Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Ein Ansatz für IoT-Sicherheitstests basierend auf dem MQTT-Protokoll

Kai Chen

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. Stefan Wagner
Betreuer/in:	Dr. Ana Cristina Franco da Silva
Beginn am:	7. Juni 2021
Beendet am:	7. Dezember 2021

Kurzfassung

Das Internet der Dinge (IoT) besteht aus einer stark wachsenden Anzahl an vernetzten Geräten und gewinnt immer mehr an Bedeutung. Aufgrund der Komplexität und Heterogenität der verwendeten Technologien existieren im IoT-Bereich viele Sicherheitsprobleme. MQTT ist das meist verwendete IoT-spezifische Protokoll für die Kommunikation, wodurch es einen attraktiven Angriffspunkt darstellt. Daher muss die Sicherheit bei MQTT-Systemen gewährleistet sein. Durch eine Literaturrecherche wurden als Hauptprobleme im Zusammenhang mit der Sicherheit von MQTT die unsichere Standardkonfiguration der Broker, sowie Schwachstellen im Umgang mit fehlerhaften Paketen identifiziert. Das Ziel dieser Arbeit ist, einen Testansatz zu entwerfen, der die Sicherheitsprobleme von MQTT-Broker-Implementierungen mittels automatisierten Sicherheitstests untersucht. Der Ansatz, genannt MQTT-AIO, besteht aus drei Testkomponenten und ist in der Lage, die Konfiguration des Brokers zu analysieren, Angriffe basierend auf Angriffsmustern auszuführen und weitere Schwachstellen mithilfe von Fuzzing zu finden. Eine weitere Komponente überwacht das System während des Testprozesses und zeichnet relevante Daten auf. Die Ergebnisse der Testdurchläufe werden als Bericht ausgegeben und können weiter analysiert werden. Der Testansatz MQTT-AIO wird im Rahmen dieser Masterarbeit prototypisch implementiert und anhand einer Fallstudie validiert.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Abkürzungsverzeichnis

DDoS Distributed Denial of Service. 37

DoS Denial of Service. 34

IoT Internet der Dinge. 15

MBT Modellbasiertes Testen. 25

MQTT Message Queue Telemetry Transport. 29

SDA Slow DoS Attack. 39

SUT System Under Test. 22

TCP Transmission Control Protocol. 34

TLS Transport Layer Security. 34

1 Einleitung

In den letzten Jahren hat das Thema Internet der Dinge (IoT) stark an Aufmerksamkeit gewonnen, da es dem Menschen potenziell enorme Vorteile ermöglicht. IoT basiert auf der Vernetzung von intelligenten Geräten, die über das Internet miteinander kommunizieren [WB16] und umfasst zahlreiche verschiedene Anwendungsbereiche wie bspw. Smart Cities, Smart Home und Industrie 4.0 [VF13]. Laut dem Statistik-Portal Statista sind im Jahr 2021 13,8 Milliarden IoT-Geräte vorhanden und bis 2025 wird erwartet, dass die Zahl auf 30,9 Milliarden Geräte steigt. Beispiele für IoT-Geräte sind vernetzte Autos und Smart Home Geräte wie steuerbare Lampen oder Amazon Alexa [STS20].

Durch die große Anzahl an Geräten, die mit dem Internet verbunden sind, und die damit einhergehenden riesigen Datenmengen, steigt die Komplexität von IoT-Netzwerken [HKH19]. Hinzu kommt, dass das IoT verschiedene bestehende Technologien wie zum Beispiel drahtlose Sensornetzwerke oder Cloud Computing kombiniert und die Sicherheitsprobleme der einzelnen Technologien erbt [ACH15]. Dadurch nehmen die Sicherheitsherausforderungen im IoT-Bereich zu.

Die Eclipse IoT Working Group hat bei ihrer Umfrage „IoT Developer Survey Key Findings“ aus dem Jahr 2020 die wichtigsten Herausforderungen im Bereich IoT untersucht. Dabei wurde als größte Herausforderung die mangelnde Sicherheit identifiziert. Als meist genutzte Technik, um IoT-Lösungen sicherer zu gestalten, wurde von 43% der Entwickler die Kommunikationssicherheit angegeben. Das MQTT-Protokoll ist mit 41% das am häufigsten eingesetzte aller IoT-spezifischen Protokolle für die Kommunikation, weshalb es einen attraktiven Angriffspunkt darstellt [EDS20].

Das MQTT-Protokoll ist ein standardisiertes und zuverlässiges Netzwerkprotokoll für die Übermittlung von Nachrichten zwischen Geräten. Das Protokoll ist ideal geeignet für den Einsatz in Umgebungen mit eingeschränkten Ressourcen wie begrenztem Arbeitsspeicher, was oftmals bei IoT-Geräten der Fall ist. Da die IoT-Geräte potenziell sensible Daten übermitteln, müssen Implementierungen des MQTT-Protokolls sicher sein.

Das Ziel dieser Arbeit ist, einen Ansatz zu entwickeln, der verschiedene MQTT-Implementierungen automatisiert hinsichtlich der Sicherheit testet. Um den derzeitigen Stand der Technik bezüglich MQTT-Sicherheit zu untersuchen, wird im Rahmen dieser Masterarbeit eine Literaturrecherche durchgeführt. Anschließend wird in einer weiteren Literaturrecherche untersucht, welche Möglichkeiten zum Testen von MQTT-Systemen bereits bestehen. Basierend auf diesen Erkenntnissen wird ein Testansatz entwickelt, das MQTT-Implementierungen auf verschiedene Sicherheitsaspekte prüft. Dieses Konzept wird daraufhin implementiert und anhand einer Fallstudie validiert.

Die Arbeit ist in folgender Weise gegliedert: In Kapitel 2 werden die Grundlagen von IoT und IoT-Sicherheit beschrieben. Auch Möglichkeiten für das automatisierte Testen der Sicherheit von IoT-Systemen werden erläutert. Kapitel 3 behandelt die Grundlagen des MQTT-Protokolls und enthält eine Analyse der damit verbundenen Sicherheitsherausforderungen. In Kapitel 4 werden verwandte Arbeiten beschrieben, die sich in der Vergangenheit bereits mit dem Thema Sicherheitstests im MQTT-Bereich beschäftigt haben. Daraufhin wird in Kapitel 5 der Testansatz MQTT-AIO und dessen Implementierung erklärt, sowie eine Fallstudie durchgeführt, in der die Nützlichkeit des Systems validiert wird. Schließlich fasst Kapitel 6 die Ergebnisse der Arbeit zusammen und stellt Anknüpfungspunkte für zukünftige Erweiterungen vor.

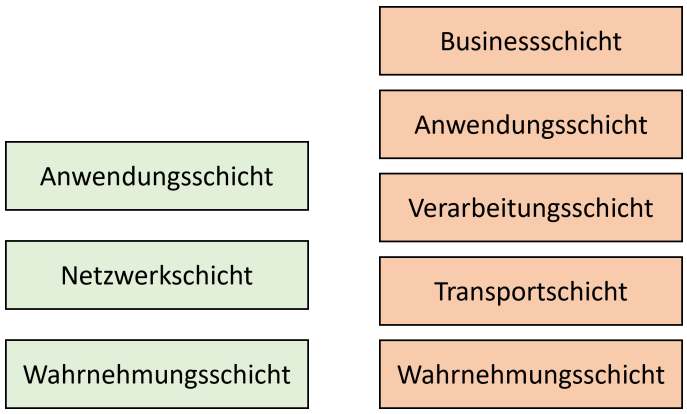
2 IoT-Sicherheit

Das Internet der Dinge (engl. Internet of Things - IoT) besteht aus hochentwickelten Sensoren, Aktoren und Chips, die in physische Dinge um uns herum eingebettet sind. Diese Dinge sind miteinander über das Internet verbunden und tauschen große Datenmengen untereinander und mit anderen digitalen Komponenten aus, ohne dass ein Mensch eingreifen muss. Hierfür werden unter anderem Technologien, wie bspw. Kommunikationstechnologien, Sensornetzwerke und Internetprotokolle verwendet [AGM+15].

Um die weitergehenden Themen einordnen zu können, werden in diesem Kapitel die Grundlagen von IoT erklärt. Zuerst wird auf die Architektur von IoT-Systemen eingegangen und mögliche Sicherheitsprobleme, die im IoT-Bereich auftreten, werden aufgezählt. Anschließend werden Ansätze zum automatisierten Testen der Sicherheit von IoT-Systemen aufgezeigt.

2.1 IoT-Architektur

Es gibt zahlreiche Ansätze für IoT-Architekturen. Die am meisten verbreitete Variante teilt die Architektur in drei Schichten auf, wie Abbildung 2.1 (a) zeigt.



(a) Drei-Schichten-Modell **(b)** Fünf-Schichten-Modell

Abbildung 2.1: IoT-Architekturen

Die Wahrnehmungsschicht ist hauptsächlich für das Sammeln von Daten von Geräten oder der Umwelt zuständig. Dies wird durch Sensoren an Geräten ermöglicht, wodurch physikalische Parameter, wie bspw. die Temperatur, erfasst werden können. Diese Daten werden anschließend an die Netzwerkschicht weitergegeben, die für die Verarbeitung der Sensordaten und die sichere Übertragung dieser Daten zwischen der Wahrnehmungsschicht und der Anwendungsschicht zuständig ist. In der Netzwerkschicht kann zudem eine Verbindung zu anderen intelligenten Geräten, Servern oder Netzwerkgeräten hergestellt werden. Die Anwendungsschicht analysiert die empfangenen Daten und stellt diese in Diensten für die Endnutzer bereit [JBH+20] [AAAC16].

Die Drei-Schichten-Architektur ist für die IoT-Forschung oft nicht ausreichend, da häufig eine genauere Unterteilung der IoT-Architektur erforderlich ist [KM18]. Eine feinere Unterteilung bildet bspw. das Fünf-Schichten-Modell nach Wu et al. [WLL+10], welches in Abbildung 2.1 (b) dargestellt ist. Die Hauptaufgabe der Wahrnehmungsschicht besteht wie beim Drei-Schichten-Modell darin, die physikalischen Eigenschaften von Objekten mit Hilfe verschiedener Sensoren zu erfassen und diese in digitale Signale umzuwandeln, die für die Übertragung im Netzwerk geeignet sind. Diese Signale werden mithilfe von Technologien wie bspw. 3G, Bluetooth oder Infrarot über die Transportschicht an die Verarbeitungsschicht weitergeleitet. Die Verarbeitungsschicht speichert, analysiert und verarbeitet die erhaltenen Informationen. Hierfür werden Technologien wie Cloud Computing und Ubiquitous Computing verwendet. Die Anwendungsschicht verwendet diese verarbeiteten Daten und umfasst verschiedene IoT-Anwendungen, wie bspw. Identitätsauthentifizierung oder standortbezogene Dienste. Die oberste Schicht ist die Businessschicht, welche für die Verwaltung von IoT-Anwendungen und die Entwicklung von Geschäfts- und Gewinnmodellen zuständig ist.

2.2 Sicherheit im IoT-Bereich

Immer mehr Unternehmen bauen Computer und Sensoren in ihre Produkte ein und ermöglichen dadurch eine Verbindung mit dem Internet. Während die Unternehmen untereinander darum konkurrieren, als erstes ein bestimmtes Produkt auf den IoT-Markt zu bringen, steht die Gerätesicherheit nicht im Vordergrund [SNF+17]. Jedoch beinhaltet IoT verschiedenste Infrastrukturen und Technologien, die jeweils eigene Herausforderungen und Schwachstellen mit sich bringen, welche im IoT-Bereich gleichzeitig beachtet werden müssen. Durch die Koexistenz und Zusammenarbeit der verschiedenen Infrastrukturen und Technologien können zusätzliche Probleme entstehen. Daher ist IoT ein Themengebiet mit besonders vielen Angriffsflächen [ACH15].

Die enorme Anzahl an IoT-Geräten auf dem Markt macht die Skalierbarkeit zu einem wichtigen Thema, wenn es um die Entwicklung effizienter Sicherheitsmechanismen geht. Der Schwerpunkt muss hierbei auf die Entwicklung dezentraler Sicherheitsmechanismen gelegt werden, anstatt auf zentralisierte, wie es in traditionellen Softwaresystemen der Fall ist. IoT-Sicherheitslösungen müssen kosteneffizient skalierbar sein, da diese häufig

auf Hunderttausende von Endpunkten angewandt werden. Außerdem kann jedes dieser Geräte im Laufe der Zeit riesige Datenströme erzeugen, sodass es wichtig ist, Sicherheitsmechanismen zu entwickeln, die eine so große Anzahl an Daten effektiv schützen können [DR19].

Ein weiteres Problem ist, dass die meisten IoT-Endgeräte nur über begrenzte Ressourcen wie zum Beispiel CPU-Leistung, Arbeitsspeicher und Übertragungsreichweite verfügen. Dies macht diese Geräte zu einem leichten Ziel für DoS-Angriffe, bei denen der Angreifer die begrenzten Ressourcen dieser Geräte überwältigen und eine Dienstunterbrechung verursachen kann. Zudem stellen die begrenzten Ressourcen der IoT-Geräte die Entwicklung von Sicherheitsprotokollen vor neue Herausforderungen, da die typischen Kryptographietechniken der Protokolle meist rechenintensiv sind [DR19].

Auch das Aktualisieren von gefundenen Schwachstellen ist eine Herausforderung im IoT-Bereich. Zum Schutz der Cybersicherheit des Systems ist es von entscheidender Bedeutung, dass die Schwachstellen sofort nach ihrer Entdeckung behoben werden. Im IoT-Bereich erfolgen Systemaktualisierungen oftmals durch Fernzugriff. Jedoch ist die Entwicklung eines sicheren Mechanismus zur Bereitstellung der Aktualisierungen eine anspruchsvolle Aufgabe [Map17].

Es existieren einige IoT-Projekte, die sich mit der Thematik der Sicherheit im IoT-Bereich auseinandergesetzt haben. Beispielsweise haben die Organisationen ENISA[1] [ENI17] und die NIST[2] [HPG+18] viele Aspekte der Cybersicherheit von IoT-Systemen untersucht und in einem sehr ausführlichen Bericht zusammengefasst. Die Organisation OWASP[3] hat eine Rangliste mit den zehn wichtigsten Aspekten, die bei der Entwicklung, Bereitstellung und Verwaltung von IoT-Systemen zu vermeiden sind, veröffentlicht. Die Liste ist in Abbildung 2.2 dargestellt [OWASP18]. Im Folgenden werden die einzelnen Punkte der Rangliste genauer erläutert:

- **Unsichere Passwörter**: Es werden schwache, leicht erratbare oder fest codierte Passwörter verwendet. Die Anmeldedaten können somit leicht über einen Brute-Force-Angriff herausgefunden und öffentlich zugänglich gemacht werden bzw. sind nicht veränderbar. Zudem können Hintertüren in Firmware oder Client-Software vorhanden sein, die Unbefugten Zugang zu den Systemen ermöglichen.
- **Unsichere Netzwerkdienste**: Auf dem Gerät laufen nicht benötigte oder unsichere Netzwerkdienste, die die Vertraulichkeit, Integrität, Authentizität oder Verfügbarkeit der Informationen gefährden oder eine nicht autorisierte Fernsteuerung ermöglichen.

[1] European Union Agency for Cybersecurity
[2] National Institute of Standards and Technology
[3] Open Web Application Security Project

- **Unsichere Ökosystem-Schnittstellen**: Es sind unsichere Schnittstellen im Ökosystem außerhalb des Geräts vorhanden, wodurch eine Kompromittierung des Geräts oder der zugehörigen Komponenten ermöglicht wird. Probleme sind beispielsweise fehlende Authentifizierung/Autorisierung oder fehlende Verschlüsselung.
- **Fehlen eines sicheren Update-Mechanismus**: Dem Gerät fehlen Möglichkeiten zur sicheren Aktualisierung der Firmware. Hierzu gehören beispielsweise mangelhafte Firmware-Validierungen oder fehlende Benachrichtigungen über neue Sicherheitsupdates.
- **Verwendung von unsicheren oder veralteten Komponenten**: Das System verwendet veraltete oder unsichere Softwarekomponenten oder -bibliotheken, die eine Kompromittierung des Gerätes ermöglichen. Hierzu gehört zum Beispiel die Verwendung von Software- oder Hardwarekomponenten von Drittanbietern aus einer kompromittierten Lieferkette.
- **Unzureichender Schutz der Privatsphäre**: Persönliche Daten des Nutzers, die auf dem Gerät oder im Ökosystem gespeichert sind, können unsachgemäß oder ohne Genehmigung verwendet werden.
- **Unsichere Datenübertragung und -speicherung**: Das gesamte Ökosystem verwendet keine Verschlüsselung oder Zugangskontrolle für sensible Daten.
- **Mangelndes Gerätemanagement**: Fehlende Sicherheitsunterstützung für Geräte, die in der Produktion eingesetzt werden. Hierzu gehören unter anderem fehlende Möglichkeiten für Asset-Management, Update-Management und Systemüberwachung.
- **Unsichere Standardeinstellungen**: Systeme werden mit unsicheren Standardeinstellungen ausgeliefert oder es fehlen Möglichkeiten, um das System über Konfigurationen sicherer zu machen.
- **Mangelnde physische Abhärtung**: Das Gerät besitzt keine physischen Schutzmaßnahmen, sodass potenzielle Angreifer an Informationen gelangen können, die zu einem zukünftigen Fernangriff führen könnten. Zudem können Angreifer lokal die Kontrolle über das Gerät übernehmen.

OWASP Internet of Things TOP 10 Themen	
1	Unsichere Passwörter
2	Unsichere Netzwerkdienste
3	Unsichere Ökosystem-Schnittstellen
4	Fehlende Update-Mechanismen
5	Unsichere und veraltete Komponenten
6	Unzureichender Schutz der Privatsphäre
7	Unsichere Datenübertragung und -speicherung
8	Unzureichendes Gerätemanagement
9	Unsichere Standardeinstellung
10	Mangelnde physische Sicherung der Geräte

Abbildung 2.2: OWASP Top 10 Herausforderungen bei IoT-Systemen [OWASP18]

Zur Bewertung der Sicherheit werden Sicherheitstests genutzt, bei denen Sicherheitseigenschaften wie bspw. Vertraulichkeit, Datenintegrität, Verfügbarkeit, Authentifizierung, Autorisierung und Nichtabstreitbarkeit verifiziert werden können [FBJ+16]. Die einzelnen Begriffe werden in Tabelle 2.1 genauer erläutert. Auf die Funktionsweise von Sicherheitstests wird im nächsten Abschnitt detailliert eingegangen.

Eigenschaft	**Beschreibung**
Vertraulichkeit	Daten sollen nicht an unbefugte Personen, Prozesse oder Geräte weitergegeben werden.
Datenintegrität	Daten dürfen nicht auf unbefugte Weise verändert werden. Die Datenintegrität muss bei der Speicherung, während der Verarbeitung und bei der Übertragung gewährleistet sein.
Authentifizierung	Die Identität eines Benutzers, Prozesses oder Geräts muss überprüft werden. Dies ist oft die Voraussetzung für den Zugriff auf Ressourcen in einem Informationssystem.
Autorisierung	Dem Benutzer, Programm oder Prozess müssen die passenden Rechte gewährt werden, um auf das System zuzugreifen.
Nichtabstreitbarkeit	Bietet die Möglichkeit festzustellen, ob eine bestimmte Person eine bestimmte Aktion durchgeführt hat, sodass sie diese im Nachhinein nicht abstreiten kann.

Tabelle 2.1: Erklärung der Sicherheitseigenschaften (angelehnt an [FBJ+16] und [Duk15])

2.3 Ansätze für automatisierte Sicherheitstests und ihre Bedeutung für IoT-Systeme

Um diese vielzähligen Sicherheitsrisiken aufdecken und anschließend beheben zu können, wird die Software der Geräte verschiedenen Tests unterzogen. Grundsätzlich besteht das Testen von Software aus der dynamischen Überprüfung des Programms hinsichtlich seines erwarteten Verhaltens bei einer endlichen Menge von Testfällen, einer sog. Testsuite [Fre01]. Das ausgeführte und beobachtete System wird als System Under Test (SUT) bezeichnet [AO16]. Getestet werden kann das funktionale Verhalten des Systems sowie die nichtfunktionalen Eigenschaften. Das Funktionale Testen befasst sich mit der Bewertung des funktionalen Verhaltens eines SUTs, also ob das System das erwartete Ergebnis liefert. Nichtfunktionales Testen zielt darauf ab, Qualitätsmerkmale wie Sicherheit, Zuverlässigkeit oder Leistung zu bewerten [FBJ+16].

Sicherheitstests können wiederum in zwei Ansätze unterteilt werden: Funktionale Sicherheitstests und das Testen auf Sicherheitslücken. Die funktionalen Sicherheitstests stellen sicher, dass die Sicherheitsfunktionen der Software korrekt implementiert sind und mit den Sicherheitsanforderungen der Spezifikation übereinstimmen. Beim Testen auf Sicherheitslücken geht es insbesondere darum, als Angreifer Sicherheitslücken zu finden. Sicherheitslücken beziehen sich auf Fehler im Systementwurf, in der Implementierung, im Betrieb und in der Verwaltung, die für Angriffe ausgenutzt werden können [TYY10].

Die Testmethoden können zudem in White-Box-, Black-Box- und Gray-Box-Testverfahren aufgeteilt werden. Für White-Box-Tests ist es notwendig, dass der Tester Zugriff auf den Quellcode hat, um diesen einer detaillierten Untersuchung der internen Logik und Struktur des Codes zu unterziehen. Bei Black-Box-Tests handelt es sich um eine Testtechnik, bei der keinerlei Kenntnisse über die interne Funktionsweise der Anwendung vorliegen müssen. Dabei werden nur die grundlegenden Aspekte des Systems, wie bspw. das Eingabe- und Ausgabeverhalten untersucht. Gray-Box-Tests sind eine Kombination von White-Box- und Black-Box-Testverfahren. Diese Art von Tests erfordern begrenztes Wissen über die interne Funktionsweise einer Anwendung. Der Tester verfügt über Wissen zu grundlegenden Aspekten des Systems, wie bspw. die verwendeten Datenstrukturen und Algorithmen, um mit diesen Informationen Testfälle zu generieren [KK+12].

Im Rahmen dieser Arbeit werden automatisierte Methoden zum Testen der Sicherheit von Systemen untersucht. Durch die immense Menge an IoT-Geräten und -Systemen ist die Skalierbarkeit des Testens ein wichtiger Aspekt. Daher werden nur Testmethoden betrachtet, die automatisiert oder halbautomatisiert durchgeführt werden können. Diese Methoden werden auf ihr Einsatzpotenzial im IoT-Kontext geprüft. Da IoT-Systeme stark heterogen sind und aus einer riesigen Anzahl verschiedener Technologien bestehen, werden keine konkreten Anwendungsfälle erläutert. Es wird lediglich ein Überblick über grundsätzliche Ansätze zur Vorgehensweise von automatisierten Sicherheitstests für IoT-Systeme gegeben. Zwei vielversprechende Ansätze sind Penetrationstests und Modellbasiertes Testen. Diese werden in den folgenden Abschnitten genauer erklärt.

2.3.1 Penetrationstests

Penetrationstests werden verwendet, um Sicherheitsbedrohungen in einem System zu erkennen. Diese Tests können auch als simulierter Cyberangriff betrachtet werden, um zu prüfen, ob das System sicher und frei von möglichen Schwachstellen ist. Als Cyberangriff wird ein Versuch in böswilliger Absicht bezeichnet, der unternommen wird, um in eine Anwendung oder ein System einzudringen [JKTG20].

Penetrationstests können entweder manuell durch einen Sicherheitsexperten durchgeführt werden oder automatisiert durch ein Programm. In vielen Fällen sind nur begrenzte Informationen über das zu testende System vorhanden. Hierfür können Black-Box-Tests durchgeführt werden, die nur mit den öffentlichen Schnittstellen des Systems interagieren. Da Schnittstellen oftmals erst relativ am Ende des Systementwicklungsprozesses implementiert werden, werden Penetrationstests meistens erst dann durchgeführt, wenn das System vollständig bzw. nahezu vollständig implementiert und mit ausreichend Daten gefüllt ist. Dadurch können die Tests alle implementierten Workflows abdecken [FBJ+16].

Automatisierte Penetrationstests im IoT-Bereich können in drei Typen unterteilt werden: Schnittstellentests, Transporttests und Systemtests [CZLS18].

Schnittstellentests zielen auf die Schnittstellen des Systems ab, die mit externen Benutzern oder Geräten interagieren. So kann es bspw. zu schwerwiegenden Problemen kommen, falls die Eingabe an die Schnittstelle ungültig ist und das System damit nicht umgehen kann. Im Gegensatz zu herkömmlichen Webschnittstellen können IoT-Schnittstellen auch mit code-orientierten Vorgängen wie der Steuerung von Systemprogrammen verknüpft sein. Code-orientierte Angriffe wie Code-Injection können daher schwerwiegende Probleme hervorrufen. Die Schnittstellentests begegnen den Herausforderungen „Unsichere Ökosystem-Schnittstellen“ und „Unsichere Netzwerkdienste“ aus den OWASP-Richtlinien.

Transporttests werden zur Prüfung der Netzwerk-Infrastruktur und der zugehörigen kryptografischen Verfahren und Kommunikationsprotokolle durchgeführt, die zum Schutz von Nachrichten verwendet werden. Diese Tests konzentrieren sich auf Designfehler und Probleme wie beispielsweise fehlende Authentifizierung oder das Verwenden eines schwachen kryptografischen Verfahrens. Mithilfe von Transporttests können die Aspekte „Unsichere Ökosystem-Schnittstellen“, „Unsichere Datenübertragung und -speicherung“ und „Unzureichender Schutz der Privatsphäre“ vermieden werden.

Bei Systemtests werden Firmware, Betriebssysteme und Systemdienste auf Implementierungsfehler, unsichere Systemeinstellungen und andere bekannte Schwachstellen untersucht. Da die zu untersuchenden Systeme oftmals als Black-Box-Systeme zur Verfügung stehen, verwenden Tester häufig automatisierte Methoden zur Generierung von Testfällen, wie bspw. Fuzzing. Aus den OWASP-Richtlinien werden durch Systemtests die Kategorien „Unsichere Standardeinstellungen“ und „Verwendung von unsicheren oder veralteten Komponenten“ abgedeckt [CZLS18].

Fuzzing ist eine effektive Penetrationstest-Methode zur Aufdeckung von Schwachstellen. Hierfür werden speziell erstellte Eingaben an das SUT gesendet, mit dem Ziel, unerwartetes Verhalten auszulösen. Dadurch können Fehler, wie bspw. Deadlocks, Endlosschleifen, falsche Null-Behandlungen oder undefinierte Verhaltensweisen gefunden werden. Fuzzing hat den Vorteil, dass es in großem Umfang und unbeaufsichtigt durchgeführt werden kann, da der Fuzzing-Prozess in der Regel automatisiert ist. Die Fuzzer können in die Kategorien Black-Box-, White-Box- oder Gray-Box-Fuzzer aufgeteilt werden [EFI21]. Ein Black-Box-Fuzzer hat keinen Zugriff auf den Quellcode und die interne Logik des SUTs, es werden bestehende korrekte Eingaben nach einem Zufallsprinzip verändert, sodass fehlerhafte Eingaben entstehen [LPJ+18]. Diese Eingabedaten werden kontinuierlich an das SUT gesendet und die daraus resultierenden Ausgaben analysiert [CCM+18]. White-Box-Fuzzer nutzen die Informationen über die interne Logik und den Quellcode des SUTs, um Testfälle zu generieren. Dadurch ist es möglich, dass die Tests den gesamten Code abdecken und somit alle Ausführungspfade des Systems geprüft werden. Gray-Box-Fuzzer verfügen über limitierte Informationen über die interne Logik des SUTs und nutzen Laufzeitinformationen des Zielprogramms, wie bspw. die Codeabdeckung, um zu entscheiden, welche Pfade bereits getestet wurden. Diese Information wird daraufhin verwendet, um die Generierung von weiteren Testfällen einzuleiten [LPJ+18].

Fuzzing steht beim Testen von IoT-Geräten durch deren begrenzte Rechenleistung oftmals vor Herausforderungen: Viele dieser Systeme verfügen nicht über ein Betriebssystem, sodass beim Auftreten eines Fehlers kein Signal ausgelöst wird, was die Folge hat, dass der Fehler nicht erkannt wird. Das Hauptproblem beim Fuzzing eines IoT-Systems ist somit die Überwachung des SUTs. Zudem ist die Wartezeit zwischen dem Senden einer Nachricht und dem Empfangen der Antwort in eingebetteten Systemen, zu denen IoT-Geräte gehören, im Allgemeinen länger als in Standardsystemen. Auch wird das Fuzzing im IoT-Bereich oftmals über eine drahtlose Verbindung durchgeführt, was Verbindungsabbrüche zur Folge haben kann. Diese Aspekte müssen beim Fuzzen von IoT-Systemen berücksichtigt werden, damit bestmögliche Ergebnisse erzielt werden können [EFI21].

Die Durchführung von Penetrationstests kann schwierig sein, da diese oft nicht direkt zu beobachtbaren Sicherheitslücken führen und weil das Testen oftmals besondere Fachkenntnisse erfordert. Die Tester müssen sich, basierend auf ihrem Wissen und ihren Erfahrungen, ein Modell im Kopf bilden, das die Sicherheitseigenschaften, Sicherheitsmechanismen und die möglichen Angriffspunkte des Systems und dessen Umgebung umfasst. All das muss beim Testen berücksichtigt werden, um die konkreten Testfälle zu spezifizieren. Wenn die Testfälle erstellt wurden, müssen diese am SUT ausgeführt werden, was oftmals eine komplexe Implementierung erfordert [FZB+16]. Im nächsten Abschnitt wird eine Methodik erläutert, die einen Teil dieser Komplexität abstrahiert und automatisiert.

2.3.2 Modellbasiertes Testen

Sicherheitstests können vereinfacht werden, wenn der Tester ein Modell definieren kann, welches maschinell verarbeitbar ist und sämtliche Sicherheitseigenschaften explizit definiert. Dieses Modell kann dann verwendet werden, um automatisiert Testfälle zu generieren und diese Testfälle auch automatisiert auf dem SUT auszuführen. Die Variante des Testens, die sich auf derartige explizite Modelle stützt, wird Modellbasiertes Testen (MBT) genannt [FZB+16].

Werden beim Modellbasierten Testen ausschließlich Sicherheitsanforderungen getestet, handelt es sich um modellbasiertes Sicherheitstesten [FZB+16]. Dies befasst sich insbesondere mit der systematischen und effizienten Spezifikation und Dokumentation von Sicherheitstestzielen, Sicherheitstestfällen und Testsuiten sowie deren automatischer und halbautomatischer Generierung [SGS12].

MBT beschreibt mithilfe von Verhaltensmodellen das beabsichtigte Verhalten eines Systems und seiner Umgebung. Diese Verhaltensmodelle bestehen aus Paaren von Eingaben und Ausgaben, die dann mit den tatsächlichen Eingaben und Ausgaben des SUTs verglichen werden. Es gibt mehrere Notationen, die für die Modellierung des Systemverhaltens verwendet werden können, wie bspw. UML-Zustandsdiagramme oder Petrinetze [UPL12]. Wichtig ist, dass die Modell-Notation maschinenlesbar und wohldefiniert ist und die Tests automatisiert durch Algorithmen erzeugt werden können. Die erzeugten Tests müssen so präzise aufgebaut sein, dass sie automatisch ausgeführt werden können [ULB+16].

Zudem muss das Verhaltensmodell abstrakter sein als das SUT, da sonst der Aufwand für die Validierung des Modells genau dem Aufwand für die Validierung des SUTs entspricht. Die Abstraktion kann bspw. durch absichtliches Weglassen von Details erfolgen. In vielen Fällen werden bestimmte Details und Informationen als unkritisch oder zu einfach angesehen, sodass es nicht notwendig ist, diese in das Modell einzubeziehen. Eine weitere Methode zur Abstraktion ist die Kapselung von Daten. Beispielsweise können Passwörter gekapselt werden, indem lediglich angegeben wird, ob das Passwort korrekt oder inkorrekt ist. Im Zusammenhang mit Protokolltests werden oft Kommunikationsabstraktionen verwendet. Mehrere Signale können zu einem einzelnen Signal einer höheren Ebene zusammengefasst werden. So ist es bspw. möglich, ein Handshaking zu Beginn der Kommunikation durch ein einziges abstraktes Signal darzustellen [PPW+05] [UPL12].

MBT kann gemäß [Ahm18] [UPL12] generell in fünf Schritte aufgeteilt werden. Diese werden im Folgenden dargelegt:

- **Schritt 1**: Ein Modell des SUTs wird auf der Grundlage von Anforderungen oder Spezifikationen erstellt. Dieses Modell enthält das beabsichtigte Verhalten und kann sich auf unterschiedlichen Abstraktionsebenen befinden.
- **Schritt 2**: Es werden Kriterien für die Testauswahl festgelegt und daraus abstrakte Testfälle erzeugt. Diese abstrakten Testfälle werden aus den Modellen generiert und enthalten keine Implementierungsdetails des SUTs. Idealerweise führen die

Testauswahlkriterien zu Testfällen, die mit vertretbarem Aufwand schwerwiegende und wahrscheinliche Fehler aufdecken und bei der Identifizierung des zugrunde liegenden Fehlers hilfreich sind.

- **Schritt 3**: Die Kriterien werden in Testfallspezifikationen umgewandelt. Diese formalisieren die Testauswahlkriterien und ermöglichen, dass ausgehend von einem Modell und einer Testfallspezifikation eine Testsuite abgeleitet werden kann.
- **Schritt 4**: Mithilfe des Modells und der Testfallspezifikation wird eine Testsuite erstellt. Diese Testsuite enthält Testfälle, die der Spezifikation entsprechen. Testfallgeneratoren wählen dann in der Regel zufällig Testfälle aus.
- **Schritt 5**: Die ausgewählten Testfälle werden ausgeführt. Um die Testausführung zu automatisieren, werden Systemadapter verwendet, die Kanäle bereitstellen, um das SUT mit der Testausführungsumgebung zu verbinden. Für jeden Testfall werden die Eingaben an das SUT gesendet und die Ausgaben gesammelt, um ein Testurteil zu vergeben. Für jeden Testfall wird ein Testurteil vergeben, das angibt, ob ein Test bestanden, fehlgeschlagen oder nicht schlüssig ist.

Im IoT-Kontext sind Modelle besonders relevant, da es so möglich wird, die fragmentierte Technologielandschaft eines bestimmten SUTs unabhängig von der zugrundeliegenden Technologie oder dem verwendeten Protokoll abzubilden [MHSB20]. Zwei Kernkonzepte des Modellbasierten Testens sind die Abstraktion und die Automatisierung. Zusammen bilden sie eine geeignete Lösung für die Herausforderungen im IoT-Bereich. Die Abstraktion erleichtert die Modellierung und damit die Spezifikation und den Entwurf komplexer IoT-Systeme [Ahm18].

Bei IoT-Netzwerken treten häufig Probleme mit der Netzwerkinfrastruktur auf, wie bspw. langsame Internetverbindungen oder überlastete Wi-Fi-Kanäle. Derartige Probleme mit der Infrastruktur können einen erheblichen Einfluss auf die Leistung der einzelnen IoT-Geräte innerhalb des Netzwerks haben, da diese auf schnelle Kommunikation angewiesen sind. Diese Rahmenbedingungen können in einem Modell formalisiert und somit in die Sicherheitstests des IoT-Netzwerks einbezogen werden. Die IoT-Geräte und -Anwendungen können so unter speziellen Bedingungen getestet werden, um sicherzustellen, dass sie korrekt und ohne Datenverlust reagieren [Ahm18].

MBT ist ein vielversprechender Ansatz, um Skalierbarkeit und Interoperabilität bei Sicherheitstests in IoT-Umgebungen zu gewährleisten. Diese beiden Aspekte stellen ein großes Hindernis für die flächendeckende Einführung und den Erfolg von IoT dar. Modelle können das SUT, dessen Umgebung oder den Test selbst repräsentieren, wodurch die Schritte Testanalyse, Planung, Kontrolle, Implementierung, Ausführung und Berichterstattung im Entwicklungsprozess unterstützt werden. MBT ist ein guter Ansatz, um so viele testbezogene Aktivitäten wie möglich zu formalisieren und zu automatisieren, und hilft somit, die Effizienz und Effektivität des Testens zu erhöhen. Im Laufe der Zeit wurden viele MBT-Werkzeuge entwickelt, um den Einsatz von MBT-Ansätzen zu unterstützen [Ahm18].

Mit deren Hilfe können Testfälle, Testdaten und auch Testskripte generiert werden, welche dann für verschiedene Testarten wie zum Beispiel Funktions- oder Leistungstests verwendet werden können [UL10].

Fazit

IoT-Sicherheit ist aufgrund der unterschiedlichen Technologien und der Vielzahl an Geräten ein komplexes Thema. Automatisierte Sicherheitstests wie Penetrationstests und Modellbasiertes Testen können eingesetzt werden, um diese Herausforderung anzugehen. Für den weiteren Verlauf der Arbeit werden diese beiden Ansätze verwendet, um die Sicherheit von IoT-Systemen zu testen.

3 Das MQTT-Protokoll - Analyse der Sicherheitsanforderungen

Da MQTT das meistverwendete der IoT-spezifischen Protokolle für die Kommunikation ist, werden in diesem Kapitel die Grundlagen von MQTT detailliert dargelegt. Zudem wird eine Forschungsfrage definiert, anhand derer die Sicherheitsrisiken von MQTT herausgearbeitet werden.

3.1 Grundlagen von MQTT

Message Queue Telemetry Transport (MQTT) ist ein standardisiertes, offenes, zuverlässiges und einfach zu implementierendes Netzwerkprotokoll für die Übermittlung von Nachrichten zwischen Geräten, welches standardmäßig das Transportschichtprotokoll TCP/IP zur Kommunikation verwendet. Dank dieser Eigenschaften ist MQTT ideal geeignet für den Einsatz in eingeschränkten Umgebungen mit knapp bemessenen Ressourcen, wie es oftmals in IoT-Systemen der Fall ist. Das Protokoll basiert auf einem Client-Server Publish/Subscribe-Kommunikationsmodell, bei dem die Clients (Geräte) mit einem Broker (Server) oder mit anderen Clients über den Broker miteinander kommunizieren. MQTT-Clients können dabei die Rolle des Publishers oder des Subscribers einnehmen. Der Publisher verbindet sich mit einem Broker und kann daraufhin Nachrichten an ein beliebiges MQTT-Topic, also ein spezifisches Thema, veröffentlichen. Sämtliche Subscriber, die das Topic abonniert haben, erhalten dann die Nachricht [Sta19] [Sta14]. Nachrichten sind in diesem Kontext Informationen, die zwischen den Geräten ausgetauscht werden sollen.

Das Publish/Subscribe-Modell ermöglicht eine lose Kopplung und hohe Skalierbarkeit für IoT-Systeme: Publisher und Subscriber müssen nicht wissen, welche anderen Geräte existieren und müssen auch nicht zur gleichen Zeit online sein. Zudem unterstützt es ein Many-to-Many-Kommunikationsmodell, d.h. ein Publisher kann Nachrichten an viele Subscriber veröffentlichen und ein Subscriber kann von vielen Publishern Nachrichten erhalten [ZLG+20].

MQTT-Topics sind in einer hierarchischen Struktur aufgebaut, die, ähnlich wie ein Pfad in einem Dateisystem, durch einen Schrägstrich getrennt sind. Eine einfache Möglichkeit, mehrere zusammengehörige Topics gleichzeitig zu abonnieren, sind Wildcards. Hauptsächlich werden die folgenden Wildcards in MQTT verwendet [CPVV18]:

- Wildcard für eine Ebene (+): Anstelle des „+“ kann jedes Unterthema stehen, das sich auf der gleichen hierarchischen Ebene befindet. Beispielsweise wird die folgende hierarchische Struktur betrachtet: Stockwerk/Raum/Sensor. Durch das Abonnieren des Themas „EG/+/Temperatur“ wird die Temperatur jedes Raumes im EG abonniert, da sich das „+“ auf der Ebene des Raumes befindet.
- Wildcard für alle Ebenen (#): In diesem Fall werden sämtliche Unterthemen auf der „#“-Ebene und darunter ausgewählt. Zum Beispiel kann das MQTT-Topic „EG/#“ abonniert werden, was dazu führt, dass alle Räume und sämtliche Sensoren im EG abonniert werden.

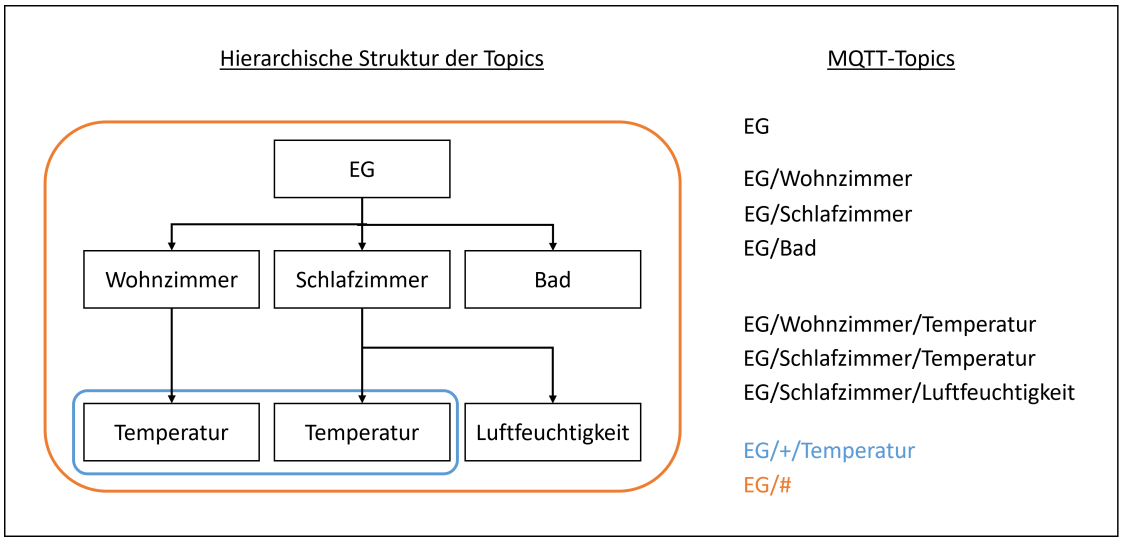


Abbildung 3.1: Topic Struktur

Eine beispielhafte hierarchische Struktur von MQTT-Topics ist in Abbildung 3.1 dargestellt. Auf der rechten Seite befinden sich die Topics, wie sie in MQTT veröffentlicht bzw. abonniert werden können. Die farblich umrahmten Kästchen stellen die Topics dar, die durch die beiden beispielhaften Wildcard-Suchen abonniert werden.

3.2 Aufbau von MQTT-Kontrollpaketen

Im Jahr 2014 wurde die MQTT-Version 3.1.1 zum OASIS Standard und 2016 zum ISO/IEC-Standard 20922:2016 ernannt. Die MQTT-Version 5.0 wurde im Jahr 2019 veröffentlicht und wurde ebenfalls zu einem OASIS-Standard. Die beiden Versionen sind nicht direkt

miteinander kompatibel, da Version 5.0 eine andere Paketstruktur und mehr Funktionen besitzt. Anbieter wie HiveMQ bieten eine Kompatibilitätsschicht an, sodass beide Versionen parallel verwendet werden können [MVH+21]. Die Kernfunktionen von Version 3.1.1 sind auch in Version 5 vorhanden und wurden durch weitere Funktionen ergänzt. In Abbildung 3.2 ist ein MQTT-Kontrollpaket in der aktuellen Version 5.0 zu sehen, die im Folgenden genauer beschrieben wird [GFT21][Sta19][MVH+21].

	Bits	7	6	5	4	3	2	1	0
Feste Kopfzeile	1 Byte	Pakettyp				DUP	QoS-Level		RETAIN
	1-4 Bytes	Restliche Länge des Pakets							
Variable Kopfzeile (Optional)	0 oder 2 Bytes	Paket-Identifikator							
	0 oder 1 Byte	Reason Code							
	1-4 Bytes	Länge der Eigenschaften							
	≥ 0 Bytes	Eigenschaften							
Payload (Optional)	≥ 0 Bytes	Payload des Pakets							

Abbildung 3.2: MQTT v5.0 (angelehnt an [Sta19] und [MVH+21])

Ein MQTT-Kontrollpaket ist ein Datenpaket, das Informationen enthält und über eine Netzwerkverbindung zwischen dem Broker und den Clients ausgetauscht wird. Die MQTT-Spezifikation definiert 15 verschiedene Arten von Paketen, von denen bspw. das PUBLISH-Paket zur Übermittlung von Anwendungsnachrichten verwendet wird. Ein MQTT-Kontrollpaket enthält zwingend eine feste Kopfzeile, sowie optional eine Kopfzeile mit variabler Länge und einen Payload. Das erste Byte der festen Kopfzeile besteht immer aus dem Typ des MQTT-Kontrollpakets und Flags, die für jeden Pakettyp spezifisch sind. Bei den Flags gibt es das DUP-Flag, das QoS-Level und das RETAIN-Flag. Zusätzlich hat die feste Kopfzeile auch immer ein Kontrollfeld, das die restliche Länge des Pakets angibt.

In Tabelle 3.1 sind sämtliche MQTT-Pakettypen, die dazugehörigen Rückgabewerte, die Richtung des Versandes, sowie eine kurze Beschreibung jedes Pakets aufgelistet. Die meisten Pakete funktionieren dabei in Paaren, d.h. auf ein Paket wird jeweils ein Bestätigungspaket zurückgeschickt. Wenn sich bspw. ein Client mit dem Broker verbinden will, wird eine CONNECT-Nachricht vom Client zum Server geschickt, worauf der Broker mit einer CONNACK-Nachricht antwortet, in der die Verbindung bestätigt oder abgelehnt wird.

Das DUP-Flag beschreibt, ob die Nachricht ein Duplikat ist und wiederholt versendet wurde, falls das Paket zuvor nicht bestätigt wurde. Das QoS-Level stellt den Grad der Zuverlässigkeit für die Zustellung der Nachricht dar und wird in Tabelle 3.2 genauer

Name	Wert	Richtung	Beschreibung
Reserved	0	Verboten	Reserviert
CONNECT	1	Client →Server	Anfrage zur Verbindung zum Server
CONNACK	2	Server→Client	Bestätigung der Verbindung
PUBLISH	3	Client ↔ Server	Veröffentlichen der Nachricht
PUBACK	4	Client ↔ Server	Bestätigung der Veröffentlichung (QoS 1)
PUBREC	5	Client ↔ Server	Veröffentlichung empfangen (Teil 1 für QoS 2)
PUBREL	6	Server ↔ Client	Veröffentlichung freigeben (Teil 2 für QoS 2)
PUBCOMP	7	Client ↔ Server	Veröffentlichung fertig (Teil 3 für QoS 2)
SUBSCRIBE	8	Client→Server	Abonnement-Anfrage vom Client
SUBACK	9	Server→Client	Bestätigung des Abonnements
UNSUBSCRIBE	10	Client→Server	Anfrage zur Abbestellung des Abonnements
UNSUBACK	11	Server→Client	Bestätigung der Abbestellung des Abonnements
PINGREQ	12	Client→Server	Ping-Anfrage
PINGRESP	13	Server→Client	Ping-Antwort
DISCONNECT	14	Client ↔ Server	Trennen der Verbindung mit dem Client
AUTH	15	Client ↔ Server	Austausch der Authentifizierung (neu in v5.0)

Tabelle 3.1: MQTT-Pakettypen (angelehnt an [Sta19])

QoS-Wert	Bit 2	Bit 1	Beschreibung
0	0	0	Nachricht wird höchstens einmal zugestellt
1	0	1	Nachricht wird mindestens einmal zugestellt
2	1	0	Nachricht wird genau einmal zugestellt
3	1	1	Verboten & Reserviert

Tabelle 3.2: MQTT QoS-Level (angelehnt an [Sta19])

beschrieben. Das DUP-Flag ist nur dann sinnvoll, wenn der QoS-Level über 0 ist, da bei einem QoS-Level von 0 die Nachricht nur ein einziges Mal versendet wird, auch wenn sie nicht ankommt. Das RETAIN-Flag sagt aus, ob die Nachricht vom Broker aufbewahrt werden soll. Die Länge des Pakets ist ein Ein bis Vier Bytes langes Kontrollfeld, das die Anzahl der im aktuellen Paket verbleibenden Bytes darstellt, einschließlich der Bytes in der variablen Kopfzeile und im Payload.

Die variable Kopfzeile wird nur von bestimmten Nachrichtentypen verwendet und enthält einen Zwei-Byte-langen Paket-Identifikator zur Identifikation des Kontrollpakets. Die Pakete, die dieses Feld verwenden, sind PUBLISH, PUBACK, PUBREC, PUBREL, PUBCOMP, SUBSCRIBE, SUBACK, UNSUBSCRIBE und UNSUBACK. Bei PUBLISH wird nur ein Paket-Identifikator verwendet, falls ein QoS-Wert größer als 0 gesetzt ist. Dieser Identifikator darf nicht 0 sein und muss eindeutig einem Pakettyp zuordenbar sein. In den meisten der MQTT-Pakete, die für die Bestätigung anderer Pakete zuständig sind, ist ein Reason Code in der variablen Kopfzeile enthalten. Lediglich in SUBACK und

UNSUBACK wird der Reason Code in dem Payload mitgesendet. Ein Reason Code ist ein Ein-Byte-Wert, der das Ergebnis eines Vorgangs angibt. Reason Codes kleiner als 0x80 zeigen den erfolgreichen Abschluss einer Operation an, Reason Code-Werte von 0x80 oder höher indizieren einen Fehler.

Seit MQTT Version 5.0 können Pakete in der variablen Kopfzeile ein Set von Eigenschaften enthalten. Dieses besteht aus einer Eigenschaftslänge, gefolgt von den Eigenschaften. Eine Eigenschaft besteht aus einem Bezeichner, der die Verwendung und den Datentyp der Eigenschaft definiert, gefolgt von einem Wert. Falls der Bezeichner und der Datentyp des Werts ungültig sind, gilt das Paket als fehlerhaft. Wenn es keine Eigenschaften gibt, muss dies durch die Angabe einer Eigenschaftslänge von 0 angegeben werden.

Das Payload ist nur für bestimmte Pakete erforderlich und enthält Nutzdaten des Pakets. CONNECT-, SUBSCRIBE-, SUBACK-, UNSUBSCRIBE-, UNSUBACK-Pakete enthalten zwingend und PUBLISH-Pakete optional einen Payload. Die Nutzdaten eines PUBLISH-Pakets sind beispielsweise die Nachrichten der Anwendung, die übermittelt werden sollen.

3.3 Sicherheitsprobleme bei MQTT

Die Umfrage der Eclipse IoT Working Group hat ergeben, dass das MQTT-Protokoll das am häufigsten eingesetzte IoT-spezifische Protokoll für die Kommunikation ist und die Kommunikationssicherheit eine der größten Gefahren im IoT-Bereich darstellt [EDS20]. Für den weiteren Verlauf dieser Arbeit wurde daher folgende Hypothese aufgestellt:

H - Bei aktuellen MQTT-Implementierungen bestehen immer noch Sicherheitsprobleme, die durch automatisierte Sicherheitstests entdeckt werden können.

Zur Beantwortung dieser Hypothese wurden zwei Forschungsfragen formuliert, die im Rahmen der vorliegenden Masterarbeit beantwortet werden. Außerdem werden mit Hilfe eines Testansatzes und eines Prototypen MQTT-Implementierungen auf bestehende Sicherheitslücken getestet und die Hypothese somit anhand von Literatur sowie praktischen Sicherheitstests geprüft. Die erste Forschungsfrage befasst sich mit den in der Literatur zu findenden Sicherheitsproblemen des MQTT-Protokolls und lautet:

RQ1 - Welche Sicherheitsprobleme existieren im Zusammenhang mit dem MQTT-Protokoll?

Zur Beantwortung dieser Forschungsfrage wurde eine Literaturrecherche durchgeführt, um den durch die oben genannten Problematiken entstehenden Angriffspunkt zu untersuchen. Im folgenden Abschnitt werden die bestehenden Probleme hinsichtlich der Sicherheit des MQTT-Protokolls sowie potenzielle Angriffspunkte, die mithilfe der Literaturrecherche identifiziert werden konnten, erläutert.

MQTT wurde ursprünglich von der Öl- und Gasindustrie für private Netzwerke entwickelt, daher hatte die Sicherheit keine hohe Priorität. Bereits einfache, fehlerhafte Pakete können zu Abstürzen führen, was bspw. Denial of Service (DoS)-Angriffe und das Ausführen von bösartigem Code ermöglicht [SFR20b]. Die MQTT-Spezifikation enthält außerdem keine verpflichtenden Anforderungen für Authentifizierung, Autorisierung, Datenintegrität, Vertraulichkeit und Datenschutz. Dies liegt zum Teil daran, dass das MQTT-Protokoll so leichtgewichtig wie möglich gehalten werden soll und Sicherheitsfunktionen oftmals einen Overhead mit sich bringen, der ressourcenbeschränkte Geräte überlasten kann [PVPG+17]. Ressourcenbeschränkte IoT-Geräte können basierend auf den ihnen zur Verfügung stehenden Ressourcen in die Klassen 0, 1 und 2 aufgeteilt werden, wie in Tabelle 3.3 zu sehen ist. Aufgrund der sehr begrenzten Rechenleistung sind für Geräte der Klasse 0 die meisten Sicherheitsansätze schwer umsetzbar, insbesondere rechenintensivere Funktionen, wie bspw. die Verschlüsselung über das Transport Layer Security (TLS)-Verschlüsselungsprotokoll [KA16]. Um auch diese Geräte zu schützen, müssen ressourcenschonendere Möglichkeiten entwickelt werden, um die Sicherheit der Geräte zu gewährleisten.

Klasse	**RAM-Größe**	**ROM-Größe**
Klasse 0	$<$ 10KB	$<$ 100KB
Klasse 1	$\approx$ 10KB	$\approx$ 100KB
Klasse 2	$\approx$ 50KB	$\approx$ 250KB

Tabelle 3.3: Verfügbare Ressourcen bei ressourcenbeschränkten IoT-Geräten [KA16]

Eine der Hauptursachen für Sicherheits- und Datenschutzprobleme im IoT-Bereich liegt in der unsicheren Standardkonfiguration von MQTT-Broker-Implementierungen. Ein MQTT-Server, bei dem die voreingestellten Werte beibehalten und keine Konfigurationsänderungen vorgenommen werden, enthält keine Authentifizierung und jeder Client kann sich mit dem Broker verbinden. Ein Angreifer kann bei einem unsicher konfigurierten MQTT-Server Zugriff auf sämtliche Nachrichten erhalten, die durch das Netzwerk fließen [AK19]. Optional kann ein Nutzername und Passwort für die Authentifizierung verwendet werden, die anschließend vom Client in einem CONNECT-Paket mitgeschickt werden. Jedoch stellt dies ebenfalls keinen hinreichenden Schutz dar, da diese Information in Klartext versendet wird. Ein Angreifer kann ein CONNECT-Paket abfangen und hat dann Zugriff auf die Nutzername/Passwort-Kombination [PVPG+17]. Zusätzlich zur fehlenden Authentifizierung bietet das MQTT-Protokoll auch keine Möglichkeit an, nur bestimmten Clients den Zugriff zu bestimmten Topics zu gewähren. MQTT-Clients, die sich authentifiziert haben und mit dem Broker verbunden sind, können Nachrichten an beliebige Topics veröffentlichen und auch alle Topics abonnieren. Das MQTT-Protokoll selbst beinhaltet keine Autorisierungsmechanismen, sondern dies muss vom Broker implementiert werden [HVL18].

Die unsichere Standardkonfiguration führt zudem zu Problemen mit der Datenintegrität, welche die Aufrechterhaltung von Konsistenz, Genauigkeit und Vertrauenswürdigkeit der Daten umfasst [TSTB19]. Standardmäßig wird das Transmission Control Protocol (TCP)-Netzwerkprotokoll ohne TLS verwendet, d.h. die Nachrichten werden unverschlüsselt

versendet. Das MQTT-Protokoll enthält keine direkte Funktionalität zur Verschlüsselung von Daten. Dies wird zumeist über TLS auf der Netzwerkschicht bereitgestellt. TLS ermöglicht eine Ende-zu-Ende gesicherte Kommunikation zwischen den beiden Geräten, die miteinander kommunizieren. Hierfür wird Kryptografie verwendet, um die Authentizität der Daten und des Endpunkts zu garantieren [LHH+15]. Falls kein TLS verwendet wird, kann ein Angreifer die Pakete, die zwischen Client und Broker ausgetauscht werden, abfangen und modifizieren. Beispielsweise kann der Topic-Name oder die Nachricht selbst verändert werden. Zahlreiche IoT-Geräte treffen Entscheidungen basierend auf Nachrichten aus der Umgebung, was eine hohe Integrität der Nachrichten voraussetzt. Modifizierte Nachrichten können die Folge haben, dass IoT-Geräte falsche, nicht gewünschte Entscheidungen treffen. Dies kann immense Auswirkungen haben: Ein IoT-Gerät kann bspw. über MQTT einen Link für ein Firmware-Update erhalten. Falls ein Angreifer Firmware-Update-Links identifizieren kann, kann dieser den Link mit einem maliziösen Link ersetzen, wodurch er Zugriff auf das IoT-Gerät erlangt. Dadurch kann der Angreifer die Kontrolle über das Gerät übernehmen und es als Teil eines Botnets nutzen, um Angriffe durchzuführen [FBVI17] [ARH17].

Da die Nachrichten standardmäßig unverschlüsselt versendet werden, gibt es auch Probleme mit der Vertraulichkeit der Daten. Die Vertraulichkeit bezieht sich auf die Fähigkeit des Systems, die Privatsphäre des Nutzers zu gewährleisten, indem eine sichere Verbindung nur für den zugelassenen Benutzer hergestellt wird. Andere Benutzer oder benachbarte Clients sollen keinen Zugriff auf die Daten erhalten, was im MQTT-Kontext oftmals ein Problem darstellt, da die Nachricht in Klartext gesendet wird und die Pakete abgefangen werden können [SJK17]. Das Abfangen der Pakete kann bspw. mit Wireshark[1] durchgeführt werden. Wireshark ist ein Programm zur Analyse von Netzwerkprotokollen. Abbildung 3.3 enthält ein abgefangenes PUBLISH-Paket. Das Payload des PUBLISH-Pakets, d.h. die Nachricht, die an alle Abonnenten geschickt wird, ist der blau markierte Teil und kann in Wireshark einfach gelesen werden.

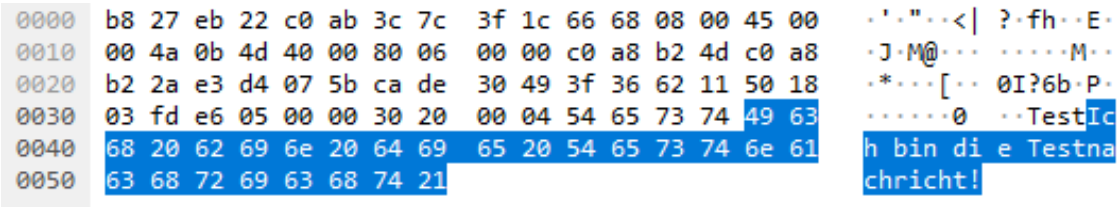


Abbildung 3.3: Abgefangenes MQTT/PUBLISH-Paket in Wireshark

Die Verwendung des TLS-Protokolls wird vom MQTT-Standard dringend empfohlen, um eine sichere Kommunikation zu gewährleisten, allerdings existieren dennoch weiterhin Sicherheitsprobleme im Zusammenhang mit dem Protokoll. Es ist bekannt, dass gerade ältere Versionen von TLS durch die Verwendung schwacher Chiffre-Suiten MQTT-Systeme

[1] https://www.wireshark.org/

anfällig für Angriffe machen [NC20]. Sheffer et al. haben in ihrem Artikel die bekannten Angriffe, die gegen TLS ausgenutzt werden, zusammengefasst [SHS15]. Beispielsweise reicht es bei den meisten Verschlüsselungsalgorithmen aus, wenn der Angreifer Zugriff auf den privaten Schlüssel des Servers hat, um sämtliche vergangenen und zukünftigen Sitzungen zu entschlüsseln. TLS-geschützte Verbindungen können mithilfe des privaten Schlüssels in Wireshark untersucht werden. Dieser Schlüssel ist die Voraussetzung, um Sicherheitstests für MQTT-Anwendungen mit TLS-basierter Kommunikation durchzuführen.

Ein weiteres Problem im MQTT-Bereich ist das unbefugte Veröffentlichen und das unbefugte Abonnieren von Nachrichten. MQTT erlaubt es Clients, Wildcards zu verwenden, um Nachrichten von allen Themen zu empfangen, die mit der Wildcard übereinstimmen. Ein Angreifer kann „#“ abonnieren, um Zugriff auf sämtliche versendeten Nachrichten in beliebigen Topics zu erlangen. Diese Nachrichten können bspw. sensible Informationen über die Umgebung des Endnutzers enthalten. Falls keine Authentifizierung voreingestellt ist, kann der Angreifer auch Nachrichten veröffentlichen, da dieser sämtliche Topics in dem Netzwerk kennt. Dies kann großen Schaden verursachen, da IoT-Geräte oftmals über Nachrichten gesteuert werden. So könnte es bspw. passieren, dass ein IoT-Gerät von Angreifern mit einer „OFF“-Nachricht ausgeschaltet wird und nicht mehr mit anderen Geräten im Netzwerk kommunizieren kann. Wird von einem Angreifer ein „RESTORE FACTORY SETTINGS“-Befehl ausgeführt, wird das Gerät zurückgesetzt und die vorhandenen Einstellungen werden überschrieben [FBVI17].

Die Überwachung des Systems ist sehr wichtig für Produktionsumgebungen, daher haben viele MQTT-Broker sogenannte SYS-Topics implementiert, die Informationen über den MQTT-Broker vermitteln. Mithilfe dieser Topics kann der Server überwacht und auch debugged werden. Clients können nicht in die SYS-Topics publishen, sie jedoch abonnieren und lesen. Das Risiko hierbei ist, dass Clients kritische Informationen über die Broker-Software und die Softwareversion erhalten können, was für Angreifer eine wertvolle Information sein kann, jedoch für normale Clients kaum nützlich ist. Wenn der Angreifer die eingesetzte Software und deren Version kennt, kann beispielsweise bei Open Source MQTT-Brokern der Quellcode zum Stand der Version direkt angegriffen werden. Daher ist es empfehlenswert, die SYS-Topics zu deaktivieren und die Systemüberwachung direkt vom Broker durchführen zu lassen [KSR19].

Die Webseiten CVE Details[2] und National Vulnerability Database[3] listen Schwachstellen von MQTT-Software mit den zugehörigen Versionen auf. Diese können sowohl von einem Angreifer als auch für Sicherheitstests genutzt werden. Ein Beispiel für eine Schwachstelle beim Broker Mosquitto in der Version 1.5.0 bis 1.6.5 ist das Senden eines SUBSCRIBE-Pakets mit 65400 Topic-Trennzeichen, was einen Systemabsturz zur Folge hat [NVD19]. 71,15% aller bereits identifizierten Schwachstellen werden durch fehlerhafte MQTT-Kontrollpakete ausgelöst. Fehlerhafte Pakete sind bspw. jene Pakete, die nicht dem MQTT-Protokoll entsprechen und aufgrund von Bugs oder absichtlicher

[2] https://www.cvedetails.com/
[3] https://nvd.nist.gov/

Modifizierung entstanden sind. Diese können die Offenlegung von Informationen, die Ausführung von Schadcode und die Verweigerung von Diensten ermöglichen und damit die Vertraulichkeit, Integrität und Verfügbarkeit beeinträchtigen. Um eine vorhandene Schwachstelle zu beheben, muss auf die MQTT-Software ein entsprechendes Update aufgespielt werden. Das ist allerdings in IoT-Umgebungen komplex, da viele Geräte keine automatische Aktualisierungsfunktion besitzen und Nutzer oftmals nicht die Kenntnisse haben, um solche Sicherheitsupdates manuell auszuführen. Dies führt dazu, dass sehr viele Schwachstellen selten gepatched werden, wodurch eine große Angriffsfläche entsteht [AM20].

Die Suchmaschine Shodan liefert Informationen zu angreifbaren IoT-Systemen und deren IP-Adressen. Mithilfe von Suchfiltern können in Shodan Geräte und Dienste, die mit dem Internet verbunden sind, gefunden werden [SHO21]. Ein Suchfilter für ein Angriffsszenario auf MQTT-Systemen ist bspw. «port:1883 „MQTT“». Der Port 1883 ist der Standardport für MQTT-Broker ohne Verschlüsselungsprotokoll, was also ein einfaches Angriffsziel ist. Der Port kann vom MQTT-Broker beliebig ausgewählt werden, daher kann auch mit TLS verschlüsselte Kommunikation auf den Port 1883 gebunden werden. Jedoch ist die Wahrscheinlichkeit hoch, dass die Kommunikation auf dem Port 1883 unverschlüsselt ist, da dies der Standardkonfiguration entspricht. Das Ergebnis der Suchanfrage in Abbildung 3.4 zeigt, dass am 04.08.2021 mit dem obigen Suchfilter 312.632 MQTT-Broker mit dem Port 1883 gefunden wurden. Für jeden gefundenen Broker sind die IP-Adresse, die dazugehörige Organisation, das Land, der MQTT-Verbindungscode sowie die MQTT-Topics aufgelistet. Broker mit dem Verbindungscode 0 sind noch einfacher anzugreifen, da bei diesen keine Authentifizierungmechanismen konfiguriert sind. Dadurch können sich alle Publisher und Subscriber anonym, ohne Benutzername und Passwort, mit dem Broker verbinden [ARH17].

Die Shadowserver Foundation[4] führt ein Projekt zur Untersuchung aller öffentlich zugänglichen MQTT-Broker durch. Diese Systeme werden identifiziert und eine Nachricht an die Netzwerkbesitzer gesendet, sodass diese die Probleme beheben können. Am 04.08.2021 wurden 293.736 individuelle IP-Adressen gefunden, die auf die Nachricht reagiert haben, wobei 250.358 Geräte keine Authentifizierung benötigen. Das bedeutet, dass über 85% aller MQTT-Broker die Standardkonfiguration oder eine unsichere Konfiguration ohne Authentifizierung verwenden und dadurch einfache Angriffsziele sind.

DoS- und Distributed Denial of Service (DDoS)-Angriffe stellen potenzielle Angriffe mit gravierenden Auswirkungen im MQTT-Kontext dar. DoS-Angriffe werden von einzelnen Angreifern durchgeführt und haben das Ziel, Geräte so außer Kraft zu setzen, dass Benutzer oder Organisationen die Ressourcen dieser nicht mehr nutzen können. DDoS-Angriffe dagegen werden von mehreren Angreifern verursacht und sorgen dafür, dass der Zugriff auf Dienste eingeschränkt wird. DoS/DDoS-Angriffe, die durch kompromittierte IoT-Geräte auf Anwendungsebene stattfinden, werden zu einem kritischen Problem. Beispielsweise können DoS-Angriffe dazu führen, dass der Server abstürzt oder dass ein Angreifer eine anonyme

[4] https://scan.shadowserver.org/mqtt/

Abbildung 3.4: Ergebnis der Suche «port:1883 „MQTT“» auf www.shodan.io

Verbindung zum Broker aufbauen kann. Der Hauptangriffspunkt für DDoS-Attacken im MQTT-Kontext ist der Broker, das Herz der IoT-Infrastruktur [KSR19]. Ein externer Angreifer kann beispielsweise Pakete mit riesigem Payload an den Broker schicken, um dessen Ressourcen zu erschöpfen, was zum Absturz des Servers führt. Zusätzlich kann der Angreifer eine höhere QoS-Stufe für die Nachrichten verwenden, sodass der Broker die riesigen Pakete speichern muss, wodurch der Ressourcenverbrauch nochmals ansteigt [FBVI17].

Da das MQTT-Protokoll ein TCP-basiertes Protokoll ist, sind die Systeme auch anfällig für TCP-basierte Angriffe, wie bspw. SYN-Flooding [FBVI17]. Ein SYN-Flooding-Angriff nutzt den Drei-Wege-Handshake-Mechanismus von TCP und die Aufrechterhaltung von halboffenen Verbindungen aus. Wenn eine TCP-Verbindung aufgebaut werden soll, wird zuerst ein SYN-Paket vom Client an den Server geschickt, woraufhin der Server mit einem SYN/ACK-Paket antwortet, um den Erhalt der SYN-Nachricht zu bestätigen. Im dritten Schritt soll der Client nun ein ACK-Paket schicken, um den Empfang des SYN/ACK-Pakets zu bestätigen. Falls dieses Paket nicht an den Server versendet wird, bleibt die Verbindung vom Server zum Client bis zum TCP-Verbindungstimeout in einem halboffenen Zustand. Der Server kann nur eine limitierte Anzahl dieser Verbindungen aufrechterhalten. Wenn dieses Limit erreicht wird, werden sämtliche, auch legitime, TCP-Verbindungsanfragen verworfen, sodass sich kein Gerät mehr mit dem Broker verbinden kann. Das kann ein Angreifer nutzen, um das System in einen DoS-Zustand zu versetzen, in dem es auf keine Anfragen mehr reagiert [SP04] [WZS02].

Vaccari et al. stellten mit SlowITe einen neuartigen DoS-Angriff auf MQTT-Dienste vor, der darauf abzielt, das Zielsystem zum Absturz zu bringen. Anders als andere DoS-Angriffe gehört dieser zur Kategorie Slow DoS Attack (SDA), was den Angriff um einiges gefährlicher macht, da nur eine kleine Bandbreite benötigt wird. SDAs können nur auf TCP-basierten Protokollen durchgeführt werden, MQTT-Systeme sind daher attraktive Angriffsziele, da diese über ein TCP-Netzwerk laufen. Das Gefährliche an SlowITe ist, dass dieser Angriff sogar von einem einzelnen Gerät mit einer begrenzten Internetverbindung gestartet werden kann, da nur eine geringe Menge an Bytes über eine längere Zeitspanne versendet wird. Um einen SDA durchzuführen, werden oft die Warteschleifen des Zielsystems mit Angriffsnachrichten gefüllt, sodass das System keine Kapazität mehr zum Bearbeiten von Anfragen von legitimen Benutzern hat [CPCA13] [VAC20].

SlowITe versucht, sämtliche Verbindungen zum MQTT-Broker beschäftigt zu halten, wodurch ein DoS-Zustand entsteht. Das wird erreicht, indem CONNECT-Pakete zum MQTT-Broker gesendet werden, woraufhin der Server mit einem CONNACK-Paket antwortet. Wenn nun vom Client keine weitere Nachricht geschickt wird, muss der Server die Verbindung für k = 1,5 multipliziert mit dem Keep-Alive-Parameter T, welcher standardmäßig bei T = 60 Sekunden liegt, aufrecht erhalten. Demnach wartet der Server für k * T = 90 Sekunden, bevor dieser die Verbindung trennt. Diese Art von Verbindung kann nun so oft durchgeführt werden, bis sich sämtliche Verbindungen zum Broker in diesem Wartezustand befinden und keine weiteren Verbindungen von legitimen Clients erlaubt werden. Die Autoren haben zusätzlich ein gravierendes Problem im MQTT-Protokoll gefunden, mit dem der DoS-Zustand sehr stark verlängert werden kann. Das MQTT-Protokoll erlaubt es, den Keep-Alive-Parameter T in dem CONNECT-Paket mit einem Höchstwert von T = 65535 zu spezifizieren. Wenn sich alle erlaubten Verbindungen in diesem DoS-Zustand befinden, können für mehr als 27 Stunden keine weiteren Netzwerkpakete zwischen Client und Server ausgetauscht werden. SlowITe ist zudem in der Lage, nicht nur Klartext-Kommunikation, sondern auch TLS-basierte, also vermeintlich sichere Kommunikation, anzugreifen [VAC20].

Fazit

Im Zusammenhang mit dem MQTT-Protokoll besteht eine Vielzahl an Sicherheitsproblemen. Für den weiteren Verlauf dieser Arbeit wird der Fokus zum einen auf die mangelhafte Standardkonfiguration von MQTT-Brokern gelegt, da diese ein Sicherheitsrisiko darstellt, das einfach zu beheben wäre. Zum anderen wird der Umgang des Brokers mit fehlerhaften Paketen untersucht, da diese in der Vergangenheit die häufigste Ursache für Sicherheitslücken darstellten.

4 Verwandte Arbeiten

Durch das Testen von MQTT-Anwendungen können einige der in Abschnitt 3.3 beschriebenen Probleme angegangen werden. Um herauszufinden, welche Ansätze zum Testen von MQTT-Anwendungen bereits existieren, wurde eine Literaturrecherche mit folgender Forschungsfrage durchgeführt:

RQ2 - Wie ist der Stand der Forschung bezüglich automatisierter Sicherheitstests von MQTT-Systemen?

Der Fokus liegt auf automatisierten Sicherheitstests, da diese eine hohe Skalierbarkeit des Testens ermöglichen und die Komplexität der Thematik abstrahieren. Beim MQTT-Protokoll sind die zwei größten Sicherheitsprobleme die unsichere Standardkonfiguration der MQTT-Broker und die fehlerhaften Pakete, die einen DoS-Zustand des Systems auslösen können. Die Literaturrecherche und der folgende Abschnitt beschäftigen sich daher vorwiegend mit Testsystemen, die diese beiden Probleme angehen. Die Manipulation von Paketen über Fuzzing gilt als die vielversprechendste Methode zur Aufdeckung von Protokollschwachstellen im Bereich IoT [LSCL18], weshalb sich der Großteil der Literatur über MQTT-Sicherheitstests mit diesem Thema befasst. Weitere Ansätze sind das Verwenden von Angriffsmustern, die auf bekannten, in der Vergangenheit gefundenen Schwachstellen basieren sowie Modellbasiertes Testen. Außerdem ist das Testen des Systems bezüglich der Konformität zur MQTT-Spezifikation eine weitere Möglichkeit.

4.1 Fuzzing

Hernández et al. präsentierten in ihrem Forschungsartikel „MQTT Security: A Novel Fuzzing Approach“ [HVL18] ein Framework basierend auf der Fuzzing-Strategie, um die Sicherheit von Applikationen, die das MQTT-Protokoll implementieren, zu verbessern. Die Autoren nutzten eine Strategie, die Proxy Fuzzing genannt wird, bei der der Fuzzer zwischen dem Client und dem Server platziert wird, um als Vermittler zu agieren. Dies wird beispielsweise mit der ARP-Spoofing-Technik durchgeführt. Das ARP-Protokoll wird verwendet, um die MAC-Adresse eines Hosts anhand seiner IP-Adresse zu ermitteln. Mit ARP-Spoofing werden die ARP-Pakete gefälscht, sodass sich ein Angreifer als ein anderer Host im Netzwerk ausgeben kann [RN05]. Dadurch sieht der Client den Proxy nun als Server und der Server den Proxy als Client. Durch die Nutzung des Proxies liegt das Hauptaugenmerk auf den Paketen, die zwischen dem Server und dem Client geschickt werden. Da die Spezifikation der Pakete für alle Anwendungen, die das MQTT-Protokoll

implementieren, standardisiert ist, entkoppelt sich der Fuzzing-Prozess vollständig von der Verbindung zwischen Server und Client und konzentriert sich auf die ausgetauschten Pakete. Ein weiterer Grund für die Nutzung eines Proxies ist, dass einige der Felder von einer früheren Nachricht abhängig sind. Wenn das der Fall ist, kann nicht einfach eine Verbindung zum Broker hergestellt und das Paket versendet werden, da diese Pakete wegen eines falschen Feldes zurückgewiesen werden würden. Durch den Proxy-Ansatz kommen die Nachrichten von einem rechtmäßigen Client und Broker, daher bleiben diese Art von Feldern intakt und können mit dem richtigen Wert gefüllt und somit ohne Probleme gefuzzt werden.

Um die MQTT-Nachrichten effizient fuzzen zu können, ist es wichtig, ausreichend Kenntnisse über die MQTT-Spezifikation zu besitzen. Durch die Spezifikation sind die Pakete und Felder bekannt, dadurch können diese interpretiert und interessante Felder ausgewählt werden, um sie zu modifizieren. Zuerst werden die relevanten Arten von Paketen identifiziert und deren variable Kopfzeilen untersucht, um die Art der Felder und die Positionen der Felder in Bytes zu finden. Daraufhin wird die Position der Kontrollfelder gesucht, da diese nach dem Fuzzen der variablen Felder neu berechnet werden müssen, damit die Pakete nicht als deformiert angezeigt werden. Um den Fuzzing-Prozess für den Tester zu vereinfachen, stellten Hernández et al. einen Template-basierten Ansatz für die Auswahl der Felder, die gefuzzt werden sollen, zur Verfügung.

Ein weiterer Ansatz für das Testen des MQTT-Protokolls, das auch auf Fuzzing basiert, wird von Casteur et al. vorgestellt [CAB+20]. Die Autoren haben zusätzlich zum Fuzzing den Fokus auf die Virtualisierung der Clients und der Broker gelegt. Hierfür wurden Docker Container mit Eclipse-Mosquitto eingesetzt, die über das MQTT-Protokoll miteinander kommunizieren. Auf einen Container wurde ein Client, der als Angreifer agiert, mit einem Angriffsskript geladen, während auf vielen anderen Containern Broker ausgeführt wurden. Das Angriffsskript beinhaltet ein Set von Paketen, wobei die Sets für jeden Broker unterschiedlich sein können, um unterschiedliche Szenarien abzudecken. Für jedes Szenario wird dann ein Wert basierend auf den Fehlern der zugrundeliegenden Pakete berechnet. Ein Fehler ist bspw., wenn das „User Name“-Flag auf 1 gestellt ist, jedoch keine Nutzerinformationen vorhanden sind. Beim Ausführen des Angriffs werden weitere Werte durch einen Algorithmus berechnet, die u. a. durch Fehler in den Logs oder der Dauer der einzelnen Szenarios zustande kommen. Wenn die Dauer eines Szenarios überdurchschnittlich hoch ist, liegt das Problem meist an fehlerhaften Paketen, die dazu führen, dass das Zielsystem ungewöhnlich reagiert und dadurch länger für die Antwort benötigt. Die Szenarien mit den höchsten Werten werden gespeichert, modifiziert und weitere Szenarien daraus gebildet, anschließend wird der gesamte Vorgang wiederholt. Die Autoren haben 600 Szenarien versendet und kamen zu dem Ergebnis, dass 35% davon negative Auswirkungen auf die Broker hatten.

Araujo Rodriguez und Macêdo Batista fanden in ihrer Recherche heraus, dass die meisten Schwachstellen des MQTT-Protokolls durch fehlerhafte MQTT-Kontrollpakete ausgelöst werden [AM20]. Der Fokus ihres veröffentlichten Artikels „Program-Aware Fuzzing for MQTT Applications“ lag daher auf Fuzzing, wobei untersucht wurde, ob Implementierungen

von Nicht-Blackbox-Fuzzern im MQTT-Bereich existieren. Ein Blackbox-Fuzzer hat keine Informationen über das SUT, dementsprechend wird eine riesige Menge an Testfällen benötigt, um Protokollzustände auf tieferen Ebenen zu erreichen. Bei der Recherche kam heraus, dass bisher keine Greybox- und Whitebox-Fuzzer im MQTT-Bereich existierten, jedoch diese durchaus Sinn ergeben, da die bisher genutzten Blackbox-Fuzzer nur eine Code-Abdeckung von 25% bis 33% erreichen. Die Autoren stellten daher ein Architekturkonzept für einen Greybox-Fuzzer für MQTT-Systeme vor, das allerdings bei der Veröffentlichung des Artikels noch nicht implementiert wurde. Um initiale Testfälle zu generieren, gibt es zwei Möglichkeiten: Es können MQTT-Pakete abgefangen oder effiziente evolutionäre Algorithmen zur Generierung der Startpakete verwendet werden. Die Pakete werden gefuzzt und direkt an das Zielsystem gesendet, wodurch es möglich ist, Grammatiken zu verwenden oder Whitebox-Fuzzing durchzuführen. Hierfür planen die Autoren Techniken, wie bspw. Stateful Fuzzing zu verwenden, das sich bereits in der Vergangenheit für Netzwerkprotokolle bewährt hat. Die effektivsten der erfolgreichen Testfälle sollen für zukünftige Iterationen aufbewahrt werden.

Ein weiterer Fuzzer, der neben MQTT auch andere Publish/Subscribe-Protokolle, wie bspw. CoAP unterstützt, wird von Zeng et al. vorgestellt [ZLG+20]. MultiFuzz ist ein abdeckungsbasierter Fuzzer, der kein Wissen über das Protokoll benötigt. Er speichert interessante Ausführungspfade von vorherigen Testläufen und verwendet diese als Eingabe für weitere Mutationen, um neue Pfade zu finden. Dabei beachtet der Fuzzer explizit, dass es bei Publish/Subscribe-Protokollen drei Parteien gibt, wodurch auch Fälle abgedeckt werden, in denen bspw. ein zweites Gerät Daten während des Fuzzings versendet. Die Architektur von MultiFuzz ist in vier Module aufgeteilt. Der Seed-Pool beinhaltet die Seed-Eingaben und weitere, während des Testens gefundene Eingaben. Das Scheduling-Modul wählt Seeds aus dem Seed-Pool aus und sendet diese an das Mutations-Modul, wo sie modifiziert und anschließend an das Ausführungs- und Überwachungsmodul weitergegeben werden. Dieses führt das SUT aus und beobachtet die Ergebnisse, wie Abstürze oder anderes Fehlverhalten. Interessante Pfade werden vom Scheduling-Modul für eine spätere Nutzung in den Seed-Pool gespeichert. Die Autoren haben den Fuzzer auf zwei bekannten MQTT-Implementierungen getestet - Eclipse Mosquitto und libCoAP - und gezeigt, dass MultiFuzz mehr Pfade und Abstürze gefunden hat als andere moderne Fuzzer, wie bspw. MOPT und AFLNET.

4.2 Standardkonfiguration und Angriffsmuster

Die Forschung im MQTT-Bereich konzentriert sich insbesondere auf die Bereitstellung von Sicherheitsschichten für die Transportprotokolle [SFR20b]. Es gibt einige Vorschläge, wie die Sicherheit des MQTT-Protokolls verbessert werden kann, die beispielsweise in [HBK18], [MG19] und [SNL19] beschrieben wurden, jedoch sind dies zum Großteil zusätzliche Ergänzungen zur MQTT-Spezifikation. Die Spezifikation selbst gibt Empfehlungen für die Verschlüsselung auf TLS-Ebene und den Einsatz von Authentifizierung, die jedoch

standardmäßig nicht aktiviert sind, sondern vom Nutzer zusätzlich konfiguriert werden müssen. Daher ist es wichtig, dass Tools existieren, die die Konfigurationen von MQTT-Brokern testen und nach weiteren Problemen bei der Bereitstellung des Brokers suchen.

Das Programm MQTT Security Assistant, kurz MQTTSA, ist ein Tool, welches automatisch unsichere Konfigurationen bei MQTT-basierten Systemen entdeckt und IoT-Entwicklern hilft, eine sichere Konfiguration des Brokers zu verwenden. Zudem kann MQTTSA potenzielle Schwachstellen des MQTT-Brokers mithilfe von einigen bekannten Angriffsmustern entdecken, generiert einen Bericht mit den gefundenen Problemen und liefert Tipps, wie diese gelöst werden können. Das Tool versucht zuerst, sich mit dem Broker zu verbinden und abhängig davon, ob die Authentifizierung erfolgreich war, werden weitere Module wie Daten-Sniffing oder das Brute-Forcing von Passwörtern durchgeführt. Daraufhin versucht MQTTSA alle SYS-Topics zu abonnieren und führt eine Aufnahme von allen Daten und Topics für 60 Sekunden durch, die dann an das Datenmanipulations-Modul gesendet werden. Dieses versucht Nachrichten an die Topics zu senden, um die Autorisierungseinstellungen des Brokers zu überprüfen. Außerdem versucht es, MQTT-Pakete zu modifizieren, mit dem Ziel, den Broker zum Absturz zu bringen. MQTTSA verwendet keinen Fuzzer und versendet keine fehlerhaften Pakete. Es wird stattdessen lediglich eine vordefinierte Liste an ungewöhnlichen Werten für bestimmte Felder eingefügt. Ein weiteres Modul ist für einen DoS-Angriff zuständig, indem riesige Pakete versendet werden und mehrere gleichzeitige Anfragen von einem einzigen Prozess mit mehreren Threads ausgeführt werden. Das letzte Modul ist für die Generierung des PDF-Berichts mit den Ergebnissen der ausgeführten Angriffe zuständig und beinhaltet auch Code-Schnipsel, um die gefundenen Probleme zu beheben.

Ein weiteres System, das Angriffsmuster verwendet, um die MQTT-Implementierung zu testen, wird von Sochor et al. in [SFR20a] und [SFR20b] vorgestellt. Angriffsmuster sind Abstraktionen von einzelnen Angriffen, die aus konkreten Beispielen von Exploits aus der Praxis bestehen. Sie enthalten Anweisungen über die allgemeinen Methoden zur Ausnutzung von Softwareschwachstellen und umfassen das Ziel, die Vorbedingungen, die einzelnen Aktionen und die Nachbedingungen des Angriffs. Für die Tests haben die Autoren einen Adapter in der Programmiersprache Java entwickelt, der einen MQTT-Client darstellt und mit Mosquitto, dem SUT, kommuniziert. Die Testsequenzen werden über Randoop generiert, der diese als JUnit Testfälle erstellt. Der Client sendet die Nachrichten der Testsequenzen an den Broker und vergleicht die Rückgabewerte mit den erwarteten Ergebnissen. Ein fehlgeschlagener Test gibt einen Hinweis auf ein mögliches Sicherheitsproblem. Jedoch sind nicht alle fehlgeschlagenen Tests gefährlich, da diese nur eine Abweichung im Verhalten andeuten, das Verhalten jedoch in manchen Fällen nicht klar in der Spezifikation definiert ist. Ein System Monitor ist für die Überwachung des Zustands des MQTT-Brokers und die Erkennung von abnormalem Verhalten, wie bspw. Nichtreagieren aufgrund stark erhöhter Ressourcennutzung, zuständig. Die Angriffsmuster

werden aus Kollektionen wie CAPEC[1] oder CVE[2] abgeleitet oder sind Fälle, die gegen die MQTT-Spezifikation verstoßen. Eines der Ziele des Tools ist, mithilfe von Regressionstests Diskrepanzen zwischen verschiedenen Implementierungen von MQTT-Brokern zu finden, die auf Fehler oder potenzielle Sicherheitsbedrohungen hinweisen können. Die Tests werden auf einer anderen Version des Brokers ausgeführt, um Unterschiede zwischen den verschiedenen Broker-Versionen zu finden. Alternativ zu Regressionstests können weitere Implementierungen des MQTT-Protokolls, wie bspw. ActiveMQ und Moquette, getestet werden.

4.3 Modellbasiertes Testen

Modellbasiertes Testen ist eine weitere vielversprechende Methode, um Sicherheitsprobleme bei MQTT-Systemen zu finden. Hierfür werden Modelle, die das SUT beschreiben, entworfen und das System damit analysiert. Verschiedene Modelle können unterschiedliche Aspekte des Systems abdecken, daher kann je nach Modelltyp bspw. die Implementierung eines MQTT-Brokers oder auch die Kommunikation zwischen verschiedenen Geräten untersucht werden. Im Folgenden werden zwei Forschungsarbeiten genauer beschrieben, die sich explizit mit dem Thema des Modellbasierten Testens in Verbindung mit dem MQTT-Protokoll beschäftigt haben.

Tanabe et al. stellten in [TTH20] einen Ansatz zum Testen eines gesamten MQTT-Netzwerks mithilfe von Modellen vor. Hierfür wird das zu untersuchende System als erweiterter, deterministischer Zustandsautomat modelliert. Die Zustände der einzelnen Geräte im MQTT-Netzwerk werden als eine Verteilung von Zuständen in einem integrierten Modell behandelt, wodurch das Verhalten von mehreren Geräten auf einmal beschrieben werden kann. Die Ankunft von MQTT-Nachrichten ist die Bedingung für einen Zustandswechsel. Eine weitere, interessante Erweiterung ist die Einführung des Begriffs "Timeout", wodurch es ermöglicht wird, Systeme zu beschreiben, die von der Zeit beeinflusst werden. Das ist wichtig, da im IoT-Kontext oftmals die Netzwerkkommunikation instabil wird, wenn beispielsweise die Geräte weit vom Router entfernt sind oder es sich um mobile Geräte handelt. Durch das Verwenden von Timeouts im Modell kann bspw. das Warten auf bestimmte Eingaben auf eine begrenzte Zeit limitiert werden. Das Modell wird verwendet, um automatisch Testfälle mithilfe des Programms Modbat zu generieren. Falls während des Testens ein Problem auftritt, wird ein Alarm an den Betreiber des Systems gesendet. Die Autoren haben in ihrem Experiment keine Sicherheitstests durchgeführt. Ein solches Modell kann jedoch verwendet werden, um Testfälle zu generieren, die die Sicherheit eines gesamten IoT-Netzwerkes untersuchen, da das Verhalten sämtlicher Geräte und ihrer Kommunikation miteinander beschrieben werden kann.

[1] https://capec.mitre.org/
[2] https://cve.mitre.org/

Ein weiterer Modellbasierter Testansatz für MQTT-Broker-Implementierungen wurde von Tappler und Aichernig [TAB17] vorgestellt. Im ersten Schritt wurden Modelle mehrerer Implementierungen von MQTT-Brokern generiert und paarweise untersucht, ob diese sich unterschiedlich verhalten. Die Autoren verwendeten für die Erstellung der Modelle die Technik „Active Automata Learning“, um Wissen über das Verhalten des MQTT-Brokers zu gewinnen. Es wurden Tests und Abfragen durchgeführt und die entsprechenden Beobachtungen analysiert, wodurch ein Modell trainiert wurde, welches das System beschreibt. Falls Unterschiede zwischen den Broker-Implementierungen auftraten, wurden diese Problemfälle manuell anhand der MQTT-Spezifikation überprüft, da diese Unterschiede wahrscheinlich auf ein fehlerhaftes Verhalten hinweisen. Der Fokus des beschriebenen Ansatzes liegt auf dem Testen des Systems bezüglich der Konformität der Spezifikation, also ob die Systeme die MQTT-Spezifikation richtig implementieren. Die Autoren fanden in allen untersuchten Implementierungen außer in Mosquitto fehlerhaftes Verhalten und Verstöße gegen die MQTT-Spezifikation.

Das Verwenden von Standards ist eine bekannte Methode, um das Vertrauen der Benutzer in ein Produkt zu stärken. Konformitätstestsuiten erhöhen die Wahrscheinlichkeit, dass Softwareprodukte mit ihrer Spezifikation übereinstimmen. Die korrekte Implementierung und Nutzung von Standards führen zu Portabilität und Interoperabilität, aber auch zu einer Erhöhung der Sicherheit. In der Spezifikation der Standards sind oftmals Sicherheitsanforderungen definiert, die die Implementierungen des Standards einhalten müssen [Ahm18]. Darum sind Tests bezüglich der Konformität zum MQTT-Standard auch eine Möglichkeit, um automatisiert die Sicherheit von MQTT-Systemen zu testen. Es gibt zahlreiche Artikel, die sich mit Testmethoden befassen, die die Konformität der Implementierungen mit dem MQTT-Protokoll untersuchen [MVMC17] [SKRW17] [AZ21]. Da Sicherheitsrichtlinien jedoch nur einen kleinen Teil der MQTT-Spezifikation ausmachen, werden im Rahmen dieser Masterarbeit nur die Teile der Spezifikation, die sich mit der Sicherheit des Protokolls befassen, untersucht. Es wird nicht genauer auf Konformitätstestsuiten eingegangen, die das gesamte Protokoll betreffen.

Fazit

In der Literatur existieren viele Ansätze zum automatisierten Testen von MQTT-Brokern. Die zuvor beschriebenen Ansätze legen den Fokus jeweils auf einen speziellen Aspekt der Sicherheit, wodurch es nicht möglich ist, einen umfassenden Blick auf den Sicherheitszustand von MQTT-Implementierungen zu erlangen. Der Testansatz, der in dieser Masterarbeit ausgearbeitet wird, nutzt Konzepte der dargestellten Artikel, um ein System zu entwickeln, das mehrere Aspekte des Sicherheitstestens kombiniert und in einem einzelnen Konzept abdeckt. Das hier entworfene Systemkonzept kombiniert die Aspekte Standardkonfiguration, Angriffsmuster und Fuzzing. Modellbasiertes Testen spielt dabei lediglich im Zusammenhang mit Angriffsmustern eine Rolle, da diese ein abstraktes Modell eines Angriffs sind. Das aus dem resultierenden Konzept und den implementierten Tests

abgeleitete Ergebnis ist somit aufgrund der unterschiedlichen Sichtweisen aussagekräftiger, als wenn nur ein einzelner Sicherheitsaspekt untersucht wird. Der Testansatz wird im nächsten Kapitel genauer vorgestellt.

5 Ansatz für MQTT-Sicherheitstests

In Abschnitt 3.3 wurden die Sicherheitsprobleme, die im Zusammenhang mit dem MQTT Protokoll auftreten, zusammengefasst. Dabei wurden zwei Hauptprobleme identifiziert: Das erste Problem ist die unsichere Standardkonfiguration der meisten MQTT-Broker, die keine Authentifizierung, Autorisierung und Verschlüsselung beinhaltet. Wenn ein Angreifer die IP-Adresse des Brokers kennt, können in diesem Fall sämtliche Nachrichten abgehört und an beliebige Topics im Netzwerk versendet werden. Die zweite identifizierte Herausforderung ist, dass MQTT-Broker oder MQTT-Clients oftmals nicht mit fehlerhaften Datenpaketen umgehen können, was im schlimmsten Fall zu einem Komplettausfall der Systeme führen kann. Der MQTT-Broker ist das Herzstück der MQTT-Systeme. Falls dieser abstürzt, können die Geräte im Netzwerk nicht mehr miteinander kommunizieren. Um solche Sicherheitsprobleme frühzeitig zu entdecken, wird im Rahmen dieser Masterarbeit ein Ansatz entworfen und implementiert, der die Sicherheit von MQTT-Brokern testet. Dieser Ansatz wird MQTT-AIO genannt und im Folgenden detailliert beschrieben.

5.1 Testansatz MQTT-AIO

Die grundlegende Idee ist, die Hauptprobleme von MQTT - mangelhafte Standardkonfiguration und fehlerhafte Pakete - mit geeigneten automatisierten Testmethoden, wie bspw. Angriffsmustern und Fuzzing, zu testen. Eine High-Level-Übersicht des Testansatzes ist in Abbildung 5.1 dargestellt.

Bei einer Umsetzung des Ansatzes nimmt das System die Rolle eines Test-Clients ein und besteht aus insgesamt fünf Komponenten. Drei dieser Komponenten testen mit unterschiedlichen Ansätzen den MQTT-Broker, eine Komponente ist für die Überwachung des SUTs zuständig und eine für die Generierung der Testberichte. Das SUT ist eine Referenzimplementierung eines MQTT-Brokers, dessen Sicherheit analysiert werden soll. Relevante Daten und Parameter für die jeweiligen Komponenten werden über eine config.json-Datei in das Programm eingelesen. Mit der Datei kann der Tester bspw. die Authentifizierungsdaten übergeben oder auswählen, welche Tests durchgeführt werden sollen. Der Ausgabebericht wird in JSON- und PDF-Format generiert, sodass die Ergebnisse der Sicherheitstests sowohl maschinen- als auch menschenlesbar sind.

Die erste Testkomponente überprüft die Konfiguration des MQTT-Brokers. Hier wird beispielsweise untersucht, ob sich ein Client anonym ohne Authentifizierung oder mit unvollständiger Authentifizierung mit dem Broker verbinden kann. Ein weiterer Test ist,

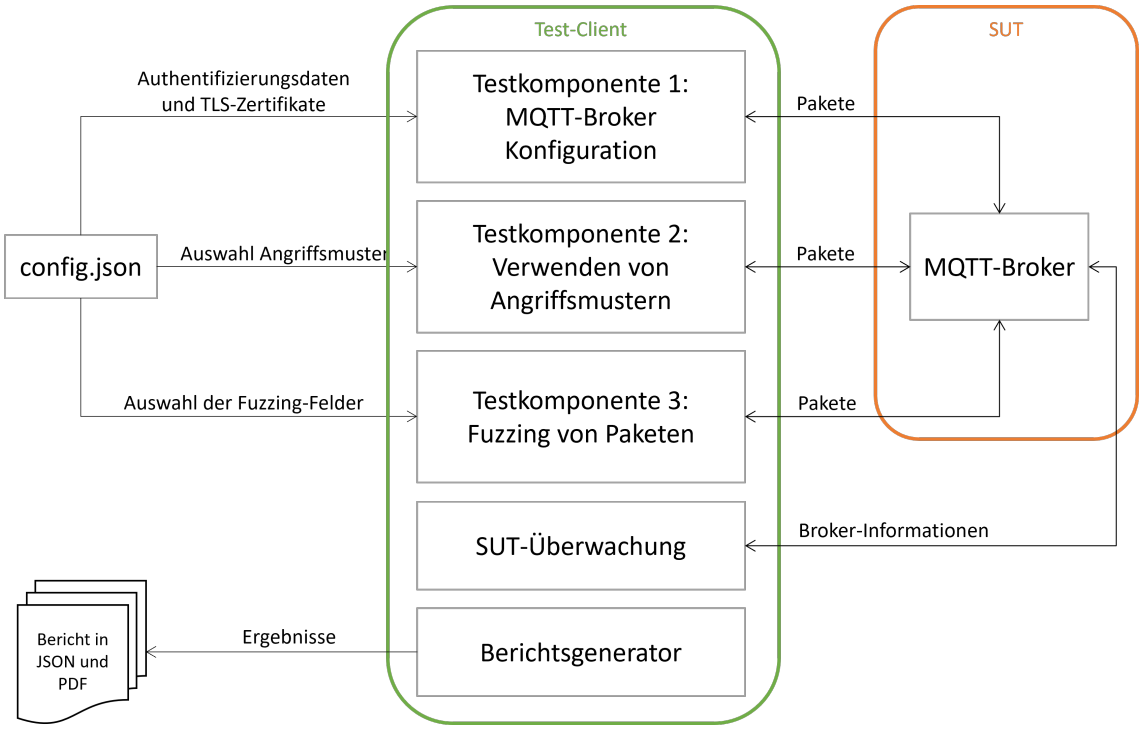


Abbildung 5.1: High-Level-Übersicht von MQTT-AIO

ob sich ein Client ohne TLS verbinden kann, was zur Folge hat, dass die Kommunikation durch einen Man-in-the-Middle-Angriff abgehört werden kann. Ein Client soll zudem nicht die Möglichkeit haben, sämtliche Topics über „#“ oder die SYS-Topics zu abonnieren. Solche Berechtigungen sind für einen zufälligen Client selten notwendig, und stellen ein Sicherheitsrisiko dar, da ein böswilliger Client alle Nachrichten abhören kann und über die SYS-Topics Informationen über den Broker erhält. Auch wenn Authentifizierung verwendet wird, können Sicherheitsprobleme vorhanden sein. So kann der Nutzer bspw. die voreingestellte Benutzername/Passwort-Kombination unverändert verwenden, was von einem Angreifer einfach ausgenutzt werden kann. Dieser kann mithilfe von Passwortlisten Zugriff auf das System erlangen.

Die zweite Komponente nutzt Angriffsmuster, um explizit einige bekannte Sicherheitsprobleme bei MQTT-Brokern zu testen. Angriffsmuster sind Beschreibungen gängiger Methoden zur Ausnutzung von Softwareschwachstellen und werden aus Perspektive des Angreifers formuliert. Die Schwachstellen können u. a. aus der Analyse von konkreten Beispielen aus der realen Welt gewonnen werden [ATP21]. In einem Angriffsmuster werden die einzelnen Schritte zur Erstellung des Angriffs definiert und das erwartete Ergebnis beschrieben. Diese Ergebnisse können bspw. aus den in der MQTT-Spezifikation definierten Sicherheitsanforderungen oder aus den Rückgabewerten einer Referenzimplementierung entnommen werden. Auf Basis dieser Angriffsmuster können dann ausführbare Angriffe definiert werden, die sich auf spezifische Schwachstellen von MQTT-Systemen beziehen. In diesem Zusammenhang wird auch die Konformität zur MQTT-Spezifikation mitgeprüft.

In [SFR20a] und [SFR20b] wurden einige explizite Angriffsmuster auf MQTT-Broker definiert, die aus den Seiten CAPEC[1], CVE[2], CWE[3] und der MQTT-Spezifikation abgeleitet wurden. Diese Angriffsmuster werden bei der prototypischen Implementierung dieses Testansatzes verwendet.

Die dritte Testkomponente ist ein Fuzzer. Ein Protokoll-Fuzzer modifiziert Felder im Paket und testet, wie das SUT mit den zum Teil fehlerhaften Paketen umgeht. Im schlimmsten Fall stürzt das SUT komplett ab. Im Gegensatz zu den Tests mit Angriffsmustern können Fuzzer unerwartete Probleme automatisiert finden. Falls kein Zugriff auf den Quellcode des Brokers besteht, können Black-Box-Fuzzer verwendet werden, welche bestehende Daten nach einem Zufallsprinzip verändern. Alternativ können auch White-Box- und Grey-Box-Fuzzer eingesetzt werden, die durch die Verwendung zusätzlicher Informationen über die interne Logik und den Quellcode des SUTs eine größere Abdeckung erreichen.

Ein Fuzzer kann beispielsweise als Proxy-Server zwischen dem MQTT-Broker und den MQTT-Clients stehen und somit die Pakete abfangen, modifizieren und an die andere Partei weiterleiten. Diese Art von Fuzzer wird unter anderem in [HVL18] und [Ram] genauer erläutert. Ein Proxy-Fuzzer hat den Vorteil, dass mutationsbasierte Fuzzer auf Live-Pakete angewandt werden können. Diese können eine Funktion auf einen bestehenden Wert eines Feldes anwenden und diesen so verändern. Dadurch ist gewährleistet, dass die ursprünglichen Paketsequenzen korrekt sind und auch die versendeten Pakete nicht fehlerhaft waren. Es ist wichtig, dass nicht alle Pakete gefuzzt werden, da Pakete wie SUBSCRIBE oder PUBLISH darauf basieren, dass sich der Client mithilfe eines korrekten CONNECT-Pakets mit dem Broker verbunden hat. Wenn fehlerhafte Pakete mit falschen Kontrollfeldern geschickt werden, beendet der Broker die Verbindung zum Client und die daraufhin verschickten Pakete werden ignoriert. Alternativ zu mutationsbasierten Fuzzern gibt es generierungsbasierte Fuzzer, die mit intelligenten Generierungsfunktionen Werte unabhängig vom Ursprungswert erstellen. Das ist wesentlich komplexer, da der Fuzzer über Informationen bezüglich des Protokolls und der einzelnen Felder verfügen muss. Zudem muss, wie zuvor beschrieben, der Client korrekt mit dem Broker mithilfe eines validen CONNECT-Pakets verbunden sein, um weitere Pakete versenden zu können.

Anstatt die Pakete in einem Proxy-Server abzufangen, können auch bspw. mit Wireshark[4] aufgenommene MQTT-Paketsequenzen für das Fuzzing verwendet werden. Der Client kann die Sequenz lesen, selbst modifizieren und direkt an den Broker versenden. Ein Beispiel für Fuzzing ohne Proxy-Server ist der Mutinity Fuzzer[5], der Fuzzing-Funktionen auf aufgezeichnete Paketsequenzen anwendet. Dies hat den Vorteil, dass das Netzwerk-Fuzzing so schnell wie möglich gestartet werden kann, jedoch auf Kosten der Gründlichkeit, da die Anzahl der aufgezeichneten Pakete begrenzt ist.

[1] https://capec.mitre.org/
[2] https://cve.mitre.org/
[3] https://cwe.mitre.org/
[4] https://www.wireshark.org/
[5] https://github.com/Cisco-Talos/mutiny-fuzzer

5.2 Umsetzung des Testansatzes

In diesem Kapitel wird eine prototypische Implementierung des im vorherigen Abschnitt dargestellten Ansatzes beschrieben. Das Programm ist in der Programmiersprache Python implementiert und verwendet die Bibliotheken Paho[6] und Scapy[7] zur Kommunikation mit dem Broker und bietet die Möglichkeit, Radamsa[8] zum Fuzzen der Pakete zu verwenden. Radamsa ist ein Testfallgenerator, der einen Eingabewert einliest und einen daraus modifizierten Wert ausgibt. Radamsa generiert bei gleicher Eingabe meist unterschiedliche Ausgaben, sodass bei jedem Fuzzing-Schritt unterschiedliche Modifikationen durchgeführt und damit verschiedene potenzielle Schwachstellen abgedeckt werden können. Ein wichtiger Aspekt für das Konzept ist die einfache Erweiterbarkeit, wodurch es möglich ist, im Nachhinein weitere Konfigurationstests und Angriffsmuster festzulegen. Mithilfe einer Konfigurationsdatei können die Testmethoden und Einstellungen des Systems individuell konfiguriert werden, sodass bspw. bei einer weiteren Ausführung des Programms bereits getestete Methoden nicht erneut ausgeführt werden.

Da es sich um eine prototypische Umsetzung von MQTT-AIO handelt, werden im Rahmen dieser Masterarbeit lediglich das Grundgerüst des Testsystems und einige relevante Tests bzw. Angriffe implementiert. Da nicht alle MQTT-Broker das MQTT-Protokoll in der Version 5.0 implementieren, jedoch die meisten die Version 3.1.1 unterstützen, wird das Verhalten der Version 5.0 nicht getestet. Beispielsweise unterstützt der beliebte Broker Moquette im Jahr 2021, also 2 Jahre nach Einführung von MQTT in der Version 5.0, immer noch keine v5.0 Pakete. MQTT-AIO testet jedoch, ob der Broker mit Version 5.0 Paketen umgehen kann, da diese Version Verbesserungen in Funktion und Sicherheit beinhaltet. Im Folgenden wird genauer auf die einzelnen Komponenten der prototypischen Implementierung eingegangen.

5.2.1 Konfigurationsdatei

Die config.json-Datei ist die Konfigurationsdatei von MQTT-AIO, durch die die zu untersuchenden Methoden individuell angepasst werden können. In der Datei werden unter anderem die Daten zur Kommunikation mit dem SUT übergeben. Dazu gehören Authentifizierungsdaten wie Benutzername und Passwort und Verschlüsselungsdaten, wie bspw. Zertifikatspfade. Wenn keine Authentifizierung und Verschlüsselung verwendet wird, können diese Felder leer gelassen werden.

Im Anschluss werden die Standardeinstellungen des MQTT-Brokers untersucht und definiert, welche Angriffe durchgeführt werden sollen. Hierfür werden separate Funktionen implementiert, die jeweils einem bestimmten Aspekt der Einstellung oder einem Angriff

[6] https://github.com/eclipse/paho.mqtt.python
[7] https://github.com/secdev/scapy
[8] https://gitlab.com/akihe/radamsa

zugeordnet sind. Die Namen der Funktionen werden in der config.json-Datei aufgelistet und jeder Funktion wird ein Boolean-Wert zugewiesen. Nur wenn der Wert der jeweiligen Funktion „true“ ist, wird die Funktion beim Testen aufgerufen. In Listing 5.1 ist eine beispielhafte Konfiguration von MQTT-AIO mit einem Teil der zu testenden Funktionen für die Testkomponente 1 und 2 zu sehen. Die konkret implementierten Funktionen werden in den Abschnitten 5.2.2 und 5.2.3 beschrieben.

```
"broker_settings_config": {
    "connect_without_authentication": true,
    "account_name_no_password": true,
    "subscribe_sys_topics": true,
    "get_broker_and_version": true
},
"attack_pattern_config": {
    "attack_two_connects_same_id": true,
    "connect_variable_header_minus_one": true,
    "connect_variable_header_plus_one": true,
    "attack_subscribe_without_payload": true,
    "send_subscribe_with_wildcard_in_topic": true
},
```

Listing 5.1: Beispielkonfiguration für MQTT-AIO Testkomponenten 1 und 2

Im letzten Teil der config.json-Datei werden die Fuzzing-Konfigurationen mitgegeben. Hier wird die Struktur der Pakete definiert und die zu fuzzenden Pakete und Felder werden ausgewählt. In Listing 5.2 ist ein Teilausschnitt der Fuzzing-Konfiguration für das PUBLISH-Paket dargestellt. Probabilities sagt hier aus, mit welcher Wahrscheinlichkeit ein MQTT PUBLISH-Paket gefuzzt werden soll. Da die Wahrscheinlichkeit in diesem Beispiel bei 25% liegt, wird hier nur jedes vierte PUBLISH-Paket gefuzzt. Die einzelnen Schlüssel im unteren „MQTTPublish“ sind die Namen der Felder des PUBLISH-Kontrollpakets. Ein Feld ist ein verschachteltes Objekt, welches die vier Schlüssel „layer“, „field_probability“, „fuzzer“ und „method“ enthalten kann.

Der „layer“-Schlüssel sagt aus, ob sich das Feld auf der Header-Ebene, der Payload-Ebene oder einer anderen Ebene befindet. Die Struktur und die Namen der Felder müssen dabei denen der Scapy-Bibliothek entsprechen, da die Felder über den Namen an die Bibliothek gebunden sind. Erst während der Laufzeit des Programms werden die Klassen der Scapy-Bibiothek dynamisch geladen. Die zwei Schlüssel „len“ und „length“ sind Kontrollfelder, d.h. wenn diese nicht die korrekten Werte besitzen, gilt das Paket als fehlerhaft. Der Schlüssel „field_probability“ gibt dem Programm mit, mit welcher Wahrscheinlichkeit dieses spezifische Feld modifiziert werden soll. Zum jetzigen Zeitpunkt können Feldmanipulationen entweder über Radamsa oder manuell mit selbstdefinierten Funktionen über Scapy durchgeführt werden. Falls die Manipulation manuell durchgeführt werden soll,

```
"packet_probabilities": {
    "MQTTPublish": 25
}
"packets": {
    "MQTTPublish": {
        "type": {"layer":  "header", "field_probability": 10, "fuzzer": "scapy",
                 "method": ["mutate_increment_value"]},
        "DUP": {"layer":  "header", "field_probability": 10, "fuzzer": "radamsa"},
        "QOS": {"layer":  "header", "field_probability": 10, "fuzzer": "scapy",
                "method": ["mutate_xor_value"]},
        "RETAIN": {"layer":  "header", "field_probability": 10, "fuzzer": "scapy",
                   "method": ["mutate_increment_value"]},
        "len": {"layer":  "header", "field_probability": 10, "fuzzer": "scapy",
                "method": ["generate_random_uint8"]},
        "length": {"layer":  "payload", "field_probability": 10, "fuzzer": "scapy",
                   "method": ["mutate_increment_value"]},
        "topic": {"layer":  "payload", "field_probability": 10, "fuzzer": "radamsa"},
        "msgid": {"layer":  "payload", "field_probability": 10, "fuzzer": "scapy",
                  "method": ["generate_random_uint8"]},
        "value": {"layer":  "payload", "field_probability": 25, "fuzzer": "scapy"
                  "method": ["mutate_insert_random_char"]},
    }
},
```

Listing 5.2: Beispielkonfiguration für MQTT-AIO Testkomponente 3

muss eine Liste an Funktionsnamen mitgegeben werden, wodurch zufällig eine der mit dem Namen assoziierten Funktionen dynamisch aufgerufen wird. Auf Radamsa und die Scapy-Funktionen wird in Abschnitt 5.2.4 genauer eingegangen.

5.2.2 Testen der MQTT-Broker-Konfiguration

Eine wichtige Erkenntnis aus Abschnitt 3.3 ist die Unsicherheit des Brokers bei Verwendung der Standardkonfiguration. Standardmäßig ist ein MQTT-Broker ohne Authentifizierung und ohne Verschlüsselung konfiguriert, was zu Problemen mit dem Datenschutz und der Datenintegrität führen kann. Bei fehlender Authentifizierung kann ein Angreifer die Topics abonnieren und alle Nachrichten im Netzwerk abhören. Falls keine Verschlüsselung verwendet wird, kann ein Angreifer die Pakete abfangen, lesen, modifizieren, an den Broker weiterleiten und dadurch Schaden anrichten. Die erste Komponente von MQTT-AIO testet die Konfiguration des SUTs und prüft, ob der Broker sicher konfiguriert ist. Nach dem Testen werden in dem Ausgabebericht die Ergebnisse der Analyse beschrieben und Tipps gegeben, wie Schwachstellen verbessert werden können.

In MQTT-AIO werden Überprüfungen auf Grundlage der folgenden Fragestellungen durchgeführt:

- Kann sich ein Client ohne TLS verbinden?
- Kann sich ein Client ohne Authentifizierung verbinden?
- Funktioniert die vom Nutzer in der config.json mitgegebene Nutzername/Passwort-Kombination?
- Funktionieren die vom Nutzer in der config.json mitgegebenen Zertifikate für TLS?
- Kann sich ein Client nur mit Accountname, aber ohne Passwort verbinden? (CVE-2016-9877[9])
- Kann der Broker mit MQTT v5.0 Paketen umgehen?
- Kann ein Client Informationen über den Broker über das Abonnieren von „SYS/#“ bekommen?
- Was sind die aktuellen Versionen der populären MQTT-Broker?
- Kann ein Client Informationen über den verwendeten Broker und die Brokerversion erhalten?
- Kann ein Client alle Topics abonnieren mit „#“?
- Ist die mitgegebene Account/Passwort-Kombination eine Kombination, die in öffentlichen Passwortlisten steht?

CVE-2016-9877 ist eine Schwachstelle, die in Pivotal RabbitMQ und RabbitMQ PCF in bestimmten Versionen gefunden wurde. MQTT Verbindungsversuche mit Nutzername/Passwort-Kombinationen waren erfolgreich, wenn der Nutzername vorhanden ist und das Passwortfeld ausgelassen wurde. Die Schwachstelle ist seitdem durch ein Update der Broker behoben, jedoch ist es trotzdem sinnvoll andere Broker-Implementierungen auf diese Schwachstelle zu testen.

Für diese Überprüfungen wird hauptsächlich die Eclipse Paho Bibliothek verwendet. MQTT-AIO nimmt dabei die Rolle eines MQTT-Clients ein und sendet Pakete an den Broker. Basierend auf den (fehlenden) Antwortpaketen können die obigen Fragen beantwortet und somit die Konfiguration des Servers getestet werden.

[9] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-9877

5.2.3 Testen mithilfe von Angriffsmustern

Im Folgenden werden die Angriffsmuster, die in MQTT-AIO implementiert sind, detailliert erläutert. Für die Implementierung der spezifischen Angriffe werden die Python-Bibliotheken Paho oder Scapy verwendet. Scapy ist eine Bibliothek zur Paketmanipulation und in der Lage, Pakete von einer Vielzahl an Protokollen zu erstellen und zu dekodieren. Im Gegensatz zu Paho kann Scapy Pakete mit fehlerhaften Kontrollfeldern erstellen und versenden. Paho führt eine Reihe von Überprüfungen durch, um sicherzugehen, dass die Pakete nicht fehlerhaft sind. Das Versenden von fehlerhaften Paketen ist wichtig, da einige der Angriffsmuster auf fehlerhaften Feldern basieren. Als Referenzimplementierung für die Tests wurde Mosquitto ausgewählt und die Ergebnisse der anderen Broker mit denen von Mosquitto verglichen.

Angriffsmuster bestehen immer aus einer Vorbedingung, der Methode und dem erwarteten Ergebnis. Wenn ein erwartetes Ergebnis auftritt, gibt MQTT-AIO den Rückgabewert 0 zurück und wenn das Ergebnis abweicht, wird der Wert 1 ausgegeben. Falls der Client sich bspw. nicht mit dem MQTT-Broker verbinden kann und dies in dem Angriffsmuster gefordert ist, wird der Rückgabewert 2 zurückgegeben. Idealerweise geben sämtliche Angriffe den Rückgabewert 0 zurück. So kann der Tester bei einer Abweichung die Ursache des Problems auf manuelle Weise suchen. Das ist wichtig, da eine Abweichung nicht automatisch einen Fehler bedeuten muss, sondern auch zeigen kann, dass der Broker die Spezifikation auf eine andere Weise interpretiert.

Dieser Teil von MQTT-AIO basiert auf den von Socher et al. in [SFR20b] gefundenen Angriffsmustern. Im Rahmen dieser Masterarbeit wurden die ersten 15 Angriffsmuster des Papers ausgewählt und in MQTT-AIO implementiert. Zusätzlich wurde nach weiteren Angriffsmustern gesucht, die sich aus der MQTT-Spezifikation ableiten lassen oder seit Veröffentlichung des Papers als Schwachstelle in CVE dokumentiert wurden. MQTT-4.7.3-2 besagt, dass in den Topic-Namen und Topic-Filtern das Null-Zeichen U+0000 nicht vorhanden sein darf. Aus dieser Definition wurde Angriffsmuster 16 aufgebaut.

Zwei der neu entdeckten Schwachstellen sind CVE-2021-28166[10] und CVE-2021-34432[11]. Diese wurden im Jahre 2021 entdeckt und bilden die letzten beiden Angriffsmuster 17 und 18 in MQTT-AIO. Angriffsmuster 17 testet, ob ein CONNACK-Paket vom Client an den Broker diesen zum Absturz bringen kann. Das wird getestet, weil entdeckt wurde, dass CONNACK-Pakete in der MQTT v5.0 Mosquitto in den Versionen 2.0.0-2.0.9 zum Absturz gebracht haben. Ähnlich ist es bei CVE-2021-34432, wo PUBLISH-Pakete mit einer Topic-Länge von 0 Mosquitto in den Versionen 2.06 und 2.07 abstürzen ließen. Die komplette Liste der in MQTT-AIO implementierten Angriffsmuster ist in Tabelle 5.1 zu finden.

[10] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-28166
[11] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-34432

Nr.	Vorbed.	Methode	Erwartetes Ergebnis
1	Keine	Sende zwei CONNECT-Pakete mit gleicher Client-ID.	Der Broker trennt die Verbindung und weitere Pakete werden ignoriert.
2	Keine	Sende CONNECT-Paket mit Länge der variablen Kopfzeile +1.	Der Broker akzeptiert das Paket nicht.
3	Keine	Sende CONNECT-Paket mit Länge der variablen Kopfzeile -1.	Der Broker akzeptiert das Paket nicht.
4	Verbunden	Sende ein PUBLISH-Paket mit einer Payload-Größe von 128 MB.	Der Broker akzeptiert das Paket nicht.
5	Verbunden	Sende ein SUBSCRIBE-Paket ohne Payload.	Der Broker akzeptiert das Paket nicht und trennt die Verbindung.
6	Verbunden	Sende ein SUBSCRIBE-Paket mit Wildcard im Topic-Name.	Der Broker akzeptiert das Paket nicht.
7	Verbunden	Sende ein SUBSCRIBE-Paket mit Maskierungszeichen im Topic-Name.	Der Broker akzeptiert das Paket und der Client abonniert erfolgreich das Topic.
8	Verbunden	Sende ein PUBLISH-Paket mit Maskierungszeichen in Payload.	Der Broker akzeptiert das Paket und veröffentlicht die Nachricht.
9	Verbunden	Sende ein PUBLISH-Paket mit Wildcard im Topic-Name.	Der Broker akzeptiert das Paket nicht und veröffentlicht die Nachricht nicht.
10	Keine	Sende ein CONNECT-Paket mit QoS=3.	Der Broker trennt die Verbindung.
11	Keine	Sende ein CONNECT-Paket mit langem Client-ID.	Der Broker akzeptiert das Paket und verbindet sich mit dem Client.
12	Keine	Sende ein CONNECT-Paket mit fehlerhaftem Protokoll-Namen.	Der Broker akzeptiert das Paket nicht oder trennt die Verbindung.
13	Keine	Sende ein CONNECT-Paket mit fehlerhafter Protokoll-Version.	Der Broker lehnt das Paket ab und gibt eventuell einen CONNACK-Paket mit Return Code 0x01 zurück.
14	Keine	Sende ein CONNECT-Paket mit Will-Flag=0 und Will-QoS-Flag=2.	Der Broker akzeptiert das Paket nicht.
15	Keine	Sende ein CONNECT-Paket mit aktivierten Username/Password Flags, aber ohne Authentifizierungsdaten.	Der Broker akzeptiert das Paket nicht.
16	Verbunden	Sende ein PUBLISH-Paket mit Null Charakter im Topic-Name.	Der Broker lehnt das Paket ab.
17	Verbunden	Sende ein CONNACK-Paket an den Broker.	Der Broker lehnt das Paket ab. Hat bei Mosquitto 2.0.0-2.0.9 und MQTT v5 zu einem Absturz geführt (CVE-2021-28166).
18	Verbunden	Sende ein PUBLISH-Paket mit einer Topic-Länge von 0.	Der Broker lehnt das Paket ab. Kann zu einem Absturz führen (CVE-2021-34432).

Tabelle 5.1: Implementierte Angriffsmuster (angelehnt an [SFR20b])

5.2.4 Testen mithilfe von Fuzzing

Die letzte Testkomponente stellt der Fuzzer dar. Rodriguez und Batista fanden in ihrer Studie heraus, dass 71,15% aller mit MQTT zusammenhängenden Schwachstellen, die in NVD[12] gesammelt wurden, auf fehlerhaften MQTT-Kontrollpaketen basieren [AM20]. Ein Fuzzer eignet sich hervorragend zur automatisierten Fehlerfindung anhand von fehlerhaften MQTT-Paketen. Pakete werden so modifiziert, dass größtenteils fehlerhafte Pakete entstehen, die dann an den Broker gesendet werden.

Das System agiert hier als ein TCP Proxy-Server, der zwischen einem MQTT-Broker und einem Client vermittelt. Wenn sich der Client mit dem Proxy-Server verbindet, stellt dieser eine Verbindung zum MQTT-Broker her. Der TCP Proxy empfängt die vom Client versendeten Pakete, liest und fuzzt diese und gibt sie an den Broker weiter. Die Antwortpakete vom Broker werden dann wieder an den Proxy gesendet, woraufhin diese an den Client weitergeleitet werden. Die Pakete in Richtung Client können auch gefuzzt werden, wodurch die Client-Implementierung getestet werden kann. Das ist jedoch nicht Teil dieser Masterarbeit, da hier auf die Schwachstellen der MQTT-Broker-Implementierungen fokussiert werden.

MQTT-AIO agiert zudem als MQTT-Client und generiert zusätzlich zu den Paketen, die sowieso im Netzwerk versendet werden, eine Sequenz von Kontrollpaketen, die dann auch in den Fuzzing-Prozess integriert werden. Diese können für das Testen des Brokers verwendet werden, falls das Netzwerk von sich aus keine Pakete versendet. Jedes Paket, welches abgefangen wird, wird mit einer bestimmten Wahrscheinlichkeit gefuzzt. Dies ist insofern wichtig, da nach einem fehlerhaften Paket die Verbindung vom Broker zum Client unterbrochen wird und sämtliche weiteren Pakete nicht mehr akzeptiert werden, bis eine neue Verbindung aufgebaut wurde. Zudem wird jedem einzelnen Feld eines Kontrollpakets eine Wahrscheinlichkeit, ob das jeweilige Feld gefuzzt werden soll, zugewiesen. Diese Wahrscheinlichkeiten können in der config.json-Datei mitgegeben werden. Standardmäßig wird jedem Paket und Feld der Wert 10% zugewiesen, was der Tester aber individuell an seine Bedürfnisse anpassen kann.

Für den Fuzzing-Prozess kann zwischen der Radamsa-Bibliothek und manuellen Funktionen ausgewählt werden. Radamsa ist ein Testfall-Generator, der basierend auf den Eingaben eine Reihe von unterschiedlichen Testfällen zurückgibt. MQTT-AIO kann das in dem Proxy abgefangene Paket decodieren und die Werte der einzelnen Felder an Radamsa weitergeben. Die Ausgaben werden dann in das Feld geschrieben und ein MQTT-Paket aus den Feldern instanziiert. Alternativ besitzt MQTT-AIO eine Reihe von eigenen Funktionen, die Testfälle entweder vollständig neu generieren oder aus vorhandenen Werten mutieren können. Ein Beispiel für eine Generierungsfunktion ist das Generieren eines zufälligen vorzeichenlosen Integers, der mit 16 Bit gespeichert wird. Eine Mutationsfunktion ist bspw. das Inkrementieren der Eingabe, falls die Eingabe eine Nummer ist. Über die Konfiguration

[12] https://nvd.nist.gov/

kann MQTT-AIO eine Liste von Funktionen mitgegeben werden, von der zufällig eine Funktion ausgewählt wird, die dann für das Fuzzing des jeweiligen Feldes verwendet wird.

5.2.5 SUT-Überwachung

Um Probleme bei dem SUT zu erkennen, ist eine Komponente zur Überwachung des Systems wichtig. Eine wichtige Metrik ist hier die Ressourcenbelastung der CPU und des Arbeitsspeichers. Eine stark erhöhte Auslastung könnte auf eine Schwachstelle des Brokers hindeuten, die bspw. durch Memory Leaks ausgelöst wird. Auch ist es wichtig zu wissen, ob der Broker noch erreichbar ist oder ob dieser bspw. durch ein fehlerhaftes Paket beim Fuzzen zum Absturz gebracht wurde.

MQTT-AIO überwacht den SUT, indem in regelmäßigen Abständen Pakete an den Broker geschickt werden und auf die Bestätigung gewartet wird. Wenn ein Bestätigungspaket vom Broker zurückgesendet wird, ist dieser funktionsfähig und erreichbar. Falls keine Bestätigung ankommt, wird das Programm beendet und der Tester kann in den Log-Dateien nachschauen, zu welchem Zeitpunkt der Broker abgestürzt ist und was die Ursache des Absturzes sein könnte. Zusätzlich enthält MQTT-AIO auch eine Option, sich über SSH mit dem SUT zu verbinden und die CPU- und Arbeitsspeicherauslastung der Geräte abzurufen. Diese Informationen werden in regelmäßigen Abständen in einer Log-Datei gespeichert. Dadurch kann der Tester die Ressourcenauslastung analysieren und beobachten, zu welchen Zeitpunkten eine hohe Auslastung zu finden ist.

5.2.6 Berichtsgenerator

MQTT-AIO schreibt in jedem Schritt relevante Informationen in Log-Dateien. Beispielsweise werden beim Testen der Broker-Konfiguration und der Angriffsmuster die einzelnen Schritte sowie die Ergebnisse der Tests und der Angriffe in Log-Dateien gespeichert. Zu den einzelnen Schritten wird zusätzlich auch ein Zeitstempel gespeichert, sodass diese mit den Zeitstempeln der Überwachungskomponente verglichen werden können.

Beim Fuzzing wird genau dokumentiert, welche Pakete und Felder modifiziert worden sind und wie das Startpaket und das am Ende gesendete Paket aufgebaut sind. Mithilfe dieser Logs kann im Fall eines Broker-Absturzes genau nachvollzogen werden, welches Paket dafür zuständig gewesen ist. In Listing 5.3 ist ein Teilabschnitt des Fuzzing-Logs zu sehen.

```
2021-10-16 at 03:53:11 **********Incoming packet**********
2021-10-16 at 03:53:11 MQTT subscribe: b'\x82\x19\x00\x01\x00\x14test_subscribe_topic\
x00'
2021-10-16 at 03:53:11 Fuzzed type: 8 -> 179
2021-10-16 at 03:53:11 Fuzzed len: 25 -> 35
2021-10-16 at 03:53:11 Fuzzed fields: ['type', 'len']
2021-10-16 at 03:53:11 Original packet
2021-10-16 at 03:53:11 {'type': 8, 'DUP': 0, 'QOS': 1, 'RETAIN': 0, 'len': 25,
'msgid': 1, 'topic': b'test_subscribe_topic', 'length': 20}
2021-10-16 at 03:53:11 Fuzzed packet
2021-10-16 at 03:53:11 {'type': 179, 'DUP': 0, 'QOS': 1, 'RETAIN': 0, 'len': 35,
'msgid': 1,'topic': b'test_subscribe_topic', 'length': 20}
2021-10-16 at 03:53:11 Sending packet bytes: b'2\x19\x00\x01\x00\
x14test_subscribe_topic\x00'
2021-10-16 at 03:53:11 ************************************************************
```

Listing 5.3: Beispielabschnitt des Fuzzing-Logs

Die Log-Dateien zeigen die einkommenden Bytes der Pakete und zu welchem MQTT-Kontrollpakettyp diese gehören. Daraufhin werden die modifizierten Felder und die gefuzzten Werte aufgelistet. Es folgt eine Zusammenfassung des zerlegten Originalpakets und des modifizierten Pakets mit den einzelnen Feldern und den Werten jedes Felds. Am Ende werden die ausgehenden Bytes gespeichert, damit im Falle eines Absturzes die Bytes des Pakets reproduziert werden können.

Die Ergebnisse der einzelnen Tests der Broker-Konfiguration und der ausgeführten Angriffe werden am Ende in einer JSON-Datei, sowie einer PDF-Datei zusammengefasst. Dadurch sind die Berichte sowohl in einem menschen- als auch einem maschinenlesbaren Format vorhanden. Zu jedem Test und Angriff wird der Name, eine kurze Beschreibung und das Ergebnis gespeichert.

5.3 Validierung

Die Validierung des Ansatzes erfolgt durch eine Fallstudie. MQTT-AIO führte Sicherheitstests auf fünf SUTs für jeweils drei Stunden aus. Die MQTT-Broker-Implementierungen wurden auf einem Rechner mit Ubuntu 20.04 installiert und die Tests nacheinander ausgeführt. Die getesteten Implementierungen waren Mosquitto in den Versionen 1.6.7 und 2.0.13, HiveMQ Community Edition 2021.2, KMQTT v0.2.9 und Moquette v0.15. Zu Beginn jeder Ausführung von MQTT-AIO wurde die durchschnittliche und maximale CPU- und Arbeitsspeicher-Auslastung des Systems ermittelt, indem für eine Dauer von 30 Minuten MQTT-Pakete an den Broker geschickt und währenddessen die Ressourcenauslastung gemessen wurde.

Daraufhin wurden sequenziell die weiteren Testmethoden durchgeführt. Zuerst wurde die Konfiguration des Brokers untersucht, dann Angriffe basierend auf den Angriffsmustern ausgeführt und für den Rest der dreistündigen Testdauer wurden Pakete vom Client zum Broker gefuzzt. Die sequenzielle Ausführung ist wichtig, da dadurch eine hohe Ressourcenauslastung mit einem bestimmten Paket oder Angriff assoziiert werden kann. Die Daten während des Testens wurden zu jedem Zeitpunkt in Log-Dateien gespeichert und parallel dazu wurde jede Sekunde die CPU- und Arbeitsspeicherauslastung gemessen. Diese Auslastung wurde konstant mit der am Anfang gemessenen Ressourcenauslastung verglichen und hohe Werte wurden mit einem Zeitstempel in einer weiteren Log-Datei gespeichert. Diese so aufgezeichneten Werte können als Anhaltspunkte für potenzielle Sicherheitsprobleme untersucht werden. Der gesamte Prozess wird ein weiteres Mal wiederholt, um einen größeren Stichprobenumfang für den Fuzzing-Teil zu erhalten und um die erhaltenen Ergebnisse zu validieren. Die jeweiligen Ergebnisse der Sicherheitstests werden in den folgenden Kapiteln beschrieben.

5.3.1 Untersuchung der Standardkonfiguration

MQTT-AIO prüft im ersten Schritt die aktive Konfiguration der MQTT-Broker auf Sicherheitsaspekte. Im Rahmen dieser Fallstudie wird ausschließlich die Sicherheit der jeweiligen Standardkonfigurationen der verschiedenen MQTT-Broker-Implementierungen getestet. MQTT-AIO kann auch mit TLS und Authentifizierung arbeiten, jedoch wird in dieser Arbeit die initiale Konfiguration ohne Änderungen betrachtet, da viele Nutzer keine Konfigurationsänderungen durchführen.

Bei der Untersuchung der Standardkonfiguration der fünf Broker-Implementierungen stellte sich heraus, dass alle fünf keine Verschlüsselung mit TLS nutzen. Anschließend wurde MQTT-AIO ausgeführt und zuerst auf Mosquitto in der Version 1.6.7 angewandt. Diese Version wurde ausgewählt, da es die Version ist, die ohne zusätzliche Installationsschritte vom Ubuntu-Paketmanager bereitgestellt wird. Die Standardkonfiguration von Mosquitto in dieser Version hat einige Schwachstellen. Zum einen ist die Version zu alt und kann noch nicht mit MQTT v5.0 Paketen umgehen. Ein Client kann außerdem sämtliche Topics und auch SYS-Topics im Netzwerk abonnieren. In den SYS-Topics können zudem Informationen über den Broker wie der Name und die Version des Brokers weitergegeben werden. Da Mosquitto eine Open-Source Software ist, kann ein potenzieller Angreifer mit diesen Informationen Angriffe basierend auf Schwachstellen im Quellcode durchführen. Durch das Abonnieren sämtlicher Topics kann ein Angreifer Nachrichten abhören und veröffentlichen, was negative Auswirkungen auf die Vertraulichkeit der Daten hat. Zudem können Angreifer basierend auf diesen Informationen Angriffe starten.

Im Gegensatz dazu kann Mosquitto in der Version 2.0.13 mit MQTT v5.0 Paketen umgehen und hat zudem eine sicherere Standardkonfiguration. Diese erlaubt den Zugriff ausschließlich vom lokalen Rechner und nur mit Authentifizierung. Um die Tests der Fallstudie zu ermöglichen, wurde der Broker so konfiguriert, dass Clients vom gesamten

Netzwerk darauf zugreifen können. In diesem Fall hatten Clients jedoch wieder Zugriff auf sämtliche Topics über „#“ und alle SYS-Topics inklusive des Namens und der Version des Brokers.

Die zurzeit aktuelle Version von Moquette unterstützt keine MQTT v5.0 Pakete. Positiv ist, dass Moquette das Abonnieren von sämtlichen Topics und SYS-Topics unterbindet. Dadurch ist es einem Client nicht möglich, Broker-Informationen zu erhalten. Ähnlich ist es bei KMQTT, bei welchem das Abonnieren aller Topics und SYS-Topics ebenfalls verhindert wird. Im Gegensatz zu Moquette kann dieser Broker jedoch mit MQTT v5.0 Paketen umgehen. Eine weitere interessante Standardkonfiguration weist HiveMQ auf, da diese das Abonnieren von SYS-Topics unterbindet, jedoch das Abonnieren aller Topics mit „#“ erlaubt. Dadurch kann der Client zwar keine Informationen über den Broker erhalten, jedoch hat dieser Zugriff auf sämtliche Nachrichten im Netzwerk.

5.3.2 Angriffsmuster

Im zweiten Schritt der Fallstudie wurden die implementierten Angriffe basierend auf den Angriffsmustern aus Abschnitt 5.2.3 auf den fünf Broker-Implementierungen ausgeführt und die Ergebnisse mit dem erwarteten Wert verglichen. Die Ergebnisse der Tests mit Angriffsmustern sind in Tabelle 5.2 zu sehen. Dabei heißt 0, dass das Ergebnis dem erwarteten Ergebnis entspricht, 1 bedeutet, das Ergebnis entspricht nicht dem erwarteten Ergebnis.

Bei den beiden Mosquitto-Versionen gab es mit Angriffsmuster 17 nur einen einzelnen Angriff, der ein unterschiedliches Ergebnis geliefert hat. Wenn ein CONNACK-Paket gesendet wurde, hat der Broker in Version 2.0.13 die Verbindung unterbrochen, während der Client bei Version 1.6.7 mit dem Broker verbunden blieb. Der Grund für diese Änderung ist die Schwachstelle CVE-2021-28166[13], die in Mosquitto zwischen 2.0.0-2.0.9 vorhanden war. Diese Schwachstelle führt zu einer Null-Pointer-Dereferenzierung, falls ein Client über MQTT v5.0 mit dem Broker verbunden ist und ein CONNACK-Paket an den Broker schickt.

Beide Mosquitto-Versionen akzeptierten PUBLISH-Pakete mit riesigen Payloads, was laut Angriffsmuster 4 nicht der Fall sein sollte. Außerdem hat Mosquitto Probleme mit Maskierungszeichen. SUBSCRIBE-Pakete mit Maskierungszeichen im Topic-Namen und PUBLISH-Pakete mit Maskierungszeichen im Payload wurden nicht akzeptiert, obwohl dies laut Angriffsmuster 7 und 8 der Fall sein sollte. Der Broker akzeptierte zudem keine CONNECT-Pakete mit langer Client-ID (Nr. 11), obwohl dies vom Broker akzeptiert werden sollte. Bei Angriffsmuster 15 wurde ein CONNECT-Paket mit aktiviertem Nutzernamen- und Passwort-Flag, aber ohne Authentifizierungsdaten versendet. Mosquitto akzeptierte dieses Paket, obwohl laut MQTT-3.1.2-19 und MQTT-3.1.2-21 die Authentifizierungsdaten vorhanden sein müssen.

[13] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-28166

Test Nr.	Mosquitto 1.6.7	Mosquitto 2.0.13	HiveMQ 2021.2	Moquette 0.15	KMQTT 0.2.9
1	0	0	0	0	0
2	0	0	0	0	0
3	0	0	0	0	0
4	1	1	1	0	0
5	0	0	0	0	1
6	0	0	0	1	1
7	1	1	0	0	0
8	1	1	1	0	0
9	0	0	0	0	0
10	0	0	0	0	0
11	1	1	0	1	0
12	0	0	0	0	0
13	0	0	0	0	0
14	0	0	0	1	0
15	1	1	1	0	1
16	0	0	0	1	0
17	1	0	0	1	0
18	0	0	0	0	1

0 = erwartetes Ergebnis, 1 = unerwartetes Ergebnis

Tabelle 5.2: Ergebnisse der Tests mit Angriffsmustern

Bei HiveMQ führten die Angriffe zu ähnlichen Ergebnissen. Das Positive war, dass mehr Angriffe das erwartete Ergebnis zurückgegeben haben als es bei Mosquitto der Fall war und es lediglich drei Angriffe gab, die einen unerwarteten Rückgabewert lieferten. Dies waren zudem drei Angriffe, die auch bei beiden Mosquitto-Versionen den Wert 1 zurückgegeben haben. HiveMQ akzeptierte riesige PUBLISH-Pakete mit einer Payload-Größe von 128 MB (Nr. 4) und auch CONNECT-Pakete mit aktivierten Nutzernamen/Passwort-Flag, aber mit leeren Authentifizierungsfeldern (Nr. 15). Im Gegensatz zu Mosquitto konnte HiveMQ mit SUBSCRIBE-Paketen mit Maskierungszeichen im Topic Namen umgehen (Nr. 7), jedoch nicht mit PUBLISH-Paketen, die ein Maskierungszeichen im Payload haben (Nr. 8).

Bei Moquette führten die Angriffe zu wesentlich anderen Ergebnissen als bei den zuvor beschriebenen MQTT-Brokern. Ähnlich wie Mosquitto konnte Moquette nicht mit CONNECT-Paketen umgehen, die eine lange Client-ID besitzen (Nr. 11). Moquettes Verhalten bei CONNACK-Paketen war ähnlich wie bei Mosquitto in der Version 1.6.7, der Broker lehnte das Paket nicht ab und unterbrach die Verbindung nicht (Nr. 17). Anders als bei allen bisher beschriebenen Brokern akzeptierte Moquette SUBSCRIBE-Pakete

mit Wildcard-Symbolen im Topic-Namen (Nr. 6). MQTT-3.3.2-2 besagt jedoch, dass der Topic-Name in PUBLISH-Paketen keine Wildcard-Symbole erhalten darf. Ein ähnliches Problem trat bei PUBLISH-Paketen mit einem NULL-Charakter im Topic-Name auf, was gegen MQTT-4.7.3-2 verstößt. Moquette akzeptierte diese Pakete und brach die Verbindung zum Client nicht ab (Nr. 16). Außerdem verstößt Moquette gegen MQTT-3.1.2-13, was besagt, dass das Will-QoS-Flag auf 0 gesetzt werden muss, wenn das Will-Flag 0 ist. Bei Angriffsmuster 14 wurde bei einem CONNECT-Paket der Will-Flag auf 0 und der Will-QoS-Flag auf 2 gesetzt. Das Paket wurde nicht abgelehnt und die Verbindung nicht unterbrochen.

Beim MQTT-Broker KMQTT gab es zwei Angriffe, die bei jedem anderen Broker das erwartete Ergebnis zurückgegeben haben, nicht aber bei KMQTT. Anders als die anderen Broker akzeptierte KMQTT SUBSCRIBE-Pakete ohne Payload (Nr. 5), was jedoch laut MQTT-3.8.3-3 eine Protokollverletzung ist. Außerdem akzeptierte KMQTT PUBLISH-Pakete mit einer Topic-Länge von 0 (Nr. 18), was in der Vergangenheit bei Mosquitto zu einem Absturz geführt hat und somit auch bei KMQTT eine Gefahr darstellen kann. Ähnlich wie Moquette akzeptierte KMQTT SUBSCRIBE-Pakete mit Wildcard-Zeichen im Topic-Namen (Nr. 6), was laut der MQTT-Spezifikation untersagt ist. Ein weiteres unerwartetes Ergebnis lieferte das Angriffsmuster 15, bei dem ein CONNECT-Paket mit aktiviertem Nutzername- und Passwort-Flag, aber ohne Authentifizierungsdaten geschickt wurde. Genau wie Mosquitto und HiveMQ akzeptierte KMQTT dieses Paket fälschlicherweise.

Auffällig hierbei ist insbesondere, dass es keinen Angriff gab, der bei allen fünf Brokern ein nicht erwartetes Ergebnis geliefert hat. Jedoch haben auch nur bei sieben von 18 Angriffen alle fünf Broker das erwartete Ergebnis zurückgegeben. Diese Ergebnisse zeigen, dass immer noch Unstimmigkeiten bei den verschiedenen Broker-Implementierungen vorhanden sind. Obwohl einige Regeln klar in der MQTT-Spezifikation definiert sind, halten sich immer noch nicht alle MQTT-Broker-Implementierungen daran. Einige der definierten Regeln sind sicherheitsrelevant und das Nichteinhalten dieser Regeln kann zu Problemen und im schlimmsten Fall zu Broker-Abstürzen führen. Die Angriffsmuster in MQTT-AIO können einfach erweitert werden, sobald neue Schwachstellen bei einem MQTT-Broker entdeckt und beispielsweise auf CVE hochgeladen werden, sodass das System immer aktuell bleibt.

5.3.3 Fuzzing

Im letzten Schritt der Fallstudie wurden die Pakete in Richtung der MQTT-Broker gefuzzt. MQTT-AIO enthält eine MQTT-Client-Implementierung, die Pakete an den Proxy-Server sendet. Der Proxy-Server empfängt die Pakete, modifiziert diese und leitet sie an den SUT weiter. In der Fallstudie sendete der Client immer die gleiche Sequenz an Paketen in folgender Reihenfolge: CONNECT, SUBSCRIBE, PUBLISH, UNSUBSCRIBE. Die Wahrscheinlichkeiten, dass die einzelnen Pakete gefuzzt werden, wurden in der config.json-Datei definiert und sind in Listing 5.4 dargestellt.

```
"MQTTConnect": 25,
"MQTTSubscribe": 35,
"MQTTPublish": 35,
"MQTTUnsubscribe": 50
```

Listing 5.4: Fuzzing-Wahrscheinlichkeiten für die Fallstudie

Die CONNECT-Pakete wurden bspw. mit einer Wahrscheinlichkeit von 25% gefuzzt. Dabei war zu beachten, dass die Fuzzing-Wahrscheinlichkeiten innerhalb der Sequenz ansteigen, sodass genügend valide Pakete in den jeweils nächsten Schritt gelangen. Wenn sämtliche CONNECT-Pakete abgefangen und zu einem fehlerhaften Paket modifiziert werden würden, würde die Sequenz nicht fortgesetzt werden und es würden nur CONNECT-Pakete getestet werden.

Wenn ein Paket zum Fuzzen ausgewählt wurde, mussten noch die zu fuzzenden Felder des Pakets festgelegt werden. Die Wahrscheinlichkeiten dafür wurden im Rahmen der Fallstudie je nach Feld auf Werte zwischen 10% und 30% gesetzt. Wenn das Feld einen String enthielt, wurde Radamsa zum Fuzzen verwendet. Bei Feldern mit Zahlen wurde eine der definierten Generierungs- oder Mutationsfunktionen zufällig ausgewählt und angewendet.

Während des gesamten Fuzzing-Prozesses wurde in regelmäßigen Abständen getestet, ob der Broker noch aktiv war. Wenn der Broker abgestürzt wäre, wäre MQTT-AIO beendet worden und es könnte in den Log-Dateien geschaut werden, was die Ursache des Absturzes gewesen ist. Es ist jedoch innerhalb der limitierten Testzeit kein MQTT-Broker abgestürzt. Dies war aber zu erwarten, da Black-Box-Fuzzing oft sehr viel Zeit und Expertenwissen benötigt. Ein Experte in Netzwerk- und MQTT-Sicherheit verfügt über mehr Erfahrung in Hinblick darauf, welche Pakete und Felder mit höherer Wahrscheinlichkeit zu Abstürzen führen können. Die Fuzzing-Konfigurationen können somit präziser angepasst werden und es besteht eine höhere Chance, dass Schwachstellen gefunden werden.

Zusätzlich wurde in 1-Sekunden-Abständen die CPU- und Arbeitsspeicher-Auslastung gemessen und mit der durchschnittlichen und maximalen Auslastung, die zu Beginn der Fallstudie gemessen wurde, verglichen. Wenn die aktuelle Auslastung bei dem Dreifachen der durchschnittlichen oder dem Eineinhalbfachen der maximalen Auslastung lag, wurde der Zeitpunkt und die Auslastung in einer Log-Datei gespeichert. Diese plötzlichen Spikes könnten auf ein Problem des SUTs beim Umgang mit bestimmten Paketen hindeuten.

Keine der Broker-Implementierungen hat plötzliche starke Anstiege bei der Auslastung des Arbeitsspeichers ausgelöst, jedoch gab es viele Spikes bei der CPU-Auslastung. Diese Spikes können im Nachhinein mit den Fuzzing-Logs verglichen und die Ursachen für den plötzlichen Anstieg analysiert werden.

Die Ergebnisse der einzelnen CPU-Auslastungen beider Testläufe für die fünf getesteten Broker-Implementierung sind in Abbildung 5.2 dargestellt. Tabelle 5.3 enthält für jede Implementierung die Gesamtzahl der CPU-Spikes beider Testläufe, die höchste beim Fuzzen gemessene CPU-Auslastung sowie die Durchschnitts-CPU-Auslastung, die zu Beginn des gesamten Testprozesses gemessen wurde. Diese Metriken sind jedoch nur

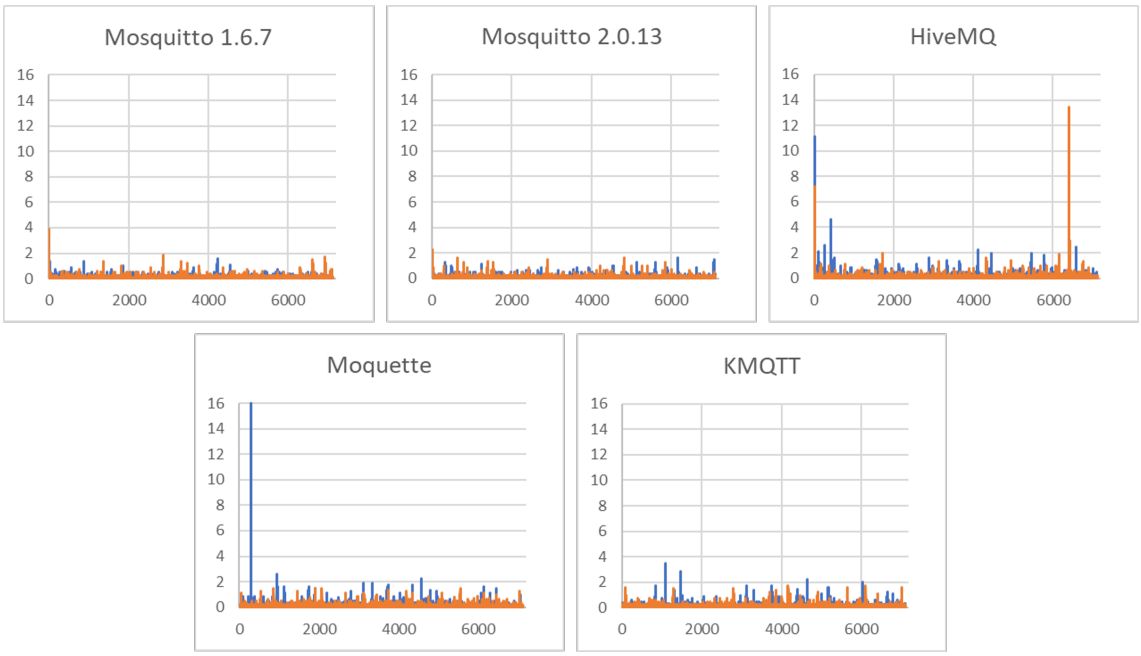


Abbildung 5.2: CPU-Auslastungen der einzelnen Broker-Implementierungen

CPU-Auslastungs-Metriken	Mosquitto 1.6.7	Mosquitto 2.0.13	HiveMQ 2021.2	Moquette v0.15	KMQTT v0.2.9
Anzahl CPU-Spikes	655	903	613	742	609
Höchstwert während Fuzzing	3,9	2,27	13,42	41,18	3,5
Durchschnitt ohne Fuzzing	0,08	0,075	0,14	0,15	0,1

Tabelle 5.3: Metriken der CPU-Auslastung

bedingt aussagekräftig, da es beim Fuzzing stark darauf ankommt, welche Pakete und welche Felder gefuzzt werden. Die meisten CPU-Spikes hatte Mosquitto in der neuesten Version mit 903 Spikes, jedoch hatte es mit 2,27% die niedrigste maximale CPU-Auslastung von allen getesteten Broker-Implementierungen. Dieser Maximalwert ist jedoch trotzdem um den Faktor 30 höher als die gemessene durchschnittliche CPU-Auslastung von Mosquitto in dieser Version. Moquette hatte mit 41,18% die höchste maximale CPU-Auslastung aller betrachteten Broker-Implementierungen. Der hohe Maximalwert könnte andeuten, dass der Broker Probleme mit einem fehlerhaften Paket oder einer Paketsequenz gehabt haben könnte.

Einen Überblick zum Vergleichen der CPU-Auslastungen der verschiedenen Implementierungen ist in Abbildung 5.3 dargestellt. Es wurden die Ergebnisse eines Durchlaufs von MQTT-AIO auf jeder Implementierung in einer Abbildung zusammengetragen. Es ist zu sehen, dass CPU-Spikes bei allen Brokern zu beobachten sind, jedoch sind diese in diesem

Testlauf bei Moquitto 2.0.13 durchgehend auf vergleichsweise niedrigem Niveau. Die gesendeten Pakete bzw. Paketsequenzen zu diesen Zeitpunkten können in den Fuzzing-Logs untersucht werden, was jedoch im Rahmen dieser Fallstudie nicht weiter betrachtet wird. Hier wird lediglich geprüft, ob mithilfe von MQTT-AIO potenzielle Sicherheitsprobleme beim Fuzzing erkennbar sind. Dies kann bestätigt werden, da bei jedem Broker CPU-Spikes zu beobachten sind, die ohne Fuzzing nicht aufgetreten sind. Diese Spikes können zu Problemen führen, wenn bspw. das Gerät, auf dem die MQTT-Broker-Implementierung installiert ist, weniger Ressourcen zur Verfügung hat. Das Gerät kann dann keine CPU-Ressourcen mehr für andere Anwendungen freigeben.

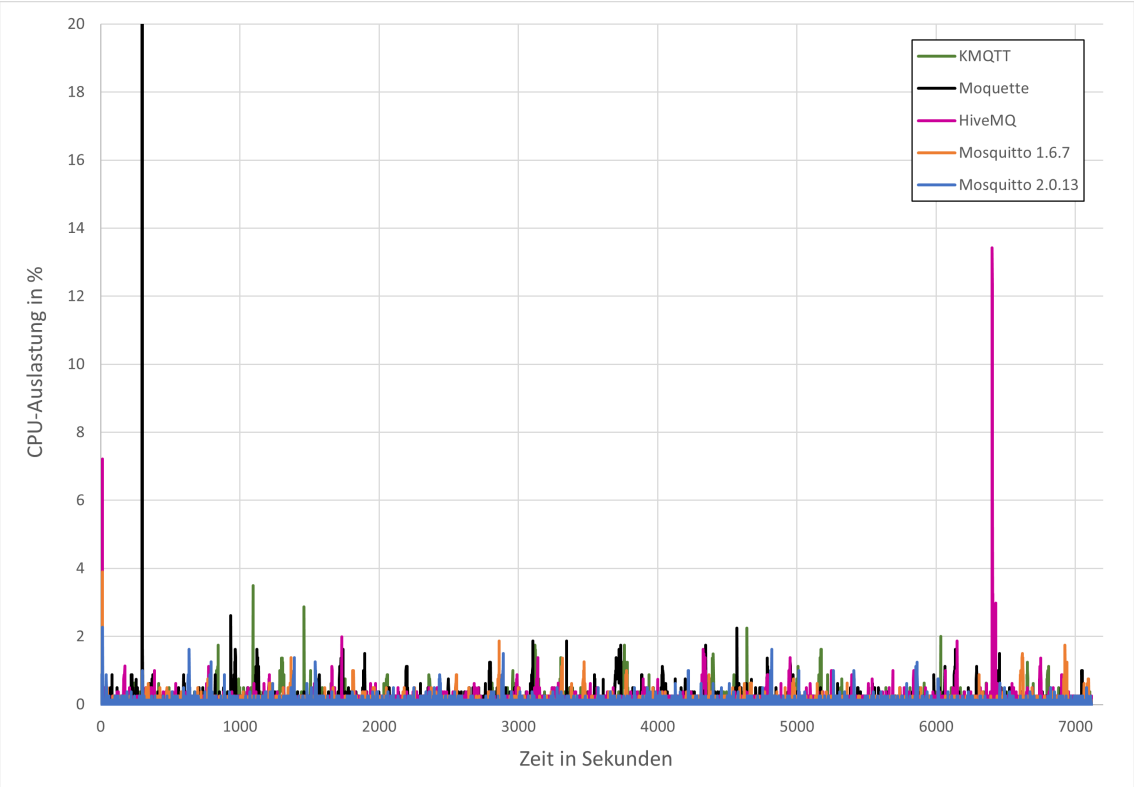


Abbildung 5.3: CPU-Auslastungen aller Broker-Implementierungen

Analyse der Hypothese

H - Bei aktuellen MQTT-Implementierungen bestehen immer noch Sicherheitsprobleme.

Zu Beginn der Arbeit wurde diese Hypothese aufgestellt. Durch die vorgenommenen Literaturrecherchen, sowie die Ergebnisse der beiden Forschungsfragen und die durchgeführte Fallstudie kann die Hypothese mit „Ja“ beantwortet werden.

Durch den entworfenen Testansatz MQTT-AIO und die mit dessen Implementierung durchgeführte Fallstudie wurden einige der in der Literaturrecherche gefundenen Sicherheitsprobleme in konkreten MQTT-Broker-Implementierungen aufgedeckt. Die Standard-

konfigurationen aller getesteten MQTT-Broker-Implementierungen weisen immer noch Sicherheitsprobleme auf. Zudem wurde durch die Tests mithilfe von Angriffsmustern bestätigt, dass die getesteten Broker die MQTT-Spezifikation nicht vollständig korrekt implementieren. Explizit definierte Anforderungen in der Spezifikation wurden nicht immer eingehalten. Bei den Fuzzing-Tests waren bei den Brokern erhöhte CPU-Auslastungen zu beobachten, was andeuten kann, dass einige der gefuzzten Pakete zu einer erhöhten Ressourcenauslastung geführt haben und dadurch potenziell einen DoS-Zustand auslösen können.

6 Zusammenfassung und Ausblick

In dieser Masterarbeit wurde mithilfe einer Literaturrecherche der aktuelle Stand der Forschung bezüglich der Sicherheit von MQTT-Systemen untersucht. Die Recherche zeigte, dass unsichere Standardkonfigurationen von MQTT-Broker-Implementierungen Sicherheitsprobleme aufweisen. Der MQTT-Broker ist die wichtigste Komponente in einem MQTT-Netzwerk, ohne den keine Kommunikation unter den Geräten möglich ist. Zahlreiche Nutzer verändern die voreingestellte Konfiguration beim Broker nicht, sodass keine Authentifizierung, Autorisierung und Verschlüsselung verwendet wird. Diese unsichere Konfiguration erlaubt Angreifern, auf einfache Weise Zugriff auf die versendeten Daten zu erhalten. Ein weiteres identifiziertes Problem ist, dass 71,15% aller in der Vergangenheit gefundenen Schwachstellen durch fehlerhafte MQTT-Kontrollpakete ausgelöst wurden. Solche fehlerhaften Pakete können zur Offenlegung von Informationen, zur Ausführung von schädlichem Code oder zu einem DoS-Zustand führen.

Um die Sicherheit von MQTT-Broker-Implementierungen zu untersuchen, wurde im Rahmen der vorliegenden Masterarbeit der Testansatz MQTT-AIO entworfen und implementiert. MQTT-AIO besteht aus drei Komponenten zum Testen der Sicherheit von MQTT-Broker-Implementierungen, sowie je einer Komponente für die Überwachung des SUTs und die Generierung von Testberichten. Die Testkomponenten wurden so entworfen, dass sie verschiedene Aspekte der Sicherheit abdecken. Die erste Testkomponente testet die Konfiguration des Brokers und untersucht unter anderem, ob sich ein Client ohne Verschlüsselung oder Authentifizierung mit dem Broker verbinden kann. Die zweite Komponente nutzt Angriffsmuster, um explizit ausgewählte Sicherheitsaspekte, die zum Beispiel aus der MQTT-Spezifikation oder aus in der Vergangenheit gefundenen Schwachstellen definiert wurden, zu testen. Die aus den Angriffsmustern abgeleiteten Angriffe werden an der Broker-Implementierung ausgeführt und die daraus resultierenden Ergebnisse mit den erwarteten Ergebnissen verglichen. Die dritte Testkomponente ist ein Protokoll-Fuzzer und testet, wie ein MQTT-Broker mit fehlerhaften Paketen umgeht. Bei diesen Tests wird nach Broker-Abstürzen und stark erhöhten Ressourcenauslastungen gesucht. Durch die Nutzung der drei verschiedenen Testkomponenten werden verschiedene Aspekte der Sicherheit von MQTT-Systemen untersucht und damit ein umfassendes Bild geschaffen.

Das Konzept MQTT-AIO wurde in dieser Arbeit validiert, indem in einer Fallstudie fünf MQTT-Broker-Implementierungen zweimal für jeweils drei Stunden getestet wurden. Hierbei zeigte sich, dass sämtliche getesteten Broker standardmäßig keine Verschlüsselung verwenden. Es wurden zudem Diskrepanzen zwischen den verschiedenen Brokern entdeckt. So erlaubt bspw. Mosquitto in beiden getesteten Versionen das Abonnieren von SYS-Topics, wodurch Brokername und -version herausgelesen werden können. Moquette, HiveMQ und

KMQTT erlaubten dies nicht. Die Tests mit Angriffsmustern zeigten, dass die Broker einige explizit in der MQTT-Spezifikation definierten Regeln nicht korrekt implementiert haben. Beim Fuzzing sind in der limitierten Testzeit keine Brokerabstürze verzeichnet worden, jedoch traten bei allen fünf getesteten Brokern CPU-Spikes auf, was darauf hindeuten kann, dass die Broker potenziell Probleme mit den gefuzzten Paketen hatten. Diese Spikes können in einem späteren Schritt ausführlich analysiert und so nach deren Ursache gesucht werden.

Ausblick

Durch die verschiedenen Testkomponenten von MQTT-AIO ist bereits ein umfassendes Sicherheitstesten von MQTT-Broker-Implementierungen möglich. Für die zukünftige Nutzung ist es wichtig, dass neu entdeckte Sicherheitsschwachstellen einfach in die Angriffsmuster-Komponente hinzugefügt werden können. Da bei der Implementierung des Konzepts auf Erweiterbarkeit geachtet wurde, können zukünftig weitere Tests unkompliziert integriert werden. Hierfür müssen lediglich die Funktionen in einem bestimmten Format definiert werden und der Name der neuen Funktion in die config.json-Datei geschrieben werden.

Für die Fuzzing-Komponente können weitere Generierungs- oder Mutationsfunktionen definiert werden, die explizit auf bestimmte Felder der verschiedenen MQTT-Pakete ausgerichtet sind. Dadurch kann die Effizienz und Effektivität des Fuzzings gesteigert werden. Eine andere interessante Erweiterungsmöglichkeit ist das Anwenden von Fuzzing in Client-Richtung. Dadurch können Client-Implementierungen ähnlich wie die Broker-Implementierungen auf ihre Sicherheit getestet werden. Um die Abdeckung des Konzepts weiter zu erhöhen, kann zudem untersucht werden, ob weitere Sicherheitsprobleme auftreten, wenn verschiedene Angriffe kombiniert werden.

Diese Erweiterungsmöglichkeiten können durch zukünftige Forschung weiter untersucht und umgesetzt werden, sodass die zahlreichen Sicherheitsherausforderungen des MQTT-Protokolls nach und nach angegangen werden können.

Literaturverzeichnis

[AAAC16] S. A. Al-Qaseemi, H. A. Almulhim, M. F. Almulhim, S. R. Chaudhry. „IoT architecture challenges and issues: Lack of standardization“. In: *2016 Future technologies conference (FTC)*. IEEE. 2016, S. 731–738 (zitiert auf S. 18).

[ACH15] I. Andrea, C. Chrysostomou, G. Hadjichristofi. „Internet of Things: Security vulnerabilities and challenges“. In: *2015 IEEE symposium on computers and communication (ISCC)*. IEEE. 2015, S. 180–187 (zitiert auf S. 15, 18).

[AGM+15] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, M. Ayyash. „Internet of things: A survey on enabling technologies, protocols, and applications“. In: *IEEE communications surveys & tutorials* 17.4 (2015), S. 2347–2376 (zitiert auf S. 17).

[Ahm18] A. Ahmad. „Model-Based Testing for IoT Systems: Methods and tools“. Diss. Bourgogne Franche-Comté, 2018 (zitiert auf S. 25, 26, 46).

[AK19] J. J. Anthraper, J. Kotak. „Security, privacy and forensic concern of MQTT protocol“. In: *Proceedings of International Conference on Sustainable Computing in Science, Technology and Management (SUSCOM), Amity University Rajasthan, Jaipur-India*. 2019 (zitiert auf S. 34).

[AM20] L. G. Araujo Rodriguez, D. Macêdo Batista. „Program-aware fuzzing for MQTT applications“. In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2020, S. 582–586 (zitiert auf S. 37, 42, 58).

[AO16] P. Ammann, J. Offutt. *Introduction to software testing*. Cambridge University Press, 2016 (zitiert auf S. 22).

[ARH17] S. Andy, B. Rahardjo, B. Hanindhito. „Attack scenarios and security analysis of MQTT communication protocol in IoT system“. In: *2017 4th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI)*. IEEE. 2017, S. 1–6 (zitiert auf S. 35, 37).

[ATP21] Cybersecurity & Infrastructure Security Agency. *Attack Patterns*. 2021. URL: https://us-cert.cisa.gov/bsi/articles/knowledge/attack-patterns (zitiert auf S. 50).

[AZ21] S. Akhtar, E. Zahoor. „Formal Specification and Verification of MQTT Protocol in PlusCal-2“. In: *Wireless Personal Communications* (2021), S. 1–18 (zitiert auf S. 46).

[CAB+20] G. Casteur, A. Aubaret, B. Blondeau, V. Clouet, A. Quemat, V. Pical, R. Zitouni. „Fuzzing attacks for vulnerability discovery within MQTT protocol“. In: *2020 International Wireless Communications and Mobile Computing (IWCMC)*. IEEE. 2020, S. 420–425 (zitiert auf S. 42).

[CCM+18] C. Chen, B. Cui, J. Ma, R. Wu, J. Guo, W. Liu. „A systematic review of fuzzing techniques“. In: *Computers & Security* 75 (2018), S. 118–137 (zitiert auf S. 24).

[CPCA13] E. Cambiaso, G. Papaleo, G. Chiola, M. Aiello. „Slow DoS attacks: definition and categorisation“. In: *International Journal of Trust Management in Computing and Communications* 1.3-4 (2013), S. 300–319 (zitiert auf S. 39).

[CPVV18] M. Calabretta, R. Pecori, M. Vecchio, L. Veltri. „MQTT-Auth: A token-based solution to endow MQTT with authentication and authorization capabilities“. In: *Journal of Communications Software and Systems* 14.4 (2018), S. 320–331 (zitiert auf S. 30).

[CZLS18] C.-K. Chen, Z.-K. Zhang, S.-H. Lee, S. Shieh. „Penetration testing in the iot age“. In: *computer* 51.4 (2018), S. 82–85 (zitiert auf S. 23).

[DR19] M. Dabbagh, A. Rayes. „Internet of things security and privacy“. In: *Internet of Things from hype to reality*. Springer, 2019, S. 211–238 (zitiert auf S. 19).

[Duk15] C. Dukes. „Committee on national security systems (CNSS) glossary“. In: *CNSSI, Fort Meade, MD, USA, Technical Report* 4009 (2015) (zitiert auf S. 21).

[EDS20] Statista. *IoT Developer Survey Key Findings*. 2020. URL: https://outreach.eclipse.foundation/eclipse-iot-developer-survey-2020 (zitiert auf S. 15, 33).

[EFI21] M. Eceiza, J. L. Flores, M. Iturbe. „Fuzzing the Internet of Things: A Review on the Techniques and Challenges for Efficient Vulnerability Discovery in Embedded Systems“. In: *IEEE Internet of Things Journal* (2021) (zitiert auf S. 24).

[ENI17] E. ENISA. *Baseline Security Recommendations for IoT in the context of Critical Information Infrastructures*. 2017 (zitiert auf S. 19).

[FBJ+16] M. Felderer, M. Büchler, M. Johns, A. D. Brucker, R. Breu, A. Pretschner. „Security testing: A survey“. In: *Advances in Computers*. Bd. 101. Elsevier, 2016, S. 1–51 (zitiert auf S. 21–23).

[FBVI17] S. N. Firdous, Z. Baig, C. Valli, A. Ibrahim. „Modelling and evaluation of malicious attacks against the iot mqtt protocol“. In: *2017 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. IEEE. 2017, S. 748–755 (zitiert auf S. 35, 36, 38).

[Fre01] P. Freeman. „Software Engineering Body of Knowledge (SWEBOK)“. In: *INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING*. Bd. 23. 2001, S. 693–696 (zitiert auf S. 22).

[FZB+16] M. Felderer, P. Zech, R. Breu, M. Büchler, A. Pretschner. „Model-based security testing: a taxonomy and systematic classification“. In: *Software Testing, Verification and Reliability* 26.2 (2016), S. 119–148 (zitiert auf S. 24, 25).

[GFT21] E. Gamess, T. N. Ford, M. Trifas. „Performance evaluation of a widely used implementation of the MQTT protocol with large payloads in normal operation and under a DoS attack“. In: *Proceedings of the 2021 ACM Southeast Conference*. 2021, S. 154–162 (zitiert auf S. 31).

[HBK18] M. Harsha, B. Bhavani, K. Kundhavai. „Analysis of vulnerabilities in MQTT security using Shodan API and implementation of its countermeasures via authentication and ACLs“. In: *2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. IEEE. 2018, S. 2244–2250 (zitiert auf S. 43).

[HKH19] S. Hameed, F. I. Khan, B. Hameed. „Understanding security requirements and challenges in Internet of Things (IoT): A review“. In: *Journal of Computer Networks and Communications* 2019 (2019) (zitiert auf S. 15).

[HPG+18] M. Hogan, B. Piccarreta, I. I. C. S. W. Group et al. *Interagency report on status of international cybersecurity standardization for the Internet of Things (IoT)*. Techn. Ber. National Institute of Standards und Technology, 2018 (zitiert auf S. 19).

[HVL18] S. Hernández Ramos, M. T. Villalba, R. Lacuesta. „Mqtt security: A novel fuzzing approach“. In: *Wireless Communications and Mobile Computing* 2018 (2018) (zitiert auf S. 34, 41, 51).

[JBH+20] M. A. J. Jamali, B. Bahrami, A. Heidari, P. Allahverdizadeh, F. Norouzi. „IoT architecture“. In: *Towards the Internet of Things* (2020), S. 9–31 (zitiert auf S. 18).

[JKTG20] R. Johari, I. Kaur, R. Tripathi, K. Gupta. „Penetration Testing in IoT Network“. In: *2020 5th International Conference on Computing, Communication and Security (ICCCS)*. IEEE. 2020, S. 1–7 (zitiert auf S. 23).

[KA16] J. King, A. I. Awad. „A distributed security mechanism for resource-constrained IoT devices“. In: *Informatica* 40.1 (2016) (zitiert auf S. 34).

[KK+12] M. E. Khan, F. Khan et al. „A comparative study of white box, black box and grey box testing techniques“. In: *Int. J. Adv. Comput. Sci. Appl* 3.6 (2012) (zitiert auf S. 22).

[KM18] N. M. Kumar, P. K. Mallick. „The Internet of Things: Insights into the building blocks, component interactions, and architecture layers“. In: *Procedia computer science* 132 (2018), S. 109–117 (zitiert auf S. 18).

[KSR19] J. Kotak, A. Shah, P. Rajdev. „A comparative analysis on security of MQTT brokers“. In: (2019) (zitiert auf S. 36, 38).

[LHH+15] C. Lesjak, D. Hein, M. Hofmann, M. Maritsch, A. Aldrian, P. Priller, T. Ebner, T. Ruprechter, G. Pregartner. „Securing smart maintenance services: Hardware-security and TLS for MQTT“. In: *2015 IEEE 13th international conference on industrial informatics (INDIN)*. IEEE. 2015, S. 1243–1250 (zitiert auf S. 35).

[LPJ+18] H. Liang, X. Pei, X. Jia, W. Shen, J. Zhang. „Fuzzing: State of the art“. In: *IEEE Transactions on Reliability* 67.3 (2018), S. 1199–1218 (zitiert auf S. 24).

[LSCL18] J.-Z. Luo, C. Shan, J. Cai, Y. Liu. „IoT Application-Layer Protocol Vulnerability Detection using Reverse Engineering“. In: *Symmetry* 10.11 (2018), S. 561 (zitiert auf S. 41).

[Map17] C. Maple. „Security and privacy in the internet of things“. In: *Journal of Cyber Policy* 2.2 (2017), S. 155–184 (zitiert auf S. 19).

[MG19] S. P. Mathews, R. R. Gondkar. „Protocol recommendation for message encryption in MQTT“. In: *2019 International Conference on Data Science and Communication (IconDSC)*. IEEE. 2019, S. 1–5 (zitiert auf S. 43).

[MHSB20] S. N. Matheu, J. L. Hernández-Ramos, A. F. Skarmeta, G. Baldini. „A survey of cybersecurity certification for the Internet of Things“. In: *ACM Computing Surveys (CSUR)* 53.6 (2020), S. 1–36 (zitiert auf S. 26).

[MVH+21] A. Mileva, A. Velinov, L. Hartmann, S. Wendzel, W. Mazurczyk. „Comprehensive analysis of MQTT 5.0 susceptibility to network covert channels“. In: *computers & security* 104 (2021), S. 102207 (zitiert auf S. 31).

[MVMC17] K. Mladenov, S. Van Winsen, C. Mavrakis, K. Cyber. „Formal verification of the implementation of the MQTT protocol in IoT devices“. In: *SNE Master Research Projects 2016–2017* (2017) (zitiert auf S. 46).

[NC20] G. Nebbione, M. C. Calzarossa. „Security of IoT application layer protocols: Challenges and findings“. In: *Future Internet* 12.3 (2020), S. 55 (zitiert auf S. 36).

[NVD19] NATIONAL VULNERABILITY DATABASE. *CVE-2019-11779*. 2019. URL: https://nvd.nist.gov/vuln/detail/CVE-2019-11779 (zitiert auf S. 36).

[OWASP18] The Apache Software Foundation. *OWASP Internet of Things*. 2018. URL: https://owasp.org/www-project-internet-of-things/ (zitiert auf S. 19, 21).

[PPW+05] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, T. Stauner. „One evaluation of model-based testing and its automation“. In: *Proceedings of the 27th international conference on Software engineering*. 2005, S. 392–401 (zitiert auf S. 25).

[PVPG+17] G. Perrone, M. Vecchio, R. Pecori, R. Giaffreda et al. „The Day After Mirai: A Survey on MQTT Security Solutions After the Largest Cyber-attack Carried Out through an Army of IoT Devices.“ In: *IoTBDS*. 2017, S. 246–253 (zitiert auf S. 34).

[Ram] T. Ramsauer. „Black-Box Live Protocol Fuzzing“. In: *Target* 2 (), S. 1–2 (zitiert auf S. 51).

[RN05] V. Ramachandran, S. Nandi. „Detecting ARP spoofing: An active technique“. In: *International conference on information systems security*. Springer. 2005, S. 239–250 (zitiert auf S. 41).

[SFR20a] H. Sochor, F. Ferrarotti, R. Ramler. „An Architecture for Automated Security Test Case Generation for MQTT Systems“. In: *International Conference on Database and Expert Systems Applications*. Springer. 2020, S. 48–62 (zitiert auf S. 44, 51).

[SFR20b] H. Sochor, F. Ferrarotti, R. Ramler. „Automated security test generation for MQTT using attack patterns“. In: *Proceedings of the 15th International Conference on Availability, Reliability and Security*. 2020, S. 1–9 (zitiert auf S. 34, 43, 44, 51, 56, 57).

[SGS12] I. Schieferdecker, J. Grossmann, M. Schneider. „Model-based security testing“. In: *arXiv preprint arXiv:1202.6118* (2012) (zitiert auf S. 25).

[SHO21] Shodan. *Shodan Search Engine*. 2021. URL: https://www.shodan.io/ (zitiert auf S. 37).

[SHS15] Y. Sheffer, R. Holz, P. Saint-Andre. „Summarizing known attacks on transport layer security (TLS) and datagram TLS (DTLS)“. In: *Internet Engineering Task Force Request for Comments* 7457 (2015) (zitiert auf S. 36).

[SJK17] S. N. Swamy, D. Jadhav, N. Kulkarni. „Security threats in the application layer in IOT applications“. In: *2017 International conference on i-SMAC (iot in social, mobile, analytics and cloud)(i-SMAC)*. IEEE. 2017, S. 477–480 (zitiert auf S. 35).

[SKRW17] I. Schieferdecker, S. Kretzschmann, A. Rennoch, M. Wagner. „IoT-testware-an eclipse project“. In: *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE. 2017, S. 1–8 (zitiert auf S. 46).

[SNF+17] M. G. Samaila, M. Neto, D. A. Fernandes, M. M. Freire, P. R. Inácio. „Security challenges of the Internet of Things“. In: *Beyond the Internet of Things*. Springer, 2017, S. 53–82 (zitiert auf S. 18).

[SNL19] O. Sadio, I. Ngom, C. Lishou. „Lightweight security scheme for mqtt/mqtt-sn protocol“. In: *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*. IEEE. 2019, S. 119–123 (zitiert auf S. 43).

[SP04] V. A. Siris, F. Papagalou. „Application of anomaly detection algorithms for detecting SYN flooding attacks“. In: *IEEE Global Telecommunications Conference, 2004. GLOBECOM'04.* Bd. 4. IEEE. 2004, S. 2050–2054 (zitiert auf S. 38).

[Sta14] O. Standard. „MQTT version 3.1. 1“. In: *URL http://docs. oasis-open. org/mqtt/mqtt/v3* 1 (2014) (zitiert auf S. 29).

[Sta19] O. Standard. „MQTT Version 5.0“. In: *Retrieved June* 22 (2019), S. 2020 (zitiert auf S. 29, 31, 32).

[STS20] Statista. *Internet of Things (IoT) and non-IoT active device connections worldwide from 2010 to 2025*. 2020. URL: https://www.statista.com/statistics/1101442/iot-number-of-connected-devices-worldwide/ (zitiert auf S. 15).

[TAB17] M. Tappler, B. K. Aichernig, R. Bloem. „Model-based testing IoT communication via active automata learning“. In: *2017 IEEE International conference on software testing, verification and validation (ICST)*. IEEE. 2017, S. 276–287 (zitiert auf S. 46).

[TSTB19] A. Tchernykh, U. Schwiegelsohn, E.-g. Talbi, M. Babenko. „Towards understanding uncertainty in cloud computing with risks of confidentiality, integrity, and availability“. In: *Journal of Computational Science* 36 (2019), S. 100581 (zitiert auf S. 34).

[TTH20] K. Tanabe, Y. Tanabe, M. Hagiya. „Model-based testing for MQTT applications“. In: *Joint Conference on Knowledge-Based Software Engineering*. Springer. 2020, S. 47–59 (zitiert auf S. 45).

[TYY10] G. Tian-yang, S. Yin-Sheng, F. You-yuan. „Research on software security testing“. In: *World Academy of science, engineering and Technology* 70 (2010), S. 647–651 (zitiert auf S. 22).

[UL10] M. Utting, B. Legeard. *Practical model-based testing: a tools approach*. Elsevier, 2010 (zitiert auf S. 27).

[ULB+16] M. Utting, B. Legeard, F. Bouquet, E. Fourneret, F. Peureux, A. Vernotte. „Recent advances in model-based testing“. In: *Advances in computers* 101 (2016), S. 53–120 (zitiert auf S. 25).

[UPL12] M. Utting, A. Pretschner, B. Legeard. „A taxonomy of model-based testing approaches“. In: *Software testing, verification and reliability* 22.5 (2012), S. 297–312 (zitiert auf S. 25).

[VAC20] I. Vaccari, M. Aiello, E. Cambiaso. „SlowITe, a novel denial of service attack affecting MQTT“. In: *Sensors* 20.10 (2020), S. 2932 (zitiert auf S. 39).

[VF13] O. Vermesan, P. Friess. *Internet of things: converging technologies for smart environments and integrated ecosystems*. River publishers, 2013 (zitiert auf S. 15).

[WB16] M. Weber, M. Boban. „Security challenges of the internet of things“. In: *2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE. 2016, S. 638–643 (zitiert auf S. 15).

[WLL+10] M. Wu, T.-J. Lu, F.-Y. Ling, J. Sun, H.-Y. Du. „Research on the architecture of Internet of Things“. In: *2010 3rd international conference on advanced computer theory and engineering (ICACTE)*. Bd. 5. IEEE. 2010, S. V5–484 (zitiert auf S. 18).

[WZS02] H. Wang, D. Zhang, K. G. Shin. „Detecting SYN flooding attacks“. In: *Proceedings. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*. Bd. 3. IEEE. 2002, S. 1530–1539 (zitiert auf S. 38).

[ZLG+20] Y. Zeng, M. Lin, S. Guo, Y. Shen, T. Cui, T. Wu, Q. Zheng, Q. Wang. „Multi-fuzz: A coverage-based multiparty-protocol fuzzer for iot publish/subscribe protocols“. In: *Sensors* 20.18 (2020), S. 5194 (zitiert auf S. 29, 43).

Alle URLs wurden zuletzt am 01. 12. 2021 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift