UNIVERSITY OF TURKU

Finding and Exploiting Vulnerabilities in Embedded TCP/IP Stacks

Cyber Security

Master's Degree Programme in Information and Communication Technology

Department of Computing, Faculty of Technology

Master of Science in Technology Thesis

Author:

Wenhui Wang

Supervisors:

Ethiopia Nigussie (University of Turku)

Stanislav Dashevskyi (Forescout Technologies, Inc.)

Antti Hakkala (University of Turku)

June 2021

The originality of this thesis has been checked in accordance with the University of Turku quality assurance system using the Turnitin Originality Check service.

Master of Science in Technology Thesis
Department of Computing, Faculty of Technology
University of Turku

Subject: Cyber Security
Programme: Master’s Degree Programme in Information and Communication Technology
Author: Wenhui Wang
Title: Finding and Exploiting Vulnerabilities in Embedded TCP/IP Stacks
Number of pages: 83 pages
Date: June 2021

In the context of the rapid development of IoT technology, cyber-attacks are becoming more frequent, and the damage caused by cyber-attacks is remaining obstinately high. How to take the initiative in the rivalry with attackers is a major problem in today's era of the Internet. Vulnerability research is of great importance in this contest, especially the study of vulnerability detection and exploitation methodologies.

The objective of the thesis is to examine vulnerabilities in DNS client implementations of embedded TCP/IP stacks, specifically in terms of vulnerability detection and vulnerability exploitation research. In the thesis, a detection method is developed for some anti-patterns in DNS client implementations using a static analysis platform. We tested it against 10 embedded TCP/IP stacks, the result shows that the developed detection method has high precision for detecting the vulnerabilities found by the Forescout research labs with a total of 88% accuracy. For different anti-patterns, the method has different detection precision and it is closely related to the implementation of the detection queries. The thesis also conducted vulnerability exploitation research for a heap overflow vulnerability that exists in a DNS client implementation of a popular TCP/IP stack. A proof-of-concept of this exploitation is developed. Though there are many constraints for successful exploitations, the ability to conduct remote code execution attacks still makes exploitation of heap overflow vulnerability dangerous. In addition, attacks against TCP/IP stacks can take advantage of the stacks and make it possible for an attacker to exploit vulnerabilities in other devices.

Often it takes a huge amount of time for researchers to have deep knowledge of a codebase and to find vulnerabilities in it. But with the developed detection method, we can automate the process of locating vulnerable code patterns to add support for detecting relevant vulnerabilities. Research on the exploitation of vulnerabilities can allow us to discover the potential impact of a vulnerability from the perspective of an attacker and implement targeted defences.

Keywords: static analysis, vulnerability detection, vulnerability exploitation, embedded TCP/IP stacks.

Table of contents

1 Introduction 9

1.1 Motivation 9

1.1.1 Disturbing trends in cyber-attacks 9

1.1.2 New threats introduced by IoT technology 11

1.1.3 The importance of vulnerability research 13

1.2 Research objective 15

1.3 Research questions 15

1.4 Thesis organization 16

2 Literature Survey 17

2.1 Embedded TCP/IP Stacks 17

2.1.1 The implementation of TCP/IP stacks 18

2.1.2 Potential threats in the embedded TCP/IP stacks 20

2.2 Vulnerability detection 21

2.2.1 Vulnerability detection methods 21

2.2.2 Static analysis 23

2.2.3 Code representations 25

2.3 Vulnerability exploitation 27

2.3.1 Stack overflow 29

2.3.2 Heap overflow 31

3 Design of vulnerability detection method and exploitation case study 33

3.1 Vulnerability detection using static analysis tool Joern 33

3.1.1 DNS name parsing in embedded TCP/IP stacks 33

3.1.2 Anti-patterns related to DNS name parsing 39

3.1.3 A static analysis tool: Joern 40

3.1.4 Selected embedded TCP/IP stacks 43

3.2 Exploitation of a heap overflow vulnerability 44

3.2.1 The designed scenario of the case study 45

3.2.2 The heap overflow vulnerability 47

3.2.3 The memory allocator 49

4 Implementation 52

4.1 Vulnerability detection using static analysis tool Joern 52

4.1.1 Query to locate the conditions of compression pointer checks 53

4.1.2 Query to filter out incorrect compression pointer checks 55

4.1.3 Query to locate compression pointer offset calculation 56
4.1.4 Query to analysis the data flow 58
4.2 Exploitation of a heap overflow vulnerability 60
4.2.1 The heap overflow 61
4.2.2 The shellcode 66
5 Results analysis 70
5.1 Vulnerability detection using static analysis tool Joern 70
5.1.1 Output analysis 70
5.2 Exploitation of a heap overflow vulnerability 73
5.2.1 Attack conditions 73
5.2.2 Potential impacts 74
6 Conclusion 75
References 77

List of Figures

FIGURE 1. TCP/IP MODEL 17
FIGURE 2. STACK VARIABLES USED IN FUNCTION CALLS 30
FIGURE 3. OVERFLOW A STACK VARIABLE 31
FIGURE 4. RECURSIVE DNS LOOKUP PROCESS 34
FIGURE 5. ITERATIVE DNS LOOKUP PROCESS 34
FIGURE 6. THE FORMAT OF A DNS MESSAGE 35
FIGURE 7. THE FORMAT OF A DNS MESSAGE HEADER 36
FIGURE 8. THE FORMAT OF A QUESTION SECTION 36
FIGURE 9. THE FORMAT OF A RESOURCE RECORD 37
FIGURE 10. DOMAIN NAME “GOOGLE.COM” 37
FIGURE 11. COMPRESSION POINTER 38
FIGURE 12. HOW JOERN WORKS 41
FIGURE 13. COMMAND TO GENERATE CPG FROM SOURCE CODE 42
FIGURE 14. COMMANDS TO RUN QUERY SCRIPT IN THE INTERACTIVE SHELL 42
FIGURE 15. COMMAND TO RUN QUERY SCRIPT OUTSIDE THE INTERACTIVE SHELL 43
FIGURE 16. THE DESIGNED ENVIRONMENT 46
FIGURE 17. THE ATTACK SCENARIO 47
FIGURE 18. CODE SNIPPET OF THE FUNCTION DNS_CALL 48
FIGURE 19. THE DNS_STRUCTURE WITH THE VULNERABLE VALUE 48
FIGURE 20. CODE SNIPPET OF THE FUNCTION VULNERABLE_CALL 49
FIGURE 21. THE STRUCTURE OF A FREE CHUNK 50
FIGURE 22. THE STRUCTURE OF AN ALLOCATED CHUNK 50
FIGURE 23. QUERY ALGORITHM FOR LOCATING THE COMPRESSION POINTER CHECKS 54
FIGURE 24. CODE SNIPPET OF CORRECT AND INCORRECT IMPLEMENTATIONS 55
FIGURE 25. QUERY ALGORITHM FOR LOCATING INCORRECT COMPRESSION POINTER CHECKS 56
FIGURE 26. AN EXAMPLE OF THE SCRIPT OUTPUT 56
FIGURE 27. CODE EXAMPLE OF COMPRESSION OFFSET CALCULATION 56
FIGURE 28. THE PROCESS OF COMPRESSION OFFSET CALCULTATION 57
FIGURE 29. QUERY ALGORITHM FOR LOCATING OFFSET CALCULATION CODE PATTERNS 57
FIGURE 30. AN EXAMPLE OF THE SCRIPT OUTPUT 58
FIGURE 31. AN EXAMPLE OF THE SCRIPT OUTPUT 59
FIGURE 32. AN EXAMPLE OF THE SCRIPT OUTPUT 59
FIGURE 33. QUERY ALGORITHM FOR DATA FLOW ANALYSIS 60
FIGURE 34. CODE SNIPPET OF THE FUNCTION UNLINK 61
FIGURE 35. THE OVERFLOW PROCESS 64
FIGURE 36. CODE SNIPPET OF THE FUNCTION NSLOOKUP 64

FIGURE 37. CODE SNIPPET OF THE FUNCTION ALERT_PRINTF 64
FIGURE 38. SIMPLIFIED UNLINK OPERATION 65
FIGURE 39. THE SHELLCODE STRUCTURE 67
FIGURE 40. CAPTURED ATTACK PACKETS 69

List of tables

TABLE 1. RESULTS OF THE QUERY SCRIPT 70

TABLE 2. MANUALLY DETECTED VULNERABILITIES AND THE DETECTION RESULTS FOR THEM 71

TABLE 3. DETECTION METRICS OF THE QUERY SCRIPT 72

1 Introduction

1.1 Motivation

The development of Internet technology has promoted the rapid progress of human society. With the development of mobile devices and Internet of Things technology, our lives are increasingly inseparable from the Internet and these electronic components connected to it. For example, mobile phones are an indispensable part of our daily lives now. According to data, as of 2020, close to 3.6 billion people in the world have smartphones. [1] This number is expected to reach 4.3 billion in ten years. From wearable devices to smart home devices that facilitate our lives, to devices used in industrial control systems, various Internet of Things devices are also appearing more and more frequently. In 2020, the number of IoT devices connected to the Internet has exceeded that of traditional personal computers and smartphones. At the end of 2020, more than half of the 21.7 billion active connections were IoT device connections. [2] The Internet of Things (IoT) market size was US$250.72 billion in 2019, and it is expected that its market size will increase by nearly 6 times by 2027. [3] And behind these exciting numbers, hidden is the growing threats to our digital lives.

These threats are the increasingly rampant cyber-attacks. Cyber-attacks are any attack forms against information systems, networks, personal computers, etc. Generally, cyber-attacks will affect the confidentiality, integrity, and availability of the attacked target through unauthorized access, modification, and destruction.

1.1.1 Disturbing trends in cyber-attacks

Cyber-attacks have shown some increasingly disturbing trends in recent years. Recently the number of cyber-attacks has shown rapid growth. According to statistics, the number of malware infection incidents has increased from 12.4 million in 2009 to 812.67 million in 2018. Although this trend has slowed down, the number of malware infection incidents is still increasing. [4] According to a survey, the number of malicious software sites declined between 2009 and 2019, but the number of phishing sites showed a rapid increase. [5] The survey also pointed out that in the past ten years, the cost of losses from consumer complaints that were reported to the FBI has also shown a disturbing growth, from approximately US$560 million in 2009 to approximately US$3.5 billion in 2019. The increase in the number of these cyber-attacks has become more prominent with the virus epidemic (COVID-19). During the covid-19 virus epidemic, the increased demand for remote work has provided attackers with a larger

attack surface, and more important business activities are also carried out in unprotected or inadequately protected environments. These all bring us more threats to our online world. A Kaspersky survey on DDOS attacks showed that in the third quarter of 2020, the overall number of DDOS attacks increased by 1.5 times compared with the same period in 2019. [6] In 2020, the number of phishing attacks reached the highest value in three years. Compared with 2019, the number of complaints about phishing attacks has doubled. [7] Overall, since the covid-19 virus epidemic started, cybercrime has increased 6 times and still growing. [4] The above data shows that despite the increasing security awareness of enterprises and individuals and more and more sophisticated protective measures against cyber-attacks, the number of cyber-attacks is still showing an increasing trend.

The economic losses caused by cyber-attacks are also showing an increasing trend. Cybersecurity ventures predict that the losses caused by cyber-attacks will grow at a growth rate of 15% within 5 years and will reach $10.5 trillion USD in 2025. [8] Among these losses, the losses caused by ransomware are worth noting. Although many security experts do not recommend companies to pay the ransom, those companies attacked by ransomware tend to agree with the attacks' deal, because compared with the losses caused by business stagnation, ransom asked by the attackers is relatively more acceptable to companies. Economic interests undoubtedly encourage this type of crime even more. It is predicted that by 2021, the loss caused by global ransomware will reach 20 billion U.S. dollars, which is 57 times that of 2015. [8]

In addition to the increasing trend of the number of cyber-attacks and the losses caused by them, other changes related to cyber-attacks in recent years are also worthy of our attention. The attacker's intentions are changing. The first computer worm was developed in 1971. [9] The worm was not malicious. It was more like a joke. After it, the first DOS attack appeared in 1988. Although the initial purpose of this attack was not to destroy, it still greatly reduced the speed of Internet access and caused losses. After these early attacks, the purpose of cyber-attacks gradually changed from innocent jokes to causing damage, and then to gaining economic benefits or making political influence. In the meantime, the techniques of cyber-attacks have also been greatly improved. The attackers have begun to use increasingly sophisticated attack techniques. For example, they have begun to control a large number of devices to carry out DDOS attacks. They also have begun to use encryption techniques to make attacks harder to be detected.

1.1.2 New threats introduced by IoT technology

Apart from these trends in cyber-attacks, the development of new technologies has also made things more complicated, especially the development of the Internet of Things (IoT). As discussed above, IoT technology has developed rapidly in recent years. From the consumer field to the industrial field, the Internet of Things technology is being applied more and more frequently.

The first thing worth noting is the development of smart home technology in recent years. Smart home technology allows the basic home amenities to have the communication ability with each other and with other types of devices. These home amenities can be TVs, refrigerators, washing machines, sweeping robots and lighting systems, etc. There are many benefits for these devices to have communication technology. Firstly, it can greatly improve the efficiency of our housework. For example, we can remotely operate the washing machine during commuting. Secondly, it is environment-friendly, for example, intelligent devices can optimize the use of resources such as water and electricity. Finally, smart devices can integrate data well and provide us with transparency in the use of the devices.

The Internet of Things technology is also being widely used in the field of healthcare. The benefits of using the Internet of Things technology are obvious. [10] The Internet of Things technology can simultaneously monitor and report the health data of patients, save lives in emergency situations. We can often see news that patients were saved in time with the help of smartwatches. [11] IoT technology can also automate patient care procedures. Patients' data can be automatically and intelligently processed. This also helps to contribute to research in the healthcare field. Finally, the use of IoT devices provides the possibility of telemedicine. It can provide patients with better medical resources, thereby increasing the cure rate of certain intractable diseases.

The application of The Internet of Things technology is also considered as a way to improve production efficiency in the industrial fields, and for sure the Internet of Things technology will gradually penetrate into more fields in the future. In addition to the wider application of Internet of Things technology. Another trend worth noting is that more and more technologies, such as artificial intelligence, cloud computing, data science, and blockchain are used in IoT devices [12], which provides more possibilities for IoT technology.

While the Internet of Things technology is becoming more mature, this technology also brings us more threats. In a study [13], the author summarized the reasons why IoT devices are more vulnerable to cyber-attacks than traditional personal computers and mobile terminals.

Firstly, IoT devices are more likely to become part of a botnet and help attackers to carry out DDOS attacks. IoT devices often connect directly to the Internet without the protection of a firewall. Even there are firewalls in place, the firewall configurations often can be easily bypassed. C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas analyzed Mirai and Other Botnets and summarized the reasons why IoT devices are easily infected and become part of a botnet. [14] In addition to the inadequate protection of IoT devices, they also pointed out that different from traditional personal computers, IoT devices often operate almost twenty-four hours per day. Also, due to the lack of an interactive interface with users, attacks on IoT devices are difficult to detect by the users and the exploitation can be constant and unobtrusive. Also, IoT devices can generate enough DDOS traffic like traditional devices. Taken together, various advantages make IoT devices favoured by hackers who want to carry out DOS attacks.

Secondly, many IoT devices are closely related to the physical world, such as smart home devices, healthcare devices, and devices used in the industrial field. The harm caused by attacks against these devices may be catastrophic, and it is likely to threaten people's lives and health. A famous example could be the Stuxnet incident. [15] Stuxnet is a worm virus that spreads by exploiting many zero-day vulnerabilities. Through complex programming, its ultimate goal is to make the centrifuges that produce enriched uranium work abnormally by controlling specific models of programmable logic controllers (PLCs). According to later assessments, the attack destroyed approximately 980 centrifuges at Iran’s nuclear facilities and delayed Iran’s overall nuclear program. [16] It can be seen that the close connection between the Internet of Things and the physical world can expand the impact of cyber-attacks and provide conditions for the weaponization of malicious code.

The last reason why IoT devices are more fragile than traditional devices is that often IoT devices cannot be updated in time. This may be due to the negligence of the system managers, or the update is simply impossible or costly. The above all shows that the development of Internet of Things technology has brought huge challenges to cyber world security. Coupled with the increasing frequency of cyber-attacks, how to restrain the intensified cybercrime has become more and more urgent.

1.1.3 The importance of vulnerability research

What plays a key role in cybercrime are the vulnerabilities that exist in the programs. Bishop and Bailey proposed the definition of vulnerability [17]: "A vulnerable state is an authorized state from which an unauthorized state can be reached using authorized state transitions; a vulnerability is a characterization of a vulnerable state which distinguishes it from all non-vulnerable states." Software developers may write programs with bugs for various reasons. A bug may occur due to the fact that the programming does not follow the programming specifications, the software safety is not paid attention to during the programming process, or the development cycle is too short and there is no time for a full testing for the software. Programs with bugs often can run in a way that was undefined by the programmer during interaction with the users, and the undefined execution paths can threaten the security of the software under certain circumstances, thereby forming vulnerabilities.

The first vulnerability in history was discovered by McPhee in 1974. [18] Software vulnerabilities will be submitted to the vulnerability database after being discovered by researchers. Though some vendors of widely used software may use their own recording system to document detailed information about the vulnerabilities in their products, normally, publicly known vulnerabilities will be recorded in the CVE database. [19] Almost every day, new vulnerabilities are recorded in the CVE database. As of this writing, the total number of vulnerabilities recorded in CVE has exceeded 150,000.

Vulnerabilities can be divided into different categories according to different methods. For example, we can classify vulnerabilities from the perspectives of vulnerability cause, vulnerability impact, vulnerability severity, etc., but in fact, it is very difficult to classify vulnerabilities comprehensively and rigorously. The commonly used classification in the security research field is CWE (Common Weakness Enumeration) [20], CWE list describes common weaknesses in common language and provides CWE ID numbers for these weaknesses.

From the perspective of disclose time, we can divide vulnerabilities into zero-day vulnerabilities and N-day vulnerabilities. Zero-day vulnerabilities refer to those vulnerabilities that have not been disclosed but may have been discovered by some hackers. Correspondingly, 1-day and N-day vulnerabilities refer to the number of days between the vulnerability being disclosed and being exploited by hackers. The harm of zero-day vulnerabilities is usually very serious because zero-day vulnerabilities have not been publicly disclosed. Software with zero-day vulnerabilities may not have corresponding patches, and many anti-virus programs that

detect malicious code through signatures cannot detect attacks that are using these zero-day vulnerabilities. If a hacker discovered a zero-day vulnerability of a widely used program, the damage the hacker can cause is incalculable, because the hacker can attack almost any computer that running the software and the attack is difficult to detect. The Stuxnet worm mentioned earlier used multiple zero-day vulnerabilities. [15]

In the process of software development, although we can effectively reduce the vulnerabilities in software by optimizing the development process, the bugs in the software are unavoidable. If we find the vulnerabilities later than the attackers, we will lose the initiative. In the fight against cybercrime, robust vulnerability detection methods can effectively reduce the number of attacks that exploit zero-day vulnerabilities.

There is no doubt that the zero-day vulnerability can bring us great losses, but the harm of disclosed N-day vulnerabilities, especially 1-day vulnerabilities, cannot be ignored. After a vulnerability is publicly disclosed, it is very likely that the vulnerable program has not yet provided with an effective patch. Although many researchers and organizations will ensure that the vulnerability disclosure process will only disclose the relevant details of the vulnerability after the program has a valid patch, it still happens that there are no effective patches after a vulnerability is disclosed. In addition, the most common situation is that even if the program vendor provides effective vulnerability patches, users are unlikely to upgrade the program in time because they do not understand the hazards of the vulnerability. The WannaCry ransomware incident in 2017 raised our vigilance on the harm of the N-day vulnerability [21]. The vulnerability used by the WannaCry ransomware is a vulnerability that exists in the Microsoft Windows operating system called EternalBlue, which was allegedly developed by the United States National Security Agency. After the vulnerability was disclosed by a group of hackers called the Shadow Brokers, Microsoft had already issued a valid vulnerability patch two months before the WannaCry incident. However, due to a large number of users didn't update the operating system in time, it is reported that the attack hit about 230,000 computers around the world.

A skilled attacker has either the ability to discover vulnerabilities, or the ability to exploit public-known vulnerabilities, or both. If an attacker has the ability to discover vulnerabilities, he or she can initiate powerful attacks because the vulnerability he or she discovered might haven't been publicly disclosed by security researchers, and the vulnerability might be a zero-day vulnerability. If an attacker has the ability to exploit N-day vulnerabilities, the attacks

initiated by the attacker could also be dangerous to our cyber world. Because the impacts of a vulnerability might be underestimated, and the attacker might have the ability to fully utilize the vulnerability. It is important to conduct vulnerability research especially research about vulnerability detection and exploitation.

1.2 Research objective

The objective of the thesis is *to develop a robust detection method for vulnerable code patterns that exist in DNS client implementations of embedded TCP/IP stacks and to examine the impacts of exploitation of a heap overflow vulnerability*. Usually, a large amount of time and energy of skilled security experts is required to find vulnerabilities in a codebase. By automating the process of finding vulnerable code patterns, we could detect related vulnerabilities more productively. This detection method can help experts save unnecessary time for getting familiar with the codebase and locating problematic code patterns. They can be more focused on analysing code patterns and finding vulnerabilities in them. As for examining vulnerability exploitation, it could help us understand what a severe vulnerability can achieve in the wrong hands and the possible constraints for attackers. From a defence point of view, this can help us design better defence mechanisms against the detected vulnerabilities.

1.3 Research questions

The thesis will be divided into two different parts. The first part of the thesis is related to vulnerability detection research. Researchers at Forescout have found many severe vulnerabilities in embedded TCP/IP stack implementations, from those vulnerabilities they summarized several anti-patterns that often cause similar vulnerabilities. These anti-patterns occur repeatedly, and we currently lack a good method to automatically detect them. In the thesis, we used a static analysis tool to provide an automatic detection method against some of the anti-patterns to add supports to vulnerability detection. The research questions for this part are:

- What is the precision of the detection method against some manually discovered anti-patterns?
- What are the advantages and disadvantages of this detection method?

In the second part of the thesis, we conducted an exploitation case study of a heap overflow vulnerability. An incorrect assessment of the potential impact of the vulnerability is likely to

cause more damage to the attack on the vulnerability. Even if certain vulnerabilities are disclosed, it does not mean that attacks against these vulnerabilities can be effectively prevented. We need to correctly understand the potential harm of the vulnerabilities. In this case study, we took a DNS client vulnerability in an embedded TCP/IP stack as an example and designed an attack scenario to study the harm that a vulnerability can produce in the embedded TCP/IP stack. The research questions for this part are:

- What harm can a successful heap overflow cause under the designed scenario?
- How easy is it to attack this vulnerability? And what are the constraints of a successful attack?

1.4 Thesis organization

The two parts of the thesis (vulnerability detection research and vulnerability exploitation case study) are explained in parallel in each chapter, and the organization of the chapters of the thesis are explained below:

Chapter 2 introduces the relevant background knowledge involved in this article in detail, mainly introduces the background of the three aspects of TCP/IP stack, vulnerability detection, and vulnerability exploitation, and also investigates related research and work.

Chapter 3, Chapter 4, and Chapter 5 are all divided into two parts: vulnerability detection and vulnerability exploitation. These three chapters introduce the design, implementation, and results analysis of vulnerability detection methods and vulnerability exploitation development respectively.

Chapter 6 concludes the research of the thesis.

2 Literature Survey

2.1 Embedded TCP/IP Stacks

The TCP/IP model is the cornerstone on top of which the modern internet exists. It follows a four-layered fashion as shown in Figure 1. Every layer encompasses at least one protocol decapsulates the data (protocol data unit PDU) coming from the downside layer and passes it to the upward layer and in the process, it does various operations to make sense of headers that follow the convention of said protocol.

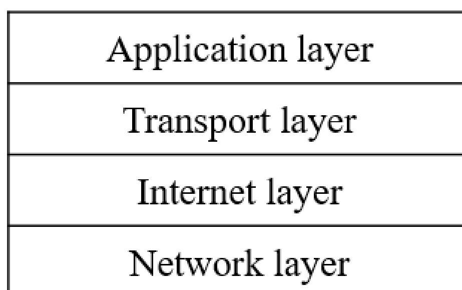

Figure 1. TCP/IP model

The functions of the TCP/IP layers are:

- The network layer: This layer corresponds to the physical layer and data link layer in the OSI (Open Systems Interconnection) model [22]. This layer is responsible for data transmission inside a network, it defines how should two devices in a network physically exchange data.
- The internet layer: It transmits data from the source node to the destination node through several intermediate nodes. The IP protocol plays an important role in this layer.
- The transport layer: It ensures that the data is delivered to the correct application. Two widely used protocols in this layer are TCP (Transmission Control Protocol) and UDP (User Datagram Protocol). TCP is responsible for providing reliable, connection-oriented service between two processes, while UDP is responsible for providing unreliable, connectionless service between two network endpoints.
- The application layer: It provides protocols for communication between software applications. There are a lot of widely used protocols running on this layer, for example, HTTP, FTP, SMTP, etc.

Protocols from the network layer up to the transport layer are often implemented in software that is called a TCP/IP stack which enables devices to be networked. The abundance in number and functionality of embedded devices makes it pertinent to be able to transmit/receive the data from/to them, it is inevitable that TCP/IP stacks are an integral part of modern embedded systems. Many microprocessor manufacturers have implemented an Ethernet controller with a complete TCP/IP protocol stack in the chip. [23] Embedded devices usually have slow processors (compare it with the IT counterparts), low storage space, and small volatile memory, they also usually run a lightweight operating system (instead of a full-fledged modern OS). Therefore, a traditional, extensive, and generic TCP/IP implementation may not be usable in a lot of embedded devices. For such devices, a TCP/IP stack is usually developed. That leads immediately to the fact that embedded TCP/IP stacks are numerous and are fertile ground for long persisting bugs (since the wheel is getting reinvented again and again with the same mistakes happening).

2.1.1 The implementation of TCP/IP stacks

The implementation of TCP/IP can generally be divided into two categories, one is inspired by (or even adopts) the BSD TCP/IP implementation, and the other is independently written. [24] BSD (Berkeley Software Distribution) [25] is an operating system derived from the Unix system. It is often used as an operating system for workstations. Therefore, many TCP/IP implementations changed from BSD are suitable for larger architectures.

In order to function within the performance limitations of some embedded devices, some TCP/IP stack implementations simplify the complete TCP/IP stack implementation. But in the meantime, they always try their best to ensure that the simplified TCP/IP stack can always be compatible with the standard complete TCP /IP stack for communication. The key to embedded system design is to optimize the TCP/IP protocol stack according to hardware and software conditions. Developers must build the TCP/IP protocol stack according to the requirements of the running program and the processor on the embedded system and simplify the TCP/IP protocol stack. [26]

There are many strategies to simplify the TCP/IP protocol stack. We can simplify the TCP/IP protocol stack according to a specific application. We only need to implement the part of the TCP/IP protocol that the application requires. For this simplified method, the embedded web server is a good example. The HTTP protocol and the HTTPS protocol are widely used. Often, we only need to provide a web interface in the embedded device to satisfy many functions. T.

Lin, H. Zhao, et al. proposed the implementation of an embedded web server called Webit. [27] In this implementation, they retained the protocols necessary for the webserver and some basic network function protocols. These protocols are ARP, RARP, ICMP, TCP, and HTTP. They removed some other complicated and rarely used protocols, such as FTP and SNMP. In addition to customizing the protocol stack for the application, they also used another simplified method, which can simplify the TCP protocol. The TCP protocol, aiming to provide connection and reliability, tends to be complex in nature, as such, programmers tend to simplify the TCP protocol in embedded systems. In Webit, the authors found that the functions of connection maintenance and the retransmission timer computation [28] in the standard TCP implementation have performance implications. So, the authors simplified the TCP protocol implementation by minimizing and thus simplifying the timer and reliability code. Many other implementations follow the same approach, such as Microchip's TCP/IP Stacks. [29]

Other simplification methods are [24]: Firstly, limiting the number of TCP connections. Some TCP/IP protocol stacks reduce the complexity of protocol stack implementation by restricting only one TCP connection at a time. Secondly, omitting IP fragmentation support. However, the latter method has a disadvantage. Normally, we will not encounter fragmentation, but when fragmented packets are received, the simplified TCP/IP protocol stack cannot reassemble them and drop them instead.

The uIP TCP/IP stack [30] developed by A. Dunkels applied the simplification approaches that we mentioned above, and it only implements four protocols, which are ARP, IP, ICMP, and TCP. For the application layer protocols above TCP, such as HTTP, FTP, and SMTP, the developers can implement them as applications on top of uIP. The support for IP fragmentation and segmentation was dropped from the stack. This greatly reduced the footprint of the stack.

Another well-known open-source TCP/IP protocol stack, lwIP [31], developed by the same author, also uses various simplification strategies so that the protocol stack can run on some 8-bit and 16-bit devices. For example, for the IP protocol, only the basic functions of sending, receiving, and forwarding packets were implemented, but like the implementation of uIP, the processing and reassembling functions of fragmented IP packets were dropped in addition to the support of IP options. Although the TCP protocol accounts for about half of the entire lwIP implementation, the author also simplified it from various aspects.

2.1.2 Potential threats in the embedded TCP/IP stacks

Although the implementation of various simplified TCP/IP protocol stacks meets the needs of embedded devices to communicate with other traditional devices, the implementation of various protocol stacks also poses a very big threat to these embedded devices.

The first threat is obvious. The starting point for offering the Internet communication function for embedded devices is to provide normal well-behaved users with more interfaces to interact with embedded devices and to enhance the functions of embedded devices by enabling embedded devices' Internet connectivity. But while we have made embedded devices more accessible and more open, we have also created new attack surfaces for threats to exploit. The richness of features provided by embedded devices to legitimate users adds up to the number of attack vectors malicious actors can use.

The second threat is that the original intention of embedded TCP/IP stack development is to enable TCP/IP implementation to meet the communication needs of embedded devices with limited resources. When developers design the TCP/IP protocol stack, security considerations are not ranked first, or security is not considered at all in the design phase. And as mentioned above, usually the development of the TCP/IP protocol stack needs to simplify the standard TCP/IP implementation. In the process of simplification, many security-related mechanisms may be overlooked, because even in the standard TCP/IP implementation, some of these mechanisms are often not present. For TCP/IP developers, these security mechanisms may be regarded as the culprit for the excessively large code space of the protocol stack that will impact the performance of serving the functional needs of users.

Thirdly, many standard TCP/IP protocol implementations have been used for a long time, and some of the obvious vulnerabilities have been discovered by security researchers and mitigated by developers. Implementing a TCP/IP protocol stack from scratch is technically difficult and error prone (standards can even be subject to speculation). In addition to that, the difficulty of updating a large number of embedded devices makes patching vulnerabilities hard.

Last but not least, since many embedded TCP/IP stacks are open-source programs, many other developers will reuse some open-source codes when designing their own embedded TCP/IP stacks. For example, wattcp is developed based on tinytcp. [32] Watt-32 [33] is developed on the basis of wattcp. The implementation of TCP/IP support in Contiki and its branch version Contiki-ng (new generation) also uses uIP TCP/IP stacks. [34] One problem with this high

frequency of code reuse is that if there is a security vulnerability in the implementation of a TCP/IP stack, then other TCP/IP stacks developed based on this TCP/IP stack may also have this vulnerability. Attacks against devices running a certain TCP/IP stack can quickly propagate to other devices sharing the vulnerable implementations of a TCP/IP stack.

These threats have attracted the attention of security researchers. For example, the Armis research team found more than 11 zero-day vulnerabilities in VxWorks and named them URGENT/11. [35] These vulnerabilities exist in the TCP/IP stack of VxWorks, according to Armis's follow-up observation shows that nearly 97% of the affected devices have not been patched 18 months after the vulnerability was disclosed. Forescout's research team found 33 vulnerabilities in a series of open-source TCP/IP stacks and named them AMNESIA:33. [36] It is estimated that these vulnerabilities affected more than 150 vendors and more than 1 million vulnerable device units. In their study NUMBER:JACK [37], they found 9 new vulnerabilities affecting embedded TCP/IP stacks. These vulnerabilities are related to weak Initial Sequence Number (ISN) generation. They are not as critical as some of the vulnerabilities found in the study AMNESIA:33, but they are much easier for attackers to discover. This further illustrates how dangerous some of the bad embedded TCP/IP stack implementations are because weak ISN generation vulnerability was discovered and fixed in traditional devices decades ago. Another study of Forescout namely NAME:WRECK [38] discovered some underlying issues related to domain name system message parsing. They found 9 vulnerabilities related and they estimated that these vulnerabilities affect approximately 100 million devices.

2.2 Vulnerability detection

2.2.1 Vulnerability detection methods

Vulnerability detection methods can be divided into multiple categories according to different factors. For example, from an automation point of view, vulnerability detection methods can be either manual, semi-automated, or fully automated. Manual vulnerability detection methods often require the participation of security experts or experienced developers. Those who perform vulnerability detection first need to have a relatively deep understanding of the principles of security vulnerabilities and their impact. Secondly, they need to have a comprehensive understanding of the software or system being tested. They need to understand what functions these software and systems have, and understand the standard implementation of these functions, as well as the bugs that can often be found in these functions. This requires

the richer experience and knowledge of those who perform vulnerability detection. Not only that, but manual vulnerability detection also requires experts to spend a lot of time and energy to locate vulnerable code. It is often the case that experts may not find any vulnerabilities after testing for a long time. Usually, vulnerability detection requires a certain amount of luck and fleeting inspiration. Because of these characteristics of vulnerability detection, researchers are more motivated to develop automated-assisted methods and fully automated methods.

A straightforward distinction factor between vulnerability detection methods is the amount of visibility within the source code of the analyzed artifact: white box testing, black box testing, and gray box testing.

1. **White box method**: The white box method requires us to analyze the existing security issues after we have a more comprehensive and in-depth understanding of the internal structure and logic of the software or system. To gain such knowledge, the source code must be available and analyzed. The advantage of this method is that because we have a good understanding of the target's logic, we can quickly and accurately analyze the causes of vulnerabilities when we discover security vulnerabilities in the target. Usually, in this way, we will analyze the source code of the software through code reviewing or use the disassembler to analyze the binary files and find the existing security vulnerabilities. But this method has some shortcomings. First of all, this method may require the source code of the software which may not be publicly available for a lot of software. Also, researchers need to have knowledge of not only the logic and function of the program but also the quirks and shortcomings of the language used to write it. Apart from that, this method usually requires a lot of researchers' time, energy, and possession.

2. **Black box method**: Contrary to the white box method, the black box method does not require us to have any internal knowledge about the target. The method of finding vulnerabilities considers the program to be a black box and we only know about the input and the supposed output. The shortcomings of this approach are obvious. Due to the lack of understanding of target structure and logic, our testing process is often blind and directionless. Like the white box method, it may also consume a lot of our time, but in a different way. We often use automated tools to test the target, which doesn't actively consume too much researcher time. Finding vulnerabilities through this test often requires a certain amount of direction, otherwise it will be based on sheer luck. Unlike the white box method, we cannot understand the causes and triggering conditions of security issues

after they are discovered. A lot of research is often needed after we discover potential security issues. The advantage of the black box method is that it can result in erroneous behaviour without a lot of investment, but the root cause analysis would be demanding in time and expertise (reverse engineering is a challenge by itself).

3. **Gray box method**: The gray box method is a middle ground between the white and black box methods. Usually, we only need a certain understanding of the target and its inner modules (and or state), or we can also analyze the target in a phased manner. As mentioned before, we can abstract certain parts or behaviors as a black box, to quickly find unexpected behavior, and then a granular analysis is followed to determine the source of the issue. The gray box method provides both the advantages of the white box method where thorough information is needed and the black box method were avoiding details saves time, and the gray box method balances the shortcomings of the white box method while it saves time and the black box method by having more knowledge of the appropriate entities within the program and isolating them.

From another perspective, we can divide vulnerability detection methods into static analysis and dynamic analysis. Static analysis is a process of analyzing the structure, code, and logic, etc. of the target without executing the program. Dynamic analysis, on the other hand, requires interaction between the researcher and the target.

Although there are various perspectives for us to classify vulnerability detection methods, we can usually describe a vulnerability detection approach by combining them. For example, a certain vulnerability detection method can be a white box automatic dynamic detection method. In the first part of the thesis, we used a white box, automated assisted static analysis methods for adding support for finding vulnerabilities in DNS clients' implementations in embedded TCP/IP stacks. Next, we will focus on the background discussion related to static analysis methods.

2.2.2 Static analysis

Static analysis can be used to find vulnerabilities in source code. By applying this technique, we could understand detailed information of the software we are testing and all paths in the code can be considered. In some cases, we may not be able to obtain all the source code of the software, we can still reverse-engineer the binary files for static analysis. Regardless of whether performing static analysis on source code or decompiled code, we can't find all the

vulnerabilities in the software according to Rice's theorem.[39] Static analysis tools can only approximate programs’ behaviors. In addition, static analysis tools may also generate false negatives and false positives. Further analysis by security experts is needed for the results produced by static analysis tools. Therefore, most static analysis tools are automated-assisted vulnerability discovery methods. In this section, we will discuss different static analysis techniques and some code representations used in these techniques.

1. **Pattern matching**: The simplest static vulnerability search tool may be the Unix utility grep [40]. Grep is a powerful string-matching tool. By matching the strings, we can find some vulnerable patterns that exist in the codebase. We do not need to understand the logic of the program, we only need to pay attention to whether the same vulnerable patterns appear in the codebase. This static analysis method is called pattern matching. ASIDE (abbreviation for Application Security plugin for Integrated Development Environment) [41], developed by M. Mohammadi et al., is an Eclipse [42] plug-in that uses pattern matching static analysis technique. In addition to detecting and identifying vulnerable code, the plugin also provides informative fixes for developers. Since the tool carries out detailed vulnerability-related information in the early stage (development phase) of the program release cycle, it can effectively reduce the bugs that occurred during the programming.

2. **Lexical analysis based pattern matching**: Although not a method on its own, lexical analysis is commonly within static analysis tools, when combined with searching for problematic patterns, we would have an improved version of pattern matching. Unlike blind pattern matching, it first pre-processes the source code of the program, converts the source code into tokens, and then uses these tokens to identify vulnerable patterns. Tools that use this static analysis method are RATS [43] and ITS4 [44]. Lexical analysis can greatly improve the accuracy of pattern recognition, but this method still produces many false positives and false negatives.

3. **Data-flow analysis**: The purpose of the data analysis is to compute information for program points statically. It is a well-known technique for compiler optimizations and can be used to assist vulnerability detection. What we would like to know for a program is that whether tainted data can reach certain sensitive sinks and there is no sanitization for the data. We can easily obtain this information through data analysis. Two popular techniques used in data analysis are reaching definitions analysis and taint analysis. Using reaching definitions analysis, we can statically determine which definitions may reach a program

point. Tainted data is data that can be introduced or modified by program users. Malicious users often take advantage of user input under his or her control and introduce tainted data by the input. The tainted data will propagate in the program, and if it influences a certain value in a way that the attacker wants, it will eventually make the program run abnormally. Using taint analysis, we can track all tainted data to see if it can reach sensitive sinks. Data analysis can greatly improve the vulnerability detection process compared with pattern matching. There are many studies using this technique to detect vulnerabilities. N. Jovanovic, C. Kruegel and E. Kirda developed a tool called Pixy that can detect cross-site scripting vulnerabilities in PHP scripts. [45] Using this tool they discovered 15 new vulnerabilities. J. Kronjee, A. Hommersom and H. Vranken applied some of the techniques of data flow analysis to extract data set features. [46] They proposed a novel method of using machine learning to predict vulnerabilities in software. Using this method, they generated data sets for SQL injection and Cross-Site Scripting (XSS) types of vulnerabilities. And they found that the tools they developed can effectively find two types of vulnerabilities in the program. The data flow analysis techniques they used are reaching definitions analysis, taint analysis, and reaching constants analysis. H. Kim, T. Choi, et al. developed a vulnerability checker. [47] They designed a special language to express vulnerability patterns and applied lightweight data flow analysis and control flow analysis.

2.2.3 Code representations

In order to describe the properties of the program, many different code representations have been developed. They are often used in the field of code analysis. We can obtain the semantic information of the code by using code representations to analyze the program. These code representations are also widely used in the field of vulnerability detection, specifically used in the static analysis techniques discussed above.

1. **Abstract syntax trees:** The first most widely used code representation is abstract syntax trees (ASTs). ASTs are ordered trees. They use inner nodes to represent operators that appear in the program, such as arithmetic operators, logical operators, assignment operators, etc. They use leaf nodes to represent the operands of each operator. In this way, ASTs can show us how the various expressions that make up the program are nested. F. Yamaguchi, M. Lottmann, and K. Rieck proposed a method to use ASTs to assist researchers in auditing source code. [48] In their method, they extract the AST from the source code and identifies structural patterns in these trees. The functions in the source code can all be represented by

a mixture of these patterns. The relevant properties of the code are stored in the subtrees of these structural patterns. By using AST code representation in vulnerability detection, the method can find structural patterns with similar properties in the codebase and then discover vulnerabilities. This is a bit similar to the lexical analysis-based pattern matching method we mentioned earlier. They both pre-process the code first and then identifies vulnerable patterns of known vulnerabilities in programs. The difference is that compare with tokens, the structure in ASTs has rich semantics. Fabian Yamaguchi's team tested their method against some open-source programs, and they were able to find zero-day vulnerabilities after only checking a small part of the codebase. H. Feng, X. Fu, et al. proposed a vulnerability detection framework that uses machine learning methods to predict existing vulnerabilities from source code. [49] Other methods that use machine learning technique often only use vulnerable codes of known vulnerabilities as data set to train their models. However, this framework proposed by them extracts the AST of the vulnerable code, and it removes other redundant information from it. The framework only keeps the syntax information of the vulnerable code. Using this framework, they can find vulnerabilities accurately and the false positive rate of this method is low. ASTs can provide us with the semantic information of the program, but because this code representation method lacks information about control flow and data dependencies, we still cannot perform advanced code analysis.

2. **Control flow graphs:** Another code representation method, control flow graphs (CFGs), can well represent the execution order of the statements in the program code, and we can also analyze the conditions that some program execution paths need to meet. S. Sparks, S. Embleton, et al. proposed a novel black box fuzzing method [50], in which static analysis methods are combined with dynamic analysis methods. They further developed the traditional fuzzing method by using a genetic algorithm based upon the Dynamic Markov Model fitness heuristic. Coupled with the analysis of the control flow of the binary file, the framework provides the fuzzer the ability to intelligently guide the fuzzing inputs, thus, achieving good code coverage and penetration depth into a program's control flow logic. Although the CFGs provide more semantic information than only using ASTs, it is far from enough for robust vulnerability detection. This is mainly because that ASTs and CFGs lack information about the flow of data, which is very important information in vulnerability

detection because through the data flow analysis technique we can accurately track user input to see if user input ends up in sensitive sinks.

3. **Program dependence graphs:** The code representation containing data flow-related information includes program dependence graphs (PDGs), which was proposed by J. Ferrante, K. J. Ottenstein, and J. D. Warren. [51] Its edges can be generated by the control flow graph. These edges are divided into two types, one is data dependency edges, which can reflect the relationship between two different variables, and the other is control dependency edges, which can reflect the relationship between predicates and variables.

4. **Code property graphs:** Though we can utilize code representation in static analysis to support vulnerability detection, each code representation alone has its own limitations. Single code representation cannot provide comprehensive program semantic information, so the detection ability of vulnerability detection methods driven from it will not be robust enough, and detection methods may prone to false positives. F. Yamaguchi, N. Golde, et al. proposed a novel code representation, which is called code property graphs. [52] It combines three different code representations, namely AST, CFGs, and PDGs. The open-source vulnerability detection tool Joern we will use is designed based on this concept of CPG. [53] An advantage of combining all three code representations together is that a combined code representation can provide us with richer semantic information, we can easily use it to model known vulnerability patterns. The open-source tool we will use in the vulnerability detection part is based on the idea of CPG. We will introduce it in detail in Chapter 3.

2.3 Vulnerability exploitation

For attackers, the purpose of attacking information systems is to use unauthorized operations to obtain unauthorized privileges so that they can perform malicious actions against the information systems. Attackers often want to have the highest privileges for a device. For a device using the Windows operating system, the ultimate goal of the attacker is to obtain the administrator level or system-level privileges, and for a device that uses Unix-like systems, the ultimate goal of the attacker is to obtain root-level privileges. Exploits that allow attackers to

obtain these administrative privileges are very attractive to attackers, especially when attackers can execute these exploits remotely. [54]

There are many techniques that allow an attacker to obtain administrative privileges remotely. One is called remote arbitrary command execution. There are many types of vulnerabilities that allow the attacker to execute arbitrary commands remotely, especially in web applications. These vulnerabilities in web applications often unique to web applications and related to the programming language of the web application, the middleware used, or the back-end SQL service. For example, file inclusion vulnerabilities are often found in web applications written in PHP. File inclusion vulnerabilities [55] usually require a file inclusion call in the currently running script, and the parameters of this call are user controllable. In this way, when the script gets executed, the included files will also be executed. In the file defined by the attacker, the attacker can call the function that can call system commands so that the remote command execution can be achieved. SQL injection vulnerabilities can also make it possible for an attacker to run arbitrary remote commands. [56] The possibility and attack method are related to the database application used. For example, in MSSQL, there is a process called xp_cmdshell, we can use it to execute windows commands. In the combination of MYSQL and PHP, we can upload a PHP script file that can execute commands using MYSQL commands. After we browse the file using the web interface, PHP will parse the file and finally achieve remote command execution. The flexibility of various components of web services provides multiple possibilities for executing remote commands. Deserialization and object injection vulnerabilities are very dangerous vulnerabilities in web applications written in JAVA or PHP language. This kind of vulnerability can also achieve the effect of executing remote commands. When the user-controllable data is deserialized, the user can introduce malicious objects, thereby interfering with the logic of the program, and in some cases, enabling the user to execute remote commands. N. Koutroumpouchos, G. Lavdanis, et al. have developed an open-source tool called ObjectMap that can detect such vulnerabilities. [57]

Another attack technique that allows an attacker to obtain administrative privileges remotely is remote code execution. And exploitation of a buffer overflow vulnerability can allow us to perform this attack technique. In a study [58], researchers analyzed the relationship between vulnerability severity and vulnerability type. The study found that high-severity vulnerabilities accounted for half of the number of vulnerabilities, making it the most common vulnerability. Among these high-risk vulnerabilities, buffer overflow vulnerabilities account for a large proportion. Buffer overflow vulnerabilities can be found in the C language and C++ language

because these programming languages do not automatically check whether the data written into the array exceeds the boundary. Buffer overflow vulnerabilities are not unique to web applications. Any software written in C language or C++ language may have this type of vulnerability. In the second part of this study, vulnerability exploitation research, we will illustrate the impact of this type of vulnerability by analyzing an exploitation case study of a buffer vulnerability. Buffer overflow vulnerabilities are mainly divided into two categories, one is stack overflow vulnerabilities, and the other is heap overflow vulnerabilities. The two types of vulnerabilities will be explained in detail below.

2.3.1 Stack overflow

Almost every modern runtime provides a stack a medium for inter procedural calls and local variable management. Functions use it to allocate local variables, to pass arguments to sub-functions, to restore the state of registers to what it was before the call, and to return the execution flow from the called function to the callee function. For this to happen, every function has an associated stack frame. The stack allocates a stack frame for each function and stores the frame pointer in the stack to optimize the process of finding each variable by the processor.

As shown in Figure 2, when the program encounters a function call instruction, it first pushes the arguments needed to call the function into the stack. And it then pushes the return address of the current function, that is, the address after the function call instruction, into the stack. After completing these, the program will adjust the stack frame, and the data previously pushed onto the stack belongs to the stack frame of the parent function. At this time, in order to optimize the addressing process of the processor, the program will also push the frame pointer of the previous stack frame into the stack. During the execution of the called function, the initialized local variables will also be pushed onto the stack. The data starting from the frame pointer all belong to the stack frame of this function. When the called function returns after completion, the program will return to the parent function to continue execution according to the return address stored in the stack.

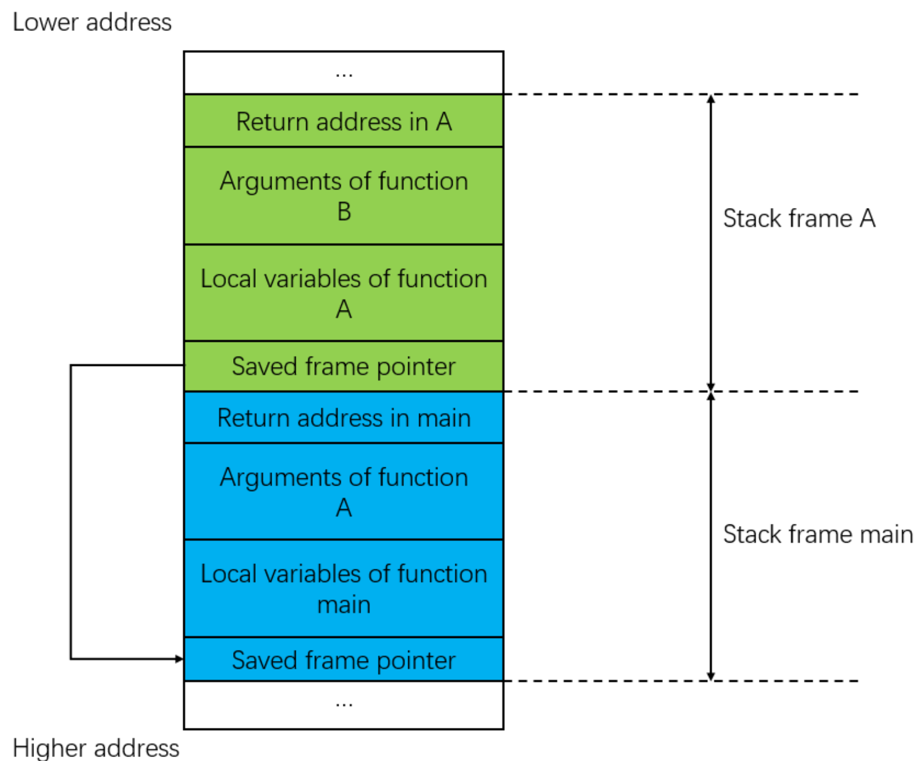

Figure 2. Stack variables used in function calls

For convenience, in Figure 2 the upper part of the stack is the lower address, and the lower part of the stack is the higher address. Usually, the stack grows toward the lower address of the program. The growth direction of the stack in this figure is upward.

Overflowing a local variable in function A (in Figure 3), means having the ability to shape the stack space through the stack overflow vulnerability. The attacker can overwrite the contents of high memory addresses from the local variable part of function A, such as the frame pointer, return address, and contents in the stack frame of the main function. However, under normal circumstances, the overwriting return address can allow an attacker to successfully hijack the control flow of the program. When Address Space Layout Randomization (ASLR) [59] is not enabled, the memory address is fixed, the attacker can put the instruction he wants to run in a controlled address and overwrite the return address by overflowing and place a pointer to his own instruction in the return address. In this way, when the function returns, the program can return to the address specified by the attacker and start executing the attacker's instructions. With some memory protection measures enabled, the exploitation of the stack overflow vulnerability is still possible. For example, ASLR randomizes the memory address of the external libraries linked on the targeted binary each time it gets executed to avoid attackers take

advantage of the provided instructions. Attackers can use an information leak vulnerability to leak an address that can be further used by stack overflow vulnerabilities to bypass ASLR.

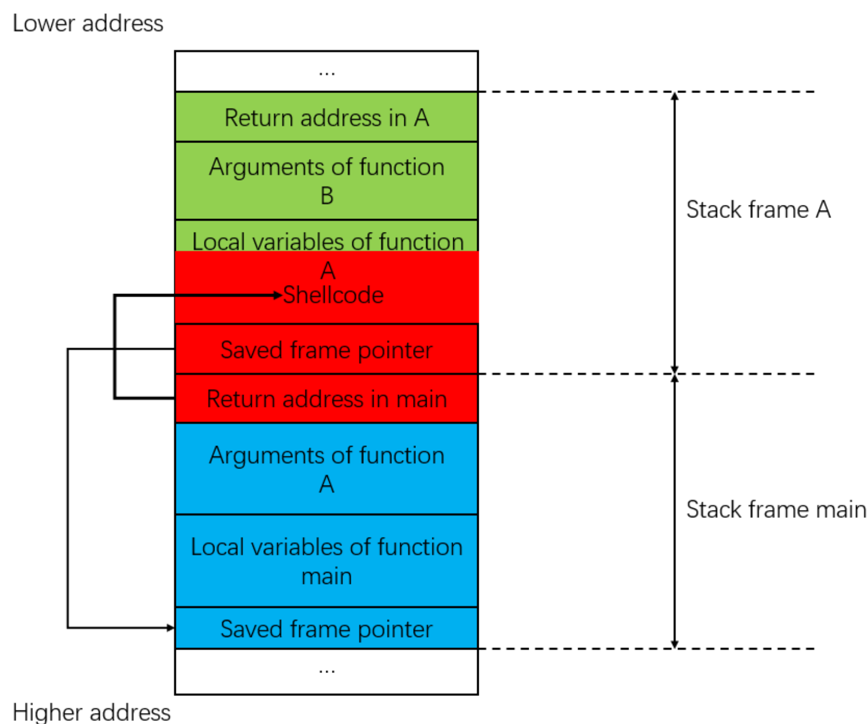

Figure 3. Overflow a Stack variable

2.3.2 Heap overflow

Heap is a memory region that is used to provide dynamically allocated buffer to running programs. Unlike the stack, the heap memory space grows toward higher addresses, and because modern calling conventions don't use the heap for inter procedural calls, we cannot hijack the execution flow using the same techniques.

If we are able to overflow a dynamically allocated buffer, there are several approaches that might be helpful for exploitation. Firstly, if the value we are overflowing is a member of a structure, and there are other members after this value, we can change these members by overflowing the value. Secondly, if there is a memory chunk within the overflow distance, we can change the metadata of the memory management data structure in the heap space. Finally, in some rare cases, we might be able to write beyond the heap area. For example, we can overwrite values inside the stack memory area. For the first approach, exploitation highly depends on the program logic. For example, in some cases, function pointers or object's virtual function pointers may be stored in the heap. We can hijack the control flow by overwriting

these pointers, but these two techniques are not always available. Under normal circumstances, we can overwrite a normal pointer value, and when the pointer gets dereferenced after the overflow, we can trigger an exception by setting a malformed pointer. But if the pointer never gets dereferenced, we can't exploit this bug.

The exploitation of heap overflow vulnerabilities is usually very complicated and highly sensitive towards the target logic and the memory allocator implementation. If we want to overflow the metadata of the heap memory management data structure, the different methods are also closely related to the allocator used by the system. One of the most frequently used techniques is to use the unlink function to hijack the control flow. [60] This is also the technique we will use in the vulnerability exploitation case study. The unlink function is usually called in free functions, it is used to delete members from a doubly linked list.

By reassigning the pointers in the node to be deleted and the nodes before and after it, unlink can make the node that needs to be deleted disappear from the doubly linked list, and the assignment of pointers gives the attacker the ability to write arbitrary data to any address. If an attacker can find a way to trigger an unlink call and carefully overflow to overwrite metadata of a memory chunk in the heap area, the attacker may be able to hijack the control flow of the program. Unlink technique is a classic and maybe the simplest way to exploit a heap overflow vulnerability. Security researchers have developed many different attack techniques for different allocators. For example, for glibc's allocator, there are shrinking free chunks, house of spirit, house of lore and house of force, etc. [61]

3 Design of vulnerability detection method and exploitation case study

3.1 Vulnerability detection using static analysis tool Joern

In this study, we will use an open-source static code analysis tool called Joern to check three vulnerability patterns that exist in embedded TCP/IP DNS name parsing implementations. We will use the query language provided by Joern to create a script that can detect these three vulnerability patterns. Then we will test the script against several embedded TCP/IP stacks to find similar vulnerability patterns in the source code of these stacks. After that, we will analyze the outputs and compare them with the results of discovering vulnerability manually.

3.1.1 DNS name parsing in embedded TCP/IP stacks

The basis for two devices on the Internet to communicate with each other is their IP address. Each device on the Internet, or more accurately, each interface on the Internet has a unique IP address. It provides the location information of the device interface on a network and provides path information for two devices that need to communicate with each other through some complex protocols. In IPv4, the IP address is composed of 32-bit numbers. Because the protocol design did not consider the exhaustion of the IP address pool, so a new version of the 128-bit IP address IPv6 was designed. There are some problems with the use of IP addresses. For an IPv4 address, 32-bit numbers are not easy to be remembered by humans, not to mention the IPv6 address with 128-bit numbers. It is hard to imagine that whenever we want to visit a certain website, we need to enter the IP address of the website in the address bar of the browser. Moreover, the IP address of many websites is not fixed. Administrators need to change the IP address of the website from time to time for various reasons. This provides a greater obstacle for users to access the websites because whenever users want to visit certain websites, they first need to obtain the updated IP address. Domain name system (DNS) was proposed to solve these problems.

DNS is a hierarchical and decentralized naming system that maps domain names, represented as human readable strings, and IP addresses. It is like the phone book on the Internet. There are two main ways of DNS lookup, namely recursive and iterative. In the recursive lookup mode, the DNS client first sends a query request to the local DNS server. If there are cache records of the query in the local DNS server, the local DNS server directly replies response of this record to the DNS client. If there are no cache records of the query in the local DNS server, the local

DNS server will initiate a query to the DNS servers at all levels and returns the final query results to the DNS client. The local DNS server will perform all needed requests on the client's behalf. In the iterative lookup method, if there are no cache records in the local DNS server the DNS client will query with DNS servers at all levels. Each level of DNS server will return the address of the next level of DNS server to the DNS client, and the DNS client will continue the query according to the response returned by each level. The client makes all needed requests untile it receives the final answer. The two lookup methods are shown in Figure 4 and Figure 5.

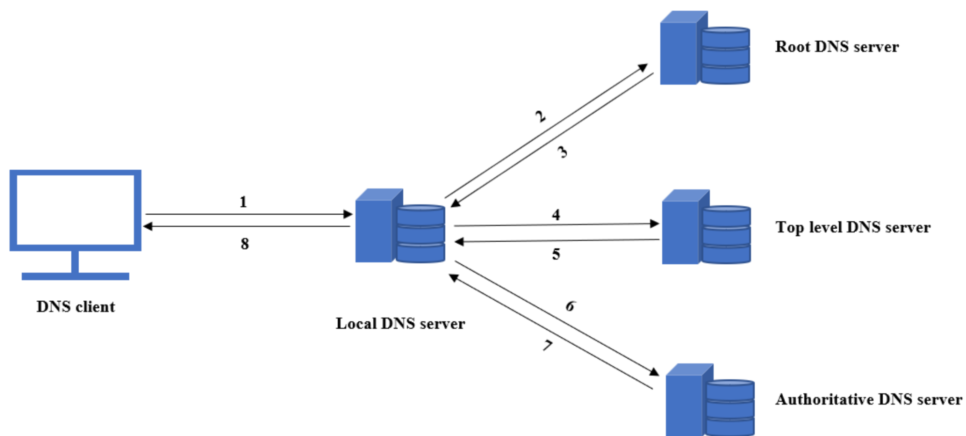

Figure 4. Recursive DNS lookup process

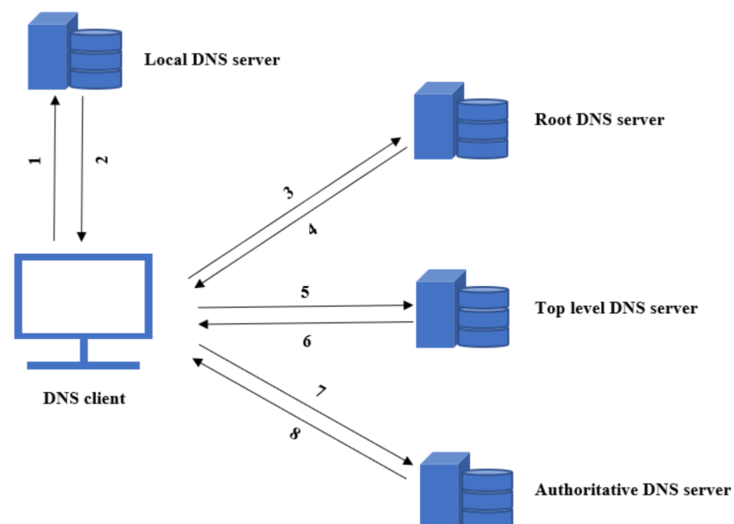

Figure 5. Iterative DNS lookup process

DNS request and response are completed by the DNS protocol, which is an application layer protocol, mainly using user datagram protocol (UDP) for data transmission. The DNS query process is completed by the DNS client sending a DNS request and the DNS server sending a DNS response. The DNS protocol runs on port 53 of the server. The structure of the DNS request format and response format is basically similar. They are all called messages. The structure of the message is explained in detail in RFC1035 [62]. The format of a message consists of the following sections, namely DNS header section, question section, answer section, authority section, and additional records section (shown in Figure 6). The field in the question section defines the relevant information of the domain name that the DNS client requests from the DNS server. The answer section contains response resource records (RRs) used to reply to the query. The RRs in the authority section correspond to the relevant information of the authoritative name server. Finally, the RRs in the additional records section contains some additional information about the query.

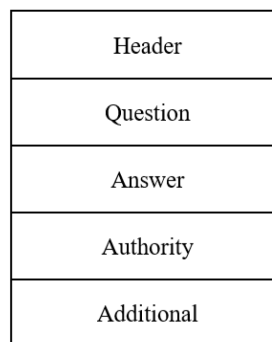

Figure 6. The format of a DNS message

The header section in the message always exists. The structure of this section is shown in Figure 7. The fields in the header determine the existence of other sections and the number of RRs in them. In addition, other fields determine whether the message is a query or a response, etc. For example, if the data in the qdcount field in the header section is 1, it means that there is only one question in the questions section. Several other fields: ancount, nscount, and arcount have similar functions. The ID field in the header defines the transaction id of this message. The DNS client defines this field when sending a DNS lookup request. The DNS server will use the same ID when replying to the query request, and the client will match it with the query sent by itself. The QR field defines whether the message is a query or a response. When the field is set

to 0, it means that the message is a query. When the field is set to 1, it means that the message is a response.

<table>
<tr><td colspan="9">ID</td></tr>
<tr><td>Q
R</td><td>Opcode</td><td>A
A</td><td>T
C</td><td>R
D</td><td>R
A</td><td>Z</td><td>RCODE</td></tr>
<tr><td colspan="9">QCOUNT</td></tr>
<tr><td colspan="9">ANCOUNT</td></tr>
<tr><td colspan="9">NSCOUNT</td></tr>
<tr><td colspan="9">ARCOUNT</td></tr>
</table>

Figure 7. The format of a DNS message header

The questions in the question section mainly consist of three fields (shown in Figure 8), which are name field, type field, and class field. What is stored in the name field is the domain name being queried. The type field stores the resource type in the query request, which can be A-type, CNAME type, PTR-type, etc. The address type is stored in the class field, usually, it is set to 1, which means it is the Internet address.

QNAME
QTYPE
QCLASS

Figure 8. The format of a question section

The resource record in the other three sections (answer section, authority section, and additional records section) mainly consists of six fields (shown in Figure 9). In addition to the name field, type field, and class field, the three additional fields they have are the time to live field, the rdlength field, and the rdata field. Time to live field tells us that how long to cache a query before it should be discarded and requesting a new one. Rdlength field specifies the length of the rdata field in octets. The length of the Rdata field is variable, and it mainly contains a string of octets to indicate the records it contains.

NAME
TYPE
CLASS
TTL
RDLENGTH
RDATA

Figure 9. The format of a resource record

A domain name in the message consists of a series of labels, and the domain name is terminated by the NULL byte (0x00). Take the domain name google.com as an example (shown in Figure 10). In the message, this domain name consists of the following two labels, namely google (0x67 0x6f 0x6f 0x67 0x6c 0x65) and com (0x63 0x6f 0x6d). There are two length fields before these two labels. The length fields indicate the respective lengths of the two labels. The length field before label google is 0x06, and the length field before label com is 0x03. We can easily see that the length of the string "google" is 6, and the length of the string "com" is 3. After the two labels, there is a NULL character (0x00) to indicate that the domain name ends here. The length of various parameters and objects related to DNS is defined in RFC1035. It stipulates that the length of each label cannot be greater than 63 octets, the length of a name field cannot be greater than 255 octets, and the length of the entire UDP messages cannot be greater than 512 octets.

len:6	**g**	**o**	**o**	**g**	**l**	**e**	**len:3**	**c**	**o**	**m**	**null**
0x06	0x67	0x6f	0x6f	0x67	0x6c	0x65	0x03	0x63	0x6f	0x6d	0x00

Figure 10. Domain name "google.com"

A situation we often encounter is that the same domain name is repeated in the message, which brings a higher amount of resource occupation for the message transmission in the DNS protocol. In order to reduce the size of the message, RFC1035 also provides us with a name compression scheme. The two anti-patterns we need to analyze are closely related to it. This compression scheme replaces a series of labels with a pointer to a previous occurrence of the same label sequence. The entire compression pointer consists of two bytes, and the first two

bits are set to 1 to indicate that this "label" is a compression pointer. As mentioned before, the length of the label is limited to no more than 63 octets. We can distinguish pointers from normal labels by setting the first two bits to 1. The remaining part (14 bits) in the compressed pointer represents the offset of the pointer from the start of the DNS header. Take the previous google.com as an example, if there is a www.google.com domain name in our message, we can use the domain name compression scheme to represent it as shown in the figure below.

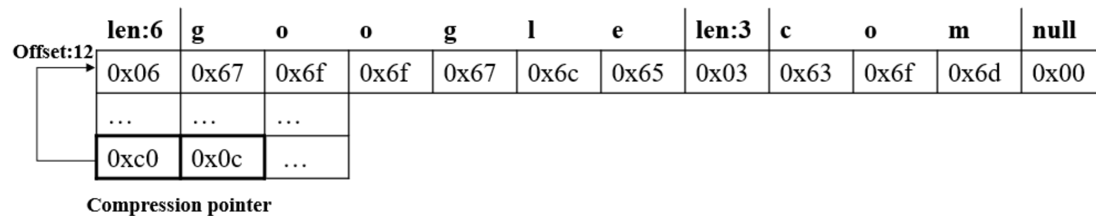

Figure 11. Compression pointer

In Forescout's research "NAME:WRECK", researchers have found a variety of vulnerable patterns related to DNS parsing in the embedded TCP/IP stack implementation. [38] They found that many embedded TCP/IP implementations often make mistakes:

- If there is a lack of verification of the transaction ID field or insufficient randomization of the transaction ID and the port number in a DNS client implementation, it makes it easier for an attacker to obtain the transaction ID and source port of a DNS query, so that the attacker has the ability to forge DNS replies.
- Some implementations lack the validation of domain name character, any character can be accepted as part of a domain name. This itself does not directly lead to vulnerabilities, but it gives the attacker more freedom. If there is a buffer overflow vulnerability in the domain name related variables, the attacker can conduct remote code execution attacks without any character restrictions.
- In some implementations, there is no validation of NULL termination. These implementations assume that users will always provide NULL termination, but attackers can take advantage of this trust. In some cases, some attacks can be achieved by placing NULL termination in other offsets.
- Some implementations lack verification of some count fields in the message header. Attackers can achieve some attacks by setting the count field that does not match the actual number of questions or the number of resource records.

They identified 6 anti-patterns in total. Next, we will focus on the rest of the anti-patterns discussed in the "NAME:WRECK" report. Later in the implementation section, we will use a static analysis tool to provide a detection method for these anti-patterns.

3.1.2 Anti-patterns related to DNS name parsing

1. **Anti-pattern 1: Wrong compression pointer checks**: According to RFC1035, the first two bits of the compression pointer are set to 1 to indicate that the current label is a compression pointer. In the process of parsing DNS messages, the program will first read the length field of a label. When the currently processed data is a normal label, this field is a normal length of the label. The common method to implement the function of detecting whether a label is a compression pointer is to perform logic AND to the data with 0xC0 (11000000) to determine whether the two most significant bits are both set to 1. However, a common mistake here is that the programmer may only check either of the two most significant bits of a label length is set to 1. This will cause the label length with the highest bit of 1 or the second-highest bit of 1 to be regarded as part of the compression pointer. In the following code example, we can find that there are three types of logic AND operation results in the if condition, which is 0xC0, 0x40, and 0x80. Only the label length that produces the 0xC0 result indicates that the data currently encountered is a compression pointer. This anti-pattern allows an attacker to set the label length field to a number whose highest bit is 1 or the second-highest bit is 1. In this way, during the domain name parsing, the program will treat the currently processing data as a compression pointer. This may cause the program to run abnormally.

2. **Anti-pattern 2: Lack of compression pointer offset validation**: As mentioned earlier, for a compression pointer, among the two bytes, the highest two bits are used to indicate that the label here is a compression pointer, and the other 14 bits indicate the pointer's offset. In lots of implementations, the program first clears the highest two bits to zero, leaving only the rest of the first byte, and then shifts this byte and adds it to the remaining byte of the offset and finally get the full offset. Different code implementations will use different ways to clear and shift bits, but the calculation process is relatively similar.

   The problem we will encounter here is that if there is no offset validation in the implementation, the attacker can define any offset, which means that the attacker can define an offset with a minimum of 0 and a maximum of 16383 (0x3fff) to make the

compression pointer point to the corresponding position after the DNS header since the offset is a 14 bits value. RFC1035 defines compression pointer as follows: a pointer that points to a prior occurrence of the same name of a label or a domain name. This means that the pointer needs to point backward and land on a valid uncompressed domain name. If the pointer controlled by the attacker points forward, or even points outside the entire message, it can make the program execute abnormally. In some cases, the attacker can launch a denial of service (DOS) attack or remote code execution attack using this malformed offset. A special case is that the attacker can set the offset to make the compression pointer point to itself so that the program will fall into an infinite loop.

3. **Anti-pattern 3: Lack of label length and domain name length validation**: The label length is limited to less than or equal to 63 characters. However, in some implementations, out of trust in user inputs or negligence in security, developers do not validate this field. This is problematic because an attacker can customize this field to facilitate the exploitation of other vulnerabilities. It is often can be seen that when performing DNS name parsing, the program will allocate memory space to a buffer with the value in this field to be the size of the buffer. Sometimes, the program will use the value in this field as the third argument of a memcpy call or any other argument of dangerous calls. With a customized label length field, the attacker can easily perform out-of-bounds read/write attacks. Even if the label length field is checked in the program, sometimes the program will not check the actual label size of a label, which is also not a good way to implement it. Usually, we calculate the total domain name length from the label length. We can check if the domain name length is exceeding 255 characters. If this type of check does not exist, the attacker can craft a very long domain name length and make the program run abnormally. As long as the attacker can control the label length field, the attacker can bypass the checks in the program to achieve malicious purposes.

3.1.3 A static analysis tool: Joern

Joern is a powerful vulnerability detection platform for C/C++ source code. By writing queries for patterns of some already discovered vulnerabilities, it can use static analysis methods to analyze the source code we provide and find the same vulnerable patterns from it. After we provide the source code, Joern will first use its fuzzy parser to parse the C/C++ source code and generate the cross-language code representation, code property graph from the output of the

fuzzy parser. It will then store the generated code property graph in a graph database. This graph database contains several properties of the program we need for static analysis. With the combination of abstract syntax trees, control flow graphs, and program dependence graphs, the code property graph generated by Joern provides us with semantic information of a program. Joern can use static analysis techniques driven from these code representations to analyze this rich and comprehensive information, and we can use the code property graph query language (CPGQL) to describe the previously discovered vulnerability patterns. Using the same queries, we can detect the same vulnerable code patterns (anti-patterns) in the codebase we want to check. The process of using Joern to detect anti-patterns in a codebase is shown in the figure below.

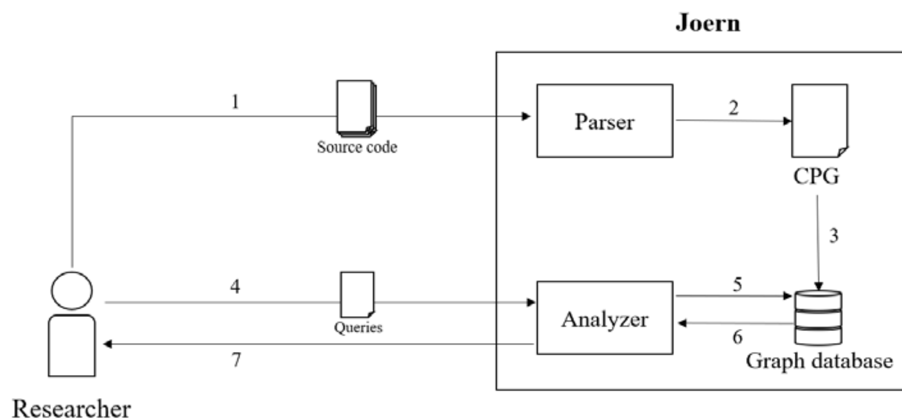

Figure 12. How Joern works

Joern's fuzzy parser can analyze any C/C++ code, even if the working environment does not support it or the source code provided is incomplete. The powerful parser provides a strong foundation for Joern's static analysis capabilities. Joern’s core idea of code property graphs integrates several different code representations so that it has a more comprehensive static analysis capability and can use a variety of static analysis techniques. The graph database and query language used by Joern provide security researchers with access to these capabilities, allowing security researchers to detect security vulnerabilities in a codebase. One advantage of using Joern is that the use of code property graphs avoids the limitations of using only one code representation. Reasonably exploiting the static analysis method provided by Joern can effectively reduce the false positives rate and can also allow us to use the query language to better locate vulnerabile code patterns. In the thesis, we will take advantage of the powerful static analysis capabilities provided by Joern and use the code property graph query language to define the discussed anti-patterns related to DNS name parsing in TCP/IP stack

implementations. And use these queries to test against the source code of some TCP/IP stack implementations.

There are two ways to run query scripts with Joern. One is to run our query in the interactive shell provided by the Joern platform, and the other is to directly provide the Joern tool with the query that needs to be run in the form of command parameters. Before running the query script, we first need to use the parsing tool in the Joern platform to generate the code property graph of the source code that needs to be tested (command is in Figure 13). The generated code property graph can be provided to the Joern tool together with our query script, allowing Joern to perform static analysis on the source code under test. We use the following command to generate the code property graph, where the out parameter specifies the path of the generated code property graph, and the source specifies the path of the source code to be analyzed.

```
1 joern-parse --out output source
```

Figure 13. Command to generate CPG from source code

After generating the code property graph, we can use the first method to run our query script. We can enter Joern's interactive shell using command "joern". In the shell, we use the following two commands to run our query script (Figure 14). We use the first command to import the generated code property graph into the current workspace. We can also directly import the source code path here using command "importCode", and Joern will automatically generate the code property graph of the code and import into the workspace for us. The second command will make Joern to run our query script.

```
1 importCpg(Path of the code property graph file)
2 cpg.runScript(Path of the query script file)
```

Figure 14. Commands to run query script in the interactive shell

In the non-interactive approach, we can directly provide Joern with code property graph and query script in the form of parameters. In this way, we need to import the required code property graph file in the script.

```
1 joern --script script_file --params cpgFile=cpg_file
```

Figure 15. Command to run query script outside the interactive shell

3.1.4 Selected embedded TCP/IP stacks

In the thesis, we mainly use Joern queries to test the following TCP/IP stacks to evaluate the detection ability of our queries. These TCP/IP stacks are uIP, Contiki, Contiki-ng, FNET, lwIP, Nucleus Net, Nutnet, NuttX, PicoTCP and PicoTCP-ng. In the following, we will briefly introduce these TCP/IP stacks.

1. **uIP, Contiki and Contiki-ng**: uIP (microIP) [30] was originally an open-source TCP/IP stack developed by Adam Dunkels, which was mainly intended to be used in 8-bit and 16-bit microcontrollers. We have discussed the design and streamlining of the TCP/IP protocol stack by developers in Chapter 2. Due to the small amount of code and RAM requirements, it is very suitable for use in embedded devices. Cisco, Atmel, and SICS have implemented ipv6 extension for it, and this new implementation is called uIPv6 [63]. Some of its subsequent versions, including uIPv6, are integrated in the operating system Contiki [34], and its later version, Contiki-ng (Contiki operating system for next-generation IoT devices). The versions used in the thesis are uIP v1.0, Contiki v3.0, and Contiki-ng v4.5. uIP implements four basic protocols, namely ARP, IP, ICMP, and TCP. The application layer protocol can be implemented on top of uIP. As for the DNS protocol, uIP also implements the DNS resolver function based on the UDP protocol.

2. **FNET**: FNET [64] is a free and open-source TCP/IP stack ordinally developed by Freescale, and currently maintained by Andrey Butok. The version we are going to study is FNET v4.6.3. FNET provides many basic services, including HTTP/HTTPS server, Telnet server, TFTP server/client, and DNS client (resolver) etc.

3. **lwIP**: lwIP [31] is also a lightweight open-source TCP/IP stack developed by A. Dunkels. After a large number of developers' improvements and bug fixes, lwIP has become a very popular TCP/IP stack in embedded devices. Like the uIP protocol stack, it implements a complete TCP/IP stack while reducing resource usage. There are a variety of services running on it, such as common HTTP services, iPerf server, MQTT

client, etc. The DNS resolve function we are concerned about is also implemented. The version we will use is lwIP v2.1.2.

4. **Nucleus Net**: Nucleus Net [65] is a TCP/IP stack used in Nucleus RTOS, and Nucleus RTOS is used in a variety of fields with high security requirements, such as industrial systems, medical equipment, airborne systems, etc. The version we will test is v4.3.

5. **Nutnet**: The Nutnet TCP/IP protocol stack in Nut/OS [66] is an open-source TCP/IP stack implementation developed by Ethernut. In addition to the basic protocols such as IP, TCP, and ARP under the application layer, it is implemented protocols include DHCP, DNS, and HTTP. The version we will use in this article is v5.1.

6. **NuttX**: Apache NuttX [67] is a powerful ROTS that can provide Internet connections. It was originally released by Gregory Nutt in 2007 and can be expanded from an 8-bit microcontroller environment to a 32-bit microcontroller environment. Some functions of uIP are referenced in the implementation of NuttX's TCP/IP stack, such as the state machine of uIP and other implementation techniques. It supports IP forwarding, user-defined sockets, and of course the DNS resolve function we will test. The version we will use is 9.1.1.

7. **PicoTCP and PicoTCP-ng**: PicoTCP [68] is a modular open-source TCP/IP stack that takes up few resources and is designed for embedded systems and the Internet of Things. It is developed by Altran Belgium and supports DHCP server and client as well as DNS client. PicoTCP-ng is a fork of the PicoTCP. The versions we will be testing are v1.7.0.

3.2 Exploitation of a heap overflow vulnerability

In this section, we will discuss the design of a case study for the heap overflow vulnerability exploitation and the related background knowledge. This vulnerability is a classic heap overflow vulnerability. The reason why we choose this vulnerability to exploit is that it is highly likely that we can perform remote code execution attacks by exploiting this vulnerability. The static analysis tool we will use in the detection research part can let us discover vulnerable code patterns in DNS client implementations. But the detection ability alone can’t show us the whole picture of a vulnerability. We also need to study the exploitation of the vulnerabilities to understand their potential impacts. The vulnerability we will be exploiting also exists in the DNS client of a widely used TCP/IP stack. And the ability to perform remote code execution

attacks can enable us to fully control the device and to explore the potential impact of the vulnerability.

3.2.1 The designed scenario of the case study

In the thesis, we selected a heap overflow vulnerability that exists in a widely used TCP/IP stack. The vulnerability was found by the Forescout research labs. We will use this vulnerability to develop a proof-of-concept exploit. Since at the time of this writing, the vulnerability is not yet publicly disclosed, we will not disclose the name of the stack, and we have anonymized the relevant technical information as much as possible.

We flashed a demo version of the stack into a microchip xplained pro MCU evaluation kit called ATSAM4E-xpro [69]. This kit contains an onboard microcontroller ATSAM4E16E (a microcontroller based on the high-performance 32-bit ARM Cortex-M4 RISC processor), two 512K (8-bit) SRAMs, and a 2 Gb (8-bit) NAND Flash. The demo version of the stack provides us with a web server, an FTP server, a Telnet server, a DHCP client/server, a TFTP client, and a DNS client. A Command-line interface (CLI) console is also available, it provides the basic commands and commands to interact with the device, to do networking configuration, and to interact with some of the protocols. In the web interface, we can also run all the commands provided by the CLI console using a web tool.

We designed a simulation environment of an industrial network, which contains our demo board, a DNS server, a programmable logic controller (PLC), a device controlled by the PLC, and an attacker's computer. In this designed simulation environment, our demo board is open to the public network. There is a DNS server in this public network, and the hacker’s computer is also located in this public network. To be specific, the hacker’s computer is between the DNS server and the intranet that contains the demo board. The attacker can sniff the packets between the DNS server and this intranet. In addition to our demo board in this intranet, there is also a vulnerable PLC and a device controlled by the logic in the PLC. The entire simulation environment is shown in the figure below.

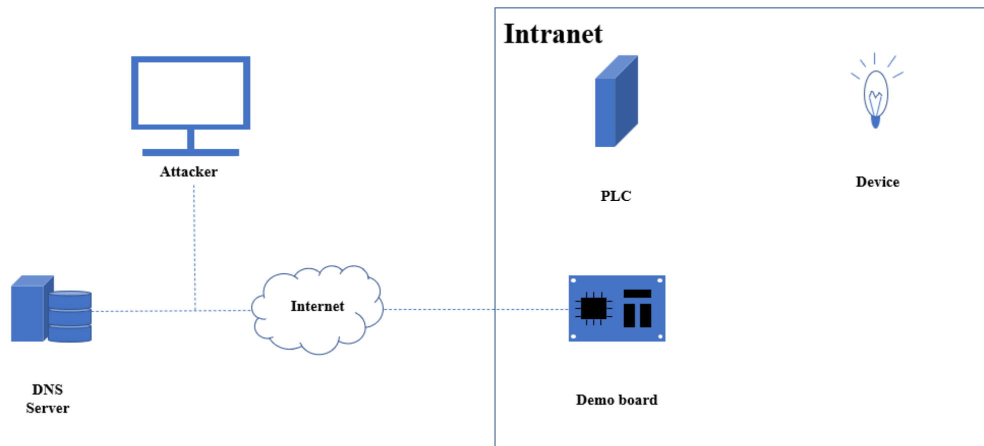

Figure 16. The designed environment

Since our demo board is open to this simulated public network, the attacker can interact with the any interface provided by this demo board. The attacker can access its web server, telnet server and FTP server. The web server allows users to access all content anonymously. So, for attackers, all content in the web server can be accessed without any permission. The FTP service on the other hand is protected by a password, and the username of the FTP service is also unknown to the attacker.

In this study case, we are a hypothetical attacker aiming to make the device controlled by the PLC in the intranet to behave abnormally. We need to conduct a DOS attack against the PLC. Since the PLC is in the intranet, we cannot directly interact with it. But the public accessible demo board come in handy. Once we take control over the demo board, we can have access to the PLC in the same intranet. The vulnerability we want to trigger on the PLC is at application layer, it is nature to attempt to take the advantage of the network connectivity of the TCP/IP stack. The goals for us attackers are as follows:

1. Infect the vulnerable demo board via a heap overflow vulnerability.
2. Conduct remote code execution attack against the vulnerability.
3. Establish a TCP connection between the vulnerable demo board and the vulnerable PLC service by structuring the shellcode to take advantage of the TCP/IP stack functionality of the vulnerable demo board.
4. Send the attack payload on the TCP connection to trigger the vulnerability in PLC and disable its control over the device in the intranet.

In our study case scenario, the ultimate goal is to disable the PLC inside the intranet. But due to the fact that we are leveraging the network connectivity of the TCP/IP stack. The scenario can also be generalized to infecting the demo board to be able to both attacks inward and outward. We can also add the demo board to be a part of a botnet to launch large scale distributed denial of service attacks. The designed attack scenario is as follows:

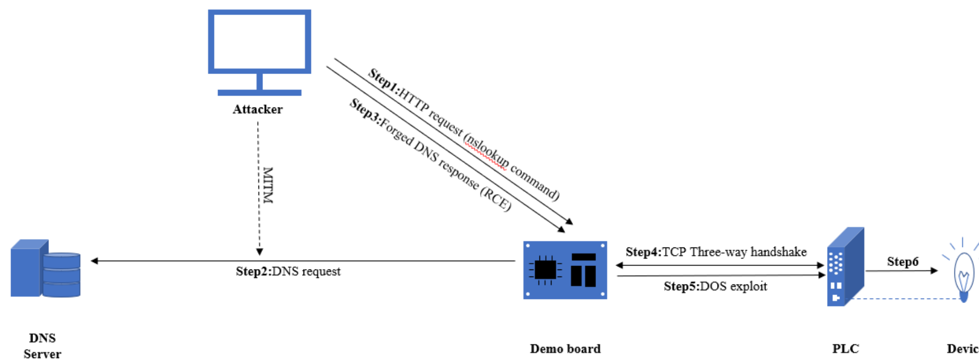

Figure 17. The attack scenario

We will first make the demo board send DNS query message to the DNS server by using the web interface in step 1 (shown in Figure 17). After the vulnerable demo board send a DNS query message to the DNS server, we can perform man in the middle attack and try to sniff the traffic between the demo board and the DNS server. The goal of this step is to capture the sent query message and obtain some important information from it, like the transaction id and the request source port. In step 3, we will send our forged DNS response to the demo board. The forged response packet contains a malicious shellcode. After successfully triggering the vulnerability and hijacking the control flow, we can make the vulnerable board establish a TCP connection with the PLC in step 4, and then we will make it to perform a DOS attack against the PLC in step 5. In step 6, the DOS attack will make the PLC crash, and the feedback loop between the PLC and the device it controls will be broken. The device will eventually go haywire.

3.2.2 The heap overflow vulnerability

Next, we will detail the heap overflow vulnerability in the DNS client of the stack, found by the Forescout research labs. As we mentioned before, the relevant technical information has been anonymized. The vulnerability is related to DNS response parsing function in the DNS client of the stack. The attacker can perform man in the middle attack against any DNS query

request sent by it. So, it is easy for the attacker to know the related information and forge the DNS response packet.

```
1  while (i < (int)(uint)records) {
2    pointer = getoffset(pointer,(char *)dnshdr,&offset);
3    pointer = getshort(pointer,&type);
4    pointer = getshort(pointer,&netclass);
5    if (netclass != 1) {
6      err = 6;
7    }
8    if ((err == 0) && ((int)(uint)queries <= i)) {
9      pointer = getlong(pointer,&ttl);
10     pointer = getshort(pointer,&rdlength);
11     switch(type) {
12     case 1:
13     case 0xc:
14       //...
15       if (i < (int)((uint)queries + (uint)answers)) {
16         if (nameoffset == 0) {
17           nameoffset = offset;
18           //...
19         }
20         vulnerable_call(dns_structure,type,(char *)pointer,(uint)
               rdlength);
```

Figure 18. Code snippet of the function dns_call

The vulnerability occurs during the parsing of a resource record (RR) of a DNS response packet. After receiving a DNS response packet, the program will call dns_call() function (shown in Figure 18). It has a for loop, and it iterates over the available DNS records and extract the corresponding fields. The for loop will also check the type of the record. And if the type of the record is a pointer record (PTR), which is represented by 0x0C, the program will call the vulnerable function vulnerable_call(). Function vulnerable_call() accept four arguments, namely dns_structure, type, pointer and rdlength. Type and rdlength are extracted from the packet in the for loop. Argument “type” is extracted from the type field of the DNS message, and it is the type of the record. Argument “rdlength” is extracted from the rdlength field, and it is a two-byte field that specifies the length of the response data. Argument “pointer” points to the current processing byte. Argument “dns_structure” is a pointer points to a structure (shown in the following figure).

```
1  typedef struct dns_structure{
2      //...
3      char vulnerable_value[128];
4      //...
5  }
```

Figure 19. The dns_structure with the vulnerable value

Inside the function vulnerable_call() (shown in Figure 20), the program will check the type of the record. If it is a PTR record(0x0C), the program will call function memcpy (Figure 20 line 15). This memcpy call will copy the size of “rdlength + 1” data from “pointer + 1” to the vulnerable_value member of the dns_structure. It is easy to be noticed that the rdlength is extracted directly from the DNS response packet and it is never checked along the way. Due to the fact that the rdlength value is controlled by attackers and there is no proper validation to this value, we can easily conclude that there is a buffer overflow vulnerability in this memcpy call.

```
1   switch(type) {
2   case 1:
3     //...
4   case 5:
5     break;
6   case 0xb:
7     if (dns_structure->ipaddrs < 5) {
8       iVar4 = dns_structure->ipaddrs;
9       dns_structure->ipaddrs = iVar4 + 1;
10      memcpy(dns_structure->ipaddr_list + iVar4,cp,4);
11    }
12    dns_structure->protocol = (uint)(byte)cp[4];
13    break;
14  case 0xc:
15    memcpy(dns_structure->vulnerable_value,cp + 1,rdlen - 1);
```

Figure 20. Code snippet of the function vulnerable_call

The memory of the dns_structure is allocated in the heap area, and the size of the member vulnerable_value is 128 bytes. If we specify a rdlength larger than 127 we can easily overflow this vulnerable_value member. This is a classic heap overflow vulnerability and there are no limitations for bad characters (Certain characters cannot be used in the overflowing data). We can easily achieve remote code execution based on this vulnerability. Because the exploitation of a heap overflow vulnerability is highly specific to the heap allocator the system use. Next, we will introduce the allocator used in the TCP/IP stack.

3.2.3 The memory allocator

The memory allocator used in the stack is similar to the memory allocator inside the newlib [70] library. In this version of the memory allocator implementation, there are two fundamental structures. These structures are essential for our exploitation case study.

The first structure is called malloc_chunk. It is the structure of the memory chunks, which are allocated or free area of memory. The structure is shown in the following figures (Figure 21

and Figure 22). A free chunk consists of four different fields. The first field is called prev_size, which indicates the size of the previous chunk of the current chunk. The next field is called size, which illustrate the size of the current chunk. After these two fields are two pointers, which are used for free chunks doubly linked to a list. An allocated chunk does not need these two fields. The first pointer is called fd, which is a forward pointer. This pointer points to the previous chunk that linked to the same list. The second pointer is called bk. It is similar to the fd pointer, but it points to the next chunk linked to the same list.

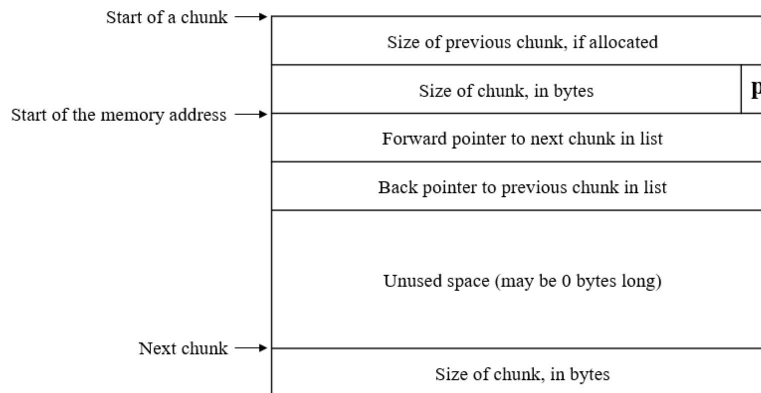

Figure 21. The structure of a free chunk

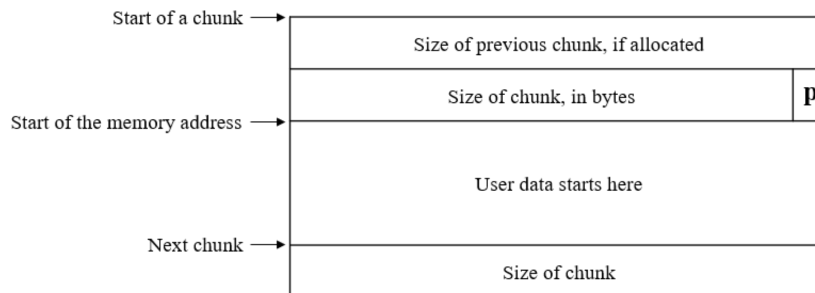

Figure 22. The structure of an allocated chunk

The second important structure is the bins, which are basically doubly linked list of free chunks. There are three different kinds of bins, normal bins, top bin and last_remainder bin. Normal bins are normal doubly linked lists, the indexes of these bins indicate the size of the free chunk inside these lists. Top bin is the top-most available chunk. After the initialization, it will always exist. Because the top bin is not a properly doubly linked list, it doesn’t have the fd and bk field. We will mainly leverage some functions related to top bin in our exploitation implementation. last_remainder is a bin that records the remainder of the most recently split chunk. It is mainly

used to record the bookkeeping information of the remainder to optimize the computing consumption of the allocator. It is also possible to leverage the normal bins and the last_remainder bin to hijack the control flow from a heap overflow vulnerability, but we will mainly focus on the top bin in the implementation chapter.

4 Implementation

4.1 Vulnerability detection using static analysis tool Joern

We discussed three vulnerability anti-patterns related to DNS name parsing functionality in embedded TCP/IP stacks earlier. In this chapter, we will transfer these anit-patterns into Joern queries and test these queries against several embedded TCP/IP stacks.

We can use Joern queries to describe a code pattern. Joern queries are actually graph traversals that we can execute on a code property graph. The code property graph consists of three parts, they are nodes and their types, labeled edges, and key-value pairs. Among them, nodes represent program constructs, including methods, variables, control structures, etc. And type field stands for which type a node belongs to. For example, there are “Literal” nodes (literal numbers), “Identifier” nodes (name used to identify a variable) and “Call” nodes (can be function calls, control structures, operations etc.). As for the labels, there can be multiple labeled edges between two nodes. These edges represent the relationship between the two nodes, and these relationships are directional. For example, we can create a “contain” edge between two nodes to indicate that node A contains node B, but this does not mean that node B contains node A because of the direction of the edge. Finally, each node contains many key-value pairs (attributes), and for different node types, there are different key-value pairs. Joern queries take advantage of the nodes, edges, and key-value pairs that exist in these code property graphs to form graph traversals.

To write a Joern query, we firstly need to represent a reference to a code property graph as our root node of a traversal. Starting from this root node, we need to define different steps to execute the traversal. These steps include node-type steps, filter steps, map steps, and repeat steps. Node-type steps can return all the nodes of the same type for us. We use filter steps to filter out the nodes we need from the nodes of the selected type. Combining with other complex steps, the traversal can provide us with more powerful query functions. In these steps, we can use different code property graph query language (CPGQL) directives. Property directives allow us to express the property of a node. Augmentation directives allow us to expand the current code property graph by adding new nodes, edges and properties. We can add an execution directive at the end of a traversal and indicate an output format, which allows Joern to execute the current traversal and return the results in the corresponding format. Joern's query language is based on

the Scala programming language, so we can write Scala scripts to implement more complex queries.

Our Joern queries mainly focus on matching three anti-patterns in a code base of a TCP/IP stack implementation, these anti-patterns are:

1. Anti-pattern 1: Wrong compression pointer checks.
2. Anti-pattern 2: Lack of compression pointer offset validation.
3. Anti-pattern 3: Lack of label length and domain name length validation.

For convenience, we will use anti-pattern 1-3 to refer to each of the above anti-patterns. It should be noted that in the implementation of the script, the output results are not strictly classified according to the anti-pattern types. For example, an output alert of the script could relate to either Anti-pattern 2 or Anti-pattern 3. To distinguish which type of anti-pattern an alert belongs to, further analysis by researchers is needed.

The script performs the following steps:

1. Locate the conditions of compression pointer checks.
2. Filter out incorrect compression pointer checks.
3. Locate compression pointer offset calculation.
4. Analysis the data flow.

4.1.1 Query to locate the conditions of compression pointer checks

Anti-patterns 1-3 are all related to DNS name parsing. An important goal of our Joern queries is to locate the relevant code area. If we want to locate the code area related to DNS name parsing, we need to locate the conditions of the DNS name compression pointer checks. If there are functions related to DNS name parsing in an TCP/IP stack, the implementation of the functions often requires a check for the compression pointer in the DNS message, regardless of whether the TCP/IP stack implementation chooses to parse the compression pointer or choose to skip it. If the implementation doesn't check whether a label in a DNS message is a normal label or a compression pointer, unpredictable errors will occur if the compression pointer is encountered in the process of parsing the DNS message. When the parsing process treats the compression pointer as a normal label, since the most significant two bits of the compression pointer are set to 1 and the normal label uses its first byte to represent the length of the actual label, the program will treat the first byte of the compression pointer as the label length. Thus,

the parsing process will become unpredictable. Compare with finding the relevant code of DNS name parsing in a codebase base on the file name, method name, variable name etc., it is simpler and more feasible to locate the condition for DNS name compression pointer check from the codebase. And because analysis of DNS name compression pointer offset, label length and name length are most likely near the code area of compression pointer check condition, as long as our query can locate the condition for DNS name compression pointer check, we can locate the relevant code areas of the anti-patterns we want to detect. The usual way to detect compression pointer is to perform logical AND to the currently parsed byte with 0xC0 and determine whether it is a compression pointer based on the result. If the result of the logical AND operation indicates the most significant bit of the byte is set to 1, the currently parsed byte will be considered to be a part of a compression pointer. This provides us with ideas for finding the condition of the DNS name compression pointer check. We can use the following natural language to describe a condition which is like the condition of DNS name compression pointer check.

“In a control structure type node of the code property graph, there is a logical AND operation in it, and an operand of this logical AND is literal 0xC0.”

The algorithm of this part of the query is as follows:

Algorithm 1: getCompressionChecks

Output: controlStructure

controlStructure = cpg.controlStructure.filter{contains ’&’ operation with operand 0xc0};
return controlStructure

Figure 23. Query algorithm for locating the compression pointer checks

It is easy for us to find all control structure nodes through the node-type step of Joern query. By analyzing the abstract syntax tree of each control structure nodes using Joern query, we can easily locate the conditions of compression pointer check. Obviously, the query here will produce false positive results because there may be other conditions that also meet the above description of the condition but are not used to detect the condition of the compression pointer check. For analyzing DNS name parsing functions of TCP/IP stacks, these false positives are tolerable because it is easy for the researchers to identify a false positive in the later analysis phase and further analysis phase is required to identify vulnerabilities base on the output of the query script.

4.1.2 Query to filter out incorrect compression pointer checks

After finding the condition of the compressed pointer check, we can check the conditions against the anti-pattern 1. The root cause of anti-pattern 1 is a mistake that is easy to make in programming. When we use the logical AND operation to check the highest two digits of a value, under normal circumstances we expect the result to be 0xC0. However, it is easy for the programmer to make the mistake that only check the final result is 1 instead of 0xC0. In this case, when the final result is 0x80 (the highest bit is set to 1) or the result is 0x40 (the second highest bit is set to 1), the program will also treat the currently parsed data as a compression pointer. But in fact, according to RFC 1035 [62], these two conditions are reserved for other uses. A large label length that meets these conditions (which is not a valid label length due to the size requirement of this field) should not be considered to be a compression pointer. In this way, we can use the following natural language to express the wrong compression pointer check:

“In the conditions obtained in the previous step, keep the conditions where the result of the logical AND operation is not 0xC0 (or 192), and raise alerts for them.”

What we excluded in this step is the correct compression pointer check implementation, the following C language snippet (Figure 24) is an example of a correct implementation and a wrong implementation:

```
1 #Correct implementation
2 if ((n & 0xc0) == 0xc0) {
3 ...
4 }
5
6 #Wrong implementation
7 if (n & 0xc0) {
8 ...
9 }
```

Figure 24. Code snippet of correct and incorrect implementations

The algorithm of this part of the query is as follows:

Algorithm 2: Query to filter out incorrect compression pointer checks

```
compressionChecks = getCompressionChecks();
foreach compressionChecks do
    if The result of the operation is not 0xC0 or 192 then
        raiseAlert(Check);
    end
end
```

Figure 25. Query algorithm for locating incorrect compression pointer checks

An example of the query script output for this part of the query is shown as follows:

```
--------------------------------------------------------------------------------
WARNING: BAD DNS COMPRESSION POINTER CHECK.
--------------------------------------------------------------------------------

The check shown below may be a DNS compression pointer check that does not
respect [RFC 1035]. The check must ensure that both of the 2 most significant
bits of the label length octet are set. This may be a sign of a "sloppy" domain
name parser implementation, therefore it must be examined carefully.

The location of the offending statement is shown below:

[/home/wenhui.wang/code-analysis/picotcp-historical/modules/pico_dns_common.c]
||
==> pico_dns_namelen_comp()
   ||
   ===> 63: if ((0xC0 & *label))
```

Figure 26. An example of the script output

4.1.3 Query to locate compression pointer offset calculation

After finding the condition of the compressed pointer check, we can also exam where the compression pointer offset is calculated. There are different ways to implement the compression pointer offset calculation, but the calculation steps are roughly the same. Firstly, we need to set the two most significant bits to 0. This can be done by performing logic AND operation to the first byte of the compression pointer with 0x3F. Then, we can put the result in the high byte of a two bytes value and add it with the second byte of the compression pointer. Take the following c code snippet (Figure 27) as an example, the computation process is shown in Figure 28.

```c
unsigned char n = *query++;
const uint16_t offset = query[0] + ((n & ~0xC0) << 8);
```

Figure 27. Code example of compression offset calculation

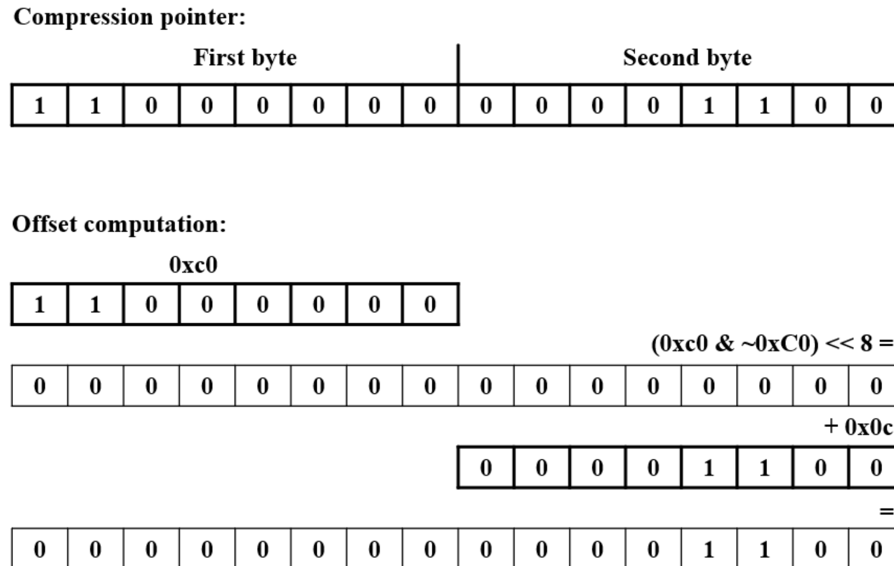

Figure 28. The process of compression offset calcultation

We first need to find all assignments in the true branch and else branch of the condition of the compressed pointer check. The code representation we used here is the control flow graph. From these assignments, we can analyze various operations and operands in an assignment through the abstract syntax tree. If the patterns of these operations and operands is similar to the compression pointer offset calculation, we can raise alert. Here we raise alerts for all offset calculations, and we will investigate whether this offset is validated through further manual analysis by researchers. To describe in natural language:

“Find all assignments dominated by a compression check, check the operations and operands in this assignment, if it meets the characteristics of the compression pointer offset calculation, raise alert.”

The algorithm of this part of the query is as follows:

Algorithm 3: Query to locate compression pointer offset calculation

```
compressionChecks = getCompressionChecks();
foreach compressionChecks do
    dominatedByCompCheck = compCheck.condition.dominates.assignments.toSet;
    offsetComputation = dominatedByCompCheck.filter{
        assignments.source contains compression pointer identifier &&
        assignments.source contains compression offset calculation patterns
    }.toset;
    raiseAlert(offsetComputation);
end
```

Figure 29. Query algorithm for locating offset calculation code patterns

An example of the query script output for this part of the query is shown as follows:

```
--------------------------------------------------------------------------------
WARNING: POTENTIAL DNS NAME COMPRESSION POINTER OFFSET COMPUTATION.
--------------------------------------------------------------------------------

The values of these offsets are often not checked within the code that parses
DNS domain names. Depending on various implementation specifics, this may lead
to the offset computation going out of bounds of the DNS packet. Therefore, each
implementation that deals with compressed DNS names must be examined carefully.


The location of the offending statement is shown below:

[/home/wenhui.wang/code-analysis/picotcp-historical/modules/pico_dns_common.c]
||
==> pico_dns_decompress_name()
   ||
   ===> 100: ptr = (uint16_t)((((uint16_t) *iterator) & 0x003F) << 8)
```

Figure 30. An example of the script output

4.1.4 Query to analysis the data flow

For anti-pattern 2 and anti-pattern 3, it is not enough to use only two code representations of the abstract syntax tree and control flow graph to perform static analysis. We also need to utilize the data flow technique.

The data analysis method we use is taint analysis. First, we need to determine user input or user-controllable variables, which may propagate to label length or domain name length variables and taint the variables. When the DNS name compression pointer resolution is implemented in the TCP/IP stack, we define that the user-controllable variable is the variable that stores the calculation result of the compression pointer offset. On the contrary, if the DNS name compression pointer resolution is not implemented in the TCP/IP stack, we define that the user-controllable variable is the variable that stores the domain name or label. These user-controllable variables can be easily obtained by analyzing the abstract syntax tree.

After this, we have two different analysis paths. In the first path, we use data flow analysis to determine whether tainted variables are passed into some dangerous functions. In the second path, we determine whether there are checks for tainted variables.

There are many dangerous functions in C language, usually user-controllable and unchecked parameters can enable attackers to use these dangerous functions to achieve unauthorized memory read and write operations. Take the memcpy function as an example. The function accepts three parameters, which are the target array pointer, the source array pointer, and the size. By specifying these three parameters, we can copy size characters from the memory area pointed to by the source array pointer to the memory area pointed to by the target array pointer. When the third parameter size is user-controllable, it is easy to cause buffer overflow vulnerabilities.

In the query, we first use the code property graph to query all the dangerous functions in the codebase and use the data analysis method to check whether the contaminated variable will affect the parameters in the dangerous function. And we also need to determine whether the program has checks on these tainted variables before the tainted variables affect these parameters. Use natural language to describe:

"Find out those tainted variables passed into dangerous functions, which have not been checked by the program. If there is such a tainted variable, raise an alert."

An example of the query script output for this part of the query is shown as follows:

```
--------------------------------------------------------------------------------
WARNING: UNCKECTED DNS NAME LABEL LENGTH USED IN A "MEMCPY()" CALL.
--------------------------------------------------------------------------------

A value derived from an unchecked DNS name length octet has been passed into a
"memcpy()" call. This may lead to Out-of-bounds Write issues, allowing to
achieve Denial-of-Service conditions or allowing for Remote Code Execution
attacks.

The location of the offending statement is shown below:

[/home/wenhui.wang/code-analysis/picotcp-historical/modules/pico_dns_common.c]
||
==> pico_dns_decompress_name()
   ||
   ===> 107: memcpy(dest_iterator + 1, iterator + 1, *iterator)
```

Figure 31. An example of the script output

In addition to being passed into some dangerous functions, there are other circumstances that tainted variables can eventually lead to denial-of-service attacks or remote code execution. For example, NULL pointer dereferences. We also need to perform additional checks on these contaminated variables:

"Raise alerts for tainted variables that have not been checked by the program."

An example of the output of our query script for this part of the query is shown below:

```
--------------------------------------------------------------------------------
WARNING: USE OF A VALUE DERIVED FROM AN UNCHECKED COMPRESSION POINTER OFFSET OR
A DOMAIN LABEL LENGTH OCTET.
--------------------------------------------------------------------------------

The statement below may use a value (e.g., through a pointer dereference or pass
into certain calls as an argument) derived from an unchecked compression pointer
offset or a domain label length octet. Depending on how it is used, it may cause
NULL pointer dereference issues, as well as out-of-bound reads and writes. This
may further lead to Denial-of-Service condtions and Remote Code Execution
attacks.

The location of the offending statement is shown below:

[/home/wenhui.wang/code-analysis/picotcp-historical/modules/pico_dns_common.c]
||
==> pico_dns_decompress_name()
   ||
   ===> 109: dest_iterator += (*iterator) + 1
```

Figure 32. An example of the script output

The algorithm of this part of the query is as follows:

```
Algorithm 4: Query to analysis the data flow
compressionChecks = getCompressionChecks();
foreach compressionChecks do
    dominatedByCompCheck = compCheck.condition.dominates.assignments.toSet;
    offsetComputation = dominatedByCompCheck.filter{
        assignments.source contains compression pointer identifier &&
        assignments.source contains compression offset calculation patterns
    }.toset;
                                                        ▷ taint analysis
    if offsetComputation.size larger than 0 then
        offset = offsetComputation.head;
        dominatedByCompCheck -= offset;
        taintedId = offset.ast.isIdentifier.head;
        assignments = dominatedByCompCheck.toSeq.sortBy(lineNumber);
        assignments.filter{lineNumber larger than offset's lineNumber};
    end
    else
        taintedId = compCheck.ast.isIdentifier.head;
        assignments = dominatedByCompCheck.toSeq.sortBy(lineNumber);
    end
    if taintedId and assignments is not empty then
                                            ▷ path 1:analyze dangerous calls
        memcpyCalls = compCheck.method.call.filter{dangerous calls};
        foreach memcpyCalls do
            sinkFlows = memcpyCall.argument(3).reachableByFlows(taintedId).toSet;
            isSinkReachable = !sinkFlows.isEmpty or (memcpyCall.argument(3) contains
             taintedId.code);
            argUnchecked = (memcpyCall.argument(3) not checked);
            if isSinkReachable && argUnchecked then
                raiseAlert(memcpyCall);
            end
        end
    end
                                ▷ path 2:analyze unchecked tainted variables
    while assignments.size != 0 do
        foreach assignments do
            if assignment.source contains taintedId.code then
                if taintedId.code is used in a pointer dereference, array access, an
                 argument of the strlen() or recursive call then
                    taintedId = assignment.target.ast.isIdentifier.head
                end
                else if taintedId.code is checked by the program then
                    raiseAlert(assignment);
                end
            end
        end
    end
end
```

Figure 33. Query algorithm for data flow analysis

4.2 Exploitation of a heap overflow vulnerability

In the design chapter, we introduced the root cause of a heap vulnerability we will trying to exploit. In conclusion, the cause of this vulnerability is that there is no verification of the rdlength in the DNS response packet in the program. The attacker can construct a malformed

response packet to overflow the vulnerable_value member in dns_structure which located in the heap memory area.

4.2.1 The heap overflow

We want to achieve remote cod execution attack against this vulnerability in order to fully control the demo board to attack other devices inside its intranet. Naturally, we will notice that in its allocator, the free chunks are doubly linked to lists. When we remove a node from a doubly linked list, we actually modify the pointers in the two adjacent nodes of this node. This process of removing a node from the doubly linked list is called unlink (Shown in Figure 34). The unlink macro used in the allocator does the following, whenever we want to unlink a free chunk P, the allocator will first store its bk pointer into the bk field of the node after it. It will then store its fd pointer into the fd field of the node before it. And after that, the node we want to unlink will not be able to be accessed by traversing the doubly linked list, which means that we successfully removed the node from the list.

```
#define unlink(P, BK, FD)
{
  BK = P->bk;
  FD = P->fd;
  FD->bk = BK;
  BK->fd = FD;
}
```

Figure 34. Code snippet of the function unlink

In the implementation of the allocator, we can easily find that there are many unlink operations. Whether in the process of allocation (malloc call) or in the process of deallocation (free call), we may encounter unlink operations. If we can control the fd and bk pointers of the unlinked node, we may be able to write arbitrary data to arbitrary address, thereby hijacking the control flow of the program.

The unlink operation that can be easily reached is the one inside the function free. During the deallocation, if there is an adjacent free chunk near to the currently deallocating chunk, the allocator will try to consolidate the chunk with its adjacent free chunk. And during the consolidation, the allocator will unlink the adjacent free chunk from its free list. So far, to achieve write arbitrary data to arbitrary address, we only need to meet the following conditions:

1. Find a path to a function free call.

2. Force the allocator performs consolidation.
3. Have control over the pointers in the adjacent free chunk.

For the first condition, we can meet this condition by leveraging the algorithm of allocating memory space from the top bin. If the allocator chooses to use this allocating algorithm, we can reach a free call under certain conditions. To be specific, if the size of the top bin is less than the currently allocating chunk size, the allocator will try to extend the top bin. And if the size of the top bin is not less than minimal possible chunk size, the allocator will try to perform the free function.

Once the allocator performs free function and try to free the top bin, it will check whether the last bit (a bit indicates the previous chunk is in use or not) of the size is 0. If the last bit of the size is 0, which means the previous chunk is a free chunk, the allocator will try to consolidate the currently deallocating chunk with this free chunk. But before that, the allocator will also try to locate that free chunk by using the prev_size field of the top bin as offset. Now, we are able to meet the other two conditions. If we can control the size field and the prev_size field of the top chunk, we can make the allocator unlink a chunk under our control. Then we can finally achieve write-what-where [71] and hijack the control flow of the program.

We now have a general idea of how to hijack the control flow in this allocator implementation by changing some of the fields in the top bin. Next, we will start from the allocation of the vulnerable dns_structure and present the whole hijacking process step by step.

The first thing we want to achieve is that we want the allocator to choose the branch of allocate memory from top bin instead of other bins during the allocation of the vulnerable dns_structure. In this way, after the allocation, the new top bin will appear after this structure. And it will let us to have the ability to change metadata of the top bin by overflowing a member of the structure. The allocation of a chunk is performed through calling the malloc() function, and the (simplified) memory allocation algorithm behind the scenes performs the following steps:

1. If there is a recent freed chunk of memory in any of the bins, and the size of it is enough for the requested chunk, the allocator will use this free chunk.
2. If there is no such chunk in the bins, the allocator will try to allocate memory from the top bin.
3. If the top bin is too small to accommodate the requested chunk, the allocator will extend the top bin.

4. Otherwise, malloc() fails and returns NULL.

If we want the allocator to allocate memory from the top bin, we need to make sure that there are no other qualified bins that can accommodate the allocation request. It can be easily achieved because after the initialization of the demo board, all the bins are empty. The allocator will start allocating memory from the top bin. Normally the board will keep at this state. If during the running there are some free chunks linked to the other bins, we can return to that state by forcing the system to restart. It can be easily achieved by the command tool in the web interface.

Now we have the system ready to receive malicious DNS response, but we need to make the vulnerable device send DNS query requests first. Otherwise, the malicious DNS response packet will be dropped by it because there is no request record with the corresponding transaction id. So, we will again leverage the command tool in the web interface to make the demo board send a DNS request to a DNS server. The command tool is not powerful enough for us to have full control over the device, but it is enough for us to achieve what we want. Now after the board sending the DNS request, we can send our malformed response back to it. And the vulnerable_value will be overflowed by the vulnerable memcpy() call.

Figure 35 shows the state of the memory before and after we overflow the dns_structure->vulnerable_value buffer. Because the allocator used the branch of allocating memory from the top bin for the allocation of dns_structure, dns_structure will be allocated right above the top bin and is adjacent to it. In our malformed packet, we set our own rdlength field to be able to not only overflow the vulnerable_value buffer and overwrite the other members, but we also overwrite the metadata of the top bin. We need to place a fake chunk to be unlinked before the metadata of the top bin. We can simply set the prev_size field of top bin to be 0x10 (16 bytes) so that the allocator can locate this fake chunk by this offset in the function free. The value of the size field depends on the request size of the next allocation which we will discuss later.

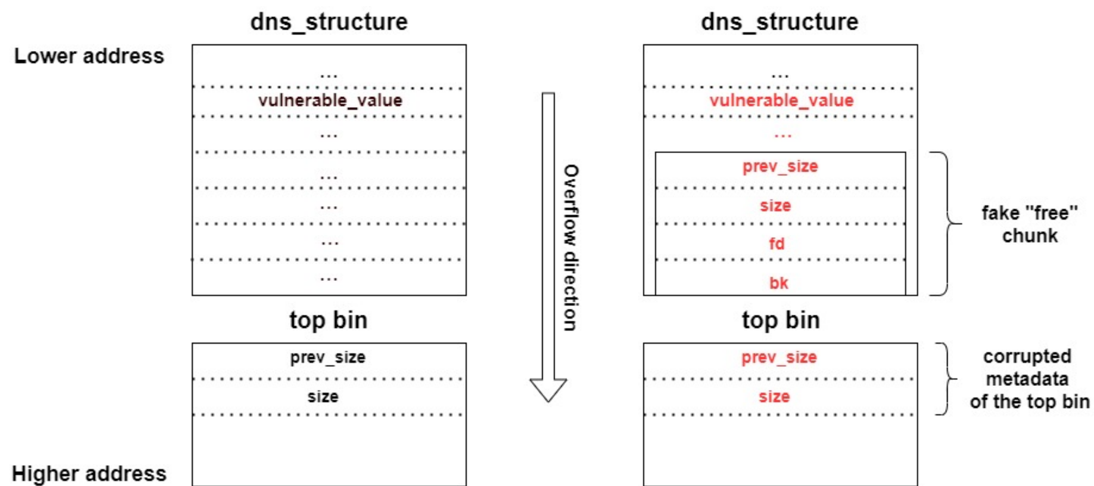

Figure 35. The overflow process

After the overflowing, since we have sent a malformed DNS response packet and the rdata field is completely different than what the board expect. Function nslookup will fail, and the board will try to return an error message back to us (Figure 36 line 4). We will encounter another allocation which is located in function alert_printf() (Figure 37 line 8).

```
1         task_lock(0x15,0x7fffffff);
2         ho = gethost(pointer,af);
3         if (ho == (hostent *)0x0) {
4            alert_printf(gio,"nslookup failed.\n");
5          }
```

Figure 36. Code snippet of the function nslookup

```
1     buf_size = 0x9c;
2     uStack8 = in_r2;
3     uStack4 = in_r3;
4     sVar1 = strlen(format);
5     if (0x9b < (int)sVar1) {
6       buf_size = sVar1 + 0x9c;
7     }
8     str = (char *)npalloc(buf_size);
9     if (str == (char *)0x0) {
10      ret_value = -0x14;
11    }
12    else {
13      vsprintf(str,format,(va_list)&uStack8);
14      sVar1 = strlen(str);
15      //...
16      gio->flags = uVar2;
17      }
18      npfree(str);
19    }
20    return ret_value;
21  }
```

Figure 37. Code snippet of the function alert_printf

This allocation will allocate memory space for the error message string buffer. Since the current state of the memory will always ensure that the allocator allocates memory from the top bin, the allocation will try to allocate memory from the already overflowed top bin. The buffer size we are about to allocate is 0x9C. In the design chapter we learned that if we want to force the allocate extends the top bin to reach the free() call, we need to set the size of the top bin to be less than the request size (masked by the malloc_align_mask) as well as bigger than the minimal possible chunk size (16 bytes). Also, we need to make sure that the last bit of the size is 0 to trigger the consolidation. Now, the allocator will deem that there is not enough space in the top bin, and it will try to extend it. During the extension, the memory allocator will free the old top bin. Because we set the last bit of the top bin size field to 0, which will let the allocator think that at the offset of prev_size before the top bin there is a free chunk, the allocator will also attempt to consolidate the top bin with the fake chunk we placed before the top bin (at offset of 16 bytes). And the allocator will perform unlink operation to the fake chunk.

If we assume the pointer of the fake chunk is P, since the board uses a 32-bit microcontroller, the unlink operation can be simplified as follows (In C programming language):

```
1 *(P->fd + 12) = P -> bk
2 *(P->bk + 8) = P -> fd
```

Figure 38. Simplified unlink operation

The metadata of the fake chunk is under our control, which means if we carefully choose the two pointers in the metadata, we can achieve write arbitrary data to arbitrary address. We can place the shellcode address of our first malicious instruction in the bk pointer or the fd pointer of the fake chunk metadata, and we can place the address to where we want to write our shellcode address in the other pointer. For example, if we store our shellcode address in the bk pointer of the fake chunk P, this shellcode address will be written to P->fd plus 12 bytes. So, we need to store the address we want to write to minus 12 into the fd pointer of the fake chunk P. One thing need to be mentioned here is that apart from writing shellcode address to our destination address, the unlink operation will also write an address back to the adjacent area of the shellcode address. So, we need to avoid writing our shellcode instructions in this part of the memory, because whatever instructions we place here by overflowing will be overwritten by the unlink operation.

Apart from the handy unlink operation, another thing makes the exploitation easy is that the allocator implementation used by the board doesn't provide us with sufficient security checks [72] unlike a newer version of glibc. [73] Next, we need to find a way to hijack the control flow using this unlink operation under our control. What naturally comes to our mind is hijacking the control flow when the program returns from the alert_printf() function to its callee. When the function is called by its callee, the return address of the function is stored in the stack. We can simply hijack the control flow by overwriting this return address stored in the stack. The when the program returns from the alert_printf() function, it will straightly goes to our shellcode instead of its callee.

So far, how to hijack the control follow is straightforward:

1. During the overflowing, we place our fake chunk in the memory, and let the value of its fd pointer to be the stack address (holds the return address of the alert_printf() call) minus 12 bytes.
2. Next, we place the address of the shellcode in the bk pointer of the fake chunk.
3. During the unlink operation, the control flow will jump to our shellcode.

4.2.2 The shellcode

After hijacking the control flow, now we need to program the assembly codes we want the device to execute and convert it to machine codes. In the design phase, we mentioned that the ultimate goal of the attacker is to infect the demo board, and from it we want to leverage the network capability of the TCP/IP stack to attack another vulnerable PLC inside the intranet. So, it is important to exam the available network APIs inside the stack. We have found several socket related APIs [74] inside the stack. The vulnerability of the PLC exists in its FTP server. This requires us to have the ability to control the application layer data. We need to first establish a TCP connection with the vulnerable PLC's FTP port (port 21), and from there we can send any data to the vulnerable FTP server over the TCP connection. Some of the APIs are useful for our goal, for example t_socket(), t_connect(), t_send(). The t_socket API allows us to create an endpoint for communication and it will return a descriptor. We can use this descriptor as an argument for other APIs. Using t_connect() API, we can establish a TCP connection with the PLC's FTP server. The t_send() API allows us to send any data over the connection we established by t_connect. We can structure our shellcode as follows (The highlighted section is the instructions of the shellcode):

Structure sockaddr_in
Network API function calls
Fix members
Jump to the network API function calls
Crafted heap metadata
TCP payload

Figure 39. The shellcode structure

In this structure, we placed the actual instructions in the section of “Jump to the network API function calls” and the section of “Network API function calls”. The reason why we need this separation of the instructions has already mentioned earlier. After the hijacking, we will land on the “Jump to the network API function calls” section. There is only one jump instruction in the section actually. Due to the fact that the unlink operation will overwrite some of the shellcode after this instruction, we need to avoid placing instructions after it. The simplest way to do so is to place a jump instruction here and start executing our actual shellcode somewhere else. So, the functional instructions are placed in the “Network API function calls” section. After jumping to the actual shellcode, we can use the t_socket API to create an endpoint for communication. API t_socket will return the descriptor to the r0 register. Before we hijack the control flow and execute our shellcode, the board was processing a DNS response packet, and the task was locked by it using. We need to unlock the task before we establish TCP connections. Then we need to use t_connect API to create a socket connection, where we will determine the IP address and the port we want to connect to as the arguments of the t_connect call. Now we can send data to this connection using t_send. After we have completed the attack on another device, we can use t_socketclose to close the connection on the socket, and then restart the device to restore the device to a normal operating state.

For some vulnerabilities, multiple packets might be needed to trigger them. But in order to reduce the number of the TCP packets and increase the efficiency of sending packets, an algorithm called Nagle's algorithm [75] is in place. Sticky packet [76] will happen when we send some small TCP packets. Nagle's algorithm will merge a number of small outgoing packets and send them at once. If the server side couldn’t handle the sticky packets properly, we won’t trigger the vulnerabilities. So, we need to disable Nagle's algorithm manually. The API we will use here is t_setsockopt by passing to it TCP_NODELAY to disable Nagle's algorithm. But this

step is not necessary for our scenario. Because in order to trigger the vulnerability that exists in the PLC, we only need to send one packet.

In the first section of the shellcode, there is a data structure called sockaddr_in, which will be passed to the t_connect() call as one of its arguments. Here we specify the address and the port we want to establish TCP connection to. The last section is also a data section. We place the data we want to send over the TCP connection here.

As we discussed earlier, in order to hijack the control flow, we need to overflow the metadata of the top bin. So, we need to calculate the offset of this part of the memory to the beginning of our shellcode, and we need to place our crafted metadata as well as the fake chunk here accordingly. Next, we will place a section of a byte sequence called “fix members” that ensures the overwritten members in the dns_structure will not cause crashes to the program.

Putting the exploitation all together we will perform the following steps:

1. Using command tool to reset the demo board to make sure to have the memory state we want.
2. Using command tool to force the board to perform the nslookup and send the DNS query request to the DNS server.
3. Perform the man in the middle attack and forge our malicious DNS response. Send it back to the board.
4. An overflow will happen, and the metadata of the top bin will be changed.
5. The allocator will try to allocate memory for a buffer from the top bin, and the top bin will be extended.
6. The allocator will try to free the old top bin.
7. The allocator will try to consolidate the old top bin with a fake chunk we placed in the memory by overflowing.
8. During the consolidation, the unlink operation will overwrite the return address of function alert_printf().
9. The program will jump to our shellcode landing address from the function alert_printf.
10. The program will jump to the actual malicious instructions.
11. Our shellcode will unlock the task lock and establish a TCP connection with the PLC using some APIs provided by the TCP/IP stack.

12. The shellcode will then send malicious data over this TCP connection and perform DOS attack to the PLC. Then PLC will be crashed, and it will lose control over the connected device.
13. Our shellcode will force the demo board to restart.

The packets we will send is as follows:

No.	Time	Source	Destination	Protocol	Length	Info
22	20.249813	192.168.100.3	192.168.100.1	DNS	81	Standard query 0x3412 PTR com.test.in-addr.arpa
24	22.331600	192.168.100.2	192.168.100.3	DNS	438	Standard query response 0x3412 PTR com.test.in-addr.arpa PTR <Root>
25	23.235449	192.168.100.3	192.168.100.1	TCP	60	1024 → 21 [SYN] Seq=0 Win=5840 Len=0 MSS=1460
26	23.235955	192.168.100.1	192.168.100.3	TCP	60	21 → 1024 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460
27	23.237418	192.168.100.3	192.168.100.1	TCP	60	1024 → 21 [ACK] Seq=1 Ack=1 Win=5840 Len=0
28	23.237418	192.168.100.3	192.168.100.1	FTP	186	Request:

Figure 40. Captured attack packets

Here, 192.168.100.1 is the IP address of the vulnerable PLC (also configured to be the DNS server IP address), IP address 192.168.100.3 represents the demo board, and 192.168.100.2 is the IP address of the attacker. NO. 24 packet indicate our forged DNS response. We can clearly see that the demo board established a TCP connection with the PLC and sent an FTP packet.

5 Results analysis

5.1 Vulnerability detection using static analysis tool Joern

5.1.1 Output analysis

After running the query script on the selected stacks, we got the following results:

Table 1. Results of the query script

Stack name	Anti-pattern 1	Anti-pattern 2	Anti-pattern 3	Total
uIP	1	0	2	3
Contiki	5	4	3	12
Contiki-ng	5	4	3	12
FNET	0	0	0	0
lwIP	0	1	0	1
Nucleus Net	0	2	1	3
Nut/Net	2	0	4	6
NuttX	2	0	0	2
PicoTCP	2	1	4	7
PicoTCP-ng	2	2	0	4
Total	19	14	17	50

The results show that in these TCP/IP stack implementations, our query script found 50 alerts in total. We divide the alerts of the script into three types, corresponding to three different anti-patterns as shown in the above table (Table 1). Since the output of our query script is not strictly classified according to different anti-patterns, after running the script to generate the outputs, manual analysis is needed to distinguish which anti-pattern each alert is about. The number of alerts related to the three anti-patterns is basically the same. Among the three types of anti-patterns, the number of alerts related to anti-pattern 1 is the largest, with a total of 19. But it contains many false positive results, we will explain in detail later. Among the selected TCP/IP stacks, Contiki and Contiki-ng generate the largest number of alerts, each with 12 alerts. After analyzing the source code of these two stacks, we found that the code patterns that triggers the alerts are basically the same, except that the file name of the code and the number of code lines are different. Since Contiki-ng is developed based on Contiki and Contiki is developed on top of uIP, a lot of code is reused. The same error occurs easily in the same code pattern. We use the following metrics to analyze the results.

- TP (True positive): The number of alerts that are related to the previously discovered vulnerable code patterns.
- FP (False positive): The number of alerts that are not related to any vulnerable code patterns.
- TN (True negative): The number of code patterns should not be reported and correctly missed by the query script.
- FN (False negative): The number of alerts that should have been reported but missed by the query script.
- Sensitivity, recall or true positive rate (TPR)
- Precision or positive predictive value (PPV)

We can use the following formulas to calculate TPR, PPV:

$$TPR = TP/(TP + FN) , \quad (1)$$

$$PPV = TP/(TP + FP) , \quad (2)$$

In these metrics, we cannot know the result of TN, because, for a codebase, there can be an infinite number of code patterns that are not related to any vulnerability. We have also simplified TP and FN accordingly because we cannot know all the vulnerable code patterns in a codebase. So, we limit the results to the previously discovered vulnerable code patterns.

In the selected stacks, there are 5 previously discovered vulnerabilities that related to the analyzed anti-patterns. The vulnerable stacks are Contiki, Contiki-ng, uIP, PicoTCP, PicoTCP-ng, Nut/Net and Nucleus Net. The following table (Table 2) shows the CVE id, affected stack name, and related anti-pattern information of these 5 vulnerabilities. In the table, we also show whether the anti-patterns related to these vulnerabilities are detected by our query script. We can see that all vulnerable code patterns that we discovered manually can be detected by our script.

Table 2. Manually detected vulnerabilities and the detection results for them

CVE ID	Affected stacks	Related Pattern	Detected
CVE-2020-24335	Contiki Contiki-ng uIP	Anti-pattern 2	YES
CVE-2020-24338	PicoTCP	Anti-pattern 2	YES

CVE ID	Affected stacks	Related Pattern	Detected
CVE-2020-24339	PicoTCP PicoTCP-ng	Anti-pattern 2	YES
CVE-2020-25110	Nut/Net	Anti-pattern 3	YES
CVE-2020-27738	Nucleus Net	Anti-pattern 2	YES

Combining the above information, we can get the metrics of the script:

Table 3. Detection metrics of the query script

metric	Anti-pattern 1	Anti-pattern 2	Anti-pattern 3	Total
TP	14	13	17	44
FP	5	1	0	6
TN	-	-	-	-
FN	0	0	0	0
TPR	100%	100%	100%	100%
PPV	73.7%	92.9%	100%	88%

Through the analysis of the above table (Table 3), we can see that for the previously discovered vulnerable code patterns, our script successfully detected them, and the number of FN is 0 for all anti-patterns. Therefore, only for the previously discovered vulnerabilities, the TPR of our script is 100%, which means that our script does not miss the detection of the code patterns of these vulnerabilities. For the detection precision, our query script has a high precision overall, which is 88%. Both anti-pattern 2 and anti-pattern 3 have high precision, and their positive predictive values are 92.9% and 100%. On the contrary, the detection precision of anti-pattern 1 is very low, and its positive predictive value is only 73.7%. This low precision of anti-pattern 1 is related to the implementation of query script. For anti-pattern 1, improving the precision of detection requires a more complex implementation to ensure that the identified code pattern belongs to the DNS name parsing function. The existing query script can only identify code patterns similar to the compression pointer check function, which is prone to false positives.

Using this detection method, we can provide researchers with an automatic way to detect vulnerable code patterns to add support to the vulnerability detection process. It can save time for researchers to locate vulnerable code patterns. Researchers also don't need to spend extra time on understanding the logic of the codes which are unrelated to the vulnerability code patterns. The process of discovering vulnerabilities will be more efficient. But the limit of this

method is obvious. The quality of the detection queries depends on their implementation. And related knowledge is required to write new detection queries or adjust the queries.

5.2 Exploitation of a heap overflow vulnerability

In the exploitation case study of the heap overflow vulnerability, we exploited the heap overflow vulnerability, and we successfully infected the vulnerable device. After the heap overflow, we successfully hijacked the control flow of the program, and we successfully triggered another vulnerability in another PLC device from this device using remote code execution. And finally, we conducted a successful denial-of-service attack on the PLC device. This also made the PLC lose control of the connected device.

5.2.1 Attack conditions

In the entire attack chain, in order to successfully exploit the vulnerability, we need to have the following conditions:

1. The attacker should be in a reasonable attack position, in order to be able to get the transaction id of the DNS query message and the source port number of the packet. In this example, we have another way to predict these two values. Another vulnerability in this TCP/IP stack provides us with this condition, because the stack did not reasonably randomize these two values. We can easily predict these two values.
2. We need to have the ability to make stack send a DNS query request. In this example, we are attacking the DNS client, so we need to meet this condition. The web interface in the stack provides us with a tool to run commands, but this tool only exists in the test version. It is possible that in reality the web server does not provide this command tool. In this case, we need to check other options that may enable stack to send DNS query requests, such as SSRF (Server-side request forgery) vulnerability.
3. The Heap overflow vulnerability should meet certain character conditions and size conditions. Some Heap overflow vulnerabilities may have character and length restrictions, which will bring difficulties to our exploitation, and even make remote code execution impossible. We need to be able to use more characters and a larger overflow size in order to successfully conduct remote code execution.
4. There are no or fewer protection measures against heap overflow vulnerability. In this case study, the stack and the device we tested doesn’t have protective measures against heap overflow vulnerability, which is very common in embedded devices. When we encounter

systems and devices that have protective measures, we need the ability to bypass these restrictions.

5. Finally, we need to have sufficient knowledge of the program and the allocator used by the system. Because the exploitation of heap overflow vulnerability is program and allocator specific. Successful exploitation is inseparable from an in-depth understanding of programs and memory allocation mechanisms.

5.2.2 Potential impacts

These conditions may make exploitation difficult, but it is also very flexible for the attacker to meet these conditions, and once the vulnerability is successfully exploited, the attacker can have a great impact on it.

In this case study, we successfully used remote code execution to attack another PLC device on the same intranet as the vulnerable device. This happens very often in reality. The attackers gain access to the network where the device exists by attacking publicly accessible devices. This allows attackers to attack other devices that are not publicly accessible. The network API provided by the embedded TCP/IP stack in this case study also makes this kind of attack easier. From another perspective, we can also use this network function to attack publicly accessible devices. Attackers can use fully controlled devices to launch attacks on other information systems for anonymous purposes. Many large-scale DDOS attacks are also launched by attackers after adding a large number of controlled devices to a botnet. Since embedded devices are often closely related to our real lives, attacks on them may cause serious damage in the physical world. In this case study, we demonstrated this threat by simulating an attack on a PLC in the intranet. We successfully executed a denial-of-service attack on this PLC. In reality, many PLCs are used in the field of industrial production. These PLCs often control some large mechanical equipment or some dangerous equipment. It is conceivable that attacks on these PLCs can cause great damage.

6 Conclusion

In the thesis, vulnerability detection method and vulnerability exploitation case study in embedded TCP/IP stacks were developed and examined. A vulnerability detection query was designed based on a static analysis platform called Joern. The query is meant to discover code patterns that are similar to three previously discovered anti-patterns in embedded TCP/IP stacks. These patterns are closely related to the DNS name parsing functionality of the embedded TCP/IP stacks. The performance of the designed query was analyzed based on several manually discovered vulnerabilities related to these anti-patterns. A case study of the exploitation of a heap overflow vulnerability was analyzed. This heap overflow vulnerability exists in a well-known embedded TCP/IP stack, and it is also related to DNS name parsing. A real-life like scenario, where the ultimate goal of the attacker is to exploit the heap overflow vulnerability in order to trigger another vulnerability that exists in another PLC device, was designed for the exploitation case study.

We tested the developed detection query script against 10 embedded TCP/IP stack implementations. Among these stacks, exist the stacks that have previously discovered vulnerabilities. The query script raised 50 alters in total for these TCP/IP stacks. Among these alerts, there are alerts for code patterns related to previously discovered vulnerabilities, confirming the successful detection capability of the query. The detection precision of the query script is high. For different anti-pattens, the query script shows different levels of precision. The script has a relatively low detection precision only for anti-pattern 1 (Wrong compression pointer checks), where it outputs many false positive results. The reason for this is related to the implementation of the query. The number of false positives can be reduced by implementing more complex queries. The powerful static analysis techniques and program semantics provided by the Joern platform facilitates the design of more robust vulnerability detection capabilities. Therefore, by designing appropriate queries in Joern vulnerabilities in embedded systems codes can be detected with high precision.

The case study of the heap overflow vulnerability exploitation demonstrated that though successful exploitation may need to meet many conditions and the attacker needs to make a lot of efforts, a vulnerability that can be exploited to execute remote code can still cause great damages. The networking capabilities provided by embedded TCP/IP stacks provide attackers with the ability to use vulnerable devices to attack other devices in the local network and on the Internet. The close connection between embedded devices and the physical world has also

increased the impacts of the attacks. For attackers, their ability is no longer limited to acquiring sensitive information, implementing denial-of-service attacks, and other attacks that exist in the cyber world. The attacker has the ability to affect the physical world which could cause greater damage to assets and disrupt the normal daily life. Therefore, vulnerability detection methods and understanding the consequences of exploitations are necessary for improving the security of embedded devices and formulating risk mitigation methods.

In the thesis, we provide high-precision detection scripts for several anti-patterns in the embedded TCP/IP stack implementations. In the future, this research can be extended to the detection of other anti-patterns or to programs that are not embedded TCP/IP stack implementations. Furthermore, additional queries can be written for detecting other vulnerable code patterns, such as dangerous function calls that are user-controllable in the program.

References

[1] OBERLO, "How many people have smartphones in 2021?", OBERLO statistics [Online], 2021. Available: https://www.oberlo.com/statistics/how-many-people-have-smartphones/. [Accessed: May 16, 2021].

[2] K. L. Lueth, "State of the IoT 2020: 12 billion IoT connections, surpassing non-IoT for the first time", IoT analytics [Online], November 19, 2020. Available: https://iot-analytics.com/state-of-the-iot-2020-12-billion-iot-connections-surpassing-non-iot-for-the-first-time/ [Accessed: May 16, 2021].

[3] GlobeNewswire, "Global iot market to be worth usd 1,463.19 billion by 2027 at 24.9% cagr; demand for real-time insights to spur growth, says fortune business insights", GlobeNewswire [Online], April 08, 2021. Available: https://www.globenewswire.com/en/news-release/2021/04/08/2206579/0/en/Global-IoT-Market-to-be-Worth-USD-1-463-19-Billion-by-2027-at-24-9-CAGR-Demand-for-Real-time-Insights-to-Spur-Growth-says-Fortune-Business-Insights.html. [Accessed: May 16, 2021].

[4] J. Firch, "9 common types of malware (and how to prevent them)", PurpleSec [Online], Mar 24, 2019. Available: https://purplesec.us/common-malware-types/. [Accessed: May 16, 2021].

[5] C. Crane, "42 cyber attack statistics by year: a look at the last decade", Sectigostore [Online], February 21, 2020. Available: https://sectigostore.com/blog/42-cyber-attack-statistics-by-year-a-look-at-the-last-decade/. [Accessed: May 16, 2021].

[6] O. Kupreev, A. Gutnikov and E. Badovskaya, "DDoS attacks in Q3 2020", Securelist [Online], October 28, 2020. Available: https://securelist.com/ddos-attacks-in-q3-2020/99171/. [Accessed: May 16, 2021].

[7] A. Zaharia, "300+ terrifying cybercrime and cybersecurity statistics & trends (2021 edition)", Comparitech [Online], June 13, 2021. Available: https://www.comparitech.com/vpn/cybersecurity-cyber-crime-statistics-facts-trends/. [Accessed: June 15, 2021].

[8] S. Calif, "Cybercrime to cost the world $10.5 trillion annually by 2025", Cybersecurity Ventures [Online], November 13, 2020. Available: https://cybersecurityventures.com/hackerpocalypse-cybercrime-report-2016/. [Accessed: May 19, 2021].

[9] C. Townsend, "A brief and incomplete history of cybersecurity", Cybersecurity Magazine [Online]. Available: https://www.uscybersecurity.net/history/. [Accessed: May 19, 2021].

[10] Peerbits, "Internet of things in healthcare: applications, benefits, and challenges", Peerbits [Online]. Available: https://www.peerbits.com/blog/internet-of-things-healthcare-applications-benefits-and-challenges.html. [Accessed: May 19, 2021].

[11] T. Desk, "7 amazing IoT (internet of things) trends to keep your eye on in 2020", The Indian Express [Online], October 21, 2020. Available: https://indianexpress.com/article/technology/gadgets/apple-watch-saves-an-indian-life-thanks-to-its-ecg-feature-report-6808698/. [Accessed: May 19, 2021].

[12] A. Laura, "Apple watch series 5's ECG feature saves life of a 61-year old indian man", Elearning Industry [Online], April 16, 2020. Available: https://elearningindustry.com/iot-internet-of-things-trends-2020. [Accessed: May 19, 2021].

[13] U. Lindqvist and P. G. Neumann, "The future of the internet of things," Communications of the ACM, vol. 60, 2, pp.26-30, January 2017.

[14] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas, "DDoS in the IoT: Mirai and Other Botnets," Computer, vol. 50, 7, pp.80-84, 2017.

[15] J. Fruhlinger, "What is Stuxnet, who created it and how does it work?", Csoonline [Online], August 22, 2017. Available: https://www.csoonline.com/article/3218104/what-is-stuxnet-who-created-it-and-how-does-it-work.html. [Accessed: May 22, 2021].

[16] J. Glaser, "Cyberwar on Iran won't work. Here's why", Defense One [Online], August 21, 2017. Available: https://www.cato.org/commentary/cyberwar-iran-wont-work-heres-why. [Accessed: May 22, 2021].

[17] M. Bishop, D. Bailey, "A critical analysis of vulnerability taxonomies.", Department of Computer Science., University of California at Davis., California, CSE-96-11,1996.

[18] W. S. McPhee, "Operating system integrity in OS/VS2," IBM Systems Journal, vol. 13, no. 3, pp. 230-252, 1974.

[19] CVE. Common vulnerabilities and exposures [Online]. Available: http://cve.mitre.org/. [Accessed: May 20, 2021].

[20] CWE. Common Weakness Enumeration [Online]. Available: https://cwe.mitre.org/index.html. [Accessed: May 20, 2021].

[21] Kaspersky. "What is WannaCry ransomware?", Kaspersky [Online]. Available: https://www.kaspersky.com/resource-center/threats/ransomware-wannacry. [Accessed: May 20, 2021].

[22] Y. Li, D. Li, W. Cui and R. Zhang, "Research based on OSI model," 2011 IEEE 3rd International Conference on Communication Software and Networks, 2011, pp. 554-557.

[23] J. Bilek and I. P. Ruzicka, "Evolutionary trends of embedded systems," IEEE International Conference on Industrial Technology, 2003, pp. 901-905 Vol.2.

[24] A. Dunkels, "Full TCP/IP for 8-bit architectures," Swedish Institute of Computer Science [Online], March 3, 2003. Available: https://www.usenix.org/legacy/publications/library/proceedings/mobisys03/tech/full_papers/dunkels/dunkels_html/mobisys.html. [Accessed: May 23, 2021].

[25] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, "The Design and Implementation of the 4.4 BSD Operating System, 2nd ed. University of Michigan: Addison-Wesley, 1996.

[26] H. RiLi, "Research and application of TCP/IP protocol in embedded system," 2011 IEEE 3rd International Conference on Communication Software and Networks, 2011, pp. 584-587.

[27] T. Lin, H. Zhao, J. Wang, G. Han and J. Wang, "An embedded Web server for equipment," 7th International Symposium on Parallel Architectures, Algorithms and Networks, 2004. Proceedings., 2004, pp. 345-350.

[28] V. Paxson, ICSI/UC Berkeley, M. Allman, J. Chu et al. (June 2011). Computing TCP's Retransmission Timer [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6298. [Accessed: May 22, 2021].

[29] Microchip. Microchip's TCP/IP Stacks [Online]. Available: https://www.microchip.com/SWLibraryWeb/product.aspx?product=TCPIPSTACK. [Accessed: May 22, 2021].

[30] A. Dunkels, "uIP-A free small TCP/IP stack," Dunkels website [Online], January 15, 2002. Available: http://www.dunkels.com/adam/download/uip-doc-0.6.pdf. [Accessed: May 22, 2021].

[31] A. Dunkels, "Design and Implementation of the lwIP TCP/IP Stack," Swedish Institute of Computer Science [Online], February 20, 2001. Available: https://www.ece.ualberta.ca/~cmpe401/fall2004/labs/docs/lwip.pdf [Accessed: May 23, 2021].

[32] E. Engelke. Networking [Online]. Available: http://www.erickengelke.com/history.html. [Accessed: June 3, 2021].

[33] G. Vinum. Watt-32 tcp/ip Homepage [Online]. Available: https://www.watt-32.net/. [Accessed: June 3, 2021].

[34] Contiki-os. Contiki-os Homepage [Online]. Available: http://www.contiki-os.org/. [Accessed: June 3, 2021].

[35] Armis research team, "URGENT/11 5 zero day vulnerabilities impacting tens of millions of devices.", Armis Labs [Online], December 15, 2020. Available: https://www.armis.com/research/urgent11/. [Accessed: June 5, 2021].

[36] Forescout Research Labs, "amnesia:33 How TCP/IP stacks breed critical vulnerabilities in IoT, OT and IT Devices.", Forescout Research Labs [Online]. Available: https://www.forescout.com/company/resources/amnesia33-how-tcp-ip-stacks-breed-critical-vulnerabilities-in-iot-ot-and-it-devices/. [Accessed: June 5, 2021].

[37] Forescout Research Labs, "number:jack Weak ISN generation in embedded TCP/IP stacks.", Forescout Research Labs [Online]. Available: https://www.forescout.com/company/resources/numberjack-weak-isn-generation-in-embedded-tcpip-stacks/. [Accessed: June 5, 2021].

[38] Forescout Research Labs, "name:wreck Breaking and fixing DNS implementations.", Forescout Research Labs [Online]. Available: https://www.forescout.com/company/resources/namewreck-breaking-and-fixing-dns-implementations/. [Accessed: June 5, 2021].

[39] B. Chess and G. McGraw, "Static analysis for security," in IEEE Security & Privacy, vol. 2, no. 6, pp. 76-79, Nov.-Dec. 2004.

[40] Man7. grep - Linux manual page [Online]. Available: https://man7.org/linux/man-pages/man1/grep.1.html. [Accessed: June 3, 2021].

[41] Owasp. (12 February 2016). OWASP ASIDE Project [Online]. Available: https://wiki.owasp.org/index.php/OWASP_ASIDE_Project#OWASP_ASIDE.2FESIDE. [Accessed: June 3, 2021].

[42] Eclipse. ECLIPSE IDE [Online]. Available: https://www.eclipse.org/eclipseide/. [Accessed: June 3, 2021].

[43] Fortify. Rough auditing tool for security [Online]. Available: https://code.google.com/archive/p/rough-auditing-tool-for-security/. [Accessed: June 3, 2021].

[44] J. Viega, J. T. Bloch, Y. Kohno and G. McGraw, "ITS4: a static vulnerability scanner for C and C++ code," Proceedings 16th Annual Computer Security Applications Conference, 2000, pp. 257-267.

[45] N. Jovanovic, C. Kruegel and E. Kirda, "Pixy: a static analysis tool for detecting Web application vulnerabilities," 2006 IEEE Symposium on Security and Privacy (S&P'06), 2006, pp. 6 pp.-263.

[46] J. Kronjee, A. Hommersom, H. Vranken, "Discovering software vulnerabilities using data-flow analysis and machine learning," In Proceedings of the 13th International Conference on Availability, Reliability and Security, 2018, pp. 1–10.

[47] H. Kim, T. Choi, S. Jung, H. Kim, O. Lee and K. Doh, "Applying Dataflow Analysis to Detecting Software Vulnerability," 2008 10th International Conference on Advanced Communication Technology, 2008, pp. 255-258.

[48] F. Yamaguchi, M. Lottmann, and K. Rieck, "Generalized vulnerability extrapolation using abstract syntax trees," In Proceedings of the 28th Annual Computer Security Applications Conference, 2012, pp. 359–368.

[49] H. Feng, X. Fu, H. Sun, H. Wang and Y. Zhang, "Efficient Vulnerability Detection based on abstract syntax tree and Deep Learning," IEEE INFOCOM 2020 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), 2020, pp. 722-727.

[50] S. Sparks, S. Embleton, R. Cunningham and C. Zou, "Automated Vulnerability Analysis: Leveraging Control Flow for Evolutionary Input Crafting," Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007), 2007, pp. 477-486.

[51] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems, vol. 9, 3, pp. 319-349, July 1987.

[52] F. Yamaguchi, N. Golde, D. Arp and K. Rieck, "Modeling and Discovering Vulnerabilities with Code Property Graphs," 2014 IEEE Symposium on Security and Privacy, 2014, pp. 590-604.

[53] Joern, Home page of a static analysis tool Joern [Online]. Available: https://joern.io/ [Accessed: June 4, 2021].

[54] H. Holm, T. Sommestad, U. Franke and M. Ekstedt, "Success Rate of Remote Code Execution Attacks Expert Assessments and Observations," Journal of Universal Computer Science, vol. 18, pp. 732-749, January 2012.

[55] G. Johnson, "Remote and Local File Inclusion Explained", Hakin9 [Online], January, 2008.Available: http://repository.root-me.org/Exploitation%20-%20Web/EN%20-%20Remote%20File%20Inclusion%20and%20Local%20File%20Inclusion%20explained.pdf. [Accessed: June 5, 2021].

[56] W. G. J. Halfond, J. Viegas, "A Classification of SQL Injection Attacks and Countermeasures," 2006.

[57] N. Koutroumpouchos, G. Lavdanis, E. Veroni, C. Ntantogian, and C. Xenakis, "ObjectMap: detecting insecure object deserialization," In Proceedings of the 23rd Pan-Hellenic Conference on Informatics, 2019, pp. 67–72.

[58] O. Alhazmi,S. Woo, Y. Malaiya, "Security vulnerability categories in major software systems," Proceedings of the Third IASTED International Conference on Communication, Network, and Information Security, 2006, pp. 138-143.

[59] S. Shea, "Address space layout randomization (ASLR)," Searchsecurity [Online], June 2014. Available: https://searchsecurity.techtarget.com/definition/address-space-layout-randomization-ASLR. [Accessed: June 4, 2021].

[60] C. Wang, "The art of exploiting heap overflow," Medium [Online], August 21, 2017. Available: https://searchsecurity.techtarget.com/definition/address-space-layout-randomization-ASLR. [Accessed: June 4, 2021].

[61] D Kapil, "Heap Exploitation", Heap-exploitation [Online], Available: https://heap-exploitation.dhavalkapil.com/attacks. [Accessed: June 4, 2021]

[62] P. Mockapetris et al. (November 1987). Domain names - implementation and specification [Online]. Available: https://datatracker.ietf.org/doc/html/rfc1035. [Accessed: May 22, 2021].

[63] Cisco, "Cisco, Atmel and the Swedish Institute of Computer Science (SICS) Collaborate to Support a Future Where Any Device Can Be Connected to the Internet", Cisco [Online], October 15, 2008. Available: https://newsroom.cisco.com/press-release-content?type=webcontent&articleId=4568692. [Accessed: June 6, 2021].

[64] A. Butok. FNET - Embedded TCP/IP stack [Online]. Available: https://fnet.sourceforge.io/. [Accessed: June 7, 2021].

[65] Siemens. Nucleus RTOS: the real-time operating system for today's advanced designs [Online]. Available: https://www.plm.automation.siemens.com/global/en/products/embedded/nucleus-rtos.html. [Accessed: June 7, 2021].

[66] Ethernut Software. Nut/OS [Online]. Available: http://www.ethernut.de/en/software/. [Accessed: June 7, 2021].

[67] Apache. NuttX Real-Time Operating System [Online]. Available: https://cwiki.apache.org/confluence/display/NUTTX/NuttX. [Accessed: June 7, 2021].

[68] Altran. picoTCP [Online]. Available: http://picotcp.altran.be/. [Accessed: June 7, 2021].

[69] Microchip. SAM4E Xplained Pro Evaluation Kit [Online]. Available: https://www.microchip.com/DevelopmentTools/ProductDetails/PartNO/ATSAM4E-XPRO. [Accessed: June 8, 2021].

[70] Googlesource. Nacl-newlib [Online]. Available: https://chromium.googlesource.com/native_client/nacl-newlib/+/master/newlib/libc/stdlib/mallocr.c. [Accessed: June 8, 2021].

[71] CWE. CWE-123: Write-what-where Condition [Online]. Available: https://cwe.mitre.org/data/definitions/123.html. [Accessed: June 8, 2021].

[72] E. Itkin, "Safe-Linking – Eliminating a 20 year-old malloc() exploit primitive," Checkpoint [Online], May 21, 2020. Available: https://research.checkpoint.com/2020/safe-linking-eliminating-a-20-year-old-malloc-exploit-primitive/. [Accessed: June 6, 2021].

[73] The GNU C Library (glibc) [Online]. Available: https://www.gnu.org/software/libc/. [Accessed: June 8, 2021].

[74] ARM. Porting TCP/IP Programmer's Guide - Socket API reference [Online]. Available: https://developer.arm.com/documentation/dui0144/b/Sockets/Socket-API-reference/. [Accessed: June 8, 2021].

[75] K. Datar, "Best practices for TCP optimization in 2019: How to use TCP performance and Nagle's algorithm to get better user experience and performance out of your network," extrahop [Online], June 29, 2019. Available: https://www.extrahop.com/company/blog/2016/tcp-nodelay-nagle-quickack-best-practices/. [Accessed: June 6, 2021].

[76] Alibabacloud, "TCP Sticky Packets," Alibabacloud [Online], June 25, 2016. Available: https://topic.alibabacloud.com/a/tcp-sticky-packets_8_8_31201106.html. [Accessed: June 6, 2021].