HAW HAMBURG

Bachelorarbeit

Sabrina Sendel

Penetrationstests der RIOT Netzwerkimplementierung

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Sabrina Sendel

Penetrationstests der RIOT Netzwerkimplementierung

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Bachelorstudiengang Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Thomas Schmidt
Zweitgutachter: Prof. Dr. Martin Becke

Eingereicht am: 07. Februar 2019

Sabrina Sendel

Thema der Arbeit

Penetrationstests der RIOT Netzwerkimplementierung

Stichworte

RIOT, IoT, Scapy, Penetrationstests, KillerBee, 6LoWPAN

Kurzzusammenfassung

Im Internet of Things ist Sicherheit ein wichtiges Thema. Eines der Betriebssysteme für diese „Dinge“ im IoT ist das RIOT-OS. Da auch hier die Sicherheit und Funktionalität an höchster Stelle stehen soll, vor allem auch in der Kommunikation der „Dinge“, werden in dieser Arbeit Penetrationstests einiger Netzwerkprotokolle dieses Betriebssystems vorgestellt und damit versucht Sicherheitslücken zu entdecken.

Sabrina Sendel

Title of Thesis

Penetration-Tests of the RIOT network-implementation

Keywords

RIOT, IoT, Scapy, Penetration-Tests, KillerBee, 6LoWPAN

Abstract

In the Internet of Things security is a relevant topic. One of the Operating Systems for these „things“ in the IoT is the RIOT-OS. Even in the IoT the security and functionality should be set at the highest level, especially in the communication of these „things“, that is why in this thesis penetration tests of some network protocols of the operating system are presented and used to try to figure out security lacks.

Inhaltsverzeichnis

Abbildungsverzeichnis v

Tabellenverzeichnis vii

Abkürzungen viii

1 Einleitung 1

1.1 Motivation . . . 1
1.2 Ziel . . . 2
1.3 Aufbau der Arbeit . . . 2

2 Das Internet der Dinge 3

2.1 Allgemeine Hintergründe . . . 3
2.2 Netzwerkprotokolle . . . 4
2.2.1 TCP . . . 4
2.2.2 UDP . . . 5
2.2.3 NDP . . . 6
2.3 Netzwerkprotokolle im IoT . . . 7
2.3.1 6LoWPAN . . . 7
2.3.2 CoAP . . . 7
2.4 RIOT . . . 8
2.4.1 Geschichte . . . 8
2.4.2 Unterstützung . . . 8
2.4.3 Eigenständige Programmierung und Entwicklung mit RIOT . . . 8
2.4.4 Relevante Bereiche . . . 9

3 Penetrationstests 10

3.1 Allgemeines Vorgehen . . . 10
3.2 Blackbox versus Whitebox Testing . . . 11
3.2.1 Blackbox Testing . . . 11

3.2.2 Whitebox Testing . . . 12
3.3 Werkzeuge . . . 13
3.3.1 Scapy . . . 13
3.3.2 KillerBee . . . 14
3.3.3 Wireshark . . . 17
3.3.4 Eigene Implementierungen . . . 18

4 Testplanung und -durchführungen **19**
4.1 Testumgebung . . . 20
4.2 Man in the Middle . . . 21
4.3 IP-Spoofing . . . 22
4.4 UDP . . . 23
4.4.1 Fault injection . . . 23
4.4.2 Testen des Packet Loss . . . 24
4.5 TCP . . . 26
4.5.1 TCP-Flooding . . . 26
4.6 Weitere Testplanungen . . . 28
4.6.1 NDP . . . 28
4.6.2 CoAP . . . 28

5 Ergebnisse **29**
5.1 IP-Spoofing . . . 29
5.2 UDP . . . 29
5.2.1 Fault injection . . . 29
5.2.2 Testen des Packet Loss . . . 30
5.3 TCP . . . 33
5.3.1 SYN-Flooding . . . 33
5.3.2 ACK- und FIN-Flooding . . . 34

6 Zusammenfassung und Ausblick **36**

Glossar **39**

7 Versicherung über Selbstständigkeit **40**

Abbildungsverzeichnis

2.1 Ablauf des Verbindungsaufbaus 5

3.1 Abbildung einer Blackbox 11
3.2 Abbildung einer Whitebox 12
3.3 Absichtlich inkorrekt erstelltes Paket per Scapy 13
3.4 Aufbau eines UDP-Paketes in Wireshark 17
3.5 Hex-Dump eines UDP-Paketes in Wireshark 17

4.1 Grafische Darstellung des Versuchsaufbaus 21
4.2 Kommunikation zwischen zwei Geräten ohne Eingriff des Angreifers 21
4.3 Kommunikation zwischen zwei Geräten mit Eingriff des Angreifers 22
4.4 TCP-Paket zum SYN-Flooden 26

5.1 Senden von nicht manipulierten und manipulierten UDP-Paketen in Wireshark 29
5.2 Output des RIOT Device under Tests's während des Testens mit manipulierten UDP-Paketen 30
5.3 Verwerfen des Paketes von dem Device under Test durch falsche Checksumme 30
5.4 Senden von 10.000 UDP-Paketen per Script 30
5.5 Senden von 100.000 UDP-Paketen per Script 31
5.6 10.000 empfangene UDP-Pakete 31
5.7 100.000 empfangene UDP-Pakete 31
5.8 Antwort des RIOT-Boards auf ein SYN-Paket in Wireshark 33
5.9 Senden von Retransmissionnachrichten vom Device under Test, aus Wireshark 33
5.10 Versuche vom Device under Test mit dem ersten Sender einer Nachricht eine Verbindung aufzubauen 34
5.11 ACK-flooding in Wireshark 34
5.12 FIN-Flooding in Wireshark 35

Tabellenverzeichnis

4.1 IoT-Devices 20

5.1 Tabelle mit Auflistung zum Packet Loss 32

Abkürzungen

6LoWPAN IPv6 over Low power Wireless Personal Area Network.

ARP Adress Resolution Protocol.

ASCII American Standard Code for Information Interchange.

BSI Bundesamt für Sicherheit in der Informationstechnik.

CoAP Constrained Application Protocol.

CPU Central Processing Unit.

DUT Device under Test.

HAW Hochschule für Angewandte Wissenschaften.

HTTP Hypertext Transfer Protocol.

INRIA Institut national de recherche en informatique et en automatique.

IoT Internet of Things.

ISN initial sequence number.

LGPLv2.1 GNU Lesser General Public License Version 2.1.

M2M Machine-to-Machine.

MAC Media Access Control.

MQTT Message Queuing Telemetry Transport.

NCT Neighbor Cache Table.

NDP Neighbor Discovery Protocol.

OS Operating System.

RAM Random Access Memory.

REST Representational State Transfer.

ROM Read-Only Memory.

TCP Transmission Control Protocol.

TI Texas Instruments.

UDP User Datagram Protocol.

WPAN Wireless Personal Area Networt.

XML Extensible Markup Language.

1 Einleitung

Das Internet of Things (IoT) ist ein stetig wachsendes Netzwerk an „Dingen“, kleinen Devices, die miteinander und auch mit anderen Komponenten kommunizieren. Nicht nur lokal untereinander, sondern auch die Kommunikation über das Internet ist möglich. Diese Geräte sind kleine Rechner mit wenig Ressourcen und beanspruchen wenig Energie.

Beispielsweise gibt es Sensoren, die Temperaturen erfassen und diese Werte an Heizungsventile senden, welche daraufhin die Heiztemperatur regeln.

Doch auch bei dieser Kommunikation können Fehler auftreten, beispielsweise werden falsche Werte an das Ventil gesendet. Auch die Manipulation von außen ist ein Schwachpunkt, sodass ein Fremder, sofern möglich, die falschen Temperaturen an diesen Geräten einstellt. Diese Manipulation kann nicht nur lokal, sondern auch weltweit durchgeführt werden, da diese Geräte mit dem Internet verbunden sind.

Dieses geschieht durch IT-Manipulation der Kommunikation dieser Geräte, weshalb es wichtig ist, auf solche Sicherheitslücken zu testen. Penetrationstests sind hier gut geeignet, da verschiedene Geräte auf verschiedene Arten getestet werden können.

Zur Zeit existieren für die Kommunikation über Wireless Personal Area Networt (WPAN) per IPv6 over Low power Wireless Personal Area Network (6LoWPAN) Komprimierung relativ wenig automatisierte Testsuits, um in diesem Bereich Fehler zu erkennen. Für diese Arbeit ist es deshalb nötig weitere Tests selbst zu programmieren und implementieren.

1.1 Motivation

Durch meine Teilnahme am Projekt „RIOT im Internet of Things“ kam ich dem Thema IoT näher. Aufgrund dessen sowie meines Interesses an Testings und der Netzwerktechnik wollte ich mich hiermit näher beschäftigen.

Da diese „Dinge“ untereinander über Netzwerke kommunizieren und somit auch sensible Daten versenden können, ist es wichtig in diesem Bereich auf Sicherheitslücken zu testen, damit diese aufgedeckt und behoben werden und nicht an Unbefugte gelangen können. In unserem Projekt wäre es z.B. durch Sicherheitslücken möglich gewesen, sich unbefugt Zugang zu gewissen Ressourcen zu beschaffen und in fremdem Namen Dinge zu entwenden.

Des Weiteren ist es wichtig diese Lücken aufzudecken, um die einwandfreie Funktionalität der Geräte zu gewährleisten. Nicht nur die Manipulation der gesendeten Daten, sondern auch der Ausfall von Geräten soll verhindert werden. Für diese Art der Sicherstellung existieren Penetrationstests um diese Lücken aufzudecken.

Folgende Fragestellung kam bei mir auf: An welchen Stellen stellt RIOT im Bereich Netzwerk und Kommunikation eine Sicherheitslücke dar?

1.2 Ziel

Das Ziel der Arbeit ist, zu testen, welche Sicherheitslücken RIOT im Wireless-Netzwerkbereich beinhaltet, das heißt in wie weit es möglich ist die Devices zum Absturz zu bringen oder ihnen falsche Daten zu senden, ohne dass die Devices die falschen Pakete erkennen. Sollte dieses möglich sein, sollen diese Lücken dargestellt werden.

1.3 Aufbau der Arbeit

Im folgenden Abschnitt 2 wird das Internet der Dinge vorgestellt, seine wichtigsten Netzwerkprotokolle erklärt und in das freie Open Source Betriebssystem RIOT eingeführt. Im Abschnitt 3 folgen Beschreibungen der Penetrationstests, dem Blackbox und Whitebox Testing und die Vorstellung der Werkzeuge. Darauf folgen die Planung und Durchführung der Tests im Abschnitt 4, in dem die Testumgebung und verschiedene Tests vorgestellt werden. Abschnitt 5 präsentiert die Ergebnisse. Die Arbeit schließt im Abschnitt 6 mit der Zusammenfassung und dem Ausblick ab.

2 Das Internet der Dinge

Das IoT ist ein Netzwerk von unterschiedlichen Devices, die oft über WPAN untereinander kommunizieren. Die Devices reichen von kleinsten Geräten die nur minimale Aufgaben erfüllen, wie beispielsweise die Messung der Temperatur im häuslichen Bereich, bis hin zu komplexen Geräten, die weitaus mehr Funktionalität bieten. Diese Devices sind miteinander über das IoT vernetzt und agieren untereinander, kommunizieren aber auch mit anderen Devices und Apps, wie beispielsweise eine App für die o.g. Temperaturüberwachung.

2.1 Allgemeine Hintergründe

Momentan befindet sich dieser Bereich der Informatik im Aufschwung. In allen Lebensbereichen haben mittlerweile Devices des IoT Einzug gehalten. Sei es im Smart Home-Bereich oder in der Industrie, beispielsweise in der Autoindustrie. [1]

Auch Privatpersonen kommen immer mehr damit in Kontakt. Zum Beispiel auf Messen (IFA Berlin [2]) oder im Elektronikfachhandel.

Durch den Einsatz dieser Technologie können diverse Abschnitte des Alltages automatisiert werden. Es können zum Beispiel Sensoren zur selbstständigen Regulierung der Temperatur oder Beleuchtung eingesetzt werden.

2.2 Netzwerkprotokolle

2.2.1 TCP

Transmission Control Protocol (TCP) (Transmission Control Protocol) ist ein verbindungsorientiertes Netzwerkprotokoll, welches im Jahre 1981 veröffentlicht wurde. [3]

Es ist ein zuverlässiges Host-2-Host Protokoll, welches in Netzwerken eingesetzt wird. Die Authentifizierung erfolgt über den TCP-Handshake.

TCP-Handshake

Das verbindungsanfragende Device sendet eine SYN-Nachricht an den Empfänger, woraufhin dieser innerhalb einer vorgegebenen Zeit die Möglichkeit hat, auf diese Nachricht mit einer SYN/ACK-Nachricht zu antworten. Der Sender reagiert darauf mit einer ACK-Nachricht an den Empfänger. Somit wurde eine Verbindung hergestellt.

SYN - Bei einer SYN-Nachricht ist die erste Sequenznummer die initial sequence number (ISN) um die Sequenz einzuleiten.

ACK - Bei einer ACK-Nachricht ist das ACK-Bit gesetzt, und im Feld "Acknowledgement Number„ ist die nächste Sequenznummer zu finden.

FIN - Bei einer FIN-Nachricht ist das FIN-Bit gesetzt.

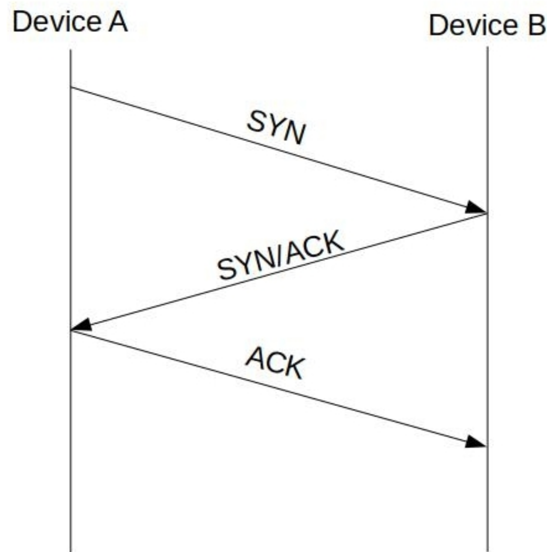

Abbildung 2.1: Ablauf des Verbindungsaufbaus

In Abbildung 2.1 wird noch einmal grafisch dargestellt, wie Device A eine SYN-Nachricht an Device B sendet, woraufhin Device B mit einer SYN/ACK-Nachricht antwortet. Device A bestätigt diese mit einer ACK-Nachricht, woraufhin die Verbindung hergestellt wurde.

Gehen Pakete verloren, werden sie erneut versendet.

2.2.2 UDP

User Datagram Protocol (UDP) ist ein verbindungsloses Protokoll, welches 1980 veröffentlicht wurde. [4]

Es wird weniger Traffic benötigt, da durch die Verbindungslosigkeit die Pakete nicht mehrfach gesendet werden. Eingesetzt wird UDP beispielsweise bei Streaming-Diensten, da diese nur im Simplex-Modus arbeiten. Dies ist positiv für IoT-Netzwerke zu sehen, da weniger Traffic verursacht und somit weniger Energie verbraucht wird.

Eine Zuverlässigkeit wie bei TCP gibt es erst auf höreren Schichten, da UDP verbindungslos arbeitet.

2.2.3 NDP

Das Neighbor Discovery Protocol (NDP) [5] regelt in IoT-Netzen die Auflösung von IPv6-Adressen in Media Access Control (MAC) Adressen, da über IPv6 kommuniziert wird. Das NDP ist unter anderem das Pendant des Adress Resolution Protocol (ARP), welches im IPv4-Bereich für die Adressauflösung sorgt.

Per NDP werden MAC-Adressen den IPv6-Adressen zugeordnet. Somit ist es den Devices nicht nur möglich neue Nachbarn zu finden, sondern auch geänderte MAC-Adressen zu aktualisieren.

2.3 Netzwerkprotokolle im IoT

Ein Netzwerk im IoT besteht aus Devices, die miteinander über Low-Power-Kommunikation per 6LoWPAN kommunizieren. In höhere IP-Modell-Schichten ist auch die Kommunikation über Protokolle wie Constrained Application Protocol (CoAP) möglich, somit lassen sich beispielsweise Sensoren auswerten und an ein Frontend senden, sodass Nutzer die Möglichkeit haben, Daten anzusehen und anschließend auszuwerten.

2.3.1 6LoWPAN

6LoWPAN ist ein standartisiertes Protokoll, welches zur Kommunikation im IoT eingesetzt wird. Es wird zur Übertragung von IPv6 benötigt. [6]

6LoWPAN beinhaltet zum Beispiel die Headerkomprimierung, Somit steht mehr Volumen für die weiteren Daten zur Verfügung. Unter anderem beinhaltet es auch die Fragmentierung. Da die Paketgröße in IPv6 mindestens 1280 Bytes groß ist, die Paketgröße über IEEE 802.15.4 aber maximal 127 Bytes beträgt, übernimmt 6LoWPAN das Fragmentieren und Defragmentieren um eine Kommunikation zu ermöglichen. [6]

2.3.2 CoAP

CoAP ist ein Web-Transfer-Protokoll, welches für die Kommunikation zwischen Low-Power-Geräten entwickelt worden ist (M2M Kommunikation).

Es basiert wie Hypertext Transfer Protocol (HTTP) auf dem Representational State Transfer (REST) Modell, wird allerdings über UDP versendet. [7]

2.4 RIOT

RIOT ist ein multithreading Betriebssystem für das IoT. [8]

2.4.1 Geschichte

Erste Varianten wurden in dem Projekt FireWhere erarbeitet, mit dem Ergebnis des Fire-Kernels. Die Entwicklung des Betriebssystems wird an der Freien Univesität Berlin, der Hochschule für Angewandte Wissenschaften (HAW)Hochschule für Angewandte Wissenschaften (HAW) Hamburg und dem Institut national de recherche en informatique et en automatique (INRIA) angeführt, aber von einer sehr großen weltweiten Community weiterentwickelt. RIOT ist unter GNU Lesser General Public License Version 2.1 (LGPLv2.1) lizensiert.

2.4.2 Unterstützung

Auch kleine Geräte, die über das Internet kommunizieren sollen, benötigen ein Betriebssystem, welches als Schnittstelle zur Hardware dient. Hier kommt RIOT ins Spiel. Unterstützt wird eine hohe Anzahl an verschiedenen Mikrocontrollern mit 8, 16 und 32 Bit Prozessoren. Durch den schlanken Aufbau benötigt RIOT nur einen minimalen Speicherplatz von etwa 1,5kB Random Access Memory (RAM) und 5kB Read-Only Memory (ROM) und ist somit auch für kleinere Geräte optimal. Es unterstützt Modularität und bietet eine Unterstützung für Real-Time an. [9]

2.4.3 Eigenständige Programmierung und Entwicklung mit RIOT

Für Beginner wird ein Tutorial [10] zur Verfügung gestellt, welches die ersten Schritte, sowie einige Grundfunktionen erläutert. RIOT unterstützt C, als auch C++ als Programmiersprache. Sollte die Hardware fehlen, ist es auch möglich seinen Code native zu starten (Central Processing Unit (CPU) und RAM werden in einem UNIX Prozess gestartet), um so bereits sehen zu können, ob es möglich ist, seinen Code zu compilieren und zu linken, sowie ihn native zu testen. Es ist möglich eigene Programme zu schreiben und somit das OS seinen Wünschen anzupassen. Zum Testen auf Sicherheitslücken

können verschiedene Module geladen werden, sodass das Device under Test (DUT) die Netzwerkprotokolle unterstützt, die getestet werden sollen.

Zur Kommunikation werden unter Anderem TCP als auch UDP als Netzwerkprotokolle angeboten. Als Nachrichtenprotokolle für die Machine-to-Machine (M2M) Kommunikation werden des Weiteren Message Queuing Telemetry Transport (MQTT) und CoAP angeboten.

2.4.4 Relevante Bereiche

Für diese Arbeit sind die Netzwerkimplementierungen von Bedeutung.

Sowohl das Modul gnrc_networking, als auch das Modul gnrc_tcp werden für Tests verwendet.

Um die Internetkonfiguration zu ändern wird auch das Shell-Modul verwendet.

3 Penetrationstests

Penetrationstests sind eine Reihe von Tests, die sich mit dem Testen der Sicherheit von einzelnen Systemen oder Netzwerken befassen. Das Ziel ist, Sicherheitslücken zu finden und somit schließen zu können. Im allgemeinen werden von Gateways über Switches bis hin zu Rechnern alle Komponenten in einem Netzwerk getestet, da jede eine Sicherheitslücke darstellen kann. Über Gateways kann beispielsweise getestet werden, ob der Zugriff auf das Intranet möglich ist. Auch Flooding ist möglich. Intern kann versucht werden, auf Devices zuzugreifen, Dienste zu nutzen oder abzuschalten, sowie Geräte zu attackieren.

Da Penetrationstests eine gewisse Planung erfordern, gibt es ein allgemeines Vorgehen, welches im nächsten Abschnitt erklärt wird.

3.1 Allgemeines Vorgehen

Wie vom Bundesamt für Sicherheit in der Informationstechnik (BSI) empfohlen [11], wird zu Beginn mit der verantwortlichen Person die Zielsetzung besprochen. Daraufhin werden Informationen über das zu testende System ausgetauscht, wer Zugangsrechte besitzt, sowie allgemeine Eckpunkte, um schon im ersten Gespräch Sicherheitslücken lokalisieren zu können. Außerdem wird auch in Erfahrung gebracht, ob in dem System besonders zu schützende Komponenten vorhanden sind, die einer höheren Beachtung bedürfen. Außerdem werden Eckdaten wie Datenschutz, Umfang der Tests und Testzeitraum besprochen.

Nach den ersten Gesprächen wird ein Testplan erstellt, der auf das System zugeschnitten ist.

Bei dem anschließenden Testen sollten die verantwortlichen Personen der zu testenden Geräte vor Ort sein, um etwaige Fragen beantworten zu können.

Nach der Phase des Testens wird ein Abschlussbericht geschrieben, in dem die Durchführung und die Ergebnisse dargestellt werden.

Da dem Tester nicht immer alle Informationen der Systeme zur Verfügung stehen, wird unter Blackbox und Whitebox Testing unterschieden, welches im nächsten Abschnitt näher erläuter wird.

3.2 Blackbox versus Whitebox Testing

Bei dem Testen wird zwischen zwei verschiedenen Varianten, dem Blackbox und dem Whitebox Testen unterschieden.

3.2.1 Blackbox Testing

Beim Blackbox Testing ist lediglich bekannt, dass ein Device vorhanden ist. Es kann komplett unbekannt sein um welche Art Komponente es sich handelt, es kann aber auch - wie in unserem Falle - bekannt sein, dass es sich um ein Device mit RIOT Operating System (OS) handelt. Unbekannt ist, welcher Code sich auf dem Gerät befindet. Bei Netzwerkdevices kann unbekannt sein, welche Sicherheitskonfigurationen auf dem Gerät vorgenommen wurden.

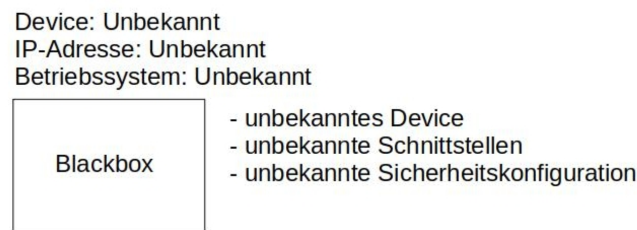

Abbildung 3.1: Abbildung einer Blackbox

In der obigen Abbildung 3.1 noch einmal grafisch dargestellt.

Somit ist das Blackbox Testing ein Eindringen von außen, wie zum Beispiel üblicherweise von einem Cracker oder Hacker.

3.2.2 Whitebox Testing

Bei dem Whitebox Testing hingegen ist das Device, sowie sämtliche Informationen bekannt. Der Tester kann so auf einem ganz anderen Niveau testen. So empfiehlt auch das BSI ein Whitebox Testing, da einige Informationen zusätzlich geliefert werden und Sicherheitslücken nicht so leicht übersehen werden können, wie es beim Blackbox Testing der Fall ist. Außerdem können Szenarien wie von einem Innentäter kaum berücksichtigt werden, da dieser über Insiderinformationen verfügt.

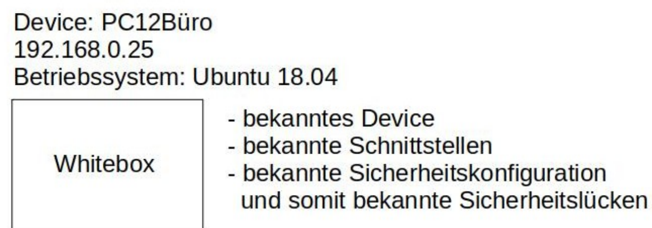

Abbildung 3.2: Abbildung einer Whitebox

Whitebox für den Tester in der obigen Abbildung 3.2 noch einmal grafisch dargestellt.

Für das Testen im IoT werden die benutzten Programme im folgenden Abschnitt erklärt.

3.3 Werkzeuge

Für das Testen im Netzwerkbereich wird ein Programm benötigt, welches erlaubt Pakete zu erstellen, zu sniffen und zu manipulieren. Hier ist Scapy das geeignete Programm der Wahl. Zum Versenden dieser Pakete über IEEE 802.15.4 wird ein Zusatzmodul benötigt. In dieser Arbeit wurde KillerBee eingesetzt. Außerdem wurden Tests auch selbst programmiert und eingebettet.

3.3.1 Scapy

Scapy [12] ist ein Programm zur Erstellung und Manipulation einzelner Pakete, sowie zum Ersniffen dieser.

Es ist in der Programmiersprache Python geschrieben und kann sowohl über die Konsole als auch per Python-Scripts bedient werden.

Es unterstützt verschiedenste Protokolle, darunter auch 6LoWPAN, wobei aber beim späteren Testen KillerBee benutzt wird, welches Scapy nutzt und einige zusätzliche Scripts bietet.

Nicht nur sniffen ist möglich, sondern auch das Versenden von selbst erstellten korrekten, bis hin zu fehlerhaften Paketen, beispielsweise eine falsche Reihenfolge der Layer. Pakete werden Layer für Layer erstellt, der im IP-Protokoll-Stack niedrigste wird als erstes benannt [13].

```
>>> paket = UDP()
>>> paket2 = IP()
>>> pkt = paket/paket2
>>> pkt
<UDP  |<IP  |>>
>>> 
```

Abbildung 3.3: Absichtlich inkorrekt erstelltes Paket per Scapy

Wie in Abbildung 3.3 zu sehen, wird zuerst ein UDP-Paket (paket) erstellt. Danach wird ein IP-Paket(paket2) erstellt. Daraufhin werden beide Pakete zu dem Paket pkt zusammengefügt. Da nun aber „paket/paket2“ erstellt wurde, und der niedrigste Layer

vorangestellt stehen muss, ist dieses Paket fehlerhaft. Das IP-Paket wurde als Payload in das UDP-Paket eingebettet. Wie auch zu sehen, warnt Scapy nicht vor inkorrekt erstellten Paketen, sodass ein Grundwissen des IP-Protokoll-Stacks nötig ist, um korrekte Pakete zu erstellen.

So kann per Scapy ein IPv6 Paket wie folgt, am Beispiel eines UDP Paketes, erstellt werden:

```
1 packet = IPv6(dst="fe80::1234:1234:1234:1234")
2 /UDP(dport=50000, sport=50000)/Raw('test')
```

Listing 3.1: Einfaches IPv6 Paket

Um dieses Paket in Listing 3.1 nun über die Devices schicken zu können, wird KillerBee benötigt, welches im nächsten Kapitel vorgestellt wird.

3.3.2 KillerBee

KillerBee [14] ist ein Testprogramm für IEEE 802.15.4 und ZigBee, welches Scapy einbindet und gleichzeitig eigene Erweiterungen anbietet.

Es werden Funktionen mitgeliefert, die zum Testen im IoT geeignet sind. Dazu gehört auch **zbsendone**, welches die Möglichkeit bietet, eigens erstellte Pakete zu senden. Beim Aufruf des Moduls wird der Kanal, die Anzahl der Pakete, sowie das zu nutzende Device angegeben.

Das im Kapitel Scapy beschriebene Listing 3.1 muss wie folgt erweitert werden, um ein gültiges UDP-Paket verschicken zu können:

```
1 packet = Dot15d4(fcf_destaddrmode=3, fcf_framever=1, fcf_srcaddrmode=3,
2 fcf_reserved_2=0, fcf_reserved_1=0, fcf_ackreq=0, fcf_pending=0,
3 seqnum=25, fcf_frametype=1, fcf_security=0, fcf_panidcompress=1)
4 /Dot15d4Data(src_addr=8748020534567945020,
5 dest_addr=8752263614135538146, src_panid=None, dest_panid=35)/
6 IPv6(dst="fe80::7b76:4962:b520:11e2")/UDP
7 (dport=53, sport=50000)/Raw('nnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnn')
```

Listing 3.2: Per zbscapy erstelltes UDP-Paket

In Listing 3.2 wird das UDP-Paket Listing 3.1 um den Dot15d4-Layer erweitert. Um das Paket zu erstellen wurde zbscapy, eine KillerBee-Erweiterung von Scapy, benutzt.

Folgend Erklärungen zu einigen Optionen:

Dot15d4

fcf_destaddrmode=3 Die Empfängeradresse wird als long addr übergeben

fcf_framever=1 Das gebaute Paket ist kompatibel mit IEEE 802.15.4-2006

fcf_srcaddrmode=3 Die Senderadresse wird als long addr übergeben

seqnum=25 Sequenznummer

fcf_frametype=1 Das gesendete Paket ist ein Datenpaket

fcf_panidcompress=1 description PAN-ID Kompression

Dot15d4Data

src_addr=8748020534567945020 Layer 2 Senderadresse

dest_addr=8752263614135538146 Layer 2 Empfängeradresse

src_panid=None PAN-ID vom Sender

dest_panid=35 PAN-ID vom Empfänger

IPv6

dst="fe80::7b76:4962:b520:11e2" IPv6Adresse vom Empfänger

UDP

dport=53 Port vom Empfänger

sport=50000 Port vom Sender

Raw

'test' Payload

Um im 6LoWPAN zu senden wird der Dot15d4-Layer, sowie der dazugehörende Dot15d4Data-Layer benötigt.

Der in Scapy genannte Layer "Dot15d4ëntspricht dem IEEE 802.15.4.

Die Pakete werden im Byteformat gesendet, beispielsweise das Paket, welches auch für einige Tests in Abbildung 5.4 bis Abbildung 5.7 benutzt wird.

Dieses Paket lässt sich per decode-Funktion auch im Hex-Format darstellen:

```
1  >>> b.decode('hex')
2  'A\xdc\x19#\x00\xe2\x11 \xb5bIvy<\xc3\xbe\xc2S6gy'
3  \x00\x00\x00\x00\x1c\x11@\xfe
4  \x80\x00\x00\x00\x00\x00\x00\xc6V Y\xe0_w\xa2\xfe
5  \x80\x00\x00\x00\x00\x00\x00{vIb\xb5 \x11\xe2\xc3P"
6  \xb8\x00\x1c\x00\xbannnnnnnnnnnntguzujnnn'
7  >>>
```

Listing 3.3: Decodetes UDP-Paket

Da bekannt ist, dass es sich um ein Dot15d4-Layer an erster Stelle des Paketes 3.3 handelt, ist es nun möglich dieses Paket auch als Dot15d4-Paket darstellen zu lassen:

```
1  >>> c=Dot15d4(b)
2  >>> c
3  <Dot15d4  fcf\_reserved\_1=0 fcf\_panidcompress=False
4  fcf\_ackreq=True fcf\_pending=True fcf\_security=False
5  fcf\_frametype=4 fcf\_srcaddrmode=None fcf\_framever=3
6  fcf\_destaddrmode=None fcf\_reserved\_2=1 seqnum=100
7  |<Raw  load='c192300e21120b5624976793cc3bec25336677960000000001
8  c1140fe80000000000000c6562059e05f77a2fe800000000000007b764962
9  b52011e2c35022b8001c00ba6e6e6e6e6e6e6e6e6e6e6e7467757a756a6e6e6e' |>>
10 >>>
```

Listing 3.4: Darstellung als Dot15d4-Paket

In Listing 3.4 erkennt man, dass der erste Teil des Paketes korrekt als Dot15d4 Paket decodiert wurde. Alle höheren Layer liegen roh als Payload vor.

3.3.3 Wireshark

Wireshark [15] ist ein Network Protocol Analyzer. Es unterstützt Hunderte von Protokollen, sowie live-capturing und bietet viele weitere Funktionen an. Die einzelnen Pakete können analysiert und sowohl gespeichert als auch exportiert werden. Es ist möglich verschiedene Einstellungen vorzunehmen, beispielsweise nur die Kommunikation der Devices über TCP anzusehen.

Für das Testen wird Wireshark nicht nur benötigt um sicherstellen zu können, dass die erstellten Pakete korrekt erstellt wurden oder ob sie fehlerhaft waren, sondern auch den ganzen Traffic zwischen den Devices zu kontrollieren.

Mit Wireshark lassen sich einzelne Pakete direkt analysieren, wie im Beispiel eines UDP-Paketes:

```
▸ Frame 274: 107 bytes on wire (856 bits), 107 bytes captured (856 bits) on interface 0
▸ Linux cooked capture
▸ IEEE 802.15.4 Data, Dst: 79:76:49:62:b5:20:11:e2, Src: 79:67:36:53:c2:be:c3:3c
▸ 6LoWPAN
▸ Internet Protocol Version 6, Src: fe80::c656:2059:e05f:77a2, Dst: fe80::7b76:4962:b520:11e2
▸ User Datagram Protocol, Src Port: 50000, Dst Port: 8888
▾ Data (20 bytes)
    Data: 6e6e6e6e6e6e6e6e6e6e6e7467757a756a6e6e6e
    [Length: 20]
```

Abbildung 3.4: Aufbau eines UDP-Paketes in Wireshark

Zu sehen in Abbildung 3.4 sind die einzelnen Layer des Paketes, vom niedrigsten oben stehend, bis zum höchsten unten stehend. Auch zu sehen sind Sender und Empfänger-Informationen, sowie die Belegung der einzelnen Flags in den jeweiligen Headern. Zusätzlich ist es auch möglich, sich den Hex-Dump anzeigen zu lassen:

```
0000  00 03 03 25 00 00 00 03  03 25 00 00 22 b8 00 f6   ···%···· ·%··"···
0010  41 dc 19 23 00 e2 11 20  b5 62 49 76 79 3c c3 be   A··#···  ·bIvy<··
0020  c2 53 36 67 79 60 00 00  00 00 1c 11 40 fe 80 00   ·S6gy`·· ····@···
0030  00 00 00 00 00 c6 56 20  59 e0 5f 77 a2 fe 80 00   ······V  Y·_w····
0040  00 00 00 00 00 7b 76 49  62 b5 20 11 e2 c3 50 22   ·····{vI b· ···P"
0050  b8 00 1c 00 ba 6e 6e 6e  6e 6e 6e 6e 6e 6e 6e 6e   ·····nnn nnnnnnnn
0060  74 67 75 7a 75 6a 6e 6e  6e 3e 0d                  tguzujnn n>·
```

Abbildung 3.5: Hex-Dump eines UDP-Paketes in Wireshark

In Abbildung 3.5 ist der Hex-Dump von dem Paket aus Abbildung 3.4 zu sehen. Rechts

daneben die Darstellung in American Standard Code for Information Interchange (ASCII).

3.3.4 Eigene Implementierungen

Bei einigen Tests wurde die Idee des Einlesens von Extensible Markup Language (XML)-Files von einem Fuzzing-Tool (6LowFuzzer) [16] übernommen, sodass einige Tests mit XML-Files und den dazugehörigen Komponenten erstellt werden konnten. Da der 6LowFuzzer ohne KillerBee-Funktionen entwickelt wurde, mussten einige Änderungen vorgenommen werden, damit der Einleseteil kompatibel zu KillerBee ist.

Module zum SYN-/ACK- und FIN-Flooden wurden ebenso wie eigene Module für einige Protokolle erstellt. Diese wurden in den Scriptbereich von KillerBee eingefügt.

Zum Testen einzelner Pakete eignete sich ein Script von KillerBee, welches mit selbst erstellten Komponenten erweitert wurde.

Somit wurde aus mehreren Programmen eines erstellt.

4 Testplanung und -durchführungen

Um Penetrationstests durchführen zu können wird zuerst geplant, welche Tests möglich und nötig sind. Da es im RIOT keine Passwörter gibt, ließen sich Tests auf Passwörter von vornherein ausschließen.

Im RIOT-Bereich eignet sich das Ersniffen von Daten per Man in the Middle (Kapitel 4.2), da auf diese Art und Weise sensible Daten ausgekundschaftet werden können, ohne direkt auf die Geräte Einfluss zu nehmen.

Weiterhin eignen sich Tests bei denen direkt mit dem Board interagiert wird, wie das Senden von Paketen an das Board und das Beobachten der Reaktion auf die Pakete dessen.

Außerdem gibt es Tests bei denen versucht werden kann, das Board zu überlasten und damit funktionsunfähig zu machen.

Um die Tests durchführen zu können wird Hardware benötigt, die im Folgenden vorgestellt wird.

4.1 Testumgebung

Die Testumgebung muss 6LoWPAN-kompatibel sein, da die Devices über IEEE 802.15.4 per 6LoWPAN kommunizieren.

Zum Testen wurden folgende Devices benutzt:

Anzahl	Hardware	Benutzung
2	Atmel SAM R21 Xplained Pro	zu testende Boards
1	Telos B	Attacker
1	TI CC2531	Sniffer

Tabelle 4.1: IoT-Devices

In Tabelle 4.1 sind die Boards aufgelistet, die im Folgenden noch einmal erläutert werden.

Das Atmel SAM R21 Xplained Pro Board wird als DUT genutzt. Es wird mittels PC mit RIOT geflashed.

Es hat alle nötigen Module zur Kommunikation im WPAN. Dieses Board wird auch im RIOT-Tutorial [17] genutzt, deswegen ist es gut zum Testen geeignet. Das Board ist verbunden mit einem PC, um so die Konsole zu verfolgen und etwaige Reaktionen auf Pakete festzustellen.

Das Telos B dient als Angreifer und wurde mit der Firmware von KillerBee geflasht. Somit ist es möglich alle Module von KillerBee, die dieses Board unterstützt, zu nutzen.

Dieses Board wurde lediglich einmal mit der KillerBee Firmware geflasht und wird nur zum Senden der Pakete genutzt.

Der Sniffer TI CC2531 von Texas Instruments dient lediglich als Sniffer um die Kommunikation via Wireshark zu überwachen.

Die Firmware ist vom Hersteller. Sowohl das Sniffen mit einer eigens von Texas Instruments (TI) veröffentlichten Software, sowie auch mit Wireshark ist möglich. Da Wireshark ein weitaus mächtigeres Tool ist, wird Wireshark verwendet.

Verwendet wird der Channel 11.

Folgend die grafische Darstellung.

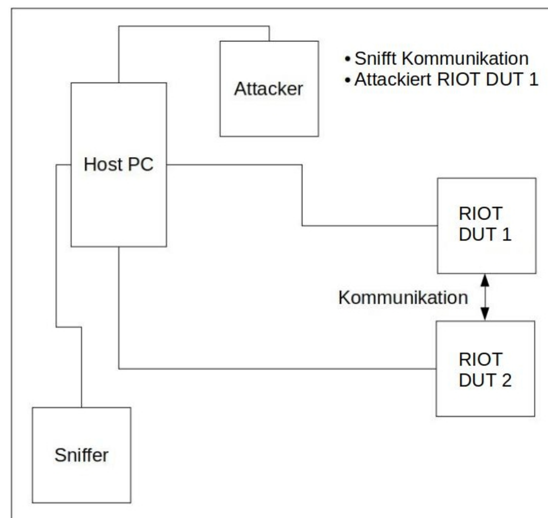

Abbildung 4.1: Grafische Darstellung des Versuchsaufbaus

4.2 Man in the Middle

Ein Man in the Middle ist ein Angriff, bei dem eine dritte Person den Traffic zwischen zwei Anderen sniffen möchte, oder sich jeweils für die andere Personen ausgibt.

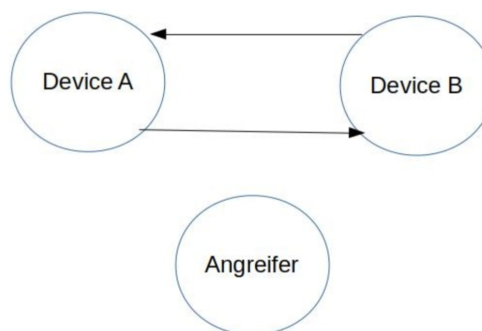

Abbildung 4.2: Kommunikation zwischen zwei Geräten ohne Eingriff des Angreifers

Wie in Abbildung 4.2 zu sehen ist, stehen als Ausgangssituation 2 Boards bereit (Board A und Board B), die miteinander kommunizieren.

In Abbildung 4.3 ist zu sehen, dass ein drittes Board eingesetzt wird, welches sich jeweils als das andere Board tarnt (für Board A ist der Angreifer als Board B identifiziert und für Board B ist der Angreifer als Board A identifiziert) und den Traffic mitliest, abspeichert, ggf. manipuliert und dann weiterleitet.

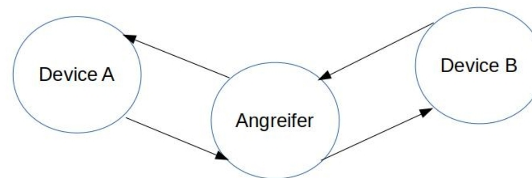

Abbildung 4.3: Kommunikation zwischen zwei Geräten mit Eingriff des Angreifers

Somit wäre das Routen manipuliert. Da aber keine Schutzmechanismen implementiert wurden, wird nicht weiter auf diese Art von Test eingegangen.

4.3 IP-Spoofing

Unter IP-Spoofing versteht man die Zusendung von IP-Paketen unter falscher Identität, um somit beispielsweise eine Kommunikation zu einem Server unter falscher Identität aufzubauen. Da diese Kommunikation allerdings nicht verbindungsorientiert ist (Antwortpakete werden an die falsche IP-Adresse gesendet), müssen die Antwortpakete ebenfalls ersniffed werden um an Informationen zu gelangen.

Spoofing ist mit Scapy möglich, da den Paketen eine falsche Source-Adresse übergeben werden kann.

Um eine zufällige IPv6-Adresse zu erstellen, wird mit einem Zufallsgenerator gearbeitet:

```
1  M=16**4
2  rnd="fe80::"+":".join(("%x" % random.randint(0, M) for i in range(4)))
```

Listing 4.1: Erstellung einer random IPv6-Adresse

Wie in Listing 4.1 zu sehen, wird die zufällige IPv6 Adresse so erstellt, dass die letzten 4 Blöcke der IP-Adresse zufällig generiert werden. Diese fügt man den ersten Blöcken ("fe80::") mit der .join-Funktion an.

Die nun erstellte IPv6-Adresse kann dem Paket bei dem Paketbau mittels 'src=rnd' übergeben werden.

Getestet wird nun, ob jegliche Pakete mit gefälschten Source-IP-Adressen von dem DUT empfangen und bearbeitet werden.

Im nächsten Abschnitt werden weitere Tests mit UDP-Paketen erläutert.

4.4 UDP

Für UDP bieten sich die Tests der Manipulation der Pakete, sowie das Testen des Paket Loss an.

4.4.1 Fault injection

Fault injection ist eine Technik zum Testen, wobei Fehler in ein System eingefügt werden um die Reaktion zu testen.[18]

Anfangs wird ein gültiges UDP-Paket gesnifft. Um Pakete zu sniffen werden mehrere Boards benötigt. Um die Kommunikation zwischen Geräten zu ersniffen, werden zwei Devices, die kommunizieren, sowie ein Sniffer, der die Pakete abfängt, benötigt.

Mit dem Sender werden nun einige UDP Pakete an das zweite Board gesendet, mit dem Sniffer ersniffed und per Scapy in einem Array gesichert. Nun ist es möglich sich die einzelnen Pakete in verschiedenen Formen anzeigen zu lassen, sowie die gesicherten Pakete selbst weiter zu senden. Die Source-IP ändert sich beim Weitersenden nicht, sofern sie nicht manuell geändert wird.

Gesnifft wird mit dem folgenden KillerBee Befehl:

```
1  a = kbsniff(iface=kbutils.devlist()[0][0],count=1,channel=11)
```

Listing 4.2: Ersniffen eines Paketes mittels KillerBee Befehl

Wie in Listing 4.2 zu sehen, ist eine KillerBee-Funktion vorhanden, mit der sich Pakete sniffen lassen. Die Übergabeparameter sind das Device, welches per 'kbutils.devlist()[0][0]' angesprochen wird, die Anzahl der Pakete (count), sowie der Channel, in dem gesnifft werden soll. A ist ein Array.

Das ersniffte Paket hat folgenden Inhalt:

```
1  packet = Dot15d4(fcf_destaddrmode=3, fcf_framever=1,
2    fcf_srcaddrmode=3, fcf_reserved_2=0, fcf_reserved_1=0,
3     fcf_ackreq=0, fcf_pending=0, seqnum=25,
4     fcf_frametype=1, fcf_security=0, fcf_panidcompress=1)/
5     Dot15d4Data(src_addr=8748020534567945026,
6     dest_addr=8752263614135538146, src_panid=None,
```

```
7    dest_panid=35)/Raw(load= '~3\xf0\xc0\x11"\xb8t\x99test')
```

Listing 4.3: Aufbau des ersnifften Paketes

Wie in Listing 4.3 zu erkennen, steht am Ende des Payloads nur 'test'. Nun wird dem Payload ein extra Buchstabe angehängt, um zu testen, wie das Device reagiert. Die übrigen Daten bleiben unverändert.

Das gesniffte Paket wird per folgendem KillerBee-Befehl versendet:

```
1  kbsendp(a[0],iface=kbutils.devlist()[0][0],channel=11)
```

Listing 4.4: Versenden eines ersnifften Paketes per KillerBee

Wie in Listing 4.4 zu sehen, wird das erste Element der ersnifften Pakete per Device, welches per 'kbutils.devlist()[0][0]' angesprochen wird, an Channel 11 gesendet.

Im nächsten Abschnitt wird sich des Packet Loss angenommen.

4.4.2 Testen des Packet Loss

Der Packet Loss ist die Angabe in Prozent der Pakete, die nicht beim Empfänger ankommen. Er sollte der Zuverlässigkeit der Kommunikation wegen relativ niedrig liegen. Durch zu viele verloren gegangene Pakete kann die einwandfreie Kommunikation nicht gewährleistet werden.

Je niedriger der Packet Loss ist, desto zuverlässiger ist die Kommunikation. Da UDP immer mit nicht bemerktem Paket Loss einhergehen kann (verbindungsloses Protokoll), wurde hierfür auf dem DUT analysiert, wie viele Pakete beim Empfänger ankommen.

Um zu analysieren wie viele Pakete das DUT erreichen, werden im DUT die empfangenen Pakete gezählt und dieser Zähler ausgegeben. Darüber hinaus wird auch ein Zähler für Pakete, die ohne Daten ankamen, ausgegeben.

Für das Ausgeben wird ein leicht abgewandelter Code aus dem RIOT-Tutorial benutzt.

Die Zähler werden bei jedem für sie empfangenen Paket inkrementiert und ausgegeben. Nach jedem Testdurchlauf wird das Board resettet, damit sich der Zähler zurücksetzt.

In KillerBee wird mit Hilfe von Scapy ein UDP Paket mit folgendem Inhalt erstellt:

```
1  packet = Dot15d4(fcf_destaddrmode=3, fcf_framever=1, fcf_srcaddrmode=3,
       fcf_reserved_2=0, fcf_reserved_1=0, fcf_ackreq=0, fcf_pending=0, seqnum=25,
       fcf_frametype=1, fcf_security=0, fcf_panidcompress=1)/Dot15d4Data(src_addr
       =8748020534567945020, dest_addr=8752263614135538146, src_panid=None, dest_panid
       =35)/IPv6(dst="fe80::7b76:4962:b520:11e2")/UDP(dport=53, sport=50000)/Raw('
       nnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnn')
```

Listing 4.5: Erstellung eines Testpaketes zum Packet Loss

In Listing 4.5 ist der Aufbau des Paketes zu sehen.

Für das Testen des Packet Loss werden schnellstmöglich hintereinander sowohl 10.000 als auch 100.000 Pakete versendet. Die Dauer der Tests beträgt, mit der Einstellung von einem minimalen Delay von 0, bei 10.000 Paketen ~5 Minuten und bei 100.000 Paketen ~50 Minuten.

Für das Testen mit TCP Paketen eignen sich andere Tests, wie folgend dargestellt.

4.5 TCP

Bei TCP bietet sich beispielsweise Tests wie das Syn-Flooding an.

4.5.1 TCP-Flooding

Zum Flooden mit SYN-/ACK- oder FIN-Paketen wurde ein Programm zum erstellen von Paketen mit gesetzten Flags erstellt. Die Idee ist einerseits zu testen, ob mehrere Verbindungen an das Board möglich sind, als auch das Füllen des Caches um das Gerät handlungsunfähig zu machen.

Sowohl das Flooden per ACK-, SYN- und per FIN-Nachrichten ist damit möglich, da jeweils die Flags pro Modul anders gesetzt werden.

Um ein TCP-Paket mit gesetztem SYN-Flag zu erstellen wird im TCP Layer flags=" gesetzt. Das jeweilige Pendant wäre das A und das "*F*" Flag.

Das Paket ist wie folgt aufgebaut:

```
>>> packet
<Dot15d4  fcf_reserved_1=0 fcf_panidcompress=True fcf_ackreq=False fcf_pending=F
alse fcf_security=False fcf_frametype=Data fcf_srcaddrmode=Long fcf_framever=1 f
cf_destaddrmode=Long fcf_reserved_2=0 seqnum=25 |<Dot15d4Data  dest_panid=0x23 d
est_addr=07:97:64:96:2b:52:01:1e src_addr=07:96:73:65:3c:2b:ec:33 |<IPv6  nh=TCP
 src=fe80::c656:2059:e05f:77a2 dst=fe80::7b76:4962:b520:11e2 |<TCP  sport=12345
dport=12345 flags=S |>>>>
>>> 
```

Abbildung 4.4: TCP-Paket zum SYN-Flooden

Wie in Abbildung 4.4 zu sehen, ist das SYN-Flag gesetzt.

In RIOT gibt es einen Counter, um die Anzahl der Verbindungen zu bestimmen, der auf 1 gesetzt wird, somit ist nur eine Verbindung pro Zeitpunkt erlaubt. Außerdem wird mit einer State Machine gearbeitet.

Als erstes werden 10 Verbindungen angestrebt, somit 10 SYN-Nachrichten an das DUT gesendet. Dies geschieht mit eines eigens dafür implementierten Programmes. Mit diesem Programm ist es möglich feste IP-Adressen und Ports zu verwenden, als auch Zufallsgenerierte zu benutzen. Geflooded wird mit der KillerBee-Funktion **kb.inject**.

Als erstes wird mit einer festen IP-Adresse und festen Source-Ports getestet. Bei dem DUT ist der Port für den TCP Server auf 8888 eingestellt.

Die Flooding-Tools lassen sich ebenfalls per Bash-Befehl aufrufen.

4.6 Weitere Testplanungen

4.6.1 NDP

Zum Testen der NDP-Implementierung eignet sich beispielsweise das Fluten der Neighbor Cache Table (NCT)'s.

Der Angreifer sendet dem DUT unter verschiedenen IP-Adressen Nachrichten, um somit die NCT zu füllen und zum überlaufen zu bringen.

Entsprechende Pakete zum Senden an das DUT sind ansatzweise implementiert.

4.6.2 CoAP

Bei CoAP lässt sich testen, ob das DUT auf Nachrichten wie GET, PUT, POST reagiert und beispielsweise der richtigen Inhalt der angeforderten Ressourcen korrekt angezeigt wird.

Implementiert sind bereits Pakete zum Verschicken per KillerBee.

5 Ergebnisse

5.1 IP-Spoofing

Die manipulierten Pakete wurden von dem DUT empfangen und bearbeitet.

Die Source-Adresse der empfangenen Pakete war irrelevant.

5.2 UDP

Im UDP wurden speziell noch die Manipulation der ersnifften Pakete, sowie das Testen des Packet Loss geprüft.

5.2.1 Fault injection

Nachdem das ersniffte Paket erfolgreich beim Empfänger ankam, wurde getestet, wie das Device auf die Manipulation dieses Paketes reagiert.

Nachdem dem Paket der Buchstabe 'a' angehängt wurde, wurde es nicht mehr vom Empfänger erkannt, da somit die Checksumme fehlerhaft war.

fe80::7b76:4962:b520:11e2	UDP	52	49169 → 8888 Len=4
fe80::7b76:4962:b520:11e2	UDP	52	49169 → 8888 Len=4
fe80::7b76:4962:b520:11e2	UDP	53	49169 → 8888 Len=5
fe80::7b76:4962:b520:11e2	UDP	53	49169 → 8888 Len=5
fe80::7b76:4962:b520:11e2	UDP	53	49169 → 8888 Len=5

Abbildung 5.1: Senden von nicht manipulierten und manipulierten UDP-Paketen in Wireshark

Wie in Abbildung 5.1 ersichtlich, wurden erst die normalen Pakete mit einem Payload von 4 (‚test‘) versendet, darauffolgend Pakete mit einem Payload von 5 (‚testa‘). Gleichzeitig kamen im RIOT DUT lediglich die Pakete mit dem Payload (‚test‘) an.

```
2018-12-05 16:28:19,212 - INFO # Recvd: test
2018-12-05 16:28:19,244 - INFO # Recvd: test
2018-12-05 16:28:19,276 - INFO # Recvd: test
2018-12-05 16:28:19,308 - INFO # Recvd: test
2018-12-05 16:28:19,341 - INFO # Recvd: test
2018-12-05 16:28:19,371 - INFO # Recvd: test
```

Abbildung 5.2: Output des RIOT Device under Tests's während des Testens mit manipulierten UDP-Paketen

Abbildung 5.2 zeigt, dass die Pakete nicht empfangen wurden. Lediglich die zuvor gesendeten Pakete mit ‚test‘als Payload wurden erkannt.

Im DUT wurde erkannt, dass das Paket durch die falsche Checksumme fehlerhaft war.

Dies zeigte auch der Output am DUT:

```
18:43:55,322 - INFO # udp: GNRC_NETAPI_MSG_TYPE_RCV
18:43:55,327 - INFO # udp: received packet with invalid checksum, dropping it
```

Abbildung 5.3: Verwerfen des Paketes von dem Device under Test durch falsche Checksumme

Wie in Abbildung 5.3 zu sehen, wurde das fehlerhafte Paket aufgrund der falschen Checksumme verworfen.

Da fehlerhafte Pakete verworfen werden sollen, war dieser Test erfolgreich.

5.2.2 Testen des Packet Loss

Um den Paket Loss zu Testen wurden schnellstmöglich viele Pakete versendet.

```
Injecting 10000 packets on channel 11 page 0 with a 89 length
sending: 41dc192300e21120b5624976793cc3bec2533667796000000001c1140fe8000000000
000c6562059e05f77a2fe800000000000007b764962b52011e2c35022b8001c00ba6e6e6e6e6e6e6
e6e6e6e6e7467757a756a6e6e6e
[sabyse@mobi37 scripts]$
```

Abbildung 5.4: Senden von 10.000 UDP-Paketen per Script

```
Injecting 100000 packets on channel 11 page 0 with a 89 length
sending: 41dc192300e21120b5624976793cc3bec25336677960000000001c1140fe80000000000
000c6562059e05f77a2fe800000000000007b764962b52011e2c35022b8001c00ba6e6e6e6e6e6e6
e6e6e6e6e7467757a756a6e6e6e
```

Abbildung 5.5: Senden von 100.000 UDP-Paketen per Script

Wie zu sehen, wurden 10.000 (Abbildung 5.4) und 100.000 (Abbildung 5.5) Pakete mit dem Inhalt in Bytes dargestellt, versendet.

Näher beschrieben im Kapitel 3.3.2.

Durch die Implementierung der Zähler im DUT lässt sich feststellen, dass nicht alle Pakete bei dem Empfänger ankamen:

```
2018-12-05 17:05:14,906 - INFO # Recvd 9979. Packet: nnnnnnnnnnntguzujnnn
2018-12-05 17:05:14,937 - INFO # Recvd 9980. Packet: nnnnnnnnnnntguzujnnn
2018-12-05 17:05:14,969 - INFO # Recvd 9981. Packet: nnnnnnnnnnntguzujnnn
```

Abbildung 5.6: 10.000 empfangene UDP-Pakete

```
2018-12-06 16:09:51,727 - INFO # Recvd 99862. Packet: nnnnnnnnnnntguzujnnn. NoData Packets: 0
2018-12-06 16:09:51,758 - INFO # Recvd 99863. Packet: nnnnnnnnnnntguzujnnn. NoData Packets: 0
```

Abbildung 5.7: 100.000 empfangene UDP-Pakete

Durch die mehrfache Ausführung unter gleichen Bedingungen lies sich ein durchschnittlicher Paket Loss von 0,1795% feststellen.

Anzahl der gesendeten Pakete	angekommene Pakete	Paket Loss
10.000	9.997	0,03
10.000	9.981	0,19
10.000	9.976	0,24
10.000	9.977	0,23
10.000	9.975	0,25
100.000	99.863	0,137
Durchschnittlicher Paket Loss		0,1795

Tabelle 5.1: Tabelle mit Auflistung zum Packet Loss

Bei 10.000 Paketen und bei 100.000 Paketen ist der Paket Loss ähnlich.

Da über Wireless-Verindungen ein Packet Loss durch verschiedene Einflüsse entstehen kann, lässt sich sagen, dass RIOT sehr stabil ist, da dieser Wert sehr niedrig ist.

5.3 TCP

Speziell für TCP wurden SYN, ACK und FIN-Floods durchgeführt.

5.3.1 SYN-Flooding

Durch die Einstellung von nur einer Verbindung zur Zeit, wird auf das erste Paket geantwortet und alle anderen SYN-Nachrichten ignoriert. Da sich der Knoten nun in einer halb offenen Verbindung befindet, ist es nicht möglich, sich noch einmal mit dem DUT zu verbinden.

Auf die erste Nachricht reagiert das DUT wie folgt:

Protocol	Length	Info
TCP	99	2325 → 8888 [SYN] Seq=0 Win=7665 Len=0
TCP	66	8888 → 2325 [SYN, ACK] Seq=0 Ack=1 Win=1220 Len=0 MSS=1220
TCP	66	[TCP Retransmission] 8888 → 2325 [SYN, ACK] Seq=0 Ack=1 Win=1220 Len=0 MSS=1220
TCP	66	[TCP Retransmission] 8888 → 2325 [SYN, ACK] Seq=0 Ack=1 Win=1220 Len=0 MSS=1220
TCP	66	[TCP Retransmission] 8888 → 2325 [SYN, ACK] Seq=0 Ack=1 Win=1220 Len=0 MSS=1220

Abbildung 5.8: Antwort des RIOT-Boards auf ein SYN-Paket in Wireshark

In Abbildung 5.8 ist zu sehen, dass das DUT auf die erste Nachricht antwortet und die SYN/ACK-Nachricht erneut versendet. Dies geschieht, da ich nur SYN-Nachrichten versende und nicht auf die Antwort reagiere.

	ICMPv6	61	Router Solicitation from 79:76:49:62:b5:20:11:e2
9…	TCP	66	[TCP Retransmission] 8888 → 2325 [SYN, ACK] Seq=0 Ack=
9…	TCP	66	[TCP Retransmission] 8888 → 2325 [SYN, ACK] Seq=0 Ack=
9…	TCP	66	[TCP Retransmission] 8888 → 2325 [SYN, ACK] Seq=0 Ack=
9…	TCP	66	[TCP Retransmission] 8888 → 2325 [SYN, ACK] Seq=0 Ack=
	ICMPv6	61	Router Solicitation from 79:76:49:62:b5:20:11:e2
9…	TCP	66	[TCP Retransmission] 8888 → 2325 [SYN, ACK] Seq=0 Ack=
9…	TCP	66	[TCP Retransmission] 8888 → 2325 [SYN, ACK] Seq=0 Ack=
9…	TCP	66	[TCP Retransmission] 8888 → 2325 [SYN, ACK] Seq=0 Ack=
9…	TCP	66	[TCP Retransmission] 8888 → 2325 [SYN, ACK] Seq=0 Ack=
	ICMPv6	61	Router Solicitation from 79:76:49:62:b5:20:11:e2

Abbildung 5.9: Senden von Retransmissionnachrichten vom Device under Test, aus Wireshark

Das DUT schickt, wie in Abbildung 5.9 zu sehen, in bestimmten Abständen wiederholt Retransmission-Nachrichten, um die Verbindung mit dem Absender der ersten SYN-Nachricht herzustellen.

```
99 [TCP Retransmission] 3413 → 8888 [SYN] Seq=0 Win=5628 Len=0
99 [TCP Retransmission] 3413 → 8888 [SYN] Seq=0 Win=5628 Len=0
99 [TCP Retransmission] 3413 → 8888 [SYN] Seq=0 Win=5628 Len=0
99 [TCP Retransmission] 3413 → 8888 [SYN] Seq=0 Win=5628 Len=0
66 [TCP Retransmission] 8888 → 3156 [SYN, ACK] Seq=0 Ack=1 Win=1220 Len=0 MSS=1220
66 [TCP Retransmission] 8888 → 3156 [SYN, ACK] Seq=0 Ack=1 Win=1220 Len=0 MSS=1220
66 [TCP Retransmission] 8888 → 3156 [SYN, ACK] Seq=0 Ack=1 Win=1220 Len=0 MSS=1220
66 [TCP Retransmission] 8888 → 3156 [SYN, ACK] Seq=0 Ack=1 Win=1220 Len=0 MSS=1220
```

Abbildung 5.10: Versuche vom Device under Test mit dem ersten Sender einer Nachricht eine Verbindung aufzubauen

Weitere SYN-Nachrichten werden ignoriert, da die Verbindungsanzahl auf eine Verbindung reduziert wurde und, wie in Abbildung 5.10 zu sehen, immer noch versucht wird mit dem ersten Sender eine Verbindung aufzubauen. Hierbei ist es irrelevant, ob die Nachrichten vom gleichen Sender und Port, oder mit Zufallszahlen gefüllte Pakete gesendet wurden.

5.3.2 ACK- und FIN-Flooding

Beim ACK-Flooding reagiert das Board sofort mit einer Reset Nachricht, um danach eine Verbindung neu aufbauen zu können.

```
fe80::c656:2059:e05…  fe80::7b76:4962…  TCP    99 [TCP Dup ACK 0#32] 2527 → 8888 [ACK] Seq=1 Ack=1 Win=1042 Len=0
fe80::7b76:4962:b52…  fe80::c656:2059…  TCP    62 8888 → 2527 [RST] Seq=1 Win=0 Len=0
fe80::7b76:4962:b52…  fe80::c656:2059…  TCP    62 8888 → 2527 [RST] Seq=1 Win=0 Len=0
fe80::7b76:4962:b52…  fe80::c656:2059…  TCP    62 8888 → 2527 [RST] Seq=1 Win=0 Len=0
fe80::7b76:4962:b52…  fe80::c656:2059…  TCP    62 8888 → 2527 [RST] Seq=1 Win=0 Len=0
fe80::c656:2059:e05…  fe80::7b76:4962…  TCP    99 [TCP Dup ACK 0#33] 2527 → 8888 [ACK] Seq=1 Ack=1 Win=1042 Len=0
fe80::c656:2059:e05…  fe80::7b76:4962…  TCP    99 [TCP Dup ACK 0#34] 2527 → 8888 [ACK] Seq=1 Ack=1 Win=1042 Len=0
fe80::c656:2059:e05…  fe80::7b76:4962…  TCP    99 [TCP Dup ACK 0#35] 2527 → 8888 [ACK] Seq=1 Ack=1 Win=1042 Len=0
fe80::c656:2059:e05…  fe80::7b76:4962…  TCP    99 [TCP Dup ACK 0#36] 2527 → 8888 [ACK] Seq=1 Ack=1 Win=1042 Len=0
```

Abbildung 5.11: ACK-flooding in Wireshark

Die weiteren Nachrichten, im unteren Bereich von Abbildung 5.11, werden ignoriert.

Auch beim FIN-Flooding, wie in Abbildung 5.12 zu sehen, reagiert das Board mit RST/ACK Nachrichten um die scheinbar fehlerhafte Verbindung zu resetten.

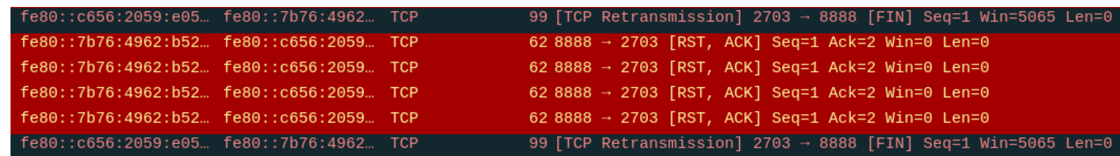

Abbildung 5.12: FIN-Flooding in Wireshark

6 Zusammenfassung und Ausblick

Das Testen im IoT ist notwendig, da sich dadurch einige Sicherheitslücken gezeigt haben. Beim SYN-Flooding wurde ein Fehler entdeckt. Auch wurde durch die Fault injection gezeigt, dass RIOT angemessen auf fehlerhafte Pakete reagiert und somit auch Tests erfolgreich abschlossen.

Da das IoT immer mehr an Bedeutung gewinnt und sich weiterentwickelt, ist es notwendig auch weiterhin auf Sicherheitslücken zu testen. Nicht nur der vorhandene Code, sondern auch hinzugefügter, mit neuen Patches hinzugefügten Features, müssen weiterhin regelmäßig getestet werden, um weitere Sicherheitslücken rechtzeitig zu erkennen und beseitigen zu können.

Vorausschauend sollte das Penetrationstesten in RIOT weiter vertieft und erweitert werden, da noch weitere Netzwerkprotokolle zum Testen vorhanden sind.

Für einige weitere Protokolle wurden bereits Ansätze implementiert, sodass weitere Tests implementiert werden können.

Literaturverzeichnis

[1] Edward Kulperger. The future of connected vehicles and the iot. https://iot.telefonica.com/blog/the-future-of-connected-vehicles-and-the-iot, Dezember 2017. Zugriffsdatum: 04. Februar 2019.

[2] IFA. Ifa berlin. https://b2b.ifa-berlin.com/de/. Zugriffsdatum: 20. Januar 2019.

[3] J. Postel. Transmission Control Protocol. RFC 793, IETF, September 1981.

[4] J. Postel. User Datagram Protocol. RFC 768, IETF, August 1980.

[5] Z. Shelby, S. Chakrabarti, E. Nordmark, and C. Bormann. Neighbor Discovery Optimization for IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs). RFC 6775, IETF, November 2012.

[6] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler. Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC 4944, IETF, September 2007.

[7] Z. Shelby, K. Hartke, and C. Bormann. The Constrained Application Protocol (CoAP). RFC 7252, IETF, June 2014.

[8] Emmanuel Baccelli, Cenk Gündogan, Oliver Hahm, Peter Kietzmann, Martine Lenders, Hauke Petersen, Kaspar Schleiser, Thomas C. Schmidt, and Matthias Wählisch. RIOT: an Open Source Operating System for Low-end Embedded Devices in the IoT. *IEEE Internet of Things Journal*, 5(6):4428–4440, December 2018.

[9] RIOT-OS. Riot in a nutshell. https://doc.riot-os.org/index.html#riot-in-a-nutshell. Zugriffsdatum: 20. Januar 2019.

[10] RIOT-OS. Riot-os/tutorial. https://github.com/RIOT-OS/Tutorials, September 2018. Zugriffsdatum: 20. Januar 2019.

[11] Bsi - startseite is-penetrationstest und is-webcheck. https://www.bsi.bund.de/DE/Themen/Cyber-Sicherheit/Dienstleistungen/ISPentest_ISWebcheck/ispentest_iswebcheck_node.html. Zugriffsdatum: 04. Februar 2019.

[12] Philippe Biondi and the Scapy community. Welcome to scapy's documentation! https://scapy.readthedocs.io/en/latest/introduction.html, Dezember 2018. Zugriffsdatum: 23. Januar 2019.

[13] Oliver Eggert. Ipv6 packet creation with scapy documentation. https://www.idsv6.de/Downloads/IPv6PacketCreationWithScapy.pdf, Januar 2012.

[14] LLC River Loop Security. Killerbee 2.0 - river loop security. https://www.riverloopsecurity.com/projects/killerbee, Dezember 2018.

[15] Wireshark - go deep. https://www.wireshark.org.

[16] Olivier Festor César Bernardini, Abdelkader Lahmadi. Development of a fuzzing tool for the 6LoWPAN protocol. [Technical Report] RR-7817. Technical report, 2011. pp.50. <hal-00645948>.

[17] RIOT. Tutorials/task-06. https://github.com/RIOT-OS/Tutorials/tree/master/task-06, November 2018.

[18] R. Kaksonen. A Functional Method for Assessing Protocol Implementation Security. VTT Publications 448, VTT, October 2001.

Glossar

RIOT Betriebssystem im IoT.

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, den ______________________ ______________________