty/main libpkcs11-helper1 i386 1.11-1 [41.0 kB]
Get:2 http://gb.archive.ubuntu.com/ubuntu/ trusty-updates/main openvpn i386 2.3.2-7ubuntu3.1 [391 kB]
Fetched 432 kB in 0s (1,462 kB/s)
Preconfiguring packages ...
Selecting previously unselected package libpkcs11-helper1:i386.
(Reading database ... 168283 files and directories currently installed.)
Preparing to unpack .../libpkcs11-helper1_1.11-1_i386.deb

```

Figure 26: Installation of openvpn on Virtual Machine

After the interface was created, I built and compiled the ‘gnrc_networking’ project by executing the ‘make’ command.

```

kelz@kelz-VirtualBox:~/RIOT/examples/gnrc_networking$
kelz@kelz-VirtualBox:~/RIOT/examples/gnrc_networking$
kelz@kelz-VirtualBox:~/RIOT/examples/gnrc_networking$ sudo openvpn --mktun --dev tap0
Wed Sep  9 12:39:14 2015 TUN/TAP device tap0 opened
Wed Sep  9 12:39:14 2015 Persist state set to: ON
kelz@kelz-VirtualBox:~/RIOT/examples/gnrc_networking$
kelz@kelz-VirtualBox:~/RIOT/examples/gnrc_networking$
kelz@kelz-VirtualBox:~/RIOT/examples/gnrc_networking$ sudo ip link set dev tap0
No command 'sudp' found, did you mean:
  Command 'sfdp' from package 'graphviz' (main)
  Command 'sup' from package 'sup' (universe)
  Command 'sudo' from package 'sudo' (main)
  Command 'sudo' from package 'sudo-ldap' (universe)
sudp: command not found
kelz@kelz-VirtualBox:~/RIOT/examples/gnrc_networking$ sudo ip link set dev tap0
kelz@kelz-VirtualBox:~/RIOT/examples/gnrc_networking$
kelz@kelz-VirtualBox:~/RIOT/examples/gnrc_networking$ make
Building application "gnrc_networking" for "native" with MCU "native".

"make" -C /home/kelz/RIOT/boards/native
"make" -C /home/kelz/RIOT/boards/native/drivers

```

Figure 27: Compilation of 'gnrc_networking' project

The ‘make term’ command started the project with the ‘tap0’ interface.

```

188761      584 114968 304313 4a4b9 /home/kelz/RIOT/examples/gnrc_networking/bin
kelz@kelz-VirtualBox:~/RIOT/examples/gnrc_networking$
kelz@kelz-VirtualBox:~/RIOT/examples/gnrc_networking$
kelz@kelz-VirtualBox:~/RIOT/examples/gnrc_networking$ make term
/home/kelz/RIOT/examples/gnrc_networking/bin/native/gnrc_networking.elf tap0
RIOT native interrupts/signals initialized.
LED_GREEN_OFF
LED_RED_ON
RIOT native board initialized.
RIOT native hardware initialization complete.

kernel_init(): This is RIOT! (Version: 2014.12-3183-gb4bd-kelz-VirtualBox)
kernel_init(): jumping into first task...
RIOT network stack example application
All up, running the shell now
> █

```

Figure 28: Execution of 'gnrc_networking' project

I then verified that there was connectivity between the RIOT and Linux by executing the 'ifconfig' command and pinging the IP address:

```

kernel_init(): This is RIOT! (Version: 2014.12-3183-gb4bd-kelz-VirtualBox)
kernel_init(): jumping into first task...
RIOT network stack example application
All up, running the shell now
> enableIRQ + _native_in_isr

> enableIRQ + _native_in_isr
ifconfig
ifconfig
Iface 6 HWaddr: c6:96:ca:e1:31:da

MTU:1280
Source address length: 6
Link type: wireless
inet6 addr: ff02::1/128 scope: local [multicast]
inet6 addr: fe80::c496:caff:fee1:31da/64 scope: local
inet6 addr: ff02::1:ffe1:31da/128 scope: local [multicast]
inet6 addr: ff02::2/128 scope: local [multicast]

```

Figure 29: Obtaining IPv6 address from ifconfig command output


```

kelz@kelz-VirtualBox:~/RIOT/examples/gnrc_networking$
kelz@kelz-VirtualBox:~/RIOT/examples/gnrc_networking$ ping6 fe80::c89c:afff:fe60:4a20%tap0
PING fe80::c89c:afff:fe60:4a20%tap0(fe80::c89c:afff:fe60:4a20) 56 data bytes
64 bytes from fe80::c89c:afff:fe60:4a20: icmp_seq=1 ttl=64 time=2.22 ms
64 bytes from fe80::c89c:afff:fe60:4a20: icmp_seq=2 ttl=64 time=1.23 ms
64 bytes from fe80::c89c:afff:fe60:4a20: icmp_seq=3 ttl=64 time=2.49 ms
64 bytes from fe80::c89c:afff:fe60:4a20: icmp_seq=4 ttl=64 time=0.895 ms
64 bytes from fe80::c89c:afff:fe60:4a20: icmp_seq=5 ttl=64 time=2.41 ms
64 bytes from fe80::c89c:afff:fe60:4a20: icmp_seq=6 ttl=64 time=3.41 ms
64 bytes from fe80::c89c:afff:fe60:4a20: icmp_seq=7 ttl=64 time=1.04 ms
64 bytes from fe80::c89c:afff:fe60:4a20: icmp_seq=8 ttl=64 time=2.14 ms
64 bytes from fe80::c89c:afff:fe60:4a20: icmp_seq=9 ttl=64 time=1.16 ms
64 bytes from fe80::c89c:afff:fe60:4a20: icmp_seq=10 ttl=64 time=0.996 ms
64 bytes from fe80::c89c:afff:fe60:4a20: icmp_seq=11 ttl=64 time=2.63 ms
64 bytes from fe80::c89c:afff:fe60:4a20: icmp_seq=12 ttl=64 time=4.56 ms

```

Figure 30: Pinging IPv6 address

The next step was setting up a UDP server on the RIOT node:

```

enableIRQ + _native_in_isr
enableIRQ + _native_in_isr
udp server start 8808
udp server start 8808
Success: started UDP server on port 8808
>

```

Figure 31: UDP server setup

After connecting the Linux host to the UDP server using netcat, I was able to send UDP packets between Linux and RIOT nodes.

```

kelz@kelz-VirtualBox:~/RIOT/examples/gnrc_networking$
kelz@kelz-VirtualBox:~/RIOT/examples/gnrc_networking$
kelz@kelz-VirtualBox:~/RIOT/examples/gnrc_networking$
kelz@kelz-VirtualBox:~/RIOT/examples/gnrc_networking$
kelz@kelz-VirtualBox:~/RIOT/examples/gnrc_networking$ nc -6uv fe80::c89c:afff:fe60:4a20%tap0 8808
Connection to fe80::c89c:afff:fe60:4a20%tap0 8808 port [udp/*] succeeded!

```

Figure 32: Connecting Linux host to RIOT UDP server

```

> udp send fe80::c89c:afff:fe60:4a1f 8808 testmessage
udp send fe80::c89c:afff:fe60:4a1f 8808 testmessage
Success: send 11 byte to fe80::c89c:afff:fe60:4a1f:8808

```

Figure 33: Sending UDP packets from RIOT side

```
kelz@kelz-VirtualBox:~/RIOT/examples/gnrc_networking$ nc -6ul 8808
testmessage
```

Figure 34: UDP packet received on UDP server on Linux side

5.3.1.1 Using Nmap to scan RIOT native port IP address

Network Mapper Security Scanner (Nmap) is a penetration testing tool created by Gordon Lyon in September 1997 for network discovery and auditing. The tool sends packets to the target host and examines the responses to determine characteristics such as services offered by the host, type of operating system used, type of packet filters or firewalls etc (Lyon, 2015).

And finally, after verification of the native port IP address, I executed ‘nmap’ to find open ports and services that could be exploited using the ‘top ports command’.

```
kelz@kelz-VirtualBox:~/RIOT/examples/gnrc_networking$
kelz@kelz-VirtualBox:~/RIOT/examples/gnrc_networking$ sudo nmap -6 --top-ports 10 fe80::c89c:afff:fe60:4a1f%tap0
Starting Nmap 6.40 ( http://nmap.org ) at 2015-09-09 13:44 BST
Nmap scan report for fe80::c89c:afff:fe60:4a1f
Host is up (0.00016s latency).
PORT      STATE SERVICE
21/tcp    closed ftp
22/tcp    closed ssh
23/tcp    closed telnet
25/tcp    closed smtp
80/tcp    closed http
110/tcp   closed pop3
139/tcp   closed netbios-ssn
443/tcp   closed https
445/tcp   closed microsoft-ds
3389/tcp  closed ms-wbt-server
Nmap done: 1 IP address (1 host up) scanned in 0.42 seconds
```

Figure 35: Nmap output - 'top ports' command

I executed Nmap again but with different parameters:

- 6: to specify IPv6 address
- A: for turn on OS and version detection
- T5: to scan all devices for open ports
- Pn: to scan host protected by firewall

```
kelz@kelz-VirtualBox:~/RIOT/examples/gnrc_networking$ nmap -6 -A -T5 -p- -Pn fe80::c89c:afff:fe60:4a1f%tap0
Starting Nmap 6.40 ( http://nmap.org ) at 2015-09-09 13:33 BST
Nmap scan report for fe80::c89c:afff:fe60:4a1f
Host is up (0.0035s latency).
Not shown: 65532 closed ports
PORT      STATE SERVICE VERSION
46072/tcp  open  unknown
48975/tcp  open  unknown
56230/tcp  open  unknown

Host script results:
|_ address-info:
|   IPv6 EUI-64:
|   MAC address:
|       address: ca:9c:af:60:4a:1f
|       manuf: Unknown
|_

Service detection performed. Please report any incorrect results at http://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 19.29 seconds
```

Figure 36: Nmap output

According to the nmap tool, most ports in the RIOT operating system were closed. However, some TCP ports were found to be open (see Figure 36).

5.3.2 MBED NXP LPC1768

The Mbed NXP LPC1768 device is a microcontroller with 512KB FLASH, 32KB RAM and a 32-bit ARM Cortex-M3 core running at 96MHz (NXP, 2009). It contains a LPC1768 processor and provides various interfaces such as reset button, on-board Ethernet and USB (for both client and hosts), 2 SPI, DAC (digital-analogue converter) etc. I chose this device because it is very developer-friendly.

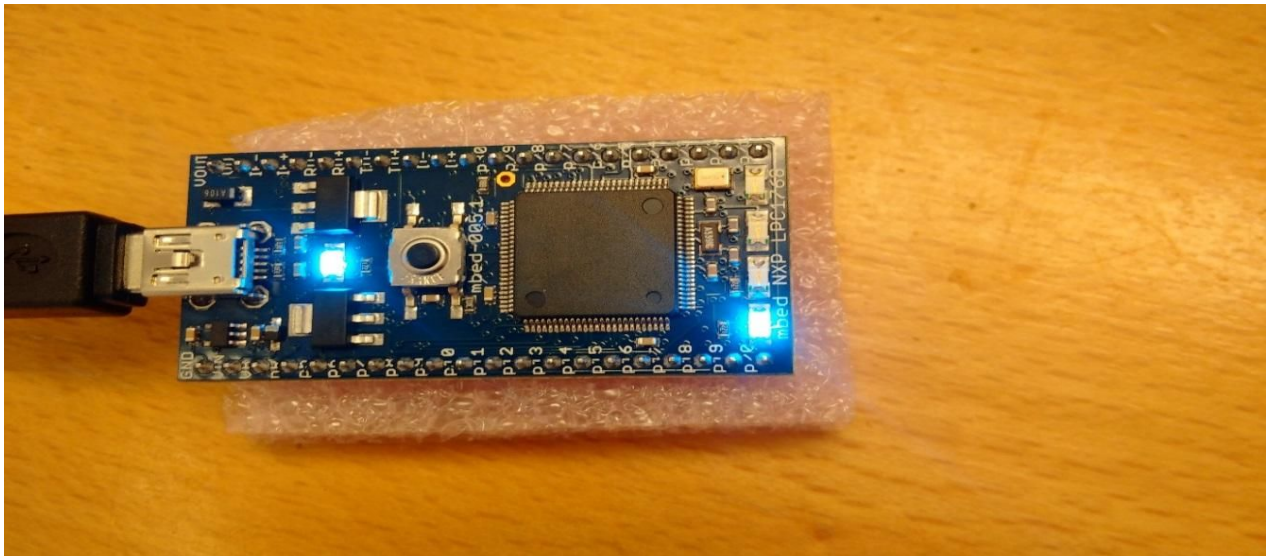


Figure 37: My Mbed NXPLPC1768 Microcontroller

The board is programmable through a USB interface; an interface controller chip is connected to the USB port. There is also an online compiler on the mbed website which can be used to compile applications for the mbed device before uploading it to the device. Finally, It can perform USB to UART conversion. That is, when the USB is connected to the computer, it can be used as either a USB to serial device or USB flash memory. After the application is compiled, the binary image file is copied to the flash memory and read by the interface controller and then programmed into the LPC microprocessor.

This was the method used to compile RIOT applications before uploading them to the mbed device. In order to obtain the IP address of the device, I attempted the following techniques:

Using the online compiler:

The Mbed online compiler is an editor that can be used to compile mbed programs. The generated 'bin' file is then uploaded to the device.

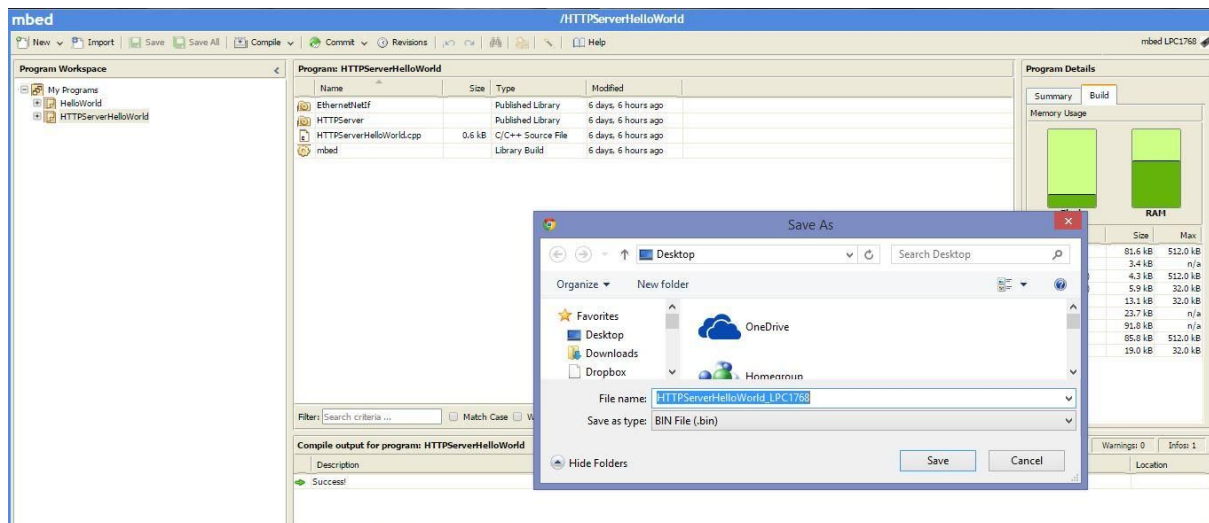


Figure 38: Compiling HTTPServerHelloWorld project for mbed microcontroller

The compiler allows for good workspace management as well as version control. It also includes a standard library from which programs can be imported and exported. I was able to import a project (HTTPServerHelloWorld) and compile it. I also installed a terminal (TeraTerm) to view printed output and for serial communication with the mbed device.

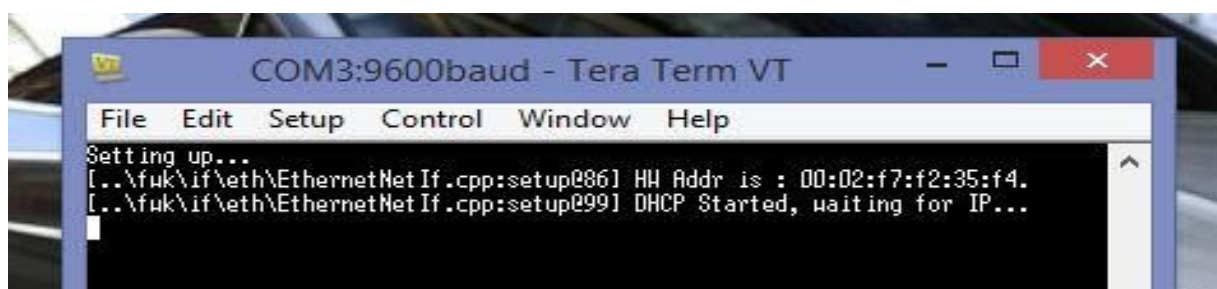


Figure 39: Output from HTTPServerHelloWorld application in TeraTerm terminal

However, I could not compile any RIOT applications via the online compiler because the Makefile is required to compile RIOT applications and this could not be used by the online compiler.

Editing existing RIOT application:

I decided to work on editing one of the applications already included in the RIOT examples directory (Hello-World). I planned to write a simple function that returns network parameters (from which I would obtain the IP address), compile the application, load it on the device and execute my newly created function from a 'tap' virtual interface. I made a backup of the directory and tested the hello-world application first to make sure it worked properly. I executed the following commands:

- make – to build the bin directory

```
knnorom1@knnorom1-VirtualBox:~/RIOT/examples/hello-world$ ls
main.c Makefile Makefile~ README.md
knnorom1@knnorom1-VirtualBox:~/RIOT/examples/hello-world$ make
Building application "hello-world" for "mbed_lpc1768" with MCU "lpc1768".

"make" -C /home/knnorom1/RIOT/boards/mbed_lpc1768
"make" -C /home/knnorom1/RIOT/core
"make" -C /home/knnorom1/RIOT/cpu/lpc1768
"make" -C /home/knnorom1/RIOT/cpu/cortexm_common
"make" -C /home/knnorom1/RIOT/cpu/lpc1768/periph
"make" -C /home/knnorom1/RIOT/drivers
"make" -C /home/knnorom1/RIOT/sys
"make" -C /home/knnorom1/RIOT/sys/auto_init
"make" -C /home/knnorom1/RIOT/sys/newlib
  text   data    bss     dec     hex filename
12360    116    2716   15192   3b58 /home/knnorom1/RIOT/examples/hello-world
/bin/mbed_lpc1768/hello-world.elf
knnorom1@knnorom1-VirtualBox:~/RIOT/examples/hello-world$
```

Figure 40: Compilation of original HelloWorld application

- make flash – to upload the compiled program onto the board

```
knnorom1@knnorom1-VirtualBox:~/RIOT/examples/hello-world$ make flash
Building application "hello-world" for "mbed_lpc1768" with MCU "lpc1768".

"make" -C /home/knnorom1/RIOT/boards/mbed_lpc1768
"make" -C /home/knnorom1/RIOT/core
"make" -C /home/knnorom1/RIOT/cpu/lpc1768
"make" -C /home/knnorom1/RIOT/cpu/cortexm_common
"make" -C /home/knnorom1/RIOT/cpu/lpc1768/periph
"make" -C /home/knnorom1/RIOT/drivers
"make" -C /home/knnorom1/RIOT/sys
"make" -C /home/knnorom1/RIOT/sys/auto_init
"make" -C /home/knnorom1/RIOT/sys/newlib
text      data      bss      dec      hex filename
12360     116     2716    15192    3b58 /home/knnorom1/RIOT/examples/hello-world
/bin/mbed_lpc1768/hello-world.elf
/home/knnorom1/RIOT/boards/mbed_lpc1768/dist/flash.sh

UPLOAD SUCCESSFUL

knnorom1@knnorom1-VirtualBox:~/RIOT/examples/hello-world$
```

Figure 41: Uploading HelloWorld Application to mbed microcontroller

- make term – to start the terminal emulator for the board.

Unfortunately, after including my new functions, I was not able to get the program to compile. This was because my code *included* some standard C header files that are usually pre-packaged in most C libraries. RIOT only contains few of these header files in its library and so when I tried to manually download the files to add to the library I was stuck in a recursive scenario. C header files are usually dependent on one another and so to download one, I had to download any other header files ‘included’ in the file.

Extending gnrc networking application:

Lastly, I decided to try an extension of the method I used to obtain the IP address for the native port. I planned to compile the 'gnrc_networking' RIOT application and upload the .bin file to the mbed device.

First, I edited the Makefile to compile the application for the mbed platform:

```
# name of your application
APPLICATION = gnrc_networking

# If no BOARD is found in the environment, use this default:
BOARD ?= mbed_lpc1768

# This has to be the absolute path to the RIOT base directory:
RIOTBASE ?= $(CURDIR)/../..

BOARD_INSUFFICIENT_MEMORY := airfy-beacon chronos msb-430 msb-430h nrf51dongle \
                             nrf6310 nucleo-f334 pca10000 pca10005 \
                             stm32f0discovery telosb wsn430-v1_3b wsn430-v1_4 \
                             yunjia-nrf51822 z1

BOARD_BLACKLIST := arduino-mega2560 spark-core
# arduino-mega2560: unknown error types (e.g. -EBADMSG)
```

Figure 42: Editing 'gnrc_networking' Application Makefile

I then compiled the application and uploaded it to the mbed lpc1768 board using 'make' and 'make flash' commands. (See Figure 43, 44).

```
kelz@kelz-VirtualBox:~/RIOT/examples/gnrc_networking$
kelz@kelz-VirtualBox:~/RIOT/examples/gnrc_networking$
kelz@kelz-VirtualBox:~/RIOT/examples/gnrc_networking$ make
Building application "gnrc_networking" for "mbed_lpc1768" with MCU "lpc1768".

"make" -C /home/kelz/RIOT/boards/mbed_lpc1768
"make" -C /home/kelz/RIOT/core
"make" -C /home/kelz/RIOT/cpu/lpc1768
"make" -C /home/kelz/RIOT/cpu/cortexm_common
"make" -C /home/kelz/RIOT/cpu/lpc1768/periph
"make" -C /home/kelz/RIOT/drivers
"make" -C /home/kelz/RIOT/sys
```

Figure 43: Compiling 'gnrc_networking' application for mbed board


```

make -C /home/kelz/RIOT/sys/etnetx
"make" -C /home/kelz/RIOT/sys/trickle
"make" -C /home/kelz/RIOT/sys/uart_stdio
"make" -C /home/kelz/RIOT/sys/universal_address
"make" -C /home/kelz/RIOT/sys/vtimer
text data bss dec hex filename
59428 200 21048 80676 13b24 /home/kelz/RIOT/examples/gnrc_networking/bin/mbed_lpc1768/gnrc_networking.elf
/home/kelz/RIOT/boards/mbed_lpc1768/dist/flash.sh
UPLOAD SUCCESSFUL

```

Figure 44: Uploading 'gnrc_networking' application to mbed board

I then executed 'make term' to start the terminal emulator for the board in order to run the application.

```

kelz@kelz-VirtualBox:~/RIOT/examples/gnrc_networking$
kelz@kelz-VirtualBox:~/RIOT/examples/gnrc_networking$ sudo make term
/home/kelz/RIOT/dist/tools/pyterm/pyterm -p "/dev/ttyACM0" -b "115200"
2015-09-11 17:58:31,803 - INFO # Connect to serial port /dev/ttyACM0
Welcome to pyterm!
Type '/exit' to exit.
ifconfig

/exit
2015-09-11 18:02:16,596 - INFO # Exiting Pyterm
kelz@kelz-VirtualBox:~/RIOT/examples/gnrc_networking$

```

Figure 45: 'gnrc_networking' application executed on mbed board via Pyterm

While I was able to execute 'ifconfig' to obtain network parameters when the application was run on the native port, I could not do the same for the mbed board. I found the reason was because the 'gnrc_networking' application had not yet been developed for compatibility with the mbed lpc1768 board. Although it had been tested on boards listed as part of the BOARD_INSUFFICIENT_MEMORY and BOARD_BLACKLIST variables (see Figure 42).

5.3.3 PENETRATION TEST RESULT ANALYSIS

The goal of penetration testing is “to identify ways to exploit vulnerabilities to circumvent or defeat the security features of security components” (Penetration Test Guidance Special Interest Group, 2015). I was able to find vulnerabilities in the system using the RIOT native port. Specifically, I found that while the application was running, some TCP service ports were open. These open ports could be used to gain unauthorised access to programs listening on that port. To reduce the number of potentially vulnerable services exposed to the internet, it is important that all ports are closed.

CHAPTER 6: PROBLEMS ENCOUNTERED

PROBLEM	SOLUTION
Slow virtual machine resulted in slow and frustrating experience	<p>I did some research and came across a site: http://askubuntu.com/questions/289677/how-to-improve-performance-of-virtualbox-when-unity-low-gfx-mode-is-not-working</p> <p>Using the information provided, I was able to modify my VM settings and improved the performance.</p>
Finding analysis tools	<p>The analysis tools used for this project were found after a lot of research. It was important to me that the tools used were highly-rated. I shortlisted tools based on quality. I also examined the tool outputs to verify highlighted vulnerabilities were not false positives.</p>

Software installation	This was mainly during the source code analysis phase of the project. I was able to find resources online on how to solve them.
Virus attack on USB	My USB stick containing all my files was infected with a virus halfway during the project. The USB stick contained all files related to the project. I leveraged Google Drive to backup all my files daily.
Laptop Motherboard	My HP laptop suddenly developed problems about a week before the project was due. It will turn on for few minutes and automatically turn back off. I suspected the motherboard was faulty. The laptop contained my virtual machine and some of my important files (the rest were on my Google Drive). I had to buy a new laptop and re-install RIOT on the VM. This problem really set me back and delayed progress with the project. You might notice different virtual machine profiles in the screenshots. This was due to this problem.

Table 10: Problems encountered during the project

CHAPTER 7: SUMMARY

7.1 CONCLUSION AND RECOMMENDATIONS

The RIOT operating system is relatively new and is still being developed and so, I expected to find vulnerabilities in the system. For instance, a lot of functions used in the source code could cause internal buffer overflow for any device running RIOT operating system. This vulnerability, if exploited, could crash the device if not handled properly. IoT devices are required to be standalone and are sometimes deployed in places with remote access. Crashed devices in the field would incur unwanted and unnecessary expenses to retrieve and repair.

It is important that the developers of the RIOT operating system adapt an iterative and incremental approach towards testing of the system. The system should be continually tested as more and more features are added.

Below are a few recommendations for reducing vulnerabilities in the operating system:

7.1.1 SOURCE CODE SYNTAX

Buffer overflow: A majority of errors highlighted by the tools (about 85%), was due to potential buffer overflow in the code. Although most of these errors were classified as low severity, the code syntax could be improved to reduce vulnerabilities. For instance:

- `strncpy()` could be replaced by `strncpy()`
- `strcat()` could be replaced by `strncat()`
- `sprintf()` could be replaced by `snprintf()`
- `gets()` could be replaced by `fgets()`
- `srand()` - does not provide sufficient entropy for true randomness

The 'n' represents the size of the buffer that is used while the 'f' in `fgets()` represents the format specifiers for the input. These substitute functions ensure the programmer defines the size of the buffer and also specifies the type of input expected so as to prevent buffer overflow. However, they also include non-null termination of overflowed buffers and no error returns on overflow (Thien, 2002).

String format: This vulnerability was mainly highlighted due to the use of the 'printf' command. It also accounted for the highest risk level when classified by the tools. This is because string format errors are closely

related to buffer overflow errors in that functions such as `sprintf()` and `vsprintf()`, assume an infinite length for the buffer. Although a string format attack is not trivial, some steps could be taken to better improve and secure the code. This can include specifying the format expected so as to prevent an attacker from inserting format specifiers in the 'string' variable that could then be used to manipulate the memory stack.

The function "`snprintf(buffer, sizeof(buffer), string)`" could be replaced by "`snprintf(buffer, sizeof(buffer), 'formatExpected', string)`".

Shell commands: – During execution of external programs, the `exec()` function should be used rather than the `system()` function. This is because the `system()` function uses the `PATH` variable which can be exploited by a hacker (Thien, L. 2002).

Race conditions: In order to prevent race conditions, functions that use file descriptors rather than path to file should be used when manipulating files e.g. `fdopen()` instead of `fopen()`. This ensures an attacker cannot use links to manipulate a file while its open. Also, files should be locked when they are being edited to prevent access from another process. This can be done using `fcntl()` or `flock()` functions.

7.1.2 MORE RIOT APPLICATIONS

I was not able to perform penetration test on my Mbed LPC1768 device. This was because the RIOT OS repository did not contain example applications that could be used for networking purposes. An attempt to

write my own application proved fruitless as the C header files needed were not present in the repository and could not all be downloaded because of dependencies between the files.

I feel an operating system that prides itself as developer-friendly should at least contain all the necessary files needed for development. I understand that this might not be possible due to limited internal memory size. However, it can be implemented by using the Mbed Operating System approach which is the use of an online compiler. This would allow a large library of applications and resources to be stored in the cloud. It would also ensure developers can easily develop applications in an environment tailored to RIOT operating system.

7.2 PERSONAL EXPERIENCE

The project was a very good learning experience for me; mainly because I had never done a security analysis project. The techniques and methods required were very new to me, but the experience gained will definitely be useful in the future. I was able to improve my programming skills in C language as well as in the Linux environment. I learnt and improved my debugging and testing skills. I also gained project management skills; including requirements analysis, time management and working with a project plan. Finally, using regular expression as a means to parse documents was always a skill I wanted in my arsenal. Analysing large file

outputs ensured I learnt and practiced using regular expressions in combination with the 'grep' command to search files.

Overall, the project was a success even though I was a bit behind in the timescale according to the project plan. This was mainly due to unforeseen circumstances such as my laptop problems. However, if I did the project over again:

- I would improve my time management – I spent a lot of time writing the early chapters(Introduction, literature review) of the report rather than working on the project. Additionally, I underestimated the amount and quality of work needed to be done for the project. For example, I planned to complete Chapter 5 of the report by the middle of August. However I found I ran into a lot of unforeseen problems, mainly software-related.

7.3 FUTURE WORK

I joined the RIOT development community on the 19th of August, 2015. I plan to share the results of the operating system analysis with the developers of the OS. I will be informing and warning them of vulnerabilities I found during the project and also offering suggestions on solutions to reducing or completely eliminating these vulnerabilities.

REFERENCES

- Baccelli, E., Hahm, O., Günes, M., Wählich, M. and Schmidt, T. (2013). *Operating Systems for the IoT – Goals, Challenges, and Solutions*. [online] ResearchGate. Available at: http://www.researchgate.net/publication/236883971_Operating_Systems_for_the_IoT_Goals_Challenges_and_Solutions [Accessed 13 July. 2015].
- B. Emmanuel, H. Oliver, G. Mesut, W. Mathias and S. Thomas (2015). *RIOT OS: Towards an OS for the Internet of Things* [online] IEEE (INFOCOM 2013). Available at: <http://www.riot-os.org/docs/riot-infocom2013-abstract.pdf> [Accessed 25 June. 2015].
- Bovet, D. and Cesati, M. (2001). *Understanding the Linux kernel*. Beijing: O'Reilly Media.

Brian Chess, Gary McGraw (2004). *Static Analysis for Security* | Cigital. [online] Cigital. Available at: <https://www.cigital.com/blog/static-analysis-for-security/> [Accessed 11 June. 2015].

Canonical (2015). *IoT: RIOT and Snappy connect the smallest of devices*. [online] Ubuntu Insights. Available at: https://insights.ubuntu.com/2015/06/05/iot-riot-and-snappy-connect-the-smallest-of-devices/?utm_source=ubunteu&utm_medium=url_shortner&utm_term=ARI3sL&utm_campaign=shortner [Accessed 1 July. 2015].

CERN Computer Security, (2015). *RATS - CERN Computer Security Information*. [online]. Security.web.cern.ch. Available at: <https://security.web.cern.ch/security/recommendations/en/codetools/rats.shtml> [Accessed 10 June. 2015].

Checkmarx, (2014). *Free Code Analysis Tools – A Step in the Right Direction*. [online] Checkmarx. Available at: <https://www.checkmarx.com/glossary/free-code-analysis-tools-a-step-in-the-right-direction/> [Accessed 14 June. 2015].

Christian Kirsch, (2013). *Introduction to Penetration Testing* | Rapid7 Community. [online] RAPID7COMMUNITY. Available at: <https://community.rapid7.com/docs/DOC-2248> [Accessed 6 Sep. 2015].

Corsaire Securing Trust, (2015). *Penetration Testing* | Corsaire. [online] Penetration-testing.com. Available at: <http://www.penetration-testing.com/> [Accessed 13 Sep. 2015].

David Wheeler, (2015). *Flawfinder Home Page*. [online] Dwheeler.com. Available at: <http://www.dwheeler.com/flawfinder/> [Accessed 14 June. 2015].

DuPaul, N. (2014). *Software Development Lifecycle (SDLC)*. [online] Veracode. Available at: <http://www.veracode.com/security/software-development-lifecycle> [Accessed 13 June. 2015].

Gordon Lyon (2015). *Nmap: the Network Mapper - Free Security Scanner*. [online] Nmap.org. Available at: <https://nmap.org/> [Accessed 16 August. 2015].

Hahm, O. (2014). *RIOT Tutorial - CodeProject*. [online] Codeproject.com. Available at: <http://www.codeproject.com/Articles/840499/RIOT-Tutorial> [Accessed 5 June. 2015].

Hauke Petersen(2015). *RIOT-OS/RIOT*. [online] GitHub. Available at: https://github.com/RIOT-OS/RIOT/tree/master/examples/gnrc_networking [Accessed 7 Sep. 2015].

Hesseldahl, A. (2015). *More Than a Billion Enterprise Devices Are on the Internet of Things* [online]. Re/code. Available at: <http://recode.net/2015/02/23/more-than-a-billion-enterprise-devices-are-on-the-internet-of-things> [Accessed 5 June. 2015].

Lakhoua, M. (2011). *Malware Analysis: Classifying with ClamAV and YARA - InfoSec Resources*. [online] Infosec Institute. Available at: <http://resources.infosecinstitute.com/malware-analysis-clamav-yara> [Accessed 8 Sep. 2015].

Louridas Panagiotis (2006). *Static Code Analysis*. [online] Greek Res. & Technology Network. Available at: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1657940> [Accessed 17 June. 2015].

Martin Johns, (2006). *A Practical Guide to Vulnerability Checkers*. [online] Secologic Project – University of Hamburg. Available at: http://www.secologic.org/downloads/testing/060313_secologic_a_practical_guide_to_vulnerability_checkers.pdf [Accessed 21 July. 2015].

Michael Scovetta, (2015). *Welcome to Scovetta.com!*. [online] Scovetta.com. Available at: <http://www.scovetta.com/yasca.html> [Accessed 28 June. 2015].

Mihai, D., (2015). *RIOT OS - An Open-Source Operating System for the Internet of Things - Smashing Robotics*. [online] Smashing Robotics. Available at: <http://www.smashingrobotics.com/riot-os-open-source-operating-system-in-ternet-things/> [Accessed 5 June. 2015].

Northcutt, S., Shenk, J., Shackleford, D., Rosenberg, T., Siles, R., Mancini, S. (2015). *Penetration Testing: Assessing Your Overall Security Before Attackers Do*. [online]. SANS Institute. Available at: <https://www.sans.org/reading-room/whitepapers/analyst/penetration-testing-assessing-security-attackers-34635> [Accessed 15 June. 2015].

NXP BV, (2009). *Rapid Prototyping for the LPC1768 MCU* [online] NXP. Available at: <http://www.nxp.com/documents/leaflet/LPC1768.pdf> [Accessed 7 Sep. 2015].

Oliver Hahm, (2015). *The friendly operating system for the IoT*. [online] RIOT. Available at: <http://wiki.eclipse.org/images/9/92/EclipseGrenoble2015-RIOT.pdf> [Accessed 13 August. 2015].

Paul Berek (2008). *Creating Clear Project Requirements – Differentiating “What” from “How”*. [online] PMP. Available at: <http://www.pmi.org/~media/PDF/Publications/ADV06NA08.ashx> [Accessed 11 June. 2015].

Pearson, A. (2014). *What is Penetration Testing and Why is It Important?*. [online] Security Innovation Europe. Available at: <http://www.securityinnovationeurope.com/blog/what-is-penetration-testing-and-why-is-it-important> [Accessed 11 Sep. 2015].

Penetration Test Guidance Special Interest Group, (2015). *Information Supplement: Penetration Testing Guidance*. [online] PCI Security Standards Council. Available at: https://www.pcisecuritystandards.org/documents/Penetration_Testing_Guidance_March_2015.pdf [Accessed 9 Sep. 2015].

Radoslav, K., Gennady, A. (2010). *Source Code Analysis – An Overview*. [online] Bulgarian Academy of Sciences. Available at: http://www.cit.iit.bas.bg/cit_2010/v10-2/60-77.pdf [Accessed 13 July. 2015].

Riot-os.org, (2015). *RIOT Documentation*. [online] Riot-os.org. Available at: <http://riot-os.org/api/> [Accessed 25 June. 2015].

SANS Institute (2002). *Penetration Testing – Is it right for you?* [online] SANS. Available at: <http://www.sans.org/reading-room/whitepapers/testing/penetration-testing-you-265> [Accessed 5 August. 2015].

Siarhei Kuryla, (2010). *RPL: IPv6 Routing Protocol for Low power and Lossy Networks* [online] Networks and Distributed Systems Seminar. Available at: <http://cnds.eecs.jacobs-university.de/courses/nds-2010/kuryla-rpl.pdf> [Accessed 1 August. 2015].

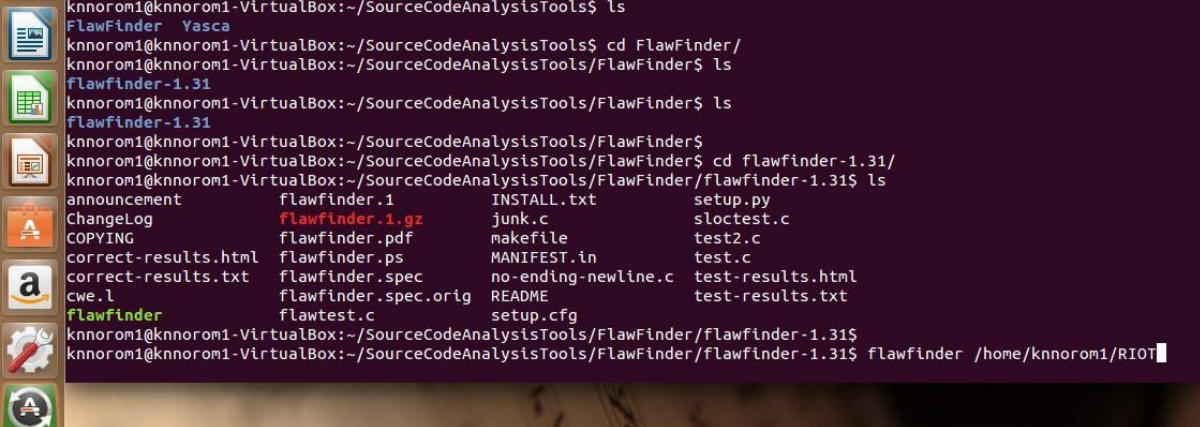
Thien La (2002). *Secure Software Development and Code Analysis Tools* [online] SANS. Available at: <https://www.sans.org/reading-room/whitepapers/securecode/secure-software-development-code-analysis-tools-389> [Accessed 8 July. 2015].

Vlad Trifa (2015). *The Web of Things - How to Give Physical Products a Digital Voice.* [online] EVERYTHING. Available at: <http://link.brightcove.com/services/player/bcpid1361944268001?bckey=AQ~~,AAABPSuWGVE~,e48IldkKVVjluELFeP7s8Y77dTKOFnP&bctid=3851152460001> [Accessed 10 June. 2015].

Askubuntu.com (2015). *How to improve performance of VirtualBox when UNITY_LOW_GFX_MODE is not working?*. [online] Askubuntu.com. Available at: <http://askubuntu.com/questions/289677/how-to-improve-performance-of-virtualbox-when-unity-low-gfx-mode-is-not-working> [Accessed 2 June. 2015].

APPENDICES

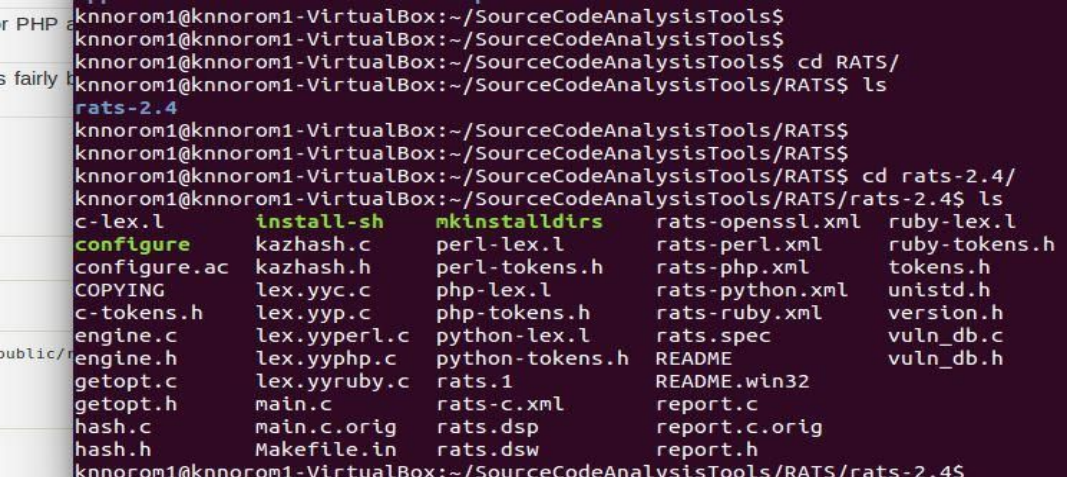
APPENDIX A: Flawfinder Installation



```
knnorom1@knnorom1-VirtualBox:~/SourceCodeAnalysisTools$ ls
FlawFinder Yasca
knnorom1@knnorom1-VirtualBox:~/SourceCodeAnalysisTools$ cd FlawFinder/
knnorom1@knnorom1-VirtualBox:~/SourceCodeAnalysisTools/FlawFinder$ ls
flawfinder-1.31
knnorom1@knnorom1-VirtualBox:~/SourceCodeAnalysisTools/FlawFinder$ ls
FlawFinder-1.31
knnorom1@knnorom1-VirtualBox:~/SourceCodeAnalysisTools/FlawFinder$
knnorom1@knnorom1-VirtualBox:~/SourceCodeAnalysisTools/FlawFinder$ cd flawfinder-1.31/
knnorom1@knnorom1-VirtualBox:~/SourceCodeAnalysisTools/FlawFinder/flawfinder-1.31$ ls
announcement      flawfinder.1      INSTALL.txt       setup.py
ChangeLog          flawfinder.1.gz   junk.c            sloctest.c
COPYING            flawfinder.pdf    makefile          test2.c
correct-results.html flawfinder.ps      MANIFEST.in       test.c
correct-results.txt flawfinder.spec    no-ending-newline.c test-results.html
cwe.l              flawfinder.spec.orig README             test-results.txt
flawfinder         flawtest.c        setup.cfg
knnorom1@knnorom1-VirtualBox:~/SourceCodeAnalysisTools/FlawFinder/flawfinder-1.31$
knnorom1@knnorom1-VirtualBox:~/SourceCodeAnalysisTools/FlawFinder/flawfinder-1.31$ flawfinder /home/knnorom1/RIOT
```

Figure 46: Flawfinder Installation

APPENDIX B: Installation of RATS and Expats Library



```
knnorom1@knnorom1-VirtualBox:~/SourceCodeAnalysisTools$
knnorom1@knnorom1-VirtualBox:~/SourceCodeAnalysisTools$
knnorom1@knnorom1-VirtualBox:~/SourceCodeAnalysisTools$ cd RATS/
knnorom1@knnorom1-VirtualBox:~/SourceCodeAnalysisTools/RATS$ ls
rats-2.4
knnorom1@knnorom1-VirtualBox:~/SourceCodeAnalysisTools/RATS$
knnorom1@knnorom1-VirtualBox:~/SourceCodeAnalysisTools/RATS$
knnorom1@knnorom1-VirtualBox:~/SourceCodeAnalysisTools/RATS$ cd rats-2.4/
knnorom1@knnorom1-VirtualBox:~/SourceCodeAnalysisTools/RATS/rats-2.4$ ls
c-lex.l      install-sh      mkinstalldirs    rats-openssl.xml  ruby-lex.l
configure    kazhash.c       perl-lex.l        rats-perl.xml      ruby-tokens.h
configure.ac  kazhash.h       perl-tokens.h     rats-php.xml       tokens.h
COPYING       lex.yyc.c       php-lex.l         rats-python.xml    unistd.h
c-tokens.h    lex.yyp.c       php-tokens.h     rats-ruby.xml      version.h
engine.c      lex.yyperl.c    python-lex.l      rats.spec          vuln_db.c
engine.h      lex.yyphp.c     python-tokens.h   README             vuln_db.h
getopt.c      lex.yruby.c     rats.1            README.win32
getopt.h      main.c          rats-c.xml        report.c
hash.c        main.c.orig     rats.dsp          report.c.orig
hash.h        Makefile.in     rats.dsw          report.h
knnorom1@knnorom1-VirtualBox:~/SourceCodeAnalysisTools/RATS/rats-2.4$
```

Figure 47: RATS Installation

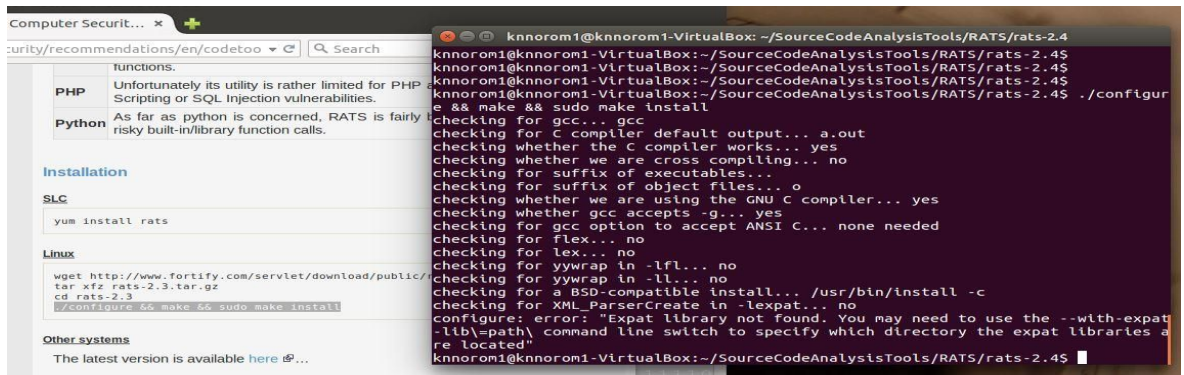


Figure 48: RATS 'Expat' Error

APPENDIX C: Visual Code Grepper Analysis Output

```

        hlist, payload, strlen(payload));

STANDARD: Potentially Unsafe Code - strlen
Line: 731 - C:\Users\user\Desktop\RIOT\RIOT\tests\netdev\main.c
Function appears in Microsoft's banned function list. For critical applications, particularly applications accepting anonymous Internet connections or unverified
similar functions can become victims of integer overflow or 'wraparound' errors.
        hlist, payload, strlen(payload));

MEDIUM: Potentially Unsafe Code - memcpy
Line: 742 - C:\Users\user\Desktop\RIOT\RIOT\tests\netdev\main.c
Function appears in Microsoft's banned function list. Can facilitate buffer overflow conditions and other memory mis-management situations.
        memcpy(s(dest[dev_address_len - sizeof(uint16_t)]), sdest_int,

HIGH: Potentially Unsafe Code - Signed/Unsigned Comparison
Line: 745 - C:\Users\user\Desktop\RIOT\RIOT\tests\netdev\main.c
The code appears to compare a signed numeric value with an unsigned numeric value. This behaviour can return unexpected results as negative numbers will be forced
positive numbers.
        for (size_t i = 0; i < dev_address_len; i++) {

STANDARD: Potentially Unsafe Code - strlen
Line: 752 - C:\Users\user\Desktop\RIOT\RIOT\tests\netdev\main.c
Function appears in Microsoft's banned function list. For critical applications, particularly applications accepting anonymous Internet connections or unverified
similar functions can become victims of integer overflow or 'wraparound' errors.
        hlist, payload, strlen(payload));

```

Figure 49: Part of VCG Analysis Output for RIOT OS

APPENDIX D: ClamAV Installation

```
kelz@kelz-VirtualBox:~$ sudo apt-get install clamav clamav-freshclam
[sudo] password for kelz:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
  clamav-base libclamav6
Suggested packages:
  clamav-docs libclamunrar6
The following NEW packages will be installed:
```

Figure 50: ClamAV Installation