FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Analysis of the Ripple20 Attacks
Student:	Jan Dvořák
Supervisor:	Ing. Josef Kokeš
Study program:	Informatics
Branch / specialization:	Computer Security and Information technology
Department:	Department of Computer Systems
Validity:	until the end of summer semester 2021/2022

Instructions

1) Research the security challenges of the Internet of Things, describe some recent major attacks and their countermeasures.
2) Study the Ripple20 family of attacks. Describe their principles, prerequisites, consequences.
3) Choose one of the Ripple20 attacks with the goal of replicating it on your hardware. Describe the attack. (Recommendation: CVE-2020-11898).
4) Attempt to replicate the attack on your hardware using your own implementation.

5) Discuss your results. If the attack succeeded, propose mitigation strategies for the user, otherwise explain the reasons for that.

Electronically approved by prof. Ing. Pavel Tvrdík, CSc. on 21 January 2021 in Prague.

Bachelor's Thesis

ANALYSIS OF THE RIPPLE20 ATTACKS

Jan Dvořák

Faculty of Information Technology CTU in Prague
Department of Computer Systems
Supervisor: Ing. Josef Kokeš
May 13, 2021

Contents

List of Figures

First and foremost, I would like to express my sincere gratitude to my supervisor, Ing. Josef Kokeš, for his invaluable advice, help and suggestions. I would also like to thank my family for having so much patience with me and for supporting me throughout my studies.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague, on May 13, 2021 .....

Abstract

In this study, we explore the security of the Internet of Things (IoT). Specifically, we focus on Ripple20, a family of 19 vulnerabilities discovered in 2020 in the TCP/IP stack by Treck, Inc. We try to reproduce at least one of the attacks on the Ripple20 vulnerabilities.

First, we present an overview of the Internet of Things, the related cybersecurity issues and a brief history of exploiting the IoT devices. We select the device for our experiments and two specific vulnerabilities to be researched. The determination whether or not the tested vulnerabilities are present in our device can be seen as another objective of this study.

We follow by performing the attacks. In one case, the device apparently demonstrated incorrect behaviour (a denial of service). Even though this response differs from the published studies (information leak from the heap), we can find indirect evidence that the vulnerability is present in the tested system. In the second case, our device successfully resisted the attacks. We conclude that the second researched vulnerability is not present in the device's implementation of the TCP/IP stack.

We succeeded in detecting one of the vulnerabilities in our IoT device and in performing the respective attack. On the contrary, the presence of the second vulnerability can be ruled out. After concluding that the tested IoT device is vulnerable, we suggest the most important risk mitigation strategies.

Keywords cybersecurity, Internet of Things, Ripple20, TCP/IP stack, Treck, Inc., IP in IP tunnelling, DNS response parsing

Abstrakt

V této bakalářské práci se zabývám bezpečností internetu věcí (IoT), konkrétně pak analýzou útoků Ripple20. Tento název označuje skupinu 19 zranitelností odhalených v roce 2020 v TCP/IP subsystému společnosti Treck, Inc. V práci si kladu za cíl alespoň jeden z těchto útoků reprodukovat.

Nejprve se zaměřím na internet věcí a kybernetickou bezpečnost a stručně zmíním s ním související útoky z minulosti. Představím též zařízení, se kterým budu experimentovat, a dvě zranitelnosti, jež budu zkoumat. Určení, zda mnou zkoumané zařízení dané chyby obsahuje, či nikoli, lze považovat za další z cílů této práce.

Následují pokusy o reprodukce útoků. V prvním případě dosáhnu zjevné chybové reakce zařízení (odmítnutí služby). Přestože se tato reakce liší od publikovaných výsledků (únik informací z haldy), je možné předložit nepřímé důkazy, že se skutečně jedná o zkoumanou zranitelnost. Ve druhém případě odolalo mé IoT zařízení útokům bezchybně. Z toho vyvozuji, že tato zranitelnost není v jeho implementaci TCP/IP subsystému přítomna.

V práci se mi podařilo potvrdit přítomnost jedné ze zranitelností ve zkoumaném zařízení a provést na ni útok. Na druhou stranu druhá testovaná zranitelnost zřejmě ve zkoumaném systému přítomna není. Po přijetí závěru, že zkoumané zařízení je zranitelné, formuluji nejdůležitější postupy ke zmírnění rizik.

Klíčová slova počítačová bezpečnost, internet věcí, Ripple20, TCP/IP subsystém, Treck, Inc., IP tunelování, zpracování odpovědí DNS

Abbreviations

API	Application Programming Interface
ARP	Address Resolution Protocol
CSMA/CD	Carrier Sense Multiple Access with Collision Detect
CVE	Common Vulnerabilities and Exposure database
CVSS	Common Vulnerability Scoring System
CVSSv3	Common Vulnerability Scoring System, version 3
DDoS	Distributed Denial of Service
DNS	Domain Name System
DoS	Denial of Service
FTP	File Transfer Protocol
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IANA	Internet Assigned Numbers Authority
ICMP	Internet Control Message Protocol
ID	Identifier
IHL	Internet Header Length
IoT	Internet of Things
IP	Internet Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
ISN	Initial Sequence Number
ISP	Internet Service Provider
MAC	Media Access Control
NAT	Network Address Translation
OSI	Open Systems Interconnection
OT	Operational Technology
PC	Personal Computer
RCE	Remote Code Execution
RTOS	Real Time Operating System
SCADA	System Control and Data Acquisition
SOHO	Small Office, Home Office
TCP	Transmission Control Protocol
TTL	Time to Live
UDP	User Datagram Protocol
Wi-Fi	Wireless Fidelity

Chapter 1
Introduction

In this study, we attempt to present a brief overview of one specific field of cybersecurity research, the security of the Internet of Things (IoT). First we deal with some fundamental concepts and recent security incidents. Then we present a description of one specific family of vulnerabilities, Ripple20. The main part of the study is a practical demonstration of attacks on two vulnerabilities belonging to the Ripple20 family, followed by their analysis.

We have selected this particular topic for our thesis because IoT security is currently a major concern and one of important topics of research. Taking into account the number of devices existing in the wild, the incredible amount of new emerging products and the specifics of the IoT concept as such, it is clear that the amount of attention of cybersecurity experts paid to this area is not going to decrease any time soon.

Unlike many studies focusing on replication of specific attacks, our research starts with a large question mark. The Ripple20 vulnerabilities are present in the TCP/IP stack implementation by Treck, Inc., which is a very widely used software suite. It exists in many versions, is highly configurable and allows for specific arrangements, e.g. to use subsystems provided by the customer. Although it is possible to find some indications which devices might be vulnerable, e.g. because their firmware was patched in response to the announcement, information about the presence or absence of the vulnerabilities in specific device types is not known to external parties. Our experiments thus have one more objective: to determine whether or not the particular selected vulnerabilities are present in our IoT device, as this is something not known to us from any sources.

1.1 Conventions Used in the Study

As regards the formal conventions used in the study, we should point out the following information:

Numerical values preceded by "0x" are hexadecimal values, e.g. 0x60 is 96 (decimal).

Code sections, names of functions, programs, command line arguments are highlighted using the following style: `framegen.cpp`.

Console output reprinted in the text is marked using the following style:

```
 Time    ID     Type Name                    Response
-------------------------------------------------------------
  0.000 64496 AAAA xmpp008.hpeprint.com. Valid response
  0.417  9955    A xmpp008.hpeprint.com. Invalid response
```

1.2 Internet of Things

Before diving into our particular area of interest, we have to introduce several fundamental notions important for our study. First and foremost, we have to clarify what is meant by the Internet of Things (IoT).

1.2.1 The Concept

The term "Internet of Things" is not a *terminus technicus* defined in a standard or RFC document, though it is widely used and its meaning is usually understandable to general public. The term was probably first used by Kevin Ashton in 1999 while he was working for Procter&Gamble. The idea described by the term is older than the term itself however. It actually dates back at least to the 1970s. Over the years, other terms were used in relation to what we call the Internet of Things today, e.g. "embedded internet" or "pervasive computing". [1]

It is therefore quite natural that any search for a definition of the Internet of Things brings a number of different results. There is a widely cited "technical" definition by McKinsey: *"In what's called the Internet of Things, sensors and actuators embedded in physical objects [...] are linked through wired and wireless networks, often using the same Internet Protocol (IP) that connects the Internet."* [2]

Hanes et al., on the other hand, suggest another definition that focuses mainly on functional aspects: *"The basic premise and goal of IoT is to 'connect the unconnected.' This means that objects that are not currently joined to a computer network, namely the Internet, will be connected so that they can communicate and interact with people and other objects."* [3]

Pallavi Sethi and Smruti R. Sarangi [4, p. 1] quote several other definitions of the Internet of Things looking at the phenomenon from other perspectives, for instance as an *"interaction between the physical and digital worlds"* ([5], cited by [4, p. 1]) or as a *"paradigm in which computing and networking capabilities are embedded in any kind of conceivable object"* [4, p. 1].

The concept of the Internet of Things is often misunderstood by the general public as a "network of home appliances". Such approach is, however, too restrictive and inaccurate. In their paper on classification of vulnerable IoT systems, Blinowski and Piotrowski enumerate the main IoT applications as follows: smart cities, smart environment, smart agriculture and farming, smart grid, manufacturing, industrial security and sensing, eHealth, home automation or "smart homes" [6, p. 2].

If we consider examples of devices from these domains—e.g. smart road systems, connected environment meters, connected industrial switches, insulin pumps, or smart fridges—we can see that such devices comply with the definitions cited above. They are physical objects that include sensors and actuators, they are linked through wired/wireless networks and they represent a class of devices that used to be unconnected.

This last aspect, i.e. "connecting the unconnected", is particularly important from the cybersecurity perspective. It actually means that connectivity technologies enter relatively new areas and that has dangerous consequences. New market players emerge, often lacking the necessary security expertise, and old players enter new areas, e.g. by changing their old "dull" products into new "smart" connected devices. Results—from the cybersecurity perspective—are actually identical in both cases and we will deal with them in more detail in the next section.

We do not have the option to roll back the current developments. *"Whether we're ready for it or not, we're rapidly evolving toward a world where just about everything will be connected"* [7]. Searching for remedial actions, preventing new damage and educating all the parties involved is our only choice at the moment.

1.2.2 Architecture and Design of IoT Devices

Taking into account the immense diversity of IoT devices it is clear that there is no unified architecture providing a universal model applicable to this entire domain. Sethi and Sarangi mention two most widespread models, viz. a three-layer model and a five-layer model [4, p. 2].

The three-layer model consists of the following layers: perception layer, network layer, and application layer. The *perception layer* is the physical layer, like sensors, and performs sensing and information gathering. The *network layer* ensures connection between smart devices, network devices and servers. The *application layer* delivers application-specific services to the user [4, p. 2].

The five-layer model was introduced because the older three-layer model was unable to fulfil the needs of research focusing on finer aspects of the Internet of Things. Its layers are: perception layer, transport layer, processing layer, application layer, business layer. In addition to the *perception layer* and the *application layer* there are three new layers in the model: the *transport layer* ensures transport of data from the *perception layer* to the *processing layer*. The *processing layer* (or the middleware layer) stores, analyses and processes data obtained from the *transport layer*. Finally the *business layer* manages the entire system [4, p. 2–3].

From cybersecurity perspective, bugs and vulnerabilities may be present throughout the entire system. Nevertheless, there is one obvious particularly vulnerable component that may ensure or compromise the security of the entire system. As the basic characteristic of the Internet of Things is its connection to the Internet, each IoT device contains a system for handling of Internet communication. Although it may take different forms, the most common and most relevant to the present study is the so-called TCP/IP stack. If we disregard physical intrusions into IoT systems which are not dealt with in this study, and if we put aside the legitimate user whose behaviour is often the most dangerous factor for IoT security (e.g. not changing the default credentials), it is the TCP/IP stack that either stops the intruder or leaves the gates open. Therefore we have to deal with this particular component in more detail.

1.2.3 TCP/IP Stack

As mentioned above, one of the main characteristics of the Internet of Things is connectivity. Internet connection is what links "things" to the "Internet". Each IoT device therefore must be capable of connecting to the Internet.

The TCP/IP stack (or more generally the "protocol stack") is a software component that implements the main connectivity functions of an IoT device. Due to the existence of such a subsystem, programs running in the device are able to connect to the Internet and use functionalities implemented at the network and transport layers (based on the OSI model, see also Section 2.1.1). It ensures that the application software does not have to deal with lower-level functions such as sending and receiving of packets, resolving of domain names, etc.

In a report on the AMNESIA:33 vulnerabilities, Forescout states that: *"TCP/IP stacks are critical components of all IP-connected devices, including IoT and OT, since they enable basic network communications. A security flaw in a TCP/IP stack can be extremely dangerous because the code in these components may be used to process every incoming network packet that reaches a device. This means that some vulnerabilities in a TCP/IP stack allow for a device to be exploited, even when it simply sits on a network without running a specific application"* [8, p. 4].

Examples of functions contained in the TCP/IP stack may include: ARP functions; Ethernet link layer functions; DNS resolver; FTP program interface; Telnet daemon; NAT implementation; or processing of IP packets (this particular list was compiled on the basis of Treck TCP/IP User Manual, see [9]).

There are numerous implementations of the TCP/IP stack. Report on AMNESIA:33 by Forescout focuses on seven open-source protocol stacks based on whether the stack is used or supported by popular RTOSes and the popularity of devices using the stack [8, p. 7]. These

open-source stacks included in the quoted report are: lwIP, uIP, Nut/Net, FNET, picoTCP, CycloneTCP, uC-TCP/IP [8, p. 8]. Another example of a TCP/IP stack is that of Treck, Inc. The Ripple20 family of vulnerabilities is tied to this particular protocol stack (although some individual vulnerabilities may be present in other stacks as well).

Protocol stacks are an attractive target for attackers for several reasons. Apart from the most obvious ones, like having a direct network exposure and being widely deployed, *"they often offer a variety of unauthenticated functionality exposing potential attack surface"* and *"they are often implemented as low-level system functionality and, as such, tend to be implemented in memory-unsafe languages such as C and C++"* [8, p. 6].

Apart from "core" dangers (actual vulnerabilities), TCP/IP stacks pose another significant cybersecurity risk, which certainly does not make things easier. Since this software component represents a relatively closed system communicating with the outer world through various APIs (*"When using the Treck TCP/IP system we want you to think of it as a 'Black Box'."*—Treck, Inc. in [9]), it is particularly suitable for reuse in many different products. This ultimately leads to multiplication and propagation of any bug found in the system. As a result, it is difficult to even identify all the affected device types, not to speak about patching (especially when some of the manufacturers may already be out of business by the time the vulnerability is discovered).

1.3 Security Challenges Related to the Internet of Things

With regard to IoT, major security challenges relate to connectivity. Users are never perfect and it is therefore inherently dangerous to connect a device to the Internet even if all the firmware and hardware is flawless. This is however not the situation we face. It has been already proved many times—by research and by actual incidents—that imperfect users are coupled with imperfect devices, with potentially immense consequences.

We are not exaggerating here. We have already mentioned that the Internet of Things includes various healthcare-related systems, like medical devices that transfer information to physicians via wireless networks, etc. A simple DoS attack on such a device can have fatal consequences.

Furthermore, numerous connected systems are utilized in public utilities, e.g. in electricity or water supply. Greenberg [10] describes the Aurora experiment that was performed in Idaho, USA, in 2007. During this experiment, cybersecurity researchers successfully destroyed a diesel generator the size of a school bus weighing twenty-seven tons using approximately 140 kilobytes of code. In the years that followed, actual attacks on electricity infrastructure occurred, e.g. in Ukraine in 2015, establishing a completely new level of dreadfulness that can be clearly designated as a "cyberwar".

Our study, however, focuses on neither medical devices nor critical infrastructure components and their security issues. From now on we will discuss the security issues of consumer IoT devices only.

To establish the magnitude of the problem, it would be useful to know the number of detected IoT vulnerabilities, either as an absolute number or as a fraction of all detected vulnerabilities. The most complete database of vulnerabilities, the Common Vulnerabilities and Exposures database (CVE, cve.mitre.org) maintained by the MITRE Corporation, does not provide any easy way of distinguishing IoT issues from the others.

Blinowski and Piotrowski [6] try to extract CVE records related to IoT devices using CPE data included in the CVE database. CPE, or Common Platform Enumeration, basically specifies the platform affected by the respective vulnerability. Although their approach incorporates an unavoidable simplification and does not claim to be absolutely precise, we can use their findings at least as an estimate (see Figure 1.1).

The Figure 1.1 clearly shows an increase in the number of IoT-related vulnerabilities, particularly after 2016 (for 2019, data for the first quarter were used only). While in the period from 2012 to 2016 the total number of detected IoT-related vulnerabilities ranged approximately from 300 to 500 per year, there was a sharp increase in 2017 (approx. 800), and 2018 (approx. 1600).

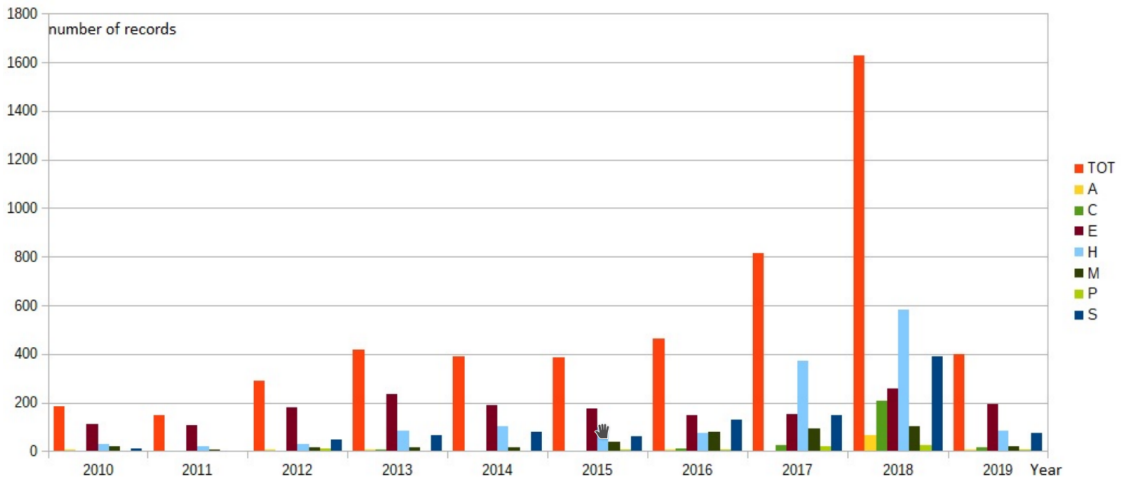

Figure 1.1 Chart showing estimated numbers of IoT vulnerabilities (reprinted from [6]); TOT = total, A = other, non-home appliances, E = enterprise, service provider hardware, H = home and SOHO devices, M = mobile phones, smart watches, portable devices, P = PCs and laptops as controllers of IoT systems, S = SCADA and industrial systems, automation components (the meaning of C is unfortunately not explained anywhere in the paper).

After combining this data with data on annual numbers of CVE records from a website allowing to browse vulnerabilities by date [11], we can establish approximate shares of IoT vulnerabilities in total detected vulnerabilities in each year from 2012 to 2018. Figure 1.2 shows the resulting table.

Year	2012	2013	2014	2015	2016	2017	2018
Total detected vulnerabilities	5297	5191	7946	6484	6447	14714	16556
IoT-related vulnerabilities (estimate)	300	410	390	390	460	810	1620
IoT / total (percentage)	6%	8%	5%	6%	7%	6%	10%

Figure 1.2 Total vulnerabilities and IoT vulnerabilities 2012–2018 (using data from [6]) and [11]).

We can conclude that the data clearly shows an increase in the number of detected vulnerabilities and although the increase follows overall growing trend of all vulnerabilities, their share in the total number of vulnerabilities might be growing too (10% in 2018 compared to 5–8% in the preceding years). Data for the entire year of 2019 and 2020 would certainly shed some light on this issue.

We can thus see that the number of IoT devices and the number of detected IoT vulnerabilities have been growing. What are the consequences of such a development? What are the implied risks? What can happen if a vulnerability is actually exploited?

A successful exploitation of vulnerabilities detected in IoT devices usually involves one or both of two types of attacks, viz. the denial of service attack and the remote code execution attack.

RFC 4732 [12] defines a Denial-of-Service (DoS) attack as: *"... an attack in which one or more machines target a victim and attempt to prevent the victim from doing useful work."*. In other words, such an attack does not imply any form of intrusion in the victim system but its consequences include unavailability of services provided by the victim (*"prevent the victim from doing useful work"*). Flooding of a website with an enormous number of http(s) requests resulting in the unavailability of the website might be a good example of a DoS attack.

A Remote Code Execution (RCE) attack is, on the other hand, an attack, in which the attacker runs an arbitrary code on the victim system. *"In an RCE attack, hackers intentionally*

exploit a remote code execution vulnerability to run malware. [...] This programming can then enable them to gain full access, steal data, carry out a full distributed denial of service (DDoS) attack, destroy files and infrastructure, or engage in illegal activity." [13] The code to be run might be provided to the victim in various ways, including a user input or a network port. The code, or a part thereof, might also be already residing in the victim system after it had been implanted there using other exploitation techniques such as social engineering.

The damaging potential of DoS attacks might seem relatively less threatening than that of RCE attacks. It is, however, necessary to emphasize that we are dealing with the Internet of Things here, which can be understood as an interface between the Internet (or attacker) and physical objects. The disruption of services of a security camera can certainly have very tangible consequences, e.g. a failure to detect the physical presence of a thief. DoS attacks on eHealth IoT systems are of course even scarier.

In the IoT domain, RCE attacks were often very closely related to DoS (or DDoS, "Distributed Denial of Service") attacks. They were used to create "armies of bots" or "botnets" that were subsequently utilized for attacking other victims.

The majority of recent attacks on IoT systems actually involved this particular scenario. The problem has proven to be very difficult to deal with, because *"the people responsible for the security of the devices commandeered for slavery don't feel the pain of the attacks, except as service users themselves. The device owners and manufacturers have no personal incentive to take responsibility, possibly perpetuating the potential for attacks in the future."* [14] Specific examples of recent attacks will be presented in the next section.

We have already discussed some possible causes of the current cheerless security situation in relation to the Internet of Things. They include, *inter alia*, inadequate experience and expertise of market players or the fact that the consequences of such security issues are not borne by those who could prevent them. Apart from that, however, there are some specific features of IoT devices that increase the magnitude of the problem even more.

The inability to update and the extremely large number of devices in the wild belong among the most problematic features inherent to the IoT domain [15]. Even when a vulnerability is detected and patched by the manufacturer, any attacker—at least if a popular consumer branch is targeted—can be virtually certain that there will be unpatched devices connected to the network. Even if the devices can be updated, there will always be many consumers who simply do not care or know enough to do so.

The mass production coupled with the modular architecture present in many IoT devices is another source of difficulty. That is what the JSOF researchers who discovered the Ripple20 vulnerabilities had in mind when speaking about "Hacking the Supply Chain" [16]. A vulnerability present in a component such as the TCP/IP stack supplied to many different manufacturers can easily find its way to an extreme number of various devices by many different manufacturers, some of whom may already be out of business when the vulnerability is discovered.

To conclude, there is no doubt that the Internet of Things can help the mankind to achieve easier and more comfortable way of living. The mode of operation of the entire industry, however, has to undergo a significant change otherwise the potential benefits could be offset by many adverse consequences. Recent developments have already proven that this is a real-world scenario.

1.4 Major Recent Attacks Related to IoT Systems

IoT security issues are not just a research concern of experts trying to make IT systems impenetrable and perfect. Attacks exploiting specific vulnerabilities of IoT devices actually happened and will likely happen again. This section presents a summary of the most notable attacks from the recent period.

1.4.1 Mirai Botnet

In September 2016, several DDoS attacks crippled a number of high-profile services, e.g. OVH, Dyn or Krebs on Security [17]. According to the attacked entities, these attacks exceeded 1 Tbps, which was an unprecedented value at that time. The unusual firepower of the attacks resulted from their unusual source: the affected websites were taken down by an army of IoT devices controlled by the attacker using the malware named Mirai. According to measurements of a security company that detected activity of the malware already in August 2016, the number of infected devices exceeded 600,000 in November 2016 when the main wave peaked. [17]

Research has shown that majority of the infected devices were routers and security cameras. The malware itself, which can be characterized as a self-propagating worm, consisted of two key components: the replication module and the attack module. When a device was infected by the replication module, it was reported to a server which injected the malware into the device. After that, the device could be controlled by a set of Command and Control servers and became a member of the enslaved botnet. [17]

Our detailed knowledge about the malware's *modus operandi* results from an unexpected event that happened on 30 September 2017. The malware source code was released on an infamous hacking forum by its alleged author under the nickname Anna-senpai [17]. This development naturally resulted in numerous attempts to follow the success of the original Mirai botnet but the successor groups were not able to gather such large numbers of controlled devices as the originator. [18]

The attacker penetrated the infected IoT devices in a simple manner: the initial version of the Mirai malware searched for IoT devices accessible from the outer network and tried a fixed set of 64 commonly used default login/password combinations. Many users did not change their default credentials after putting their devices in operation and therefore the attacker easily obtained control over their devices, though no high-tech sophisticated technique was actually employed. [17]

Not all devices could be controlled by Mirai indeed. *"Even if a device can be infected and join a Mirai botnet, many devices behave differently once enslaved. For example, some devices crash and reboot once they are issued an attack command, which flushes Mirai from its system. In another example, it was observed that certain variables, like source or destination IP, could not be accurately implemented by a specific device, thus sending the attack to the wrong destination."* [18]

Although the majority of sources analysing the Mirai attack do not deal with the infection process in detail, it is obvious that successfully guessing the right credentials does not mean that an attacker can run arbitrary code on the respective device. *"... in general, the attacker only have access to a telnet server in the device that gives access to a very constrained shell, usually a shrink version of busybox. What this means is that you will not find tools like ftp, wget or curl. For this reason the dropper may have to do some tricks to get the device infected."* [19]

By "dropper", the anonymous author of [19] means a *"program/script used to get the malware in the device and install it"*. The operation performed by the dropper consists of several steps, namely finding a folder with write and execution permissions, figuring out the device architecture and transferring the respective binary into it. The technique used by malware to complete these purposes is described in detail in [19]. Taking into account the focus of our study, we do not need to go into details now, noting that the respective process is—at least in a sufficient number of devices—achievable without the need of any other vulnerability. The anonymous author of [19] confirms this opinion too: *"Overall, there is nothing really special about the infection process. Most of those malwares do not even use an exploit."*

After the attacker had enslaved enough devices he launched his attacks on OVH, one of the Europe's largest hosting providers, the Krebs on Security website and Dyn (DNS provider). The attack targeted on OVH was actually aimed at one undisclosed customer. Reports show that Minecraft servers were the actual target and there is more information confirming that *"Mirai has been extensively used in gamer wars and is likely the reason why it was created in the first*

place" [17]. Even the Dyn attack, which was painted by the press as an attempt to "take down the Internet", was probably just another episode in a mysterious gamers' war and the unavailability of services of Internet giants like AirBnB, Amazon, Github, HBO, Netflix, Paypal, Reddit or Twitter was just a "collateral damage" [17].

The systems subject to the DDoS attacks by Mirai recorded massive traffic, peaking at 623 Gbps (Krebs on Security) or even 1 Tbps (OVH) [17]. Disproportionate number of attacking IoT devices were geographically located in South America and South East Asia (Brazil, Vietnam, and China ranked highest). Routers and cameras were the most frequent device types used in the attacks (routers and remotely installed cameras directly face the Internet).

What countermeasures could be taken to cope with botnet DDoS attacks utilizing IoT devices like the Mirai botnet attack? The reply to this question has two parts, since the attacker actually attacks two classes of systems. First, there is the attack on IoT devices in order to enslave them. Second, a botnet or an army of these enslaved devices attacks a selected target using a DDoS attack.

Taken into account the focus of the present study, we will not try to give a detailed answer to the second part of the question. There are various DDoS attack mitigation strategies. The Mirai malware was actually able to use many different vectors of attack, including DNS or HTTP, so it can be expected that a successful defence against its attacks should also employ many different strategies. In his 2018 report on Mirai malware, Ron Winward states: *"Effectively fighting these attacks requires specialized solutions, including behavioral technologies that can identify the threats posed by Mirai and other IoT botnets."* [18]

As far as the first question is concerned, The Cloudflare Blog proposes the following measures to be taken: eliminating default credentials, making auto-patching mandatory, login rate limiting. [17]

Eliminating default credentials should help defend the IoT devices against an attacker employing a list of credentials. *Making auto-patching mandatory* should eliminate the "set and forget" approach very often encountered in the IoT domain. Autopatching of these devices could be another step out of the danger zone. *Login rate limiting* should again help to defend the devices against brute-force attacks and might be another mitigation of the tendency of people to use weak passwords. [17]

1.4.2 Post-Mirai Botnet Attacks

As already mentioned above, the source code of Mirai malware was published by its author on September 30, 2017. This resulted in proliferation of new variants of the malware, creation of independent smaller botnets and new attacks.

On October 31, 2017, a Lonestar Cell, one of largest telecommunication operators in Liberia was targeted by Mirai botnet. Based on the fact that the infrastructure used for this attack showed virtually no overlap with that used for the Krebs on Security or OVH attacks, it seemed that this attack was actually performed by another actor. This was confirmed as correct later when the perpetrator confessed during a trial that he had been paid by another Liberian Internet service provider (ISP) to take down their competitors.

Ron Winward concludes: *"Although Mirai is several years old now, it is still active in its original form in addition to modern variants. Botnets such as Masuta, Owari, DaddysMirai and Orion all include Mirai attack code. Evidence also suggests that other IoT botnets like IoT_Reaper/IoTroop and Satori are based on the Mirai framework, albeit different approaches."* [18, p. 3].

Countermeasures listed in the previous section on Mirai are applicable to other botnet attacks of similar origin as well.

1.4.3 Recently Discovered IoT Vulnerabilities

The Mirai attack and subsequent attempts utilizing the same or very similar approach were real attacks. They should be considered as a dark example of what can happen when a vulnerability or a vulnerable behaviour is exploited. In the following subsections, on the other hand, we will deal with some recently detected vulnerabilities that were discovered and presented in publications by researchers and IT security companies. As far as we know (and we have to stress that our knowledge is limited) actual attacks based on the respective vulnerabilities have not been reported.

The following subsections provide brief descriptions of the most significant recent discoveries related to the vulnerabilities found in IoT devices. Individual descriptions are followed by a brief section presenting countermeasures to protect against exploits of these vulnerabilities.

1.4.3.1 Urgent/11

A group of vulnerabilities dubbed Urgent/11 was discovered in 2019 by Armis Labs. The Urgent/11 vulnerabilities are present in the TCP/IP stack of VxWorks, the most popular real-time operating system (RTOS). The identified TCP/IP stack was used in additional RTOSes as well and according to the report by Armis Labs, over 30 vendors have self-identified as being vulnerable and released patches [20, p. 3].

Apart from other systems, the vulnerability is present in industrial controllers by Rockwell Automation, Schneider Electric and Siemens. Since these three companies account for more than 60 % of global market share of programmable logic controllers, it is clear that the potential impact of Urgent/11 vulnerabilities is immense [20, p. 3].

One of the most critical vulnerabilities of this family, CVE-2019-12256, is a RCE vulnerability utilizing a stack-based buffer overflow that can occur while parsing the IP options in IPv4 packets. The vulnerability can be triggered by a specific malformed IPv4 packet, including *"a maliciously crafted broadcast IPv4 packet that can be sent to the entire LAN, and trigger a stack overflow on any vulnerable device within it."* This characteristic provides the attacker with an unusual power, allowing them *"sending the maliciously crafted broadcast packets to the network, and take-over any vulnerable devices on the same LAN, in parallel"* [20, p. 3].

1.4.3.2 Amnesia:33

Following the discovery of Urgent/11 in 2019 and Ripple20 vulnerabilities in June 2020, it became clear that the problem of numerous vulnerabilities found in TCP/IP stacks is not restricted to a few particular software suites. Forescout Research Labs based in San Jose, USA, initiated Project Memoria with the aim to perform—together with industry peers, universities and research institutes—a large study of TCP/IP stacks focusing on various vulnerabilities. [8]

The first report under Project Memoria appeared in December 2020 and presented a collection of memory corruption bugs designated as Amnesia:33. As the name suggests, this group includes 33 new vulnerabilities, which were found in 7 analysed protocol stacks used by many major IoT vendors. Four of these vulnerabilities were critical RCE vulnerabilities with all the associated potential consequences (compromising a larger network, getting control of the device, inclusion of the device in a botnet, etc.).

As an example, we can mention one of the critical vulnerabilities of this family with the CVSS score of 9.8, CVE-2020-24338, found in the picoTCP protocol stack. The report [8] describes it as *"The function that parses domain names lacks bounds checks, allowing attackers to corrupt memory with crafted DNS packets"*.

1.4.3.3 NUMBER:JACK

Another output of Project Memoria was published in February 2021, presenting a collection of 9 vulnerabilities based on TCP communication, specifically on ISN (Initial Sequence Number). Dubbed NUMBER:JACK, this new family of vulnerabilities can be exploited in order to hijack or spoof TCP connections, ultimately resulting in the possibility *"to close ongoing connections, causing limited denials of service, to inject malicious data on a device or to bypass authentication"* [21, p. 3].

Initial Sequence Number is a mechanism introduced in order to prevent connection collisions when TCP sockets or unique combinations of network address and network port are reused. ISN is a 32-bit sequence number that should be unique for every TCP connection. The ability to guess the ISN gives the attacker a wide array of possibilities, including closing a connection or impersonating a legitimate client [21, p. 6–7]. The flaw that is now considered a vulnerability (ISN was based on a 4 millisecond timer) was actually incorporated in the original design and is contained in RFC 793 "Transmission Control Protocol" [22]. It was later corrected by RFC 1948 "Defending Against Sequence Number Attacks" [23] and RFC 6528 of the same name [24] which propose a manner of computing ISN that is better protected against ISN guessing.

Although some of the reviewed protocol stacks were implemented correctly in relation to ISN generation, a vast majority of them was not. The most common description *"ISN generator is initialized with a constant value and has constant increments"* [21, p. 4] actually proves that the root cause is what can be described as *"RFC documents detailing the countermeasures are being ignored"* [21, p. 8].

1.4.3.4 NAME:WRECK

In April 2021, Forescout Research Labs and JSOF published a joint report on another group of vulnerabilities discovered within the frame of Project Memoria. The new group consisting of 9 vulnerabilities was labelled NAME:WRECK. This time, researchers focused on DNS-related bugs in several TCP/IP stacks, including some very popular brands. Therefore it may be expected that the number of IoT devices containing NAME:WRECK vulnerabilities might be very high, the authors' estimate is 100 million devices. [25]

NAME:WRECK vulnerabilities concern DNS communication and are specifically tied to the DNS compression scheme which is described in detail in Section 3.1.2. Authors of the report are convinced that *"DNS compression is neither the most efficient compression method nor the easiest to implement. [...] this compression mechanism has been problematic to implement for 20 years"* [25, p. 5].

The potential impact of the vulnerabilities includes RCE and DoS and their CVSS score ranges from 6.5 to 9.8. For instance, the vulnerability with the assigned number CVE-2020-15795 (found in the NUCLEUS Net protocol stack) results from improper validation of domain names in DNS responses. Parsing of a malformed packet could lead to writing past the allocated structure, which could result in a remote code execution or a denial of service [25].

1.4.4 Possible Countermeasures

Updating of the device firmware whenever a patch or a new version is available should be seen as the primary countermeasure protecting against exploitation of various TCP/IP stacks vulnerabilities. On a related note, users should also make sure that an IoT device they intend to purchase allows for firmware updates.

If a patch is not available or if a device cannot be patched, the respective device should be connected to an internal network behind a router and should not face the Internet directly. Although this might not be a 100% cure, a properly configured and protected network may help mitigate the risk to a large extent. Where there is neither a patch available nor a chance to hide

the device from the outer world (e.g. in case of unpatchable routers or remotely installed IP cameras), it would be wise to procure a new device.

1.5 Ripple20 Family of Vulnerabilities

Ripple20 is a group of 19 vulnerabilities found in the Treck TCP/IP stack. The vulnerabilities were discovered by the research team of JSOF lab based in Jerusalem. The first announcement was published on June 16, 2020 after a special "prolonged" grace period was given to Treck, Inc., and its affected customers to issue the necessary patches. According to the researchers, the story began in September 2019 as a "part-time" research of the Treck TCP/IP library. After discovering some problems, they kicked off a coordinated vulnerability disclosure process. *"We understood from the beginning that Treck worked with many different vendors, with the potential to impact a large number of users. However, we simply had no idea of the scale and sheer magnitude of the situation, nor how complex the supply chain had become."* [26]

JSOF researchers estimate that the number of affected devices amounts to "hundreds of millions of devices (or more)" [26]. Although the Treck TCP/IP stack is not an open-source library, there are several factors that affect our ability to trace its use in final products. The first aspect to be taken into account is that this protocol stack has been around for about 20 years. Second, Treck was unable to supply the researchers with a complete list of their clients. Third, Treck collaborated on the development of its protocol stack with another company in 1990s. The other company, Elmic Systems, now Zuken Elmic, is based in Japan and it marketed the protocol stack as Kasago TCP/IP in Asia. The two companies later separated and thus there are two separate branches with the same origin that were distributed in separate geographic areas. Researchers' attempts to communicate with Zuken Elmic were unsuccessful. The tests have shown, however, that at least some Ripple20 vulnerabilities are present in Kasago TCP/IP as well. [26]

Since it may be presumed that a lot of devices in the wild will be never patched against Ripple20 vulnerabilities, publicly available descriptions of Ripple20 vulnerabilities are mostly general, lacking the level of detail needed for a successful exploit. Three exceptions to this approach are the vulnerabilities described in detail in two technical whitepapers. The first whitepaper, published at the time of the first announcement, focused on two vulnerabilities, specifically CVE-2020-11896 and CVE-2020-11898 [27]. The second whitepaper was published in August 2020 and it concerned another vulnerability, CVE-2020-11901 [16].

All vulnerabilities detected in the Treck TCP/IP stack were grouped and given a common designation because of their presence in the same protocol stack. However, they have different principles and modes of operation, some being closely related to the processing of incoming TCP packets while others concern parsing of DNS responses, etc. This will become more apparent in the following chapters where we describe our attempts to exploit two of the vulnerabilities using two fundamentally different approaches.

A list of all the vulnerabilities included in the Ripple20 group, including their CVE ID, CVSS score, description, and version of the Treck TCP/IP stack that fixes the respective issue is presented in Appendix A. Four of the vulnerabilities are rated critical (CVSS score over 9) and allow for remote code execution. [26]

Although the principles of these vulnerabilities and methods of their exploitation may be very different, the root causes are often very similar and include the "usual suspects" like improper input validation, possible out-of-bounds read or improper handling of the length parameter inconsistency (plus some others) [26]. Detailed information on the vulnerabilities targeted in our experiments can be found in the respective sections in the following chapters.

A successful exploitation of these vulnerabilities could imply a number of risk scenarios, providing the attacker with numerous possibilities, e.g. to take control over an Internet-facing device; to target a specific device after a successful infiltration in the network (using other

means); to broadcast an attack that is able to take over all impacted devices in the network; or even to perform an attack on a device from outside the network boundaries. *"In all scenarios, an attacker can gain complete control over the targeted device remotely, with no user interaction required."* [26]

Possible consequences of such attacks depend on the specific targeted device and the network to which it is connected. Taking into account that dozens of vendors were actually affected (as of October 25, 2020, 31 vendors were confirmed as affected, 66 vendors were pending confirmation, 29 were confirmed as not affected), including companies producing medical devices or power systems, it is clear that a successful DoS attack itself could have disastrous consequences for the impacted organization.

1.6 Device Selected for Experiments

The initial announcement of the Ripple20 zero-day vulnerabilities included a relatively long list of affected manufacturers. After deciding to attempt to explore these vulnerabilities, we faced the question of which device to acquire for our experiments. Although it would be possible to use the same device as the authors of the initial whitepaper, selection of another device and exploring it without knowing the results in advance seemed like a better idea to us, although it implied a lot of uncertainty.

Hewlett Packard, one of the affected manufacturers mentioned by JSOF, published—in connection with Ripple20—a list of its products with recommended firmware updates. By comparing the types listed on the respective web page [28] with products currently available on primary and secondary markets in Czechia, we found several intersecting points. HP DeskJet Ink Advantage 3775 printer was among them.

The selected device is the HP DeskJet Ink Advantage 3700 All-in-One printer (regulatory model number 5DGOB-1621, FPU No. J9V87-64006), firmware version LAP1FN1828AR of July 13, 2018. After completing the basic experiments, a firmware update to LAP1FN2020BR was performed and the behaviour of the device with the updated firmware was tested too.

Figure 1.3 shows a photograph of the device.

Figure 1.3 HP DeskJet Ink Advantage 3775 printer used for out tests.

When describing the response of the device to various conditions or attacks, we use some indicators from its control panel. It is therefore necessary to become familiar with them. The control panel of the device can be seen in Figure 1.4.

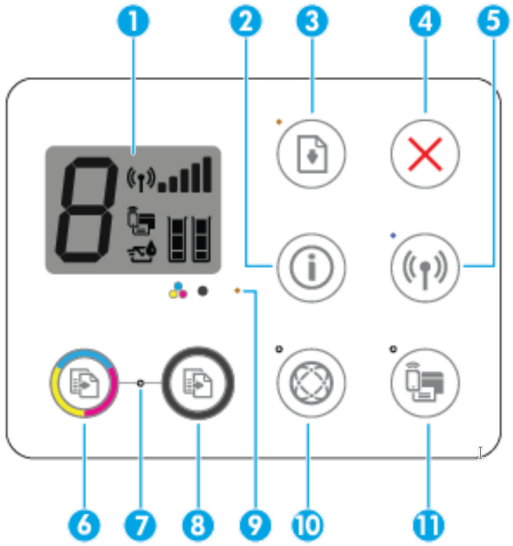

Figure 1.4 Control panel of the device (reprinted from [29], p. 5). Key: (1) Control panel display; (2) Information button; (3) Resume button with Resume light; (4) Cancel button; (5) Wireless button with Wireless light; (6) Start Copy Color button; (7) Start Copy light; (8) Start Copy Black button; (9) Ink Alert light; (10) Web Services button with Web Services light; (11) Wi-Fi Direct button with Wi-Fi Direct light. Moreover, the power button of the device features a LED light which can enter the following states: off/on/flashing.

It is possible that a network printer does not fit all the definitions of an IoT device, arguing that it lacks the perception layer, and so on. Taking into account the objectives of our study, however, we consider such distinction irrelevant. From the cybersecurity viewpoint, our device poses exactly the same risks as any "proper" IoT device and its use of protocol stack does not differ from other IoT devices either.

1.7 Selection of Vulnerabilities for Attacks

The first whitepaper published by JSOF in June 2020 [27] described in detail two Ripple20 vulnerabilities, viz. CVE-2020-11896 and CVE-2020-11898. The second whitepaper [16] that followed in August 2020 contained a description of the CVE-2020-11901 vulnerability.

First of all, it is important to note that before starting our experiments, we had no guarantee that our device would contain any of the vulnerabilities at all. The TCP/IP stack by Treck, Inc., exists in many different versions and some of the vulnerabilities described by JSOF have already been corrected before. The information whether and to what extent individual clients of Treck, Inc., i.e. hardware manufacturers, applied updates to their products is not available to general public.

Of the three Ripple20 vulnerabilities introduced in more detail by JSOF in their whitepapers, we decided to test CVE-2020-11898 and CVE-2020-11901. We ruled out the first of the three described vulnerabilities, CVE-2020-11896, as its description implied additional restrictions (unknown variables): *"The exploit is probably easily adaptable to devices using the Treck stack that use the same heap configuration and a similar (slightly older) version of Treck"* [27].

1.8 Environment for Our Experiments

Our device is a printer capable of connecting to Wi-Fi. The full setup for our experiments therefore consists of: (1) a router connected to the Internet with a Wi-Fi function, (2) a personal computer connected to the router, and (3) the printer, connected to the router.

For portability reasons, a mobile phone is used as a router (Lenovo K9, running Android OS, with the Wi-Fi hotspot function on). The personal computer connected to the router is Lenovo ThinkBook laptop running OpenSUSE Leap 15.2 version of Linux. The device is the HP DeskJet Ink Advantage 3700 printer described in Section 1.6.

Chapter 2

Implementation of an Attack on the CVE-2020-11898 Vulnerability

CVE-2020-11898 is the first vulnerability selected for our experiments. It was described in detail in the first technical whitepaper published by JSOF in June 2020 [27]. In the following sections, we will introduce some necessary technical concepts utilized in computer networks, like the OSI network layer model, Ethernet, Internet Protocol or IP in IP tunnelling.

After reviewing the theoretical background, we move on to a detailed description of the vulnerability, hardware settings, and software tools used in our experiments. A detailed description of our experiments follows, together with results and conclusions. It is again necessary to stress that we did not know whether the vulnerability was present in the device before starting our experiments. Determination of its presence is therefore one of the sub-objectives of the present study.

2.1 Theoretical Background: Networking Basics

Before we can start describing specific properties and bugs of the affected system and the method of their potential exploitation, we have to make a brief introduction to computer networks.

For this reason, we include the following sections explaining the fundamental networking concepts, like the OSI model and its layers, Ethernet, Internet Protocol, UDP protocol and IP in IP tunnelling, which might not be a fundamental concept but is needed for the particular vulnerability we are dealing with.

2.1.1 The OSI Model

Considering purely chronological criteria, the Open Systems Interconnection reference model or OSI model would not be presented as the first concept. The OSI model was created from two projects that were developed independently starting from late 1970s and it was first published in 1984 by the International Organization for Standardization as standard ISO 7498. [30]

Its prominent position in our introduction results from the fact that the OSI model offers a comprehensible overview of a layer-based network model and helps to grasp the basic notions. According to it, there are seven layers of network communication. Please note that the descriptions, which are based on [31], are highly simplified:

- *Application layer* is the human-computer interaction layer, the only layer that interacts directly with the user; example of protocols: HTTPS, SMTP.
- *Presentation layer* prepares data for the *application layer*, includes translation, encryption, compression of data.
- *Session layer* opens and closes communication between devices.
- *Transport layer* is responsible for the communication between devices. This may involve a breaking of data taken from the *session layer* into chunks (segments) and handing them over to the *network layer* as well as reassembling of data obtained from the *network layer* on the receiving device. The TCP and UDP protocol that we will deal with later in this chapter belong to this layer.
- *Network layer* ensures data transfer between networks. It breaks up segments into smaller units (packets) and reassembles them again on the receiving device. The *network layer* takes care of routing (finding the best path to the destination) too. Internet protocol (IPv4, IPv6) is one of the protocols of this layer.
- *Data link layer* does the tasks that are very similar to those of the *network layer*, but for two devices on the same network. It breaks the packets into smaller units called frames. This layer often utilizes the Ethernet protocol.
- *Physical layer* is the layer of the actual physical equipment involved in the data transfer, like cables and switches. It converts the data into a form that can be transferred by the respective medium (e.g. into a bit stream) and vice versa.

The OSI reference model is not the only available model. We can mention the so-called TCP/IP model, which is quite similar to the OSI model, although it introduces certain simplification (reduction of the number of layers). Figure 2.1 shows the comparison of the two models together with protocols associated to individual layers.

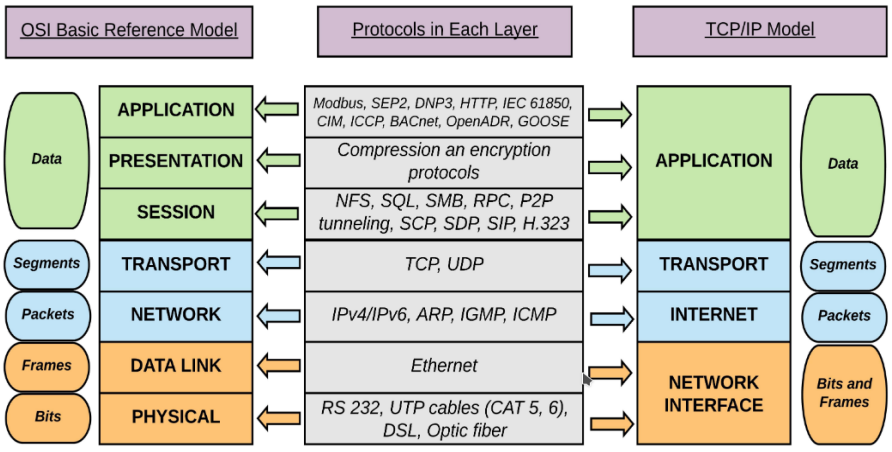

Figure 2.1 OSI reference model compared to TCP/IP model (reprinted from [32], p. 4).

2.1.2 Ethernet

The history of modern computer networks may be traced to various specific technologies and inventions. If we were to choose one which has most impacted the current form of our computer networks, including the Internet, it would certainly be the Ethernet.

Ethernet was invented in early 1970s by Robert Metcalfe, David Boggs and other members of their team charged with a task of creating a local-area network for Xerox's Palo Alto Research Center. From a closed in-house system it gradually evolved into an industry standard, beating some very serious competitors along the way. At present, networks based on non-Ethernet alternative technologies are mostly historical curiosities. (This does not apply to Wi-Fi technologies which have become very popular. *"But to supply those Wi-Fi access points with network connectivity, Ethernet will always have a role."* [33])

When we look at Figure 2.1, we find the Ethernet protocol at the *data link layer*. The layer description mentions that its task is to break the packets into smaller units called frames. [31] The most important benefit of Ethernet, however, lies somewhere else: it is the packet collision avoidance system that allowed to use the network infrastructure in a far more efficient manner than any alternative options.

The packet collision avoidance method defines what to do when two different sources are broadcasting packets at the same time. The approach used before Ethernet, i.e. packets were rebroadcast after some waiting, ensured a relatively reliable data delivery but the maximum traffic load was reached at only 17% of its potential maximum efficiency. Metcalfe studied the problem thoroughly and discovered that the right packet queuing algorithms are capable of increasing this number up to 90%. Ethernet's media access control (MAC) rules, Carrier Sense Multiple Access with Collision Detect (CSMA/CD), have been created on the basis of his work. [33]

In the CSMA/CD technique, a station that has data to send first listens to determine whether any other station is transmitting. When the channel is busy, the station waits. When the channel becomes idle, the station commences transmission. When two stations discover simultaneously an idle channel, a collision may appear. Successive collisions are prevented using a random delay scheme. [34]

In our study, we will encounter Ethernet protocol when we capture frames sent or received by our device or PC by Wireshark (software utility for analysing network traffic). The captured units are actually Ethernet frames and they consist of a header and a payload. We should therefore understand their structure. Figure 2.2 shows an actual Ethernet frame captured during our experiments.

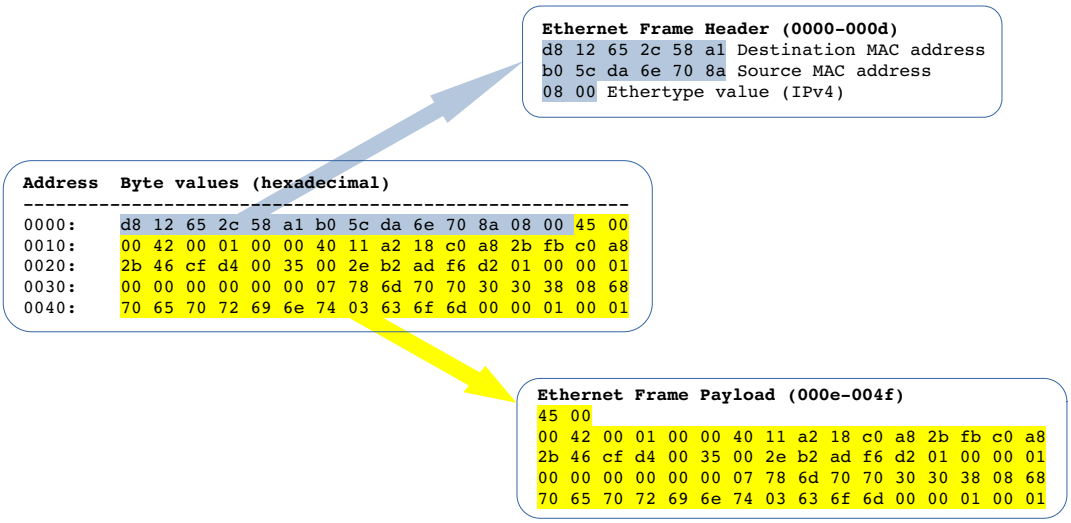

Figure 2.2 Example of an Ethernet frame.

In Figure 2.2 we can see the structure of the header. It follows the IEEE Standard for Ethernet [35, p. 120–121] and consists of these three parts: destination MAC address (6 bytes), source

MAC address (6 bytes) and Length/Type value (2 bytes). A MAC address is a 6 byte address that consists of a Company ID of 3 bytes (ID assigned to the respective hardware manufacturer) and of other 3 bytes ensuring a reasonable level of uniqueness within a local network. As regards the Length/Type field, IEEE 802.3-2018 standard [35, p. 121] states: *"If the value of this field is greater than or equal to 1536 decimal (0600 hexadecimal), then the Length/Type field indicates the Ethertype of the MAC client protocol (Type interpretation)."* In this case we have 0x0800 and therefore the field indicates the Ethertype value. According to IEEE 802-2014 standard [36, p. 28], the value of 0x0800 indicates the IPv4 protocol.

In Figure 2.2 we intentionally leave the payload as unstructured data since it is exactly how the multi-layer system operates: it takes the data coming from the neighbouring layer, "packs" it in own wrapper, disregarding the actual payload content or its structure. (For those who need to know: the packet depicted in Figure 2.2 is an IP packet with UDP datagram containing a DNS query, see Figure 3.2.)

2.1.3 Internet Protocol

Internet Protocol (IP) operates at the *network layer*. RFC 791 [37] from September 1981 defines the function of the Internet Protocol as follows: *"The internet protocol provides for transmitting blocks of data called datagrams from sources to destinations, where sources and destinations are hosts identified by fixed length addresses. The internet protocol also provides for fragmentation and reassembly of long datagrams, if necessary, for transmission through 'small packet' networks."* [37, p. 1].

Two versions of Internet Protocol are important for us. First, it is version 4 (IPv4), which has dominated the cyberspace for decades. It has clear limits, however, of which the insufficient address space is probably the most crucial. Therefore, version 6 (IPv6) was developed, which overcomes the shortcomings of version 4 and should be able to meet even future demands, hopefully forever.

In order to prevent an uncontrolled increase in the number of combinations of variables used in our experiments, we have decided to restrict our testing environment to Internet Protocol version 4 only. Only the most important information on both protocol versions will be therefore presented here. Afterwards, we will deal in detail with the structure of IPv4 packets only.

The most important difference between the two versions of Internet Protocol is related to IP addresses. As already mentioned above, IP address is a fixed length address. Since the authors of the original concept could hardly imagine in 1981 that their invention shall become one of the backbones of a flourishing world-wide network, they decided to construct the IP address as a sequence of 32 bits only. Obviously, this considerably limits the available address space (approx. to 4.295 billion addresses). IPv6, on the other hand, provides an immense address space as its addresses consist of 128 bits. It introduces some other improvements as well but—quite naturally—creates a problem related to backward (in)compatibility. [38]

During our experiments we capture IP packets (wrapped in Ethernet frames, as explained above) and we therefore have to familiarize ourselves with their form and structure. Figure 2.3 shows an example of IPv4 packet actually captured by us.

Figure 2.3 again shows the Ethernet frame header, but the payload is now structured. As it is an IPv4 packet, we can identify an IPv4 header and the packet payload (which will be dealt with later). The IPv4 header consists of the following sections (all information is based on RFC 791 [37], the specific values are those present in the example in Figure 2.3):

- The first four bits indicate the Internet protocol version, the following four bits indicate IHL (Internet header length). According to the RFC 791 document [37], *"Internet Header Length is the length of the internet header in 32 bit words"*. The IHL value of 5 therefore means that the header length is 20 bytes.

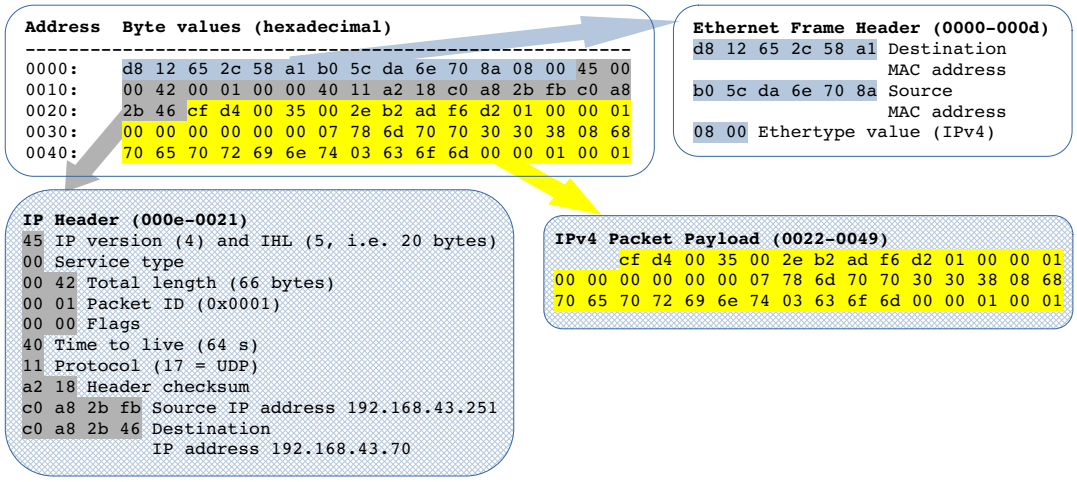

Figure 2.3 Internet Protocol packet (the hatched boxes) within an Ethernet frame.

- The second byte in the header indicates the type of service. The value consists of several flags implying precedence and levels of delay, throughput and reliability. The value we find here (0x00) is a "normal" value.
- The next two bytes indicate the total length of the packet. The value of 0x0042 means 66 bytes (indeed, the Ethernet header is not counted).
- The following two bytes indicate the packet ID (in our case: 0x0001).
- Two following bytes are divided in flags (3 bits) and fragment offset (13 bits). The first flag must be zero, the second flag indicates "May Fragment" (0) vs. "Don't Fragment" (1) and the third flag is "Last Fragment" (0) vs. "More Fragments" (1). Here we have 0x0000, b0000000000000000, meaning "may fragment", "last fragment", offset = 0.
- The next byte indicates time to live in seconds (0x40 = 64 s).
- The next byte: *"This field indicates the next level protocol used in the data portion of the internet datagram"* [37]. According to RFC 790 [39], the value of 0x11 we find here means "User Datagram".
- The following two bytes show the header checksum (here: 0xa218).
- Source IP address follows: 0xc0 0xa8 0x2b 0xfb, i.e. 192.168.43.251.
- Destination IP address occupies the last four bytes: 0xc0 0xa8 0x2b 0x46, i.e. 192.168.43.70.

In Figure 2.3, the payload is again intentionally left unstructured, although the information obtained from the header indicates the protocol used (UDP in this particular case).

2.1.4 User Datagram Protocol

The User Datagram Protocol (UDP) is one of the two protocols used at the *transport layer* based on the OSI or TCP/IP model, the other being the Transmission Control Protocol (TCP). In our experiments, we encounter UDP only and therefore we will not deal with TCP here.

User Datagram Protocol is specified in RFC 768 [40]. UDP focuses on sending messages between programs with a minimum of protocol mechanism. Under UDP, neither delivery nor

duplicate protection is guaranteed. If such guarantee is needed, it is recommended to use the Transmission Control Protocol (TCP), which has been designed specifically for such purposes. *"Applications requiring ordered reliable delivery of streams of data should use the Transmission Control Protocol (TCP)."* [40], p. 1.

Similarly to TCP, the User Datagram Protocol ensures *multiplexing*. This means that UDP can be used by multiple processes within a single host. For this purpose, UDP uses ports and sockets. This concept is described in RFC 793 for Transmission Control Protocol: *"To provide for unique addresses within each TCP, we concatenate an internet address identifying the TCP with a port identifier to create a socket which will be unique throughout all networks connected together."* [22], p. 10.

Port numbers are 16-bit numbers and are defined in RFC 6335 [41]. Ports with numbers from 0 to 1023 are *System Ports* and are assigned by IANA. Ports with numbers from 1024 to 49151 are known as *User Ports* or *Registered Ports* and are assigned by IANA too. Ports with numbers from 49152 to 65535 are never assigned and are known as *Dynamic Ports*, *Private* or *Ephemeral Ports*. [41]

In our experiments, we encounter UDP packets (in the context of attacks on CVE-2020-11901 vulnerability) and therefore we should understand their structure. There is no surprise that it follows the general structure of header and payload, as illustrated in the Figure 2.4.

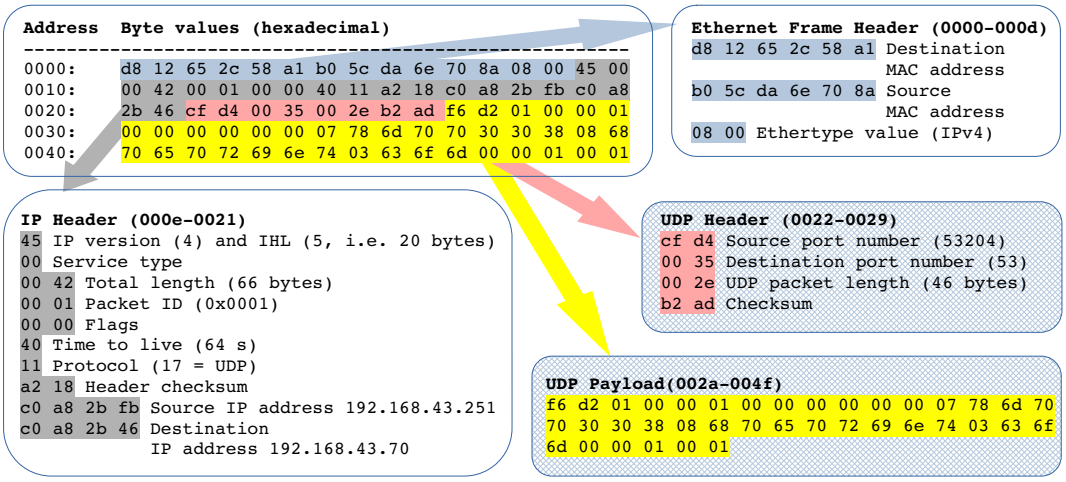

Figure 2.4 UDP packet (the hatched boxes) within an IP packet and Ethernet frame.

The UDP header is very simple and consists of the following sections (all the descriptions are based on [40]):

- Source port number (two bytes, here: 0xcfd4 or 53204)
- Destination port number (two bytes, here: 0x0035 or 53, this is the port number assigned to Domain Name System service)
- UDP packet length (two bytes, here: 0x002e or 46 bytes)
- Checksum (two bytes, here: 0xb2ad)

The payload is again intentionally displayed unstructured (it is a DNS message).

2.1.5 IP in IP Tunnelling

IP in IP encapsulation (tunnelling) is described in RFC 1853 [42]. The technique *"has long been used to bridge portions of the Internet which have disjoint capabilities or policies"* [42, p. 2].

The encapsulation is quite simple: a new IP header (the outer header) is added before the original header, with optional other headers (e.g. security headers specific for the tunnel in question) between them. [42]

As far as the content of the outer (new) IP header is concerned, RFC 1853 provides the following specification [42, pp. 3–4]:

- The type of service is copied from the inner IP header.
- A new ID number is generated for each outer header. There may be more outer IP headers due to fragmentation (e.g. when the maximum packet size within the tunnel is smaller than that of the upstream system and the incoming IP packet is too large).
- The "Don't fragment" flag is copied from the inner IP header while the "More fragments" flag is set as required by fragmentation.
- A new value for time to live is specified, the inner IP header time to live is decremented once before encapsulation.
- Protocol specifies the next header protocol. If no special header is used between the outer and inner IP headers, protocol 4 is used (Internet Protocol).
- Source and destination designate the beginning and end of the tunnel.
- Options are not copied from the inner header and new options may be used.

Figure 2.5 illustrates the concept of IP in IP tunnelling with fragmentation. (Please note that Figure 2.5 shows IP packets, not Ethernet frames, therefore there are no Ethernet headers displayed.)

2.2 Detailed Description of the Vulnerability

The vulnerability designated as CVE-2020-11898 was one of the two vulnerabilities described in detail in the whitepaper published together with the first public announcement of the Ripple20 vulnerabilities in June 2020 [27].

The National Vulnerability Database provides the following information about it [43]:

- Current description: The Treck TCP/IP stack before 6.0.1.66 improperly handles an IPv4/ICMPv4 Length Parameter Inconsistency, which might allow remote attackers to trigger an information leak.
- Severity: Base score: 9.1 Critical
- Weakness enumeration: CWE 200, Exposure of Sensitive Information to an Unauthorized Actor

When an IP packet received by the protocol stack contains an invalid IPv4 protocol number (e.g. 0), the stack creates an ICMP error message (destination unreachable, protocol unreachable). For debugging purposes, some data from the offending packet are copied to the error packet by the respective function. [27, p. 22]

The amount of data copied to the error packet is calculated as the header length + 8 bytes or the entire packet, whichever is smaller [27, pp. 22–23]. Due to the incorrect handling of IP

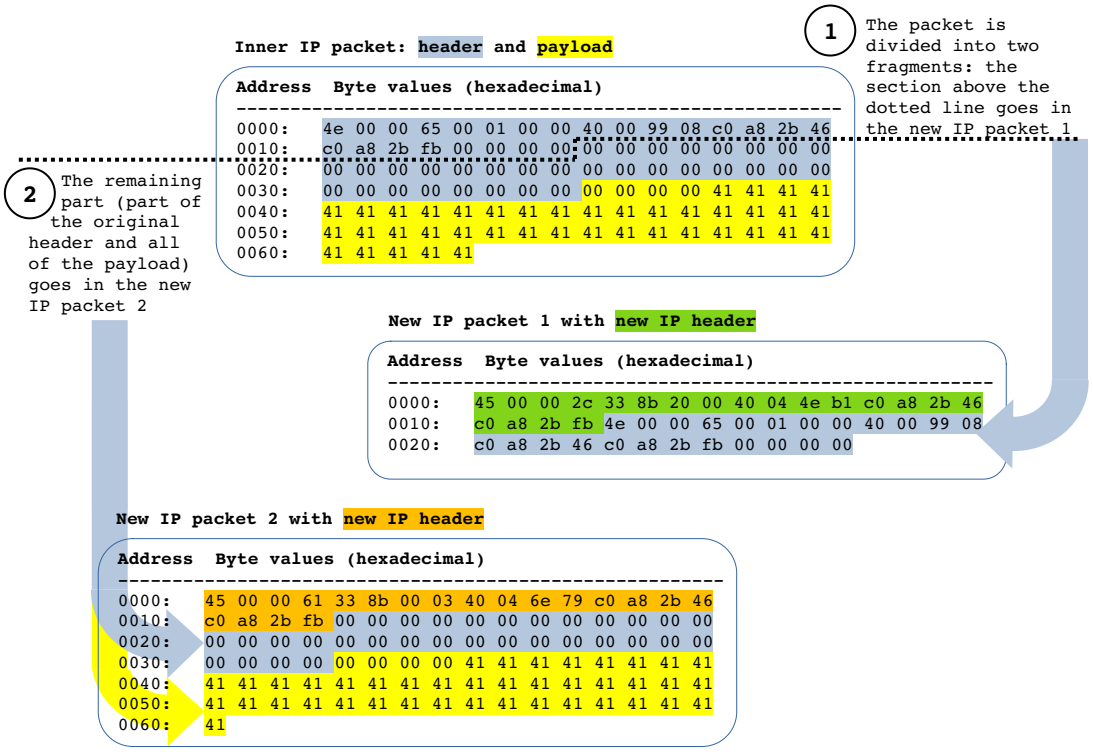

Figure 2.5 Example of IP in IP tunnelling with fragmentation.

in IP tunnelling by the TCP/IP stack, however, individual fragments in a fragmented IP packet are all assigned the total length value [27, p. 8]. This can result in a situation where a fragment is actually shorter than 68 bytes (which is the maximum length of IP header plus 8 bytes), but the protocol stack considers it as longer and therefore returns 68 bytes of data. [27]

Consider the following example from [27], p. 22:

- *Inner IP packet: IPv4{ihl=0xf, len=100, proto=0} with payload '\x00'*40+'\x41'*100.*
- *Outer IP packet (fragment 1): IPv4{frag offset=0, MF=1, proto=4, id=0xabcd} with 24 bytes from the inner IP packet payload. This means that 20 bytes of IP header will be copied, plus 4 null bytes.*
- *Outer IP packet (fragment 2): IPv4{frag offset=24, MF=0, proto=4, id=0xabcd} with the rest of the bytes from the inner IP packet as payload.*

After the two outer packets are processed, the inner packet is not assembled in a way that the entire packet is copied in a continuous memory area. Instead, the fragmented packet is stored as a linked list where each data structure contains the respective fragment data and a link to the next fragment. [27]

When the TCP/IP stack discovers that the protocol number (0) in the inner packet is invalid, it raises an error and assembles the ICMP error message mentioned above. In this moment, however, the fragment length associated with each fragment is 100. When the function checks the IHL (0xf, which means that the header is 60 bytes long), it decides to copy 68 bytes of data to the packet with the ICMP error message (as 68 is less than 100), although the fragment is actually only 24 bytes long. This results in a leak of 44 bytes of data from the heap. [27]

Improper handling of the length parameter inconsistency is the root cause of this vulnerability. [27]

2.3 Getting Everything Ready

The following sections describe the specific hardware settings, software tools and methods employed in performing an attack on the CVE-2020-11898 vulnerability.

2.3.1 Hardware Settings

The experiments performed in our attempts to exploit the CVE-2020-11898 vulnerability utilize the basic hardware settings as described in Section 1.8.

2.3.2 Software Tools Used for the Attack

In early stages of our research, we performed simulated attacks on the CVE-2020-11898 vulnerability using the Scapy tool[1]. After becoming more self-confident in packet manipulating skills, we decided to create just a simple python script capable of sending the contents of pre-prepared binary files to the selected network interface. The script named `11898.py` can be found on the attached data medium.

Our script `11898.py` is based on a minimalist approach. It does nothing more than accept command line arguments, check them, read the content of two binary files and send them as two separate Ethernet frames to the selected network interface.

To prepare files with the binary representation of the Ethernet frames, a simple C++ program, `framegen.cpp` has been written and used. It can be also found on the attached data medium, both as the source code and the compiled binary executable.

`framegen.cpp` is again a very simple program that allows for creating of a binary file with an arbitrary content. As the number of different parameters is relatively high, we have decided not to input them from the command line but directly in the source code. Each change of parameters thus requires a new compilation of the binary executable. `framegen.cpp` employs a function that determines the checksum value to be included in an IP header.

Wireshark software utility[2] was used to monitor the network traffic in order to detect any response from the device and/or any abnormalities.

2.3.3 Construction of Special Packets

Two outer IP packets and one inner IP packet described in Section 2.2, proposed by the JSOF researchers, were used for the attack. They were prepared using our custom-made software tool `framegen.cpp`. The inner packet is shown in Figure 2.6 and the outer packets in Figures 2.7 and 2.8.

```
Address  Byte values (hexadecimal)
-------------------------------------------------------
0000:   4f 00 00 64 00 01 00 00 40 00 98 07 c0 a8 2b 46
0010:   c0 a8 2b fb 00 00 00 00 00 00 00 00 00 00 00 00   Header
0020:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0030:   00 00 00 00 00 00 00 00 00 00 00 00 41 41 41 41
0040:   41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
0050:   41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
0060:   41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41   Payload
0070:   41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
0080:   41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
0090:   41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
```

Figure 2.6 Inner IP packet used for attack on the CVE-2020-11898 vulnerability.

[1] https://scapy.net/
[2] https://www.wireshark.org/

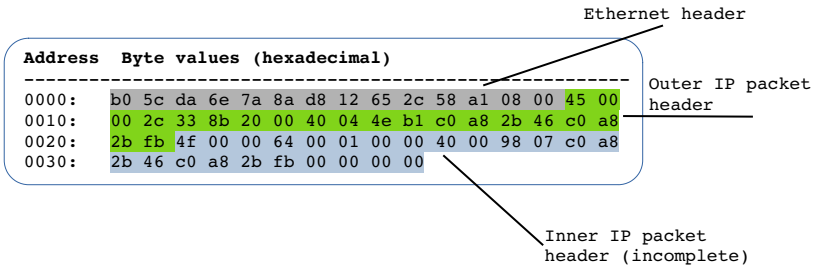

Figure 2.7 First outer IP packet used for attack on the CVE-2020-11898 vulnerability.

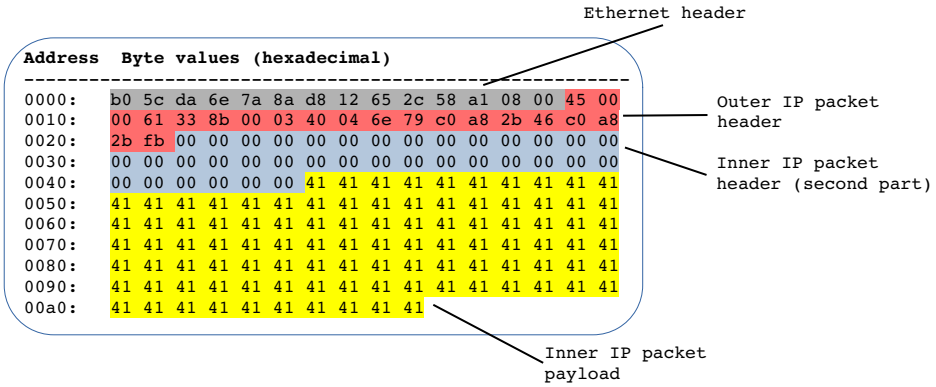

Figure 2.8 Second outer IP packet used for attack on the CVE-2020-11898 vulnerability.

Since we constructed the packets as Ethernet frames, the outer packets (where it is relevant) are displayed including the Ethernet headers.

Indeed, this was not the only pair of packets or frames used during our experiments. Quite naturally, the path was not straightforward and we tried a number of different options. We tried to change various parameters within the IP headers. Although the Total Length field in the inner IP packet has shown to be of particular importance, it was not the only parameter we tried to manipulate with. IHL of the inner packet as well as various options of fragmentation of the inner packet among the two outer packets were explored too.

Packets used for such experiments were constructed using our software tool `framegen.cpp`. We have already mentioned that our tool calculates the correct checksum values for the IP headers. Mutual internal relations between various values (e.g. the fragment offset in IP header and the actual length of the first fragment) were taken care of manually.

2.4 Attack Implementation

After preparing the correct environment and setting up all the necessary tools, as described in previous sections, the packets prepared on the basis of information specified in [27], as shown in detail in Section 2.3.3, were sent to our IoT device. When doing so, all the network communication with our IoT device was monitored using the Wireshark utility. Byte form of the frames used for the attack are available on the attached data medium as the binary files `frame1-01.bin` and `frame2-01.bin`.

The response of our IoT device was immediate: it entered an error state, stopped responding to any queries (like pings), the Resume light and Wireless light on the control panel were flashing, the power button LED was flashing and a letter E was blinking on the device's LCD display

(see Figure 2.9). We found just one way how to recover from this state, to push the power button. After doing so, the device restarts to a normal state (i.e. if we want to switch the device off, we have to push the power button once, wait until it restarts, and then push it again). Furthermore, Wireshark did not detect any response from the printer. We were able to achieve the same result repeatedly with 100% accuracy.

Figure 2.9 Device indicating error: flashing/blinking elements on control panel.

Following this initial attempt, we tried to tweak some parameters of the fragmentation, IP headers and the actual payload. In doing so, we were able to verify that the IoT device actually performs the basic sanity checks described in [27], p. 7–8: *"In addition to verifying the header checksum, it also verifies that:*

ip_version == 4 &&
data_available >= 21 &&
header_length >= 20 &&
total_length > header_length &&
total_length <= data_available."

When we changed the packets in such a way that the respective conditions were not fulfilled, our device simply did nothing. We understand such response (no error state, no response detected in Wireshark) as an indication that the packets were rightly identified as malformed by the device and were dropped before the vulnerable program part could be executed.

The byte forms of the frames used for these experiments are available on the attached data medium as the binary files `frame1-03.bin` and `frame2-03.bin` (IP version equal to 5), `frame1-04.bin` and `frame2-04.bin` (data available equal to 20), `frame1-05.bin` and `frame2-05.bin` (header length equal to 16), `frame1-06.bin` and `frame2-06.bin` (total length equal to header length), and `frame1-07.bin` and `frame2-07.bin` (total length exceeds data available).

If we change the Total Length value in the inner IP packet header to the actual correct length of the inner packet (i.e. from 100 to 160 in the example described in Section 2.3.3), we can achieve a correct-scenario behaviour, when the device responds exactly as expected. It is because in such a case, the vulnerable fragment trimming function is not called at all. We mention it because it can be seen as another evidence that the vulnerable function is actually present in the device, although it may be coupled with another problem. The byte form of the frames used for the attack are available on the attached data medium as the binary files `frame1-02.bin` and `frame2-02.bin`.

The response of the device to these "correct" frames, as captured by Wireshark, is illustrated in the Figure 2.10.

The Ethernet frame illustrated in Figure 2.10 shows that the ICMP message consists basically of two parts, the ICMP header of 8 bytes (it is specified in RFC 792, [44]) and the attached example data from the problematic fragment. The ICMP message is placed in an IP packet (with its IP header), which is then packed in Ethernet frame (with its Ethernet header). It is important to note that the amount of data from the problematic fragment sent within the ICMP message is correct, as the vulnerable trimming function was not called at all.

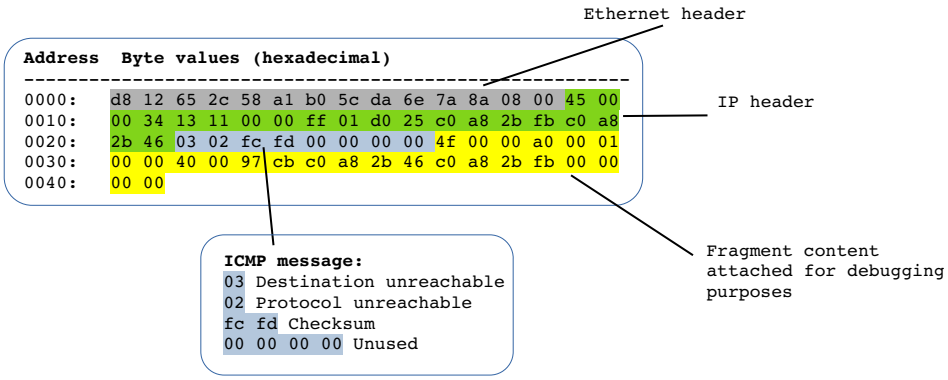

Figure 2.10 Ethernet frame containing ICMP message from the device.

2.5 Results and Their Interpretation

Our attempt at reproducing the attack on the CVE-2020-11898 did not have the same result as that of the JSOF researchers described in their whitepaper published in June [27]. Instead of an "information leak" we achieved a "denial of service".

We are convinced, however, that the attack was performed correctly. First, we were able to achieve the predicted result when using packet with the correct value in the Total Length field. Second, the predicted results were achieved in other scenarios where the packet data were intentionally malformed in order to verify that the sanity checks described in [27], pp. 7–8, are actually carried out.

The only case showing unexpected behaviour is therefore the case where the information leak vulnerability should have occurred. It is possible that (a) an attempt to correct the vulnerable function had been carried out in this particular version of the TCP/IP stack in the device; or (b) the vulnerable function exists in the device but another problem drops the device into an error state before the information leak can occur.

We have tried to examine and identify the cause of the crash of the device (its error state) but have not succeeded. We do not have any knowledge of the heap internals of the device. We suspect either a possible heap overflow or an access to unmapped memory. It is, however, significant that the error occurs regardless of the amount of data by which the Total Length value is exceeded. The difference of one byte is all that is needed, whatever the overall packet size is. Tweaking the value of the last byte (the one that overflows) did not lead to a different behaviour either. We must conclude that our current state of knowledge of the system concerned does not allow us to provide a more precise explanation.

2.6 Risk Mitigation Strategies

The list of risk mitigation strategies is based on the features of our particular IoT device, although we try to assume a general approach.

1. The user should change the default credentials (this is a general advice not directly related to this type of attack, we place it here due to its extreme importance). In our particular case, however, the device allows the user to set a password in order to protect system settings only. Network access to the device is not password protected.

2. The device should run the most up-to-date version of firmware.

3. A vulnerable device should not directly face the Internet. A properly configured internal network should be able to prevent such an attack from outside.

4. A vulnerable device should be connected to trusted networks only. The user must consider who else can get access to the network to which the device is connected.

2.7 Behaviour After a Firmware Update

After performing and evaluating all the experiments, we performed an upgrade to the current version of firmware issued after the discovery of the Ripple20 vulnerabilities (LAP1FN2020BR). Then we repeated the experiments using our device with the new firmware.

The new firmware eliminated the incorrect behaviour of the device. The device now resists the attack described in this chapter. Neither a denial of service nor an information leak occurs. This fact can be seen as another evidence supporting our opinion that the CVE-2020-11898 vulnerability was actually present in the device before the firmware upgrade.

Chapter 3

Implementation of an Attack on the CVE-2020-11901 Vulnerability

CVE-2020-11901 is the second vulnerability belonging to the Ripple20 family selected for our experiment. It was described in detail in a paper published by JSOF Labs in July 2020 [16]. This vulnerability is actually a summary designation covering 4 specific sub-vulnerabilities.

CVE-2020-11901 is closely related to the processing of DNS messages by the Treck TCP/IP stack. Before being able to exploit this vulnerability, any attacker has to overcome measures protecting the communication of the device with a DNS server. It is not an impossible task and one can find descriptions of successful attacks on various subsystems leading to such a result. However, we consider it to be out of scope of the present study and therefore we just presume that we are able to control DNS responses coming to the device from the outer world (DNS server).

In this regard, it is also noteworthy that having the ability to control DNS responses coming to the device means that we could also redirect the communication of the device to a server under our control instead of the one it tries to connect to. By doing so, we could possibly obtain an ability to plant our own code into the device, but such a task would be very difficult and is also out of scope of the present study.

Once again, we would like to stress that we have no information about the version of Treck TCP/IP stack used in our device and we do not know which parts of it were actually implemented. Due to the specifics of the business model of Treck, Inc., it is therefore quite possible that this particular vulnerability (or this group of vulnerabilities) is not present in our device at all. The following chapter of our study therefore tries to find answers to this question as well.

3.1 Theoretical Background: Communication with DNS Servers

As already mentioned above, Internet communication is transferred across the network using a system of unique numerical addresses (IP addresses). These addresses of uniform length (32 bits in case of IPv4 addresses, 128 bits in case of IPv6 addresses) are perfectly suited for use with digital machines. For an average human user, however, names consisting mainly of letters (as well as numbers and a few other characters) are easier to use. To overcome this discrepancy, a Domain Name System (DNS) was introduced.

DNS can be described as an interface that makes it possible to use human-readable domain names instead of numerical IP addresses. The entire system, which dates back to 1987 (see [45], [46]), consists basically of domain name servers and a set of rules and protocols governing the communication with them.

Figure 3.1 presents a simple illustration of the basic principle of operation of the Domain Name System. When the user's machine needs to contact a server with a certain domain name, it sends a DNS query to a DNS server. The DNS server finds the necessary data (it is not necessary to deal with details and specifics of the respective process here) and a DNS response is sent to the user containing the requested IP address (plus, optionally, some other information).

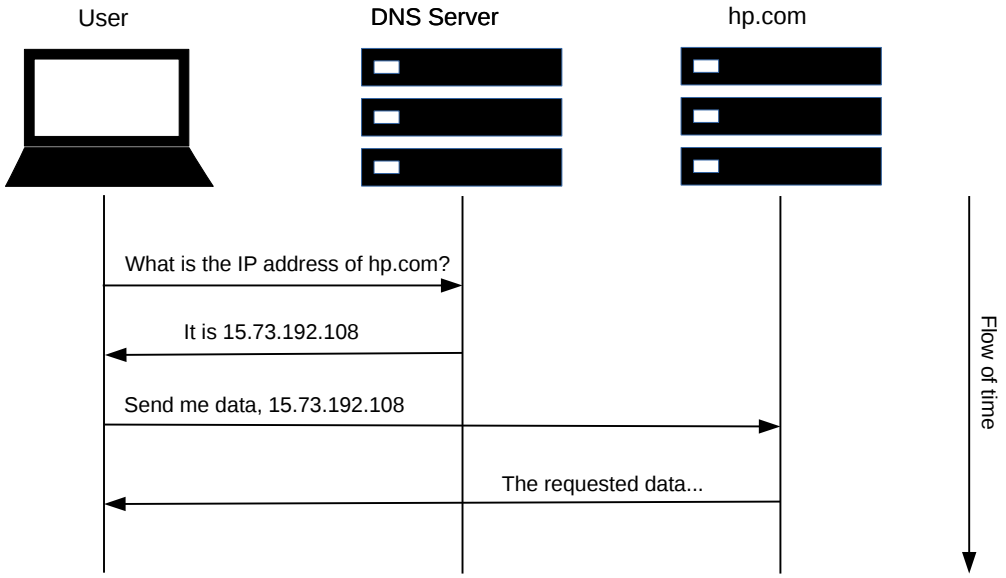

Figure 3.1 Domain Name System: basic principle of operation.

The user's machine or device then contacts the target server using the information obtained. To accelerate the process and decrease traffic load on the network, the user's machine can store IP addresses obtained from a DNS server, but usually only for a limited period because IP addresses may change over time.

For the purposes of our experiments it is necessary to understand the "language" used by the devices or computers in the system, i.e. the structure of DNS messages.

3.1.1 DNS Query

DNS query is that part of communication that originates from the user's system, device, etc. and is sent to the DNS server. A complete specification is again out of scope of this study and therefore only the query types encountered in our experiments will be dealt with below.

The general format of each DNS message is defined as follows: Header, Question, Answer, Authority, Additional. [46, p. 25]

In DNS queries, the Answer section is not used. Furthermore, the Authority and Additional sections are irrelevant for the present study and will be therefore not dealt with here. The Header and Question sections are the only parts of DNS queries relevant for our experiments.

Figure 3.2 shows an example of a packet containing a DNS query captured by Wireshark during our experiments with the device.

Ethernet header, IP header and UDP header have been already sufficiently described in Sections 2.1.2, 2.1.3, and 2.1.4. We are interested in the UDP payload, which is the actual DNS

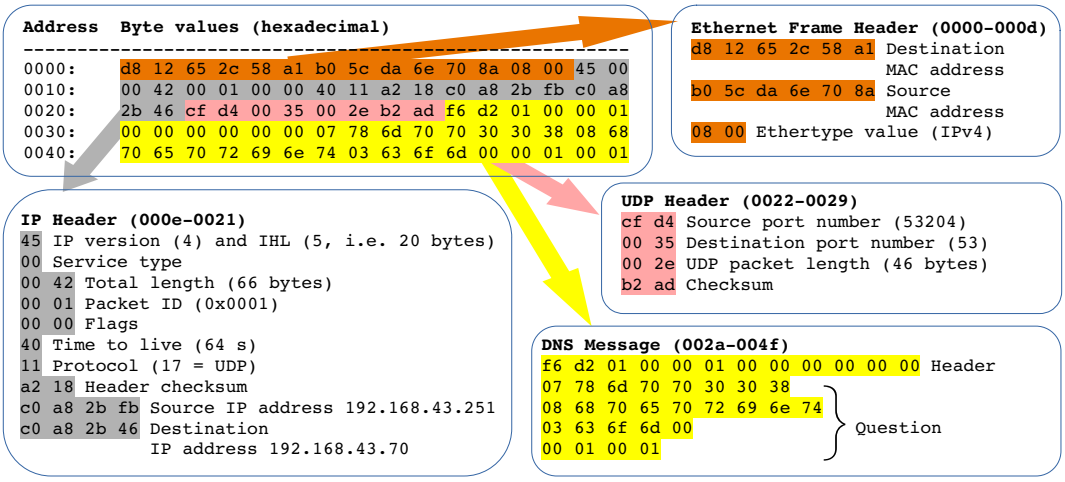

Figure 3.2 Example of an Ethernet frame containing UDP datagram with a DNS query.

message. It has its own structure that consists of header and question. It will be described in detail in the following text and in Figure 3.4.

From here on, we will be displaying various DNS messages without the Ethernet frame header, IP header and the UDP header. First, this information is not relevant for further analysis, and second, DNS messages use their own internal addressing system which assigns offset 0 to the first byte of the DNS message and we would like to adhere to that representation.

The DNS query message extracted from the above example is illustrated in Figure 3.4. The header consists of 12 bytes and its structure is uniform for all DNS messages, both queries and responses.

The rest of the payload (in our case 26 bytes) is the question. It contains QNAME, the queried domain name encoded using the following simple system. The name is *"represented as a sequence of labels, where each label consists of a length octet followed by that number of octets. The domain name terminates with the zero length octet for the null label of the root. Note that this field may be an odd number of octets; no padding is used."* [46, p. 28]. Figure 3.3 shows the encoding of domain name in the question specified in our particular example.

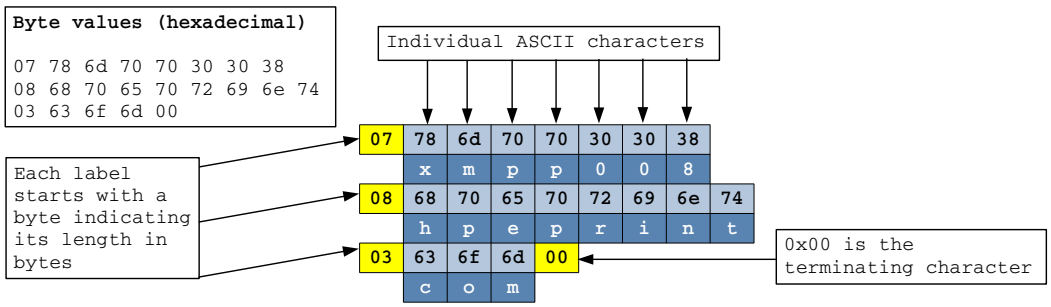

Figure 3.3 Encoding of domain names in DNS messages.

The name is followed by two-octet QTYPE code (here: 00 01) and two-octet QCLASS code (here: 00 01). Of these two codes, the former is important for us, as it defines the type of query. In our experiments, we encountered two types of queries, viz. type 1 and 28. Type 1, or A-type query, is a request for an IPv4 address of the respective domain name. On the other hand type 28, or AAAA-type query, is a request for an IPv6 address.

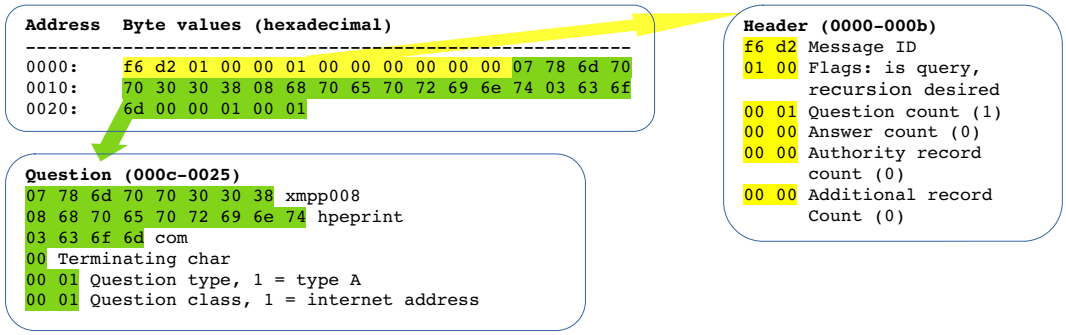

Figure 3.4 Example of a DNS query message.

3.1.2 DNS Response

When the DNS server processes the query and finds the necessary data in its database, it sends a DNS Response to the sender of the original message. The response to the query shown in Figure 3.4 can be seen in Figure 3.5.

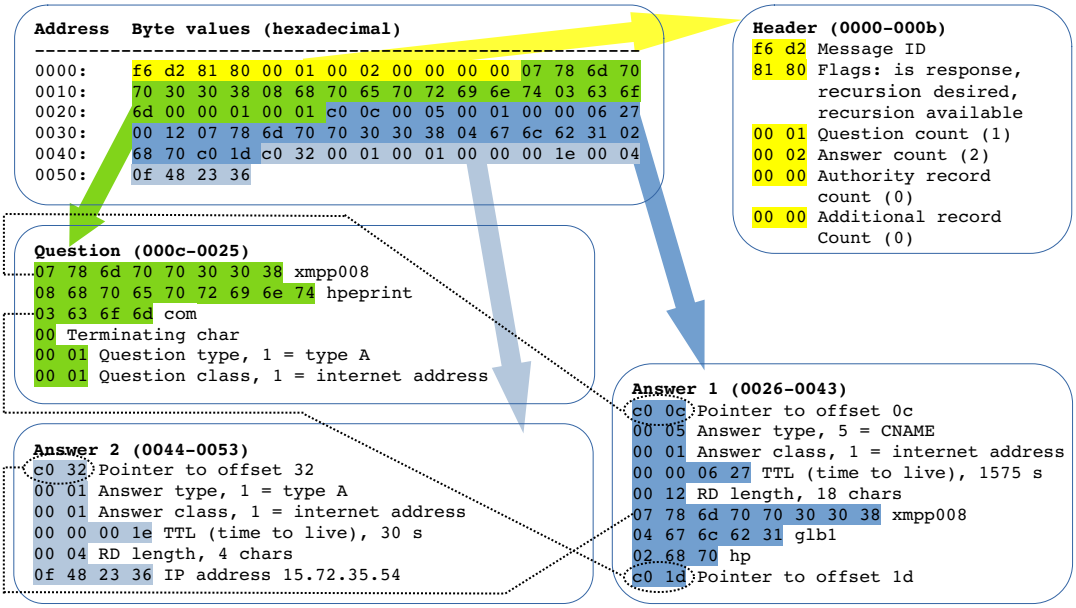

Figure 3.5 Byte-to-byte analysis of DNS response to type A query for xmpp008.hpeprint.com.

The DNS response header does not differ much from that of the DNS query but certain values are different here. Flags, which are represented by the two octets that follow the message ID (i.e. the two bytes starting at an offset of 0x02), have the value of 0x8180 or b1000000110000000. Most importantly, the first 1 denotes that the message is answer. The two other ones have the meaning “recursion desired” and “recursion available” (the meaning of these flags in actually not important for our experiments; for more information see [45], p. 26). Furthermore, the answer count should not be zero in a DNS response (in our case, the count is 2).

The header is then followed by two sections: a question section and an answer section. In this particular example, the answer section contains two answers.

The question section is an exact copy of the question from the DNS query to which the response belongs. Our experiments have shown that if the content of the question section differs from that of the DNS query, the device rejects the DNS response and does not process it. The same applies to responses with message ID different from that of the DNS query. Message ID and the copy of the original query are thus used as primary security measures ensuring that the received response is an actual response from a DNS server and not a spoofed response from attacker.

The question section thus starts at offset 0x0c (the header has 12 bytes, starting from offset 0x00) and its length is 26 bytes (in this particular case).

The question section is followed by the answer section. According to RFC 1035 [46, p. 29], an answer has the following format: NAME, TYPE, CLASS, TTL, RDLENGTH, RDATA. There are several possible types of answers, of which the following types are relevant for our experiments: type 1 (IPv4 host address), type 28 (IPv6 host address), and type 5 (CNAME).

Answer types 1 and 28 represent a straightforward reply to the question, i.e. they provide the requesting machine with the IPv4 or IPv6 host address associated with the domain name specified in the question section.

The type 5 answers work in a slightly different manner. They specify a canonical name (or CNAME) belonging to the name contained in the question section. This requires a brief explanation. According to RFC 1034: *"In existing systems, hosts and other resources often have several names that identify the same resource. [...] Most of these systems have a notion that one of the equivalent set of names is the canonical or primary name and all others are aliases."* [45, pp. 14–15]. The type 5 answer thus informs the requesting system that the queried name is an alias to another (canonical) name. A type 1 answer containing the IP address of that canonical name often follows in another answer within the same response (not necessarily).

Taking into account the frequency of DNS messages and the volume of the related traffic, it is clear that the authors of the system tried to make messages as short as possible to reduce their potential impact on network load to minimum. For this purpose, a simple compression scheme has been introduced. RFC 1035 [46, p. 30] describes it as follows: *"In this scheme, an entire domain name or a list of labels at the end of a domain name is replaced with a pointer to a prior occurance of the same name."*

The pointer is a two-byte sequence starting (at the bit level) with two ones. No label byte (i.e. a byte that could be otherwise expected here) can start with two ones because of length restriction on labels (a label cannot have more than 63 bytes). A typical pointer is thus a pair of bytes, of which the first is 0xc0 (or b11000000) and the second one indicates the offset from the beginning of the DNS message (offset value of 0x00 indicates the first byte of the header). The use of the compression scheme can be also seen in Figure 3.5 in which pointer bytes are connected with the respective targets using black dotted lines.

RFC 1035 [46, p. 30] concludes: *"The compression scheme allows a domain name in a message to be represented as either: a sequence of labels ending in a zero octet; a pointer; a sequence of labels ending with a pointer"* (Note: Interpunction added by us).

All these options are commonly used in actual DNS queries, often all of them in a single message.

It should be noted, however, that the compression scheme itself presents a threat in terms of cybersecurity. Its correct implementation is far from being simple and thus its positive impact on saved bandwidth might not be worth the risk. In a recent study on the NAME:WRECK family of vulnerabilities, the authors express the following opinion: *"It is also interesting that simply not implementing support for compression (as seen for instance in lwIP) is an effective mitigation against this type of vulnerability. Since the bandwidth saving associated to this type of compression is almost meaningless in a world of fast connectivity, we believe that support for DNS message compression currently introduces more problems than it solves."* [25, p. 26].

3.2 Detailed Description of the Vulnerability

This vulnerability was announced, like all the other Ripple20 vulnerabilities, by JSOF Labs in June 2020. Unlike most other Ripple20 vulnerabilities, it was later described in detail in a separate whitepaper [16].

The National Vulnerability Database provides the following information about it [47]:

- Current description: The Treck TCP/IP stack before 6.0.1.66 allows Remote Code execution via a single invalid DNS response.
- Severity: Base score: 9.0 Critical
- Weakness enumeration: CWE 20, Improper Input Validation

JSOF researchers studied the source code of the TCP/IP Stack by Treck, Inc., in several versions contained in multiple specific devices using reverse engineering methods. According to their findings, some versions of the Treck TCP/IP stack show the vulnerability which they describe on page 2 of their whitepaper: *"CVE-2020-11901 is a single name for several critical client-side vulnerabilities in the DNS resolver of the Treck TCP/IP stack. If successfully exploited, they allow remote code execution for an unauthenticated attacker that is able to respond to a DNS query generated from an affected device. [...] The vulnerabilities mostly stem from an incorrect DNS label length calculation. One vulnerability is triggered by specifying small RDLENGTH value, and another leverages DNS message compression scheme in order to achieve an integer overflow."* [16]

The four vulnerabilities covered by the CVE-2020-11901 designation are (numbering and short descriptions by [16], pp. 7–14):

- Vulnerability No. 1: Bad RDLENGTH Leads to Heap Overflow
- Vulnerability No. 2: From Integer Overflow to Heap Overflow
- Vulnerability No. 3: Read Out-of-Bounds
- Vulnerability No. 4: Predictable Transaction ID

3.2.1 CVE-2020-11901 Vulnerability No. 1

The RDLENGTH value, which can be fully controlled by an attacker, is used for calculation of a buffer used for storing the domain name of an MX record [16, p. 10] or a CNAME record, as explained on page 14 in the same whitepaper. *"An attacker can specify a small RDLENGTH value, causing the length calculation to stop prematurely. This results in tfDnsExpLabelLength returning small length number. As explained earlier, this leads to heap-based buffer overflow vulnerability. This vulnerability only exists in newer versions of the Treck TCP/IP stack, and affects the latest version at the time of disclosure. We do not know the exact version when this vulnerability was introduced."* [16, p. 10]

Improper input validation is the root cause of this vulnerability where input means DNS response received from a DNS server.

3.2.2 CVE-2020-11901 Vulnerability No. 2

The length of a domain name stored in a MX record or a CNAME record is—at least in some versions of the Treck TCP/IP stack—stored in a 16 bit variable of `unsigned short` type (possible values from 0 to 65535). UDP packets containing DNS responses up to the size of 1460 bytes are accepted by the Treck's resolver of DNS messages. If one can place a name longer than

65535 characters in this limited space, *"The result is heap-based buffer overflow vulnerability: due to the integer overflow, tfDnsExpLabelLength returns a labelLength of small size. Based on this size, a buffer is allocated on the heap. tfDnsLabelToAscii is then asked to copy the encoded name as ASCII, overflowing the buffer just-allocated with the payload at the beginning of the MX hostname."* [16, p. 13]

Moreover, exploit of this vulnerability is possible due to other problems. First, unlike the maximum label length in a domain name (0x3f or 63 characters), the maximum length of the entire domain name (255 characters) is not enforced. Second, the system does not validate characters inside of domain name labels (only alphanumeric characters and "-" should be allowed). [16, p. 10]

The method of overflowing the `unsigned short` length variable in the limited space provided by UDP packet is described in detail in Section 3.3.3.4.

Improper input validation is the root cause of this vulnerability where input is a DNS response received from a DNS server.

3.2.3 CVE-2020-11901 Vulnerability No. 3

In older versions of the Treck TCP/IP stack, there was another vulnerability, read out-of-bounds. As we do not know anything about the version of the TCP/IP stack used in our device, we include this vulnerability in our experiments as well in order not to miss anything significant.

This vulnerability stems from incorrect parsing of the MX (or CNAME) response section. The problem is that *"there are no checks on the packet buffer. In particular, there's no end-pointer being calculated and passed to tfDnsExpLabelLength."* [16, p. 13] The function does not have any bound checks. *"Instead, it will iterate over the length bytes until a null-byte is reached, possibly crossing buffer bounds. This issue could result in a denial-of-service vulnerability, if, for instance, the function reads from an unmapped page while iterating over the length bytes. More interestingly, the issue could result in an information leakage vulnerability."* [16, p. 13]

Interestingly, vulnerability No. 1 is actually a bad fix of this vulnerability No. 3 [16, p. 13]. Their simultaneous occurrence in a device should therefore be impossible.

Improper input validation is the root cause of this vulnerability where input is a DNS response received from a DNS server.

3.2.4 CVE-2020-11901 Vulnerability No. 4

The last vulnerability, predictable transaction ID, is characterized in the JSOF's second whitepaper as follows: *"Another vulnerability, only seen in earlier versions of the network stack, is that the DNS transaction ID is incremented serially, and starts at 0, making it easily guessable. This means that the attacker might not need to execute complex man-in-the-middle attack in order to find the value of the transaction ID."* [16, p. 14]

As mentioned already above, message ID is actually a security measure that should help to ensure that spoofing of DNS responses is not an easy task. If it is predictable it becomes virtually useless since the necessary matching of domain name and host address can be performed using the query and its copy in responses.

The root cause of this vulnerability is the use of insufficiently random values.

3.3 Getting Everything Ready

In the following sections, we deal with basic hardware setup, software tools used for the attack, construction of special DNS responses, and methods used in evaluation of results obtained.

3.3.1 Hardware Settings

The experiments described below utilize the same basic setup as described in Section 1.8. The general setup, however, had to be supplemented by some minor additions needed in order to obtain access to the DNS communication of the device.

First of all, we decided to prohibit the use of IPv6 protocol. This decision was arbitrary and it was based on the assumption that the presence of errors in software components handling the "old" IPv4 protocol could be more likely than in the case of IPv6 protocol. The required settings were carried out through the device's web-based interface and can be seen in Figure 3.6.

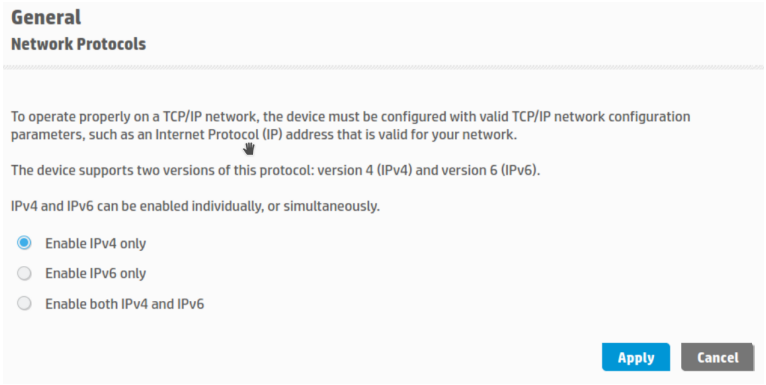

Figure 3.6 Device network protocol settings.

It is worth mentioning, however, that one of our experiments not documented in this study proved that even when the settings "Enable both IPv4 and IPv6" is selected, the device is not able to contact its server if it does not get a correct result for its IPv4 DNS query.

If we can say that the above-described change was an arbitrary decision then the other modification in settings was a sheer necessity. By changing the settings from "Automatic DNS Server" to "Manual DNS Server" and by filling in IP addresses for both the primary and secondary nameservers, we were able to achieve the state when all the DNS messages from the device are routed to our attacking computer. (The IP address typed in the Manual Alternate DNS Server field is actually not used by any device in our network.)

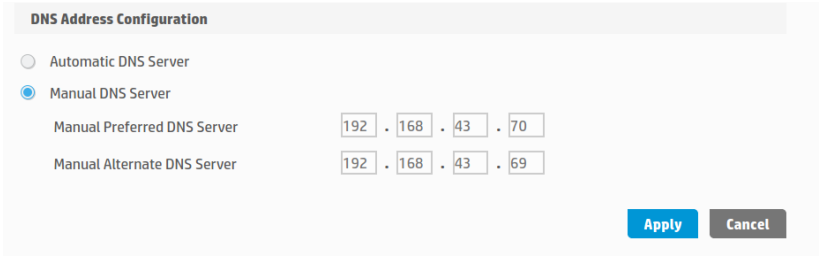

Figure 3.7 Device DNS server settings.

It is clear that a potential attacker "in wild" could hardly have such a comfortable setting for a DNS attack and would have to use other methods to achieve the man-in-the-middle position or to poison DNS cache. Taking into account the scope of the present study, however, we are convinced that such a simplification is acceptable.

And last but not least, the firewall settings on the attacking computer had to be modified in order not to block the incoming DNS messages from the device.

3.3.2 Software Tools Used for Attacks

Our software utilities used to perform the attack consist of one python script (`11901.py`) and one C++ program (`createpacket.cpp`). Although the needed functionality could be certainly ensured by one script or program only, we have come to this arrangement as a result of gradual development and it has proved to be suitable for our purposes and therefore we have retained it.

The C++ program takes care of the creation of responses to be used in the course of attacks. It contains a number of very long lists of values defining individual bytes of many types of responses used in our experiments and it is actually better for us not to place all this obtrusive material in the short and relatively well-arranged python script.

The entire system operates in this way: The device is switched on and the python script is launched. The script creates sockets for communication with the device (listening on port 53 which is used by DNS servers) and with an actual DNS server outside of the local network. When the device sends a DNS query, it is passed to our script. The script either (a) forwards the query to the actual DNS server, collects its reply and sends it back to the device, or (b) extracts the DNS query ID, passes it to the `createpacket` program as a command line argument, collects the created response which was saved by `createpacket` on the hard drive and sends it as a DNS message to the device. The choice of the course of action depends on the command line arguments selected when launching the script as well as on the parameters of the DNS query (some queries always get correct replies).

The command line arguments used with the python script `11901.py` are: (none), `allOK`, `attack`, `noreply`, `wrongip`, `invalidresponse`.

If no command line argument is specified or if the `allOK` argument is specified, the script works as a simple packet forwarder between the device and the actual DNS server.

If the `attack` command line argument is specified, it may be followed by a number specifying the type of response that is to be obtained from the `createpacket` program. Many options were used in our experiments but in the end only three of them are relevant: 1, 2, and 3. If no number is specified, value 1 is used (default value).

Command line arguments `noreply`, `wrongip` and `invalidresponse` do exactly what they suggest (in relation to one particular query only). They were used to simulate the respective comparative scenarios. While the `noreply` option causes the program not to reply to certain queries at all, the `wrongip` and `invalidresponse` options function virtually identically as the `attack` option, i.e. they get a specific malformed response from `createpacket` and send it to the device.

In order to allow a comfortable monitoring, the python script prints out information on received DNS queries and the respective replies. These console outputs are used in the present study as illustrations of the device behaviour.

Both our custom-made tools, i.e. `11901.py` and `createpacket.cpp` (and the compiled version `createpacket`) are included in the attached data medium.

Wireshark was used to monitor and check that the script really does what it should do.

3.3.3 Construction of Special Responses

This subsection presents methods and specifics of construction of DNS response packets used in our experiments either for the attack itself or for triggering various comparative scenarios.

A detailed description of structure of DNS reply was already provided in Section 3.1.2. To achieve maximum flexibility, we have decided not to utilize any specific utility for creating own "tweaked" DNS replies and to rely instead on byte-to-byte construction of these responses using our own tool, `createpacket.cpp` introduced in the previous Section 3.3.2.

As the basis, we use DNS response data captured by Wireshark during a communication with a regular DNS server after a query sent by the python script `11901.py`. Most of our modifications were made on the basis of the DNS server response to A-type query for xmpp008.hpeprint.com. A detailed overview of this original data can be seen in Figure 3.5.

The byte form of this DNS response (with message ID 0xffff) is available on the attached data medium as file `packet000.bin`.

3.3.3.1 DNS Response Used in Invalid DNS Response Scenario

The DNS response used to trigger the invalid DNS response scenario (see below in Section 3.3.4.3) is identical to the correct DNS response described above, except for one byte in the question section. The byte at offset 0x0c (length byte with the value of 0x07) was changed to 0x06. This creates two problems: First, only six characters of the label are read and the next byte is taken as the next length byte. Its value of 0x38 means that the parser is brought somewhere near the end of the question part and generates an error. The second problem is that the question section is not identical to that of the query sent by the device which also results in response rejection. We do not know which mechanism is used first but the result is what we need.

The changed byte can be seen in Figure 3.8.

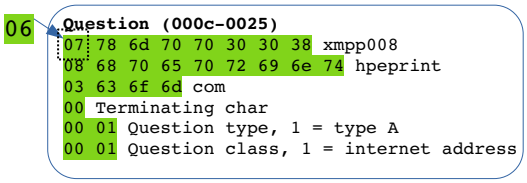

Figure 3.8 The byte changed in the DNS response triggering the invalid response scenario.

The byte form of this DNS response (with message ID 0xffff) is available on the attached data medium as the binary file `packet005.bin`.

3.3.3.2 DNS Response Used in Wrong IP Scenario

The construction of the DNS response used to trigger the wrong IP scenario was also straightforward. All that we had to do was a simple change in the last four bytes that contain the IPv4 address from the A-type record. In our case, we changed the order of the first two values in the IP address as follows: correct address = 0x0f482336, wrong address = 0x480f2336.

Figure 3.9 shows how and which bytes were modified.

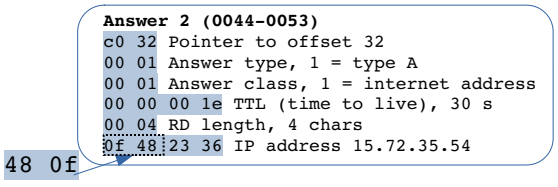

Figure 3.9 The bytes changed in the DNS response triggering the wrong IP scenario.

The byte form of this DNS response (with message ID 0xffff) is available on the attached data medium as the binary file `packet006.bin`.

3.3.3.3 DNS Response Used for Attack on Vulnerability No. 1

CVE-2020-11901 vulnerability No. 1 stems from incorrect handling of the RDLENGTH value present in the CNAME question. Attempts to exploit this vulnerability start with creating a DNS response in which the RDLENGTH value is shorter than the actual length of the domain name. This could possibly lead to heap overflow. We have therefore prepared a DNS response with an

incorrect RDLENGTH value which is set by the byte at offset 0x31. RDLENGTH was 0x12 in the original response, we changed the value to 0x02 for our purposes. Figure 3.10 illustrates this change.

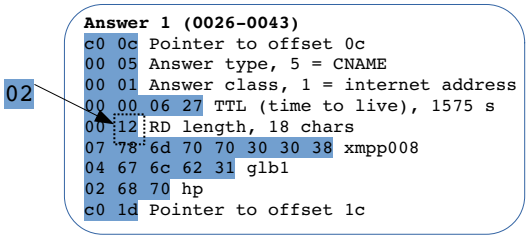

Figure 3.10 The byte changed in the DNS response used for attack on vulnerability No. 1.

After obtaining preliminary results, we have also decided to create a DNS response with RDLENGTH value higher than the correct one, specifically 0x30 instead of 0x12 (value of the byte at offset 0x31), and perform our experiment with this DNS response too. The byte forms of both these DNS responses (with message ID 0xffff) are available on the attached data medium as the binary files `packet001.bin` (with RDLENGTH 0x02) and `packet002.bin` (with RDLENGTH 0x30).

3.3.3.4 DNS Response Used for Attack on Vulnerability No. 2

According to JSOF researchers [16, p. 10], the Treck DNS resolver has a size limitation on UDP packets containing DNS responses of 1460 bytes. Taking into account this limitation, the construction of a response that should overflow the `unsigned short` maximum value seems to be impossible. Surprisingly, it can be done. The principle, which can be described as "compression pointer nesting", is dealt with in detail in [16, pp. 11–13].

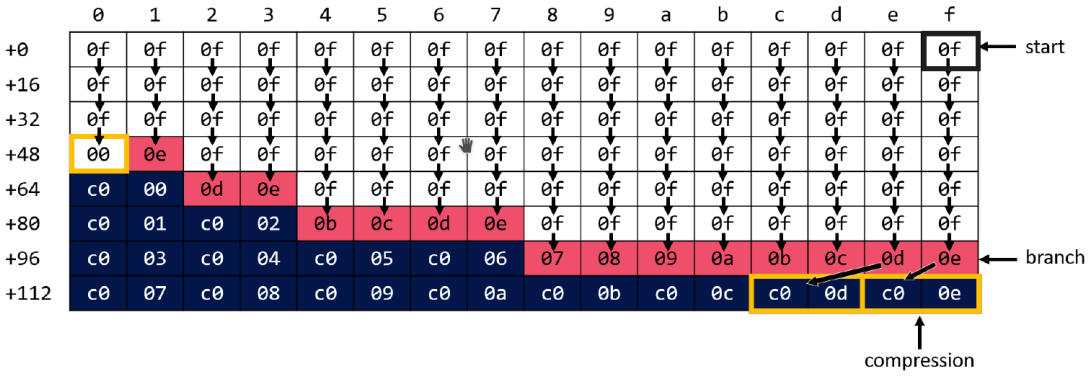

Figure 3.11 DNS compression pointer nesting (image source: [16, p. 12]).

Figure 3.11 illustrates the main idea and basic principle. It shows a matrix of 16 × 8 bytes. Let us assume that a pointer somewhere below this matrix points to the byte in the upper right corner (byte at an offset of 0x0f, marked as "start"). 0x0f is then considered a length byte, its value is 0x0f. The label is therefore 15 bytes long and thus the next length byte we read is at offset 0x1f (the one that is below the starting byte). We continue in the same manner (successfully skipping the 0x00 value at an offset of 0x48, which would otherwise terminate the string) until we reach the byte at an offset of 0x6f (red cell in the last column with the value of 0x0e, marked as "branch"). Based on the value of this length byte, the following label is only 14 bytes long and we land on the byte at offset 0x7e. This byte is not a length byte but pointer

indicator and the next byte (offset 0x7f, or the byte in the bottom right corner) tells us where to go. It points to the byte at offset 0x0e, i.e. first line, last but one byte. We have covered one column and collected already 111 bytes of domain name length ($16 \times 6 + 15$).

Now the action goes on in exactly the same manner as in the previous round. After reaching the last line, we get to another pointer indicator and the new address pointed to. We get again to the first line, one byte left from the previous starting position. (We collected 110 bytes, totalling to 221.)

After some time, "branch" bytes have to go one line up, which is the result of the fact that the pointer structure has the width of two bytes. When we finally go through all the columns, we end up in the first one where the terminating character at offset 0x30 terminates the entire domain name. By then, we have collected $860 + 374 + 157 + 63 + 48 = 1502$ bytes in total. (Length bytes are included in the calculation, we can visualize them as "dots" dividing domain name parts. The only difference is that the number of dots is equal to the number of labels, i.e. you can imagine that each domain name has a dot in its end.)

It is important that this method follows the principle that in a DNS message, any label pointed to by a pointer precedes the position from which it is pointed to. It is also necessary to stress that this idea works only due to the fact that the system does not validate characters in labels (e.g. the most often used character in the full-size matrix is 0x3f, which is "?", a character that cannot be present in a domain name).

We have seen that using a simple matrix of 128 bytes we can construct a domain name of 1502 characters. This is not enough to overflow `unsigned short` but we have not consumed much of the available space yet. If the matrix has 64 columns instead of 16 (64 columns is the maximum number, as there is a length restriction on label length which is actually enforced) and when we add some additional lines (we need 20 in total) we will be able to exceed the value of 65535. And the space consumed for the matrix part of the response is $64 \times 20 = 1280$ bytes only.

This name is then placed in the DNS response in such a manner that it can be accepted during the normal parsing of the DNS response and—at the same time—that it can be referenced from those sections of DNS response that follow. According to JSOF researchers, the most suitable place is therefore the question section which precedes all the answer sections. [16, p. 13]

We can place more than one question in a DNS response, although it is very uncommon and some systems even do not support this feature: *"The idea that only a single question is allowed is sufficiently entrenched that many DNS servers will simply return an error (or fail to response at all) if they receive a query with a question count (QDCOUNT) of more than one."* [48]

The Treck TCP/IP stack, however, apparently supports it (at least the version exploited in the JSOF's second whitepaper). We will therefore construct our malicious DNS response as follows:

1. We construct the matrix-like domain name similarly to the Figure 3.11, but with 64 columns and 20 lines. Based on the sum of bytes of the DNS message header and the first question, we determine the offset value of the byte in the upper right corner (the "start" byte). In our case, the value was 0x66. From this value, we subtract one and then we place the result in the bottom right corner. Other values used in pointers are then calculated on the basis of this value (going from right to left, 1 is always subtracted).

2. We use the original header (only the question count is now 2) and question section from the correct DNS response.

3. After the end of the first question, we place one byte with the value of 0x3f (it is interpreted as a length byte and ensures that we skip to the "start" offset after the first label).

4. After this one byte, the entire 64×20 matrix domain name is placed.

5. The name is followed by four bytes denoting QTYPE and QCLASS (0x0001 and 0x0001).

6. The CNAME type answer follows. For alias name it uses a pointer to the first question (0xc0 0x0c), for canonical name the "start" byte in our matrix domain name is used (0xc0 0x66).

7. A-type answer follows. It can point either to the canonical name or to 0xc0 0x66.

Figure 3.12 presents a basic overview of where is the malicious payload put in the DNS response. The header is yellow, the first question light green (black font), the second question is dark green (white font). The first byte of the second question is marked in red: this is the byte preceding the matrix scheme. The two blue sections represent answer 1 and answer 2.

```
Address  Byte values (hexadecimal)
-------------------------------------------------------------------------------------------
0000:    ffff 8180 0002 0002 0000 0000 0778 6d70 7030 3038 0868 7065 7072 696e 7403 636f
0020:    6d00 0001 0001 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f
0040:    3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f

                  24 lines containing only 0x3f bytes not displayed

0360:    3f3f 3f3f 3f3f 3f00 3e3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f
0380:    3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f
03a0:    3f3f 3f3f 3f3f 3fc0 273d 3e3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f
03c0:    3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f
03e0:    3f3f 3f3f 3f3f 3fc0 28c0 293b 3c3d 3e3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f
0400:    3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f
0420:    3f3f 3f3f 3f3f 3fc0 2ac0 2bc0 2cc0 2d37 3839 3a3b 3c3d 3e3f 3f3f 3f3f 3f3f 3f3f
0440:    3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f
0460:    3f3f 3f3f 3f3f 3fc0 2ec0 2fc0 30c0 31c0 32c0 33c0 34c0 352f 3031 3233 3435 3637
0480:    3839 3a3b 3c3d 3e3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f 3f3f
04a0:    3f3f 3f3f 3f3f 3fc0 36c0 37c0 38c0 39c0 3ac0 3bc0 3cc0 3dc0 3ec0 3fc0 40c0 41c0
04c0:    42c0 43c0 44c0 451f 2021 2223 2425 2627 2829 2a2b 2c2d 2e2f 3031 3233 3435 3637
04e0:    3839 3a3b 3c3d 3ec0 46c0 47c0 48c0 49c0 4ac0 4bc0 4cc0 4dc0 4ec0 4fc0 50c0 51c0
0500:    52c0 53c0 54c0 55c0 56c0 57c0 58c0 59c0 5ac0 5bc0 5cc0 5dc0 5ec0 5fc0 60c0 61c0
0520:    62c0 63c0 64c0 6500 0100 01c0 0c00 0500 0100 0001 ba00 02c0 66c0 6600 0100 0100
0540:    0000 1e00 040f 4823 36
```

Figure 3.12 Overview of the placement of the malicious payload in the DNS message used for attack on vulnerability No. 2.

The byte form of this DNS response (with message ID 0xffff) is available on the attached data medium as the binary file `packet003.bin`.

3.3.3.5 DNS Response Used for Attack on Vulnerability No. 3

To exploit this vulnerability, we need to change the data in such a manner that the parser processing label length and label data bytes overflows behind the end of the packet. Figure 3.13 illustrates this change.

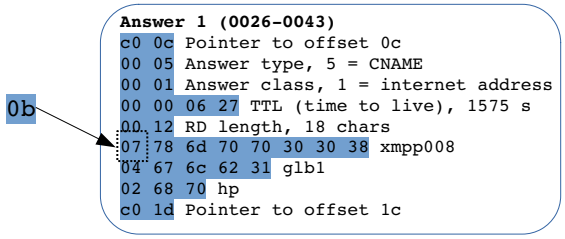

Figure 3.13 The byte changed in the DNS response used for attack on vulnerability No. 3.

By changing the byte at offset 0x32 from 0x07 to 0x0b, we force the system to interpret the next 11 bytes as a label and the byte that follows them (offset 0x3e, value of 0x31) as a label length indicator. Its value (0x31) then points to the space behind the bounds of the DNS response

and the UDP packet. The offset of the last byte in the DNS response is 0x53, while this length byte indicates that the next label starts at offset 0x70. If the vulnerability is present in the system, we would thus overflow to a "forbidden area" by 29 bytes, potentially resulting in error.

3.3.4 Comparative Analysis of Behaviour

To evaluate the response of the device to the attempted attack we have decided to apply the following method: (1) we establish a set of possible scenarios, and describe behaviour of the system in each of them; (2) we compare the observed behaviour under attack to the individual scenarios and try to find the one that is identical.

It is also necessary to clearly define what do we call "behaviour" here. Since we have no access to the internal subsystems of the device, we can observe its behaviour only on the basis of its communication with the external world through (1) data network, (2) its physical interface (display and indicators on its panel), and (3) its web-based interface. As far as the device's data network communication is concerned, we monitor DNS traffic only.

Our set of scenarios comprises the following options:

- Correct behaviour of DNS system: each DNS query of the device is followed by a valid response within reasonable period of time.
- No reply to A-type queries for xmpp008.hpeprint.com: these queries are left unanswered, other DNS queries sent by the device are followed by valid responses.
- Invalid response to A-type query for xmpp008.hpeprint.com: in this case, the reply is recognized by the DNS resolver as invalid.
- Wrong IP address received in response to A-type query for xmpp008.hpeprint.com: the DNS query is followed by a response, which is valid. The IP address contained in it, however, does not help the device to connect to the respective Internet resource.

We believe that the behaviour of the device under attack should show a similarity to one of these scenarios. The result should thus allow us to find at least an estimate of a conclusion concerning the handling of the malformed packet used for the attack within the device's system.

After the device boots, it sends several different types of DNS queries. We have selected one of them as the experimental query, while the other queries (if they are under the respective scenario sent at all) are normally replied to using valid responses. The actual attack is then also performed in response to that particular query, though in the end, we try also a different approach in order to reveal a potential heap overflow.

After powering up the device and after successfully connecting to the registered Wi-Fi network, the Wireless light stops flashing and remains on. Shortly afterwards the device starts sending DNS queries to a DNS server.

3.3.4.1 Correct Behaviour Scenario

In the default mode, which is activated by launching the script without any command-line argument or with the `allOK` argument, five DNS queries are sent to the server and all of them are responded using valid DNS replies:

```
 Time    ID     Type Name                    Response
-------------------------------------------------------------
  0.000 32622 AAAA xmpp008.hpeprint.com. Valid response
  0.329 41877    A xmpp008.hpeprint.com. Valid response
  0.472 29377 AAAA xmpp008.hpeprint.com. Valid response
  4.549 42143 AAAA chat.hpeprint.com.    Valid response
  4.789 60545    A chat.hpeprint.com.    Valid response
```

The Web Services light on the device's control panel stops flashing and remains on. The web-based interface of the device shows that the device is connected to the web services (see Figure 3.14).

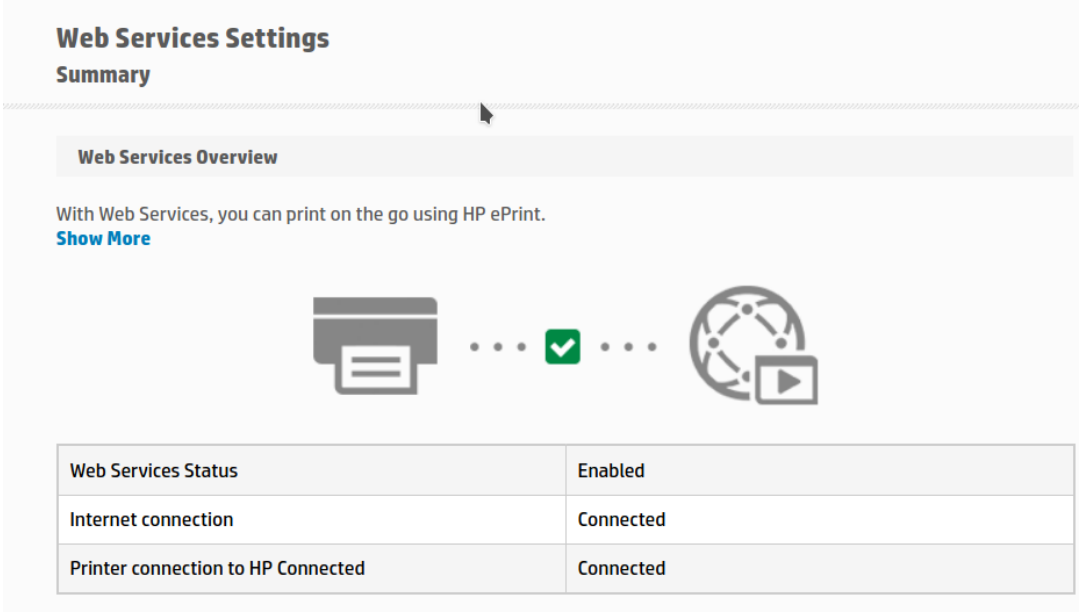

Figure 3.14 Device web interface confirming the connection to the cloud services.

3.3.4.2 No-Reply Scenario

In this scenario, which is activated by the `noreply` command-line argument, the first query (AAAA-type query for xmpp008.hpeprint.com), gets a valid response. It is followed by A-type query for xmpp008.hpeprint.com. There is no response to this query and the device resends the same query (with the same ID number) another four times in approximately 7 seconds intervals. After five attempts, the first query (AAAA type) is repeated and the entire process starts over. One round of such DNS communication looks like this:

```
 Time    ID     Type Name                    Response
-------------------------------------------------------------
  0.000 32625 AAAA xmpp008.hpeprint.com. Valid response
  0.319 17819    A xmpp008.hpeprint.com. No response
  7.233 17819    A xmpp008.hpeprint.com. No response
 14.281 17819    A xmpp008.hpeprint.com. No response
 21.283 17819    A xmpp008.hpeprint.com. No response
 28.233 17819    A xmpp008.hpeprint.com. No response
```

The Web Services light on the device's control panel does not stop flashing. The web-based interface of the device shows that the device is not connected to the web services (see Figure 3.15).

3.3.4.3 Invalid DNS Response Scenario

In this scenario, which is activated by the `invalidresponse` command-line argument, the device identifies the received response as invalid (it has an incorrect byte in domain name in the question section). The device performs five attempts to get a correct response, in each case both the

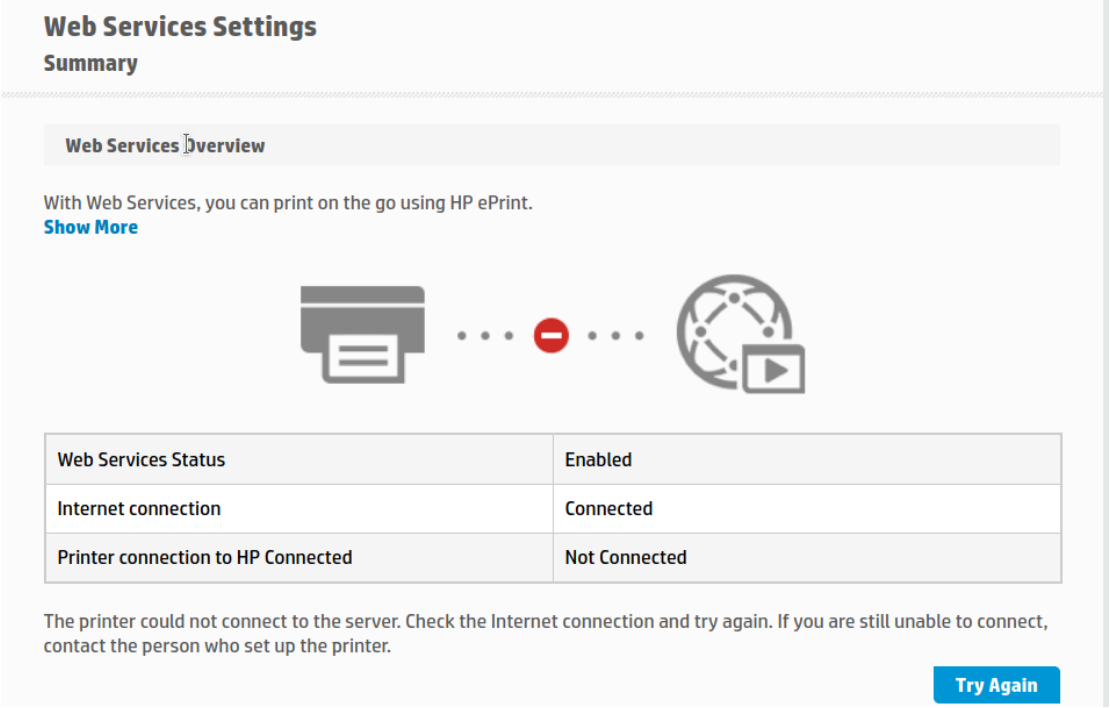

Figure 3.15 Device web interface indicating failure to connect to the cloud services.

AAAA-type and the A-type queries are resent. After five attempts, the device stops trying and sends queries for another domain name.

The pattern of the DNS communication is as follows:

```
 Time    ID     Type  Name                    Response
-------------------------------------------------------------
  0.000  64496  AAAA  xmpp008.hpeprint.com.  Valid response
  0.417   9955     A  xmpp008.hpeprint.com.  Invalid response
  0.915  43600  AAAA  xmpp008.hpeprint.com.  Valid response
  1.261  62667     A  xmpp008.hpeprint.com.  Invalid response
  1.414  38118  AAAA  xmpp008.hpeprint.com.  Valid response
  1.673   7267     A  xmpp008.hpeprint.com.  Invalid response
  1.939  19239  AAAA  xmpp008.hpeprint.com.  Valid response
  2.416  27935     A  xmpp008.hpeprint.com.  Invalid response
  2.664  13894  AAAA  xmpp008.hpeprint.com.  Valid response
  2.957  57825     A  xmpp008.hpeprint.com.  Invalid response
  3.992  24045  AAAA  chat.hpeprint.com.     Valid response
  4.282  63753     A  chat.hpeprint.com.     Valid response
```

The Web Services light on the device's control panel does not stop flashing. The web-based interface of the device shows that the device is not connected to the web services (see Figure 3.15).

3.3.4.4 Wrong IP Address Scenario

In this scenario, which is activated by the `wrongip` command-line argument, the script replies to the DNS requests of the device of the A type for xmpp008.hpeprint.com by a valid DNS response, which contains an incorrect IP address.

The pattern of the DNS communication is as follows:

```
 Time    ID     Type Name                    Response
-------------------------------------------------------------
  0.000 64466 AAAA xmpp008.hpeprint.com. Valid response
  0.467 12495    A xmpp008.hpeprint.com. Resp. OK, wrong IP
  0.954  8695 AAAA xmpp008.hpeprint.com. Valid response
  4.047 10757 AAAA chat.hpeprint.com.    Valid response
  4.536 12715    A chat.hpeprint.com.    Valid response
 28.786 49521 AAAA xmpp008.hpeprint.com. Valid response
 56.505 23522 AAAA xmpp008.hpeprint.com. Valid response
 56.773 12326    A xmpp008.hpeprint.com. Resp. OK, wrong IP
 84.724 51936 AAAA xmpp008.hpeprint.com. Valid response
```

The Web Services light on the device's control panel does not stop flashing. The web-based interface of the device shows that the device is not connected to the web services (see Figure 3.15).

3.3.5 Revealing Heap Overflow

With the exception of vulnerability No. 4, predictable transaction ID, CVE-2020-11901 vulnerabilities involve a potential heap overflow. Heap overflow is a specific case of buffer overflow that occurs on the heap. Unlike stack overflows, however, heap overflows are more difficult to detect and exploit.

When a buffer overflow occurs on the stack, the return address needed after the completion of the current sub-program can be overwritten. This is not the only path for exploiting a stack-based buffer overflow (e.g. exception registration structures stored on the stack can be overwritten too, as pointed out by Koziol et al in [49]). In case of an accidental occurrence of such a buffer overflow, however, the incorrect value in place of the return address usually makes it impossible to pass unseen. A crash of the system is the typical result of a buffer overflow with no crafted shellcode utilized.

A heap-based overflow, on the other hand, can easily occur undetected. It may happen that the overwritten memory is not currently used or that although some important metadata were overwritten, they were not needed, as e.g. no `free()` instruction was called on the respective memory chunk.

Once again we have to stress that we are not able to get to the internal memory of the device we use for our experiments. We have neither the source code nor the binary executables of any software it runs. We do not know the version of the TCP/IP Stack in the device. We can only observe its behaviour, including its communication with outer world. It is therefore quite clear that any attempt to exploit a heap overflow in order to achieve remote code execution would need skills and experience that cannot be expected at our level of expertise.

However, a detection of heap overflow manifesting itself in the form of a crash of the device's system could be achievable. The prerequisite is, of course, that the vulnerability in question is actually present in the device's firmware.

Our approach and methods are described in detail in Section 3.4.2.

3.4 Attack Implementation

Observations of the system's behaviour under the different attacks are described in this section. Relevant considerations are presented and possible explanations are proposed.

3.4.1 CVE-2020-11901 Vulnerability No. 1

We tested our device for vulnerability No. 1, bad RDLENGTH leads to heap overflow, using our testing script `11901.py` (command line arguments `attack 1`). The response used for the attack was a valid response packet with actual correct data, but with incorrect RDLENGTH value in the CNAME section (a value of 0x02 was used instead of 0x12).

The DNS communication of the device showed the following behaviour:

```
 Time    ID     Type Name                    Response
-------------------------------------------------------------
  0.000 48805 AAAA xmpp008.hpeprint.com. Valid response
  0.224 49407    A xmpp008.hpeprint.com. Malicious response
  0.567 46879 AAAA xmpp008.hpeprint.com. Valid response
  0.724 13936    A xmpp008.hpeprint.com. Malicious response
  1.023 14880 AAAA xmpp008.hpeprint.com. Valid response
  1.223 21432    A xmpp008.hpeprint.com. Malicious response
  1.587 64771 AAAA xmpp008.hpeprint.com. Valid response
  1.741 43685    A xmpp008.hpeprint.com. Malicious response
  2.040  5980 AAAA xmpp008.hpeprint.com. Valid response
  2.240  1433    A xmpp008.hpeprint.com. Malicious response
  3.299 40853 AAAA chat.hpeprint.com.    Valid response
  3.589 57923    A chat.hpeprint.com.    Valid response
```

The Web Services light on the device's control panel does not stop flashing. The web-based interface of the device shows that the device is not connected to the web services (see Figure 3.15).

The recorded data indicate that the device considers the DNS response with RDLENGTH value smaller than the correct value as invalid and rejects it. In another experiment with another DNS response, in which RDLENGTH value was larger than the actual correct value (0x30 instead of 0x12), the behaviour of the device was the same, i.e. the DNS response was rejected.

We can imagine two possible reasons for the rejection of this DNS response:

1. The system reads the RDLENGTH value and somehow compares it to the actual length of the domain name. When a discrepancy is found, the system rejects the package.
2. The system reads the RDLENGTH value and segments the subsequent bytes on its basis, i.e. either ends the domain name too early or too late, resulting in a corruption of the rest of the data.

We do not see any possibility, however, that such a behaviour could be a result of a heap overflow. That would lead to a system crash or reboot, not to a mere rejection of the respective DNS response.

3.4.2 CVE-2020-11901 Vulnerability No. 2

We tested our device for vulnerability No. 2, from integer overflow to heap overflow, using our testing script `11901.py` (command line arguments `attack 2`). For the attack we used the DNS response described above in Section 3.3.3.4.

DNS communication of the device showed the following behaviour:

```
 Time    ID     Type Name                    Response
-------------------------------------------------------------
  0.000 32625 AAAA xmpp008.hpeprint.com. Valid response
  0.318 31461    A xmpp008.hpeprint.com. Malicious response
  0.469 62329 AAAA xmpp008.hpeprint.com. Valid response
  4.286 24027 AAAA chat.hpeprint.com.    Valid response
  4.526 30627    A chat.hpeprint.com.    Valid response
```

The Web Services light on the device's control panel stops flashing and remains on. The web-based interface of the device shows that the device is connected to the web services (see Figure 3.14).

Based on these observations, the behaviour of the system under our attack perfectly corresponds to the correct behaviour as described in Section 3.3.4.1 above. We have to keep in mind, however, that our results have not provided us with any indication whether or not a heap-based overflow occurred during our experiment so far. We therefore have to adopt a more sophisticated targeted approach to shape the heap in such a way that the potential overflow would actually lead to a measurable event (probably a crash).

What we need is a sequence of two DNS entries (specifically domain names which are pointed from DNS entries stored in another place of the heap), which would be allocated on the heap one after another. To achieve this goal, we need two allocations of similar size allocated without much delay between them. In this way, we believe, we can maximize the probability that the two memory chunks are allocated in the same heap bucket (i.e. memory area) one after another. The first of these domain names would be then overflown to the following one. This would possibly lead to a crash at the moment a `free()` command is called on the memory allocated for the second domain name.

As a starting point, we take the normal behaviour of the device. After a reboot, it sends five DNS requests, as illustrated in this output from console:

```
 Time    ID     Type Name                    Response
-------------------------------------------------------------
  0.000 32612 AAAA xmpp008.hpeprint.com. Valid response
  0.468 32492    A xmpp008.hpeprint.com. Valid response
  0.746 13594 AAAA xmpp008.hpeprint.com. Valid response
  4.762 57623 AAAA chat.hpeprint.com.    Valid response
  5.002 60806    A chat.hpeprint.com.    Valid response
```

When we look at the entries we can assume the following sequence of actions:

1. Memory on the heap is allocated for the domain name returned in the CNAME answer to the AAAA-type query for xmpp008.hpeprint.com.
2. Memory on the heap is allocated for the domain name returned in the CNAME answer to the A-type query for xmpp008.hpeprint.com.
3. Memory allocated on the heap in (1) is freed (`free()` is called).
4. Memory on the heap is allocated for the domain name returned in the CNAME answer to the AAAA-type query for xmpp008.hpeprint.com.
5. Memory on the heap is allocated for the domain name returned in the CNAME answer to the AAAA-type query for chat.hpeprint.com.
6. Memory on the heap is allocated for the domain name returned in the CNAME answer to the A-type query for chat.hpeprint.com.

We can see that the `free()` call is probably used just once and in connection with the first allocated name. This would not allow us to perform the intended actions in the planned order as we need a `free()` call for memory allocated AFTER the memory chunk containing the overflowing domain name.

We have found a solution though. When the Wireless button of the device is pressed, it disconnects the device from the Internet. A repeated press connects it again, leading to a new series of 3 DNS queries as depicted below (similar effect has clicking on the Try Again button on the device's web interface, see Figure 3.15):

```
 Time    ID     Type Name                     Response
-------------------------------------------------------------
  0.000 64885 AAAA xmpp008.hpeprint.com. Valid response
  0.115 10348    A xmpp008.hpeprint.com. Valid response
  0.426 18676 AAAA xmpp008.hpeprint.com. Valid response
```

We can see that the queries related to xmpp008.hpeprint.com are repeated even though the system did not reboot. That means that the memory with the original entries was most probably freed using the `free()` call, this time not only in relation to the AAAA-type record but the A-type record as well. And the memory for the domain name of this A-type record was allocated after that for the AAAA-type record. If a heap overflow is triggered by the domain name returned in the AAAA-type record, and if it is sufficiently large, it should corrupt the memory chunk storing the domain name returned in the A-type record, which should then lead to a system crash when a `free()` command is called for the latter memory chunk.

To ensure that domain names of both the overflowing and to-be-overflown memory chunks are placed in the same memory area (bucket), we use the same domain names in both of them. This means that the second domain name also overflows the memory allocated on the heap and increases the probability of crash (it can e.g. overflow to unmapped memory or another unrelated chunk that could be freed later).

It is also important to point out that if the device contains the vulnerability CVE-2020-11901, No. 2, "from integer overflow to heap overflow", the allocated memory would be significantly smaller than the actual length of the domain name due to the integer overflow (it would be actually smaller by those 65536 bytes that "got lost" in overflowing the `unsigned short integer` value). This amount of overflow should be sufficient enough to compensate for any gaps caused by memory alignment or other factors.

Based on this strategy, the following plan was created:

1. We launch our script in the `allOK` mode, which means that after the device boots, the first five DNS queries are replied using correct DNS data from the regular server. This will allow the device to establish connection with the cloud system and enter a stable mode of operation (mostly waiting). Then we stop the script.

2. We launch our script (actually we prepared its slightly modified version specifically for this purpose, `v2_11901.py`) in the `attack` mode, which means that any new DNS queries will be replied with malicious replies utilizing the integer-overflow responses. We then press the device's Wireless button twice (with a few seconds waiting in between) and let the system send its three DNS queries and the script reply to them.

3. We can repeat the step 2 if nothing happens to increase the probability of success. It is again important to stress that we have no actual knowledge of the memory management system used in the device and therefore it is quite legitimate to repeat the process with the intent to increase the probability of achieving an optimum arrangement of memory chunks on the heap.

When we performed the experiment on the basis of the above-described plan, nothing happened. The recorded pattern of behaviour can be seen below. The first part documents the response of the device to the step 1:

```
  Time    ID     Type Name                      Response
-------------------------------------------------------------
  0.000 64363 AAAA xmpp008.hpeprint.com. Valid response
  0.285 43300    A xmpp008.hpeprint.com. Valid response
  0.426  1743 AAAA xmpp008.hpeprint.com. Valid response
  4.585  7691 AAAA chat.hpeprint.com.    Valid response
  4.826 26395    A chat.hpeprint.com.    Valid response
```

And the step 2 follows:

```
 Time    ID     Type Name                    Response
-------------------------------------------------------------
  0.000 64737 AAAA xmpp008.hpeprint.com. Malicious response
  0.137 29350    A xmpp008.hpeprint.com. Malicious response
  0.400 21409 AAAA xmpp008.hpeprint.com. Malicious response
```

The device did not crash in the manner observed in case of CVE-2020-11898 and did not reboot (which could be another symptom of a crash) as that would be visible in the console output of the script (new series of 5 DNS queries would be sent). We therefore firmly believe that the vulnerability in question, viz. CVE-2020-11901, No. 2, from integer overflow to heap overflow, is not present in the device we tested.

3.4.3 CVE-2020-11901 Vulnerability No. 3

We tested our device for vulnerability No. 3, read out-of-bounds, using our testing script `11901.py` (command line arguments `attack 3`). The DNS response used for the attack was the one described above in Section 3.3.3.4.

DNS communication of the device showed the following behaviour:

```
  Time    ID     Type Name                     Response
-------------------------------------------------------------
  0.000  7603 AAAA xmpp008.hpeprint.com. Valid response
  0.242 20242    A xmpp008.hpeprint.com. Malicious response
  0.494 27117 AAAA xmpp008.hpeprint.com. Valid response
  3.984 23078 AAAA chat.hpeprint.com.    Valid response
  4.226 21537    A chat.hpeprint.com.    Valid response
```

The Web Services light on the device's control panel stops flashing and remains on. The web-based interface of the device shows that the device is connected to the web services (see Figure 3.14).

It is apparent that the device did not have any problems to overcome our attempt to lead it to an error, possibly resulting in crash. It seems important to us that in the case of the above-described DNS response the RDLENGTH value was left unchanged (the correct value of 0x12 was used). When we tried the same attempt with changed RDLENGTH value, the DNS response was rejected in the same way as we observed when attempting to attack the vulnerability No. 1.

3.4.4 CVE-2020-11901 Vulnerability No. 4

During experimenting with DNS responses, we have captured many series of DNS queries sent by the device. Although it seems that there is no apparent regularity in message ID numbering (numbers neither start from zero nor show an easily predictable pattern), we have decided to perform some basic statistical tests (frequency test and runs test) on three samples of sequences of generated ID numbers examining their randomness. The tested sequences, test descriptions and results are presented in Appendix B. Based on the results, we can conclude that the presence of the CVE-2020-11901 vulnerability No. 4 in the tested device can be ruled out.

3.5 Results and Their Interpretation

The behaviour of the device facing various types of attacks placed in the CNAME type answer section can be summarized as follows:

1. The device is perfectly resistant to attacks based on the processing of the canonical domain name. DNS responses prepared for attacks on the vulnerabilities 2 and 3 do not lead to any deviation from normal behaviour.
2. Any DNS response containing a wrong RDLENGTH data in the CNAME type question section results in a rejection of the response. This concerns both values that are lower than the correct value and those that are higher.

We can imagine various possible approaches to DNS message parsing that could lead to this particular behaviour. We do not see, however, any reliable method how to confirm or reject such hypotheses and such effort would likely be out of the scope of this study. Therefore we just conclude that although this behaviour seems to be rather strange, it ensures quite a strong protection against the vulnerabilities of the CVE-2020-11901 group.

3.6 Risk Mitigation Strategies

Since our device has proven to be resistant against our efforts to exploit the CVE-2020-11901 vulnerabilities, risk mitigation strategies can be defined at a general level only.

1. The user should change the default credentials (this is a general advice not directly related to this type of attack, we place it here due to its extreme importance). In our particular case, the device allows the user to set a password in order to protect the system settings. It should be noted that a change in settings was one of the preconditions making the attacks much easier for us (by changing the network settings, we rerouted DNS messages to the attacking computer).
2. The device should run the most up-to-date version of firmware.
3. A vulnerable device should not directly face the Internet. A properly configured internal network should be able to prevent such an attack from the outside.
4. A vulnerable device should be connected to trusted networks only. The user must consider who else can get access to the network to which the device is connected.

3.7 Behaviour After a Firmware Update

After performing and evaluating all the experiments, we performed an upgrade to the current version of firmware issued after the discovery of the Ripple20 vulnerabilities (LAP1FN2020BR). The behaviour of the device remained unchanged, i.e. correct.

Chapter 4
Conclusion

We have performed experiments attempting to reproduce the attacks on two Ripple20 vulnerabilities, specifically CVE-2020-11898 and CVE-2020-11901, the latter being a collection of four specific sub-cases. We are convinced that we succeeded in fulfilling the necessary preconditions, preparing the hardware environment, creating software utilities as well as implementing the actual attacks.

As regards the CVE-2020-11898 vulnerability, our attack did not yield the same result as that of JSOF researchers, viz. "information leak". This is not because of incorrect performance of the attack but due to the specific implementation of the related or other functions in our IoT device. The result was different but still tangible as we achieved a "denial of service".

Concerning the CVE-2020-11901 vulnerability, our IoT device has proven to be virtually impenetrable. We are convinced that we were able to sufficiently test all of the four vulnerabilities covered by this CVE number. Our results show that this vulnerability (or none of them, to be precise) is actually present in our tested device.

From a security perspective, our IoT device has proven to be imperfect (it should be noted, however, that the firmware upgrade has resolved the problem). When experimenting with it, we have in fact discovered another reliable method of achieving a permanent denial of service (until hardware restart) by flooding one of the device's open ports with packets. This method is not described elsewhere in this study since we consider it as a "side product" of our research only. It highlights, however, our other findings concerning the immunity of the device against attacks.

The general assumption that the Internet of Things demonstrates cybersecurity issues has been basically confirmed by our study. The fact that some of the tested vulnerabilities have not been found in the device is certainly positive. On the other hand, it illustrates how difficult it may be to trace the problems of embedded software components, like protocol stacks, in the wide, diverse and difficult-to-map landscape of actual implementations.

Much attention has been devoted to this area but there is still a lot to be done. It is significant that many of the quoted papers appeared during the last months or even weeks. It can be expected that such attention of cybersecurity professionals will have an impact on the manufacturers, operators and owners of IoT devices as well. Otherwise, the boom of connected services and devices surrounding our lives might give us a very grim perspective. Let us conclude with words of the pioneer of white-hat hacking and the discoverer of a scary systemic error in DNS, the late Daniel Kaminsky: *"The internet was never designed to be secure. The internet was designed to move pictures of cats. We are very good at moving pictures of cats. [...] We didn't think you'd be moving trillions of dollars onto this. What are we going to do? And here's the answer: Some of us got to go out and fix it."* [50]

Appendix A

List of All Ripple20 Vulnerabilities

The table below has been compiled and rearranged using data from the Ripple20 initial announcement. All the text in this Appendix A is a verbatim quote from [26].

CVE ID	CVSSv3	Description	Fixed on Version
CVE-2020-11896	10	This vulnerability can be triggered by sending multiple malformed IPv4 packets to a device supporting IPv4 tunneling. It affects any device running Treck with a specific configuration. It can allow a stable remote code execution and has been demonstrated on a Digi International device. Variants of this Issue can be triggered to cause a Denial of Service or a persistent Denial of Service, requiring a hard reset. Remote Code Execution	6.0.1.66 (release 30/03/2020)
CVE-2020-11897	10	This vulnerability can be triggered by sending multiple malformed IPv6 packets to a device. It affects any device running an older version of Treck with IPv6 support, and was previously fixed as a routine code change. It can potentially allow a stable remote code execution. Out-of-Bounds Write	5.0.1.35 (release 04/06/2009)
CVE-2020-11898	9.1	Improper Handling of Length Parameter Inconsistency (CWE-130) in IPv4/ICMPv4 component, when handling a packet sent by an unauthorized network attacker. Possible Exposure of Sensitive Information (CWE-200)	6.0.1.66 (release 03/03/2020)

CVE ID	CVSSv3	Description	Fixed on Version
CVE-2020-11899	5.4	Improper Input Validation (CWE-20) in IPv6 component when handling a packet sent by an unauthorized network attacker. Possible Out-of-bounds Read (CWE-125), and Possible Denial of Service.	6.0.1.66 (release 03/03/20)
CVE-2020-11900	8.2	Possible Double Free (CWE-415) in IPv4 tunneling component when handling a packet sent by a network attacker. Use After Free (CWE-416)	6.0.1.41 (release 10/15/2014)
CVE-2020-11901	9	This vulnerability can be triggered by answering a single DNS request made from the device. It affects any device running Treck with DNS support and we have demonstrated that it can be used to perform Remote Code Execution on a Schneider Electric APC UPS. In our opinion this is the most severe of the vulnerabilities despite having a CVSS score of 9.0, due to the fact that DNS requests may leave the network in which the device is located, and a sophisticated attacker may be able to use this vulnerability to take over a device from outside the network through DNS cache poisoning, or other methods. Thus an attacker can infiltrate the network and take over the device with one vulnerability bypassing any security measures. The malformed packet is almost completely RFC compliant, and so it will likely be difficult for security products such as firewalls to detect this vulnerability. On very old versions of the Treck stack, still running on some devices, the transaction ID is not randomized making the attack easier. Remote Code Execution	6.0.1.66 (release 03/03/2020)
CVE-2020-11902	7.3	Improper Input Validation (CWE-20) in IPv6OverIPv4 tunneling component when handling a packet sent by an unauthorized network attacker. Possible Out-of-bounds Read (CWE-125)	6.0.1.66 (release 03/03/20)

CVE ID	CVSSv3	Description	Fixed on Version
CVE-2020-11903	5.3	Possible Out-of-bounds Read (CWE-125) in DHCP component when handling a packet sent by an unauthorized network attacker. Possible Exposure of Sensitive Information (CWE-200)	6.0.1.28 (release 10/10/12)
CVE-2020-11904	5.6	Possible Integer Overflow or Wraparound (CWE-190) in Memory Allocation component when handling a packet sent by an unauthorized network attacker Possible Out-of-Bounds Write (CWE-787)	6.0.1.66 (release 03/03/2020)
CVE-2020-11905	5.3	Possible Out-of-bounds Read (CWE-125) in DHCPv6 component when handling a packet sent by an unauthorized network attacker. Possible Exposure of Sensitive Information (CWE-200)	6.0.1.66 (release 03/03/20)
CVE-2020-11906	5	Improper Input Validation (CWE-20) in Ethernet Link Layer component from a packet sent by an unauthorized user. Integer Underflow (CWE-191)	6.0.1.66 (release 03/03/20)
CVE-2020-11907	5	Improper Handling of Length Parameter Inconsistency (CWE-130) in TCP component, from a packet sent by an unauthorized network attacker Integer Underflow (CWE-191)	6.0.1.66 (release 03/03/20)
CVE-2020-11908	3.1	Improper Null Termination (CWE-170) in DHCP component when handling a packet sent by an unauthorized network attacker. Possible Exposure of Sensitive Information (CWE-200)	4.7.1.27 (release 11/08/07)
CVE-2020-11909	3.7	Improper Input Validation (CWE-20) in IPv4 component when handling a packet sent by an unauthorized network attacker. Integer Underflow (CWE-191)	6.0.1.66 (release 03/03/20)
CVE-2020-11910	3.7	Improper Input Validation (CWE-20) in ICMPv4 component when handling a packet sent by an unauthorized network attacker. Possible Out-of-bounds Read (CWE-125)	6.0.1.66 (release 03/03/20)

CVE ID	CVSSv3	Description	Fixed on Version
CVE-2020-11911	3.7	Improper Access Control (CWE-284) in ICMPv4 component when handling a packet sent by an unauthorized network attacker. Incorrect Permission Assignment for Critical Resource (CWE-732)	6.0.1.66 (release 03/03/20)
CVE-2020-11912	3.7	Improper Input Validation (CWE-20) in TCP component when handling a packet sent by an unauthorized network attacker. Possible Out-of-bounds Read (CWE-125)	6.0.1.66 (release 03/03/20)
CVE-2020-11913	3.7	Improper Input Validation (CWE-20) in IPv6 component when handling a packet sent by an unauthorized network attacker. Possible Out-of-bounds Read (CWE-125)	6.0.1.66 (release 03/03/20)
CVE-2020-11914	3.1	Improper Input Validation (CWE-20) in ARP component when handling a packet sent by an unauthorized network attacker. Possible Out-of-bounds Read (CWE-125)	6.0.1.66 (release 03/03/20)

Appendix B

Statistical Analysis of Generated Random Numbers

In order to confirm or reject our hypothesis concerning the "randomness" of generated number sequences used for DNS message ID numbers, a statistical analysis had to be performed. The respective data, the tests and their results are described in this appendix.

For the purposes of this chapter, the R project for statistical computing (language and environment) has been used[1]. We do not explain the mathematical principles of the individual functions and rely on the predefined functions and features of the R language and environment.

B.1 Analysed Numbers

The sequences of IDs used for the analysis were obtained using the `11901.py` script with the `invalidresponse` command-line argument. After recording 12 DNS queries from the device, we disconnected and reconnected the device to the network (five times in total) by pushing the Wireless button twice and obtained 10 more DNS queries each time. We thus collected a sequence of 62 DNS queries generated in a row, without a reboot in-between.

In this way, we generated three sequences of 62 numbers and performed the statistical tests on each of them separately. These three sequences (s1, s2, and s3) are listed below as R commands used to create the respective numeric vectors:

```
s1 <- c(64441, 18376, 62841, 26621, 13492, 23813, 56105, 64649, 13699,
47743, 18064, 48523, 64804, 27203, 61092, 160, 53424, 7207, 1489, 24183,
42510, 7822, 64006, 55925, 2446, 50515, 25593, 47843, 45392, 38932,
34571, 59427, 15762, 45293, 43990, 45043, 44077, 57387, 58047, 56417,
3423, 20530, 64693, 35704, 20001, 12770, 49392, 17391, 24825, 24726,
22288, 3808, 8021, 23091, 38475, 9169, 13736, 47161, 7805, 23476, 29901,
15770)
```

```
s2 <- c(64385, 23791, 14096, 16549, 16183, 46978, 53257, 44090, 28272,
19412, 29355, 28989, 18683, 29050, 52745, 24805, 58020, 55116, 42050,
13984, 22514, 7216, 11007, 2392, 35530, 21563, 8168, 18929, 1862, 60534,
54849, 30759, 13386, 34047, 35131, 14182, 61776, 27618, 3827, 49688,
65395, 49828, 32539, 6255, 9129, 36737, 63117, 57698, 64887, 59579,
62327, 7377, 56657, 17949, 41319, 56226, 57570, 60250, 14871, 64600,
57295, 397)
```

[1] https://www.r-project.org

```
s3 <- c(64377, 29218, 38413, 5780, 49183, 13353, 55934, 46680, 60198,
38536, 27251, 2274, 64859, 64155, 19088, 48259, 25519, 2306, 18900,
25752, 61636, 50270, 64238, 24757, 19134, 25896, 26193, 11721, 49720,
24926, 38693, 47241, 64828, 34158, 8297, 10938, 15806, 9575, 38917,
23384, 63610, 52064, 63858, 141, 20927, 52552, 33048, 45688, 12243,
44891, 3305, 5132, 32565, 31630, 12278, 8543, 22002, 59206, 23727, 35593,
41270, 19655)
```

B.2 Tests Performed

Numerous tests are available for testing random numbers. Xiannong Meng [51] provides a list of five such tests: (1) frequency test, (2) runs test, (3) auto-correlation test, (4) gap test, and (5) poker test. The use of these tests is then described as follows: *"The two properties we are concerned most are uniformity and independence. [...] The first one tests for uniformity and the second to fifth ones test independence."* [51]

Considering the scope of this study, we have decided to perform the frequency test and one of the tests for independence only (specifically the runs test). Both tests will be carried out on the three samples collected from our device. We have decided to test multiple samples to make sure that the device does not contain just one hard-coded sequence of numbers that is used again and again (which could be perfectly random when tested alone).

B.3 Tested Hypotheses

Frequency test: Distribution of the generated random numbers in our samples shall be considered uniform if the result of the selected frequency test (Kolmogorov-Smirnov test, see below in Section B.3.1) performed on the sample falls in the 95% confidence interval. H_0 = generated number sequences show uniform distribution. H_1 = generated number sequences do not show uniform distribution. If the P value obtained from the test is higher than 0.05, we accept the H_0 hypothesis, in other cases we reject it.

Independence test: Generated random numbers in our samples shall be considered independent if the result of the runs test (see below in Section B.3.2) performed on the sample falls in the 95% confidence interval. H_0 = generated numbers are independent. H_1 = generated numbers are not independent. If the P value obtained from the test is higher than 0.05, we accept the H_0 hypothesis, in other cases we reject it.

B.3.1 Frequency Test

For the frequency test, the Kolmogorov-Smirnov test has been used. This test is a statistical test that compares two continuous distribution functions to see whether they are different [52]. We can therefore use it to determine whether or not our samples show uniform distribution. In R, this test is implemented in the `dgof` package. We therefore start with installing the package and loading the library.

```
# Installation of the required package
install.packages("dgof")
# Loading of the required library
library(dgof)
```

Afterwards, the three numeric vectors are tested using one-sample tests. The distribution used for one-sample tests is defined using parameters. In our case it is the uniform distribution

with values from 0 to 65535. The console outputs below show both the commands used and the results obtained:

```
> ks.test(s1,punif,0,65535)

        One-sample Kolmogorov-Smirnov test

data:  s1
D = 0.084909, p-value = 0.7305
alternative hypothesis: two-sided
```

```
> ks.test(s2,punif,0,65535)

        One-sample Kolmogorov-Smirnov test

data:  s2
D = 0.12742, p-value = 0.2449
alternative hypothesis: two-sided
```

```
> ks.test(s3,punif,0,65535)

        One-sample Kolmogorov-Smirnov test

data:  s3
D = 0.08353, p-value = 0.7483
alternative hypothesis: two-sided
```

We can see from the console outputs for the three test runs that the P values for all of the three samples (0.7305 for s1, 0.2449 for s2, and 0.7483 for s3) allow us to accept the H_0 hypothesis: "The generated number sequences show uniform distribution".

B.3.2 Runs Test

We have used the runs test to test the independence of numbers in the generated sequences. Run is a series of increasing values or a series of decreasing values. Length of the run is the number of such monotonous sequence. *"In a random data set, the probability that the (I+1)th value is larger or smaller than the Ith value follows a binomial distribution, which forms the basis of the runs test."* [53]

The runs test is implemented in the R language and environment in the `snpar` package. Therefore, we have to install the package and load the library first:

```
install.packages("snpar")
library(snpar)
```

The individual tests are then run on each of the samples s1, s2, and s3 using the `runs.test()` function with the `exact=TRUE` parameter. The individual commands as well as the console outputs are presented below:

```
> runs.test(s1, exact=TRUE)

        Exact runs test

data:  s1
Runs = 34, p-value = 0.521
alternative hypothesis: two.sided
```

```
> runs.test(s2, exact=TRUE)

        Exact runs test

data:  s2
Runs = 26, p-value = 0.159
alternative hypothesis: two.sided
```

```
> runs.test(s3, exact=TRUE)

        Exact runs test

data:  s3
Runs = 32, p-value = 0.8966
alternative hypothesis: two.sided
```

The P values obtained from the individual runs tests, specifically 0.521 for s1, 0.159 for s2, and 0.8966 for s3, show that we can accept the H_0 hypothesis: "The generated numbers are independent".

B.4 Conclusion

We have performed two series of tests on our samples. The Kolmogorov-Smirnov test was used to test the uniformity of the generated samples, while the runs test was used for independence testing. Results of both test series for all individual samples have confirmed the H_0 hypothesis. Therefore, we can be 95% confident that the generated sequences show uniform distribution and the numbers in these sequences are independent.

Bibliography

1. LUETH, Knud Lasse. *Why the Internet of Things is called Internet of Things: Definition, history, disambiguation* [online]. 2014-12 [visited on 2021-04-22]. Available from: `https://iot-analytics.com/internet-of-things-definition/`.
2. CHUI, Michael; LÖFFLER, Markus; ROBERTS, Roger. *The Internet of Things* [online]. 2010-03 [visited on 2021-04-10]. Available from: `https://www.mckinsey.com/industries/technology-media-and-telecommunications/our-insights/the-internet-of-things`.
3. HANES, David et al. *IOT fundamentals: networking technologies. protocols, and use cases for the internet of things*. Cisco Press, 2017. ISBN 978-1-58714-456-1.
4. SETHI, Pallavi; SARANGI, Smruti R. Internet of Things: Architectures, Protocols, and Applications. *Journal of Electrical and Computer Engineering*. 2017, vol. 2017, pp. 1–25. Available from DOI: `10.1155/2017/9324035`.
5. VERMESAN, Ovidiu et al. Internet of things strategic research roadmap. In: *Internet of Things: Global Technological and Societal Trends*. Aalnorg: River Publishers, 2011, 9–52. ISBN 9788792329738.
6. BLINOWSKI, Grzegorz J.; PIOTROWSKI, Paweł. *CVE based classification of vulnerable IoT systems*. 2020. Available from arXiv: `2006.16640 [cs.CR]`.
7. TURCK, Matt. *The Internet Of Things Is Reaching Escape Velocity* [online]. TechCrunch, 2014-12 [visited on 2021-04-22]. Available from: `https://techcrunch.com/2014/12/02/the-internet-of-things-is-reaching-escape-velocity/`.
8. SANTOS, Daniel dos et al. *AMNESIA:33* [online]. Forescout Research Labs, 2020 [visited on 2021-04-18]. Available from: `https://www.forescout.com/company/resources/amnesia33-how-tcp-ip-stacks-breed-critical-vulnerabilities-in-iot-ot-and-it-devices/`.
9. *Treck TCP/IP User Manual* [online]. Treck Incorporated, 2004 [visited on 2021-04-21]. Available from: `https://manualzz.com/doc/6645174/treck-tcp-ip-user-manual`.
10. GREENBERG, Andy. *Sandworm: a new era of cyberwar and the hunt for the Kremlins most dangerous hackers*. Anchor Books, 2020. ISBN 9780385544412.
11. *Browse Vulnerabilities By Date* [online] [visited on 2021-04-10]. Available from: `https://www.cvedetails.com/browse-by-date.php/`.
12. HANDLEY, Mark; RESCORLA, Eric (eds.). *Internet Denial-of-Service Considerations* [online]. 2006 [visited on 2021-04-22]. Available from: `https://tools.ietf.org/html/rfc4732`.

13. N-ABLE. *Remote Code Execution Overview* [online]. 2019 [visited on 2021-04-22]. Available from: https://www.n-able.com/blog/remote-code-execution.

14. *The IoT Attacks Everyone Should Know About* [online]. Pwnie Express, 2020 [visited on 2021-04-22]. Available from: https://outpost24.com/blog/the-iot-attacks-everyone-should-know-about.

15. *What Makes IoT so Vulnerable to Attack?* [Online]. Pwnie Express, 2020 [visited on 2021-04-22]. Available from: https://outpost24.com/blog/what-makes-the-iot-so-vulnerable-to-attack.

16. KOL, Moshe; SCHÖN, Ariel; OBERMANN, Shlomi. *Ripple20: CVE-2020-11901* [online]. 2020-08 [visited on 2021-03-07]. Available from: https://www.jsof-tech.com/wp-content/uploads/2020/08/Ripple20_CVE-2020-11901-August20.pdf.

17. BURSZTEIN, Elie. *Inside the infamous Mirai IoT Botnet: A Retrospective Analysis* [online]. 2017-12 [visited on 2021-04-22]. Available from: https://blog.cloudflare.com/inside-mirai-the-infamous-iot-botnet-a-retrospective-analysis/.

18. WINWARD, Ron. *IoT Attack Handbook* [online]. 2018 [visited on 2021-04-18]. Available from: https://www.radware.com/getattachment/402db7f3-0467-4fa3-bb9a-ae88b728e91b/MiraiHandbookEbookFinal_04.pdf.aspx.

19. *IoT Malware Droppers (Mirai and Hajime)* [online]. 2017 [visited on 2021-04-20]. Available from: https://0x00sec.org/t/iot-malware-droppers-mirai-and-hajime/1966.

20. HADAD, Barak; KAUFFMAN, Gal; SERI, Ben. *URGENT/11: Exploring and Exploiting Programmable Logic Controllers with URGENT/11 Vulnerabilities* [online]. Armis, 2020 [visited on 2021-04-18]. Available from: https://info.armis.com/rs/645-PDC-047/images/Armis-URGENT11-on-OT-WP.pdf.

21. *NUMBER:JACK* [online]. Forescout Research Labs, 2021 [visited on 2021-04-19]. Available from: https://www.forescout.com/company/resources/numberjack-weak-isn-generation-in-embedded-tcpip-stacks/.

22. *RFC 793: Transmission Control Protocol* [online]. 1981 [visited on 2021-04-22]. Available from: https://tools.ietf.org/html/rfc793.

23. *RFC 1948: Defending Against Sequence Number Attacks* [online]. 1996 [visited on 2021-04-22]. Available from: https://tools.ietf.org/html/rfc1948.

24. *RFC 6528: Defending Against Sequence Number Attacks* [online]. 2012 [visited on 2021-04-22]. Available from: https://tools.ietf.org/html/rfc6528.

25. *NAME:WRECK — Breaking and fixing DNS implementations* [online]. JSOF, 2021-04 [visited on 2021-04-22]. Available from: https://www.forescout.com/company/resources/namewreck-breaking-and-fixing-dns-implementations/.

26. *Ripple20: 19 Zero-Day Vulnerabilities Amplified by the Supply Chain* [online]. 2020-06 [visited on 2021-04-24]. Available from: https://www.jsof-tech.com/disclosures/ripple20/.

27. KOL, Moshe; OBERMANN, Shlomi. *Ripple20: CVE-2020-11896 RCE, CVE-2020-11898 Info Leak* [online]. 2020-06 [visited on 2021-04-24]. Available from: https://www.jsof-tech.com/wp-content/uploads/2020/06/JSOF_Ripple20_Technical_Whitepaper_June20.pdf.

28. *HPSBPI03666 rev. 3 - Certain HP and Samsung-branded Print Products - Network Stack Potential Vulnerabilities — HP® Customer Support* [online]. 2020-07 [visited on 2021-03-05]. Available from: https://support.hp.com/in-en/document/c06640149.

29. *HP DeskJet 3700 All-in-One series* [online]. Hewlett Packard, [n.d.] [visited on 2021-05-13]. Available from: http://h10032.www1.hp.com/ctg/Manual/c05152483.pdf.

30. KOZIEROK, Charles M. *The TCP/IP Guide – History of the OSI Reference Model* [online]. 2005-09 [visited on 2021-04-25]. Available from: http://www.tcpipguide.com/free/t_HistoryoftheOSIReferenceModel.htm.

31. *What Is The OSI Model?* [Online] [visited on 2021-04-25]. Available from: https://www.cloudflare.com/learning/ddos/glossary/open-systems-interconnection-model-osi/.

32. SUNDARARAJAN, Aditya et al. A Survey of Protocol-Level Challenges and Solutions for Distributed Energy Resource Cyber-Physical Security. *Energies* [online]. 2018, vol. 11, no. 9 [visited on 2021-04-25]. ISSN 1996-1073. Available from DOI: 10.3390/en11092360.

33. VAUGHAN-NICHOLS, Steven. *The birth and rise of Ethernet: A history* [online] [visited on 2021-04-25]. Available from: https://www.hpe.com/us/en/insights/articles/the-birth-and-rise-of-ethernet-a-history-1706.html.

34. RADOVANOVIC, Igor. *Optical Local Area Networks: New Solutions For Fiber-to-the-Desk Applications* [online]. Twente University Press, 2003 [visited on 2021-04-25]. ISBN 9036520029. Available from: https://pages.jh.edu/bcooper8/sigma_files/SPOT/Reference/SPE/OPTICAL%20LOCAL%20AREA%20NETWORKS.pdf.

35. *IEEE Std 802.3™-2018 Standard for Ethernet*. IEEE, 2016.

36. *IEEE Std 802®-2014 Standard for Local and Metropolitan Area Networks: Overview and Architecture*. IEEE, 2014.

37. *RFC 791: Internet Protocol* [online]. 1981 [visited on 2021-04-25]. Available from: https://tools.ietf.org/html/rfc791.

38. SATRAPA, Pavel. *IPv6: internetový protokol verze 6*. CZ.NIC, z.s.p.o., 2019. ISBN 978-80-88168-43-0.

39. *RFC 790: Assigned Numbers* [online]. 1981 [visited on 2021-04-27]. Available from: https://tools.ietf.org/html/rfc790.

40. *RFC 768: User Datagram Protocol* [online]. 1980 [visited on 2021-04-28]. Available from: https://tools.ietf.org/html/rfc768.

41. *RFC 6335: Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry* [online]. 2011 [visited on 2021-04-28]. Available from: https://tools.ietf.org/html/rfc6335.

42. *RFC 1853: IP in IP Tunneling* [online]. 1995 [visited on 2021-04-30]. Available from: https://tools.ietf.org/html/rfc1853.

43. *National Vulnerability Database: CVE-2020-11898* [online] [visited on 2021-04-25]. Available from: https://nvd.nist.gov/vuln/detail/CVE-2020-11898.

44. *RFC 792: Internet Control Message Protocol* [online]. 1981 [visited on 2021-05-02]. Available from: https://tools.ietf.org/html/rfc792.

45. MOCKAPETRIS, Paul. *RFC 1034 Domain Names - Concepts and Facilities* [online]. 1987 [visited on 2021-04-22]. Available from: https://tools.ietf.org/html/rfc1034.

46. MOCKAPETRIS, Paul. *RFC 1035 Domain Names - Implementation and Specification* [online]. 1987 [visited on 2021-04-22]. Available from: https://tools.ietf.org/html/rfc1035.

47. *National Vulnerability Database: CVE-2020-11901* [online] [visited on 2021-03-13]. Available from: https://nvd.nist.gov/vuln/detail/CVE-2020-11901.

48. *DNS Multiple QTYPEs* [online]. 2017 [visited on 2021-03-22]. Available from: https://tools.ietf.org/id/draft-bellis-dnsext-multi-qtypes-04.html.

49. KOZIOL, Jack et al. *The Shellcoders Handbook: Discovering and Exploiting Security Holes*. Wiley Pub., 2004. ISBN 0-7645-4468-3.

50. PERLROTH, Nicole. *Daniel Kaminsky, Internet Security Savior, Dies at 42* [online]. The New York Times, 2021 [visited on 2021-04-27]. Available from: `https://www.nytimes.com/2021/04/27/technology/daniel-kaminsky-dead.html`.

51. MENG, Xiannong. *Tests for Random Numbers* [online]. 2002 [visited on 2021-05-09]. Available from: `https://www.eg.bucknell.edu/~xmeng/Course/CS6337/Note/master/node42.html`.

52. SANDERS, T. J. *Simulation Notes* [online]. United States Naval Academy, 2013 [visited on 2021-05-09]. Available from: `https://www.usna.edu/Users/math/dphillip/sa421.s16/chapter02.pdf`.

53. CROARKIN, Carroll; TOBIAS, Paul (eds.). *NIST/SEMATECH e-Handbook of Statistical Methods: Runs Test for Detecting Non-randomness* [online]. NIST, 2012 [visited on 2021-05-09]. Available from: `https://www.itl.nist.gov/div898/handbook/eda/section3/eda35d.htm`.

Contents of the Enclosed Data Medium

- `00readme.txt` — Brief description of the data medium contents
- `pdf`
  - `thesis.pdf` — Bachelor's thesis (PDF)
- `tex`
  - `appendix.tex` — Source code of the appendix (TEX)
  - `assignment-include.pdf` — Assignment of bachelor's thesis (PDF)
  - `medium.tex` — Source code of the contents of the data medium (TEX)
  - `thesis.tex` — Source code of the thesis (TEX)
  - `img` — Directory containing images used in the thesis
  - `bib` — Directory containing the Biblatex database file
- `utils`
  - `11898` — Utilities used for the attack on the CVE-2020-11898 vulnerability
    - `frames` — The byte forms of the frames used in our experiments
    - `source` — Source code of the framegen.cpp utility written in C++
  - `11901` — Utilities used for the attack on the CVE-2020-11901 vulnerabilities
    - `responses` — The byte forms of the DNS responses used in our experiments
    - `source` — Source code of the createpacket.cpp utility written in C++