# Architektur und Implementierung einer sicheren IoT-Managementlösung

Leon Lukas


BACHELORARBEIT

Matrikelnummer: 58771316

Studiengruppe: IF7

Studiengang: Informatik

Betreuung:

Prof. Dr. Thomas Schreck

# Erklärung

Ich erkläre eidesstattlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche gekennzeichnet habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

München, am 6. September 2020

Leon Lukas

Studiengruppe IF7W / SS / 2020
Geburtsdatum 21.11.1997

# Kurzfassung

Trotz des starken Wachstums der IoT-Branche werden entsprechende Verwaltungssysteme meist ohne die nötigen Sicherheitsvorkehrungen entwickelt. So entstehen weitreichende Gefahren für Firmen, Privatpersonen sowie die gesamte Gesellschaft.
In dieser Abschlussarbeit werden die Gefahren für eine IoT-Managementlösung analysiert und Techniken zur Sicherung solcher Systeme erläutert. Dabei wird auf die Sicherheit des Updateprozesses sowie die Sicherheit der Online-Verwaltungssysteme eingegangen. Anschließend wird mithilfe der Frameworks *The Update Framework (TUF)* und *Django* die Architektur einer sicheren Managementlösung vorgeschlagen. Die Architektur sieht einen Server vor, der Benutzern die Installation und Konfiguration von Anwendungen auf den IoT-Geräten über ein Webinterface ermöglicht und Software-Updates als auch Konfigurations-Daten an die Clients verteilt. Die clientseitige Implementierung lädt Konfigurationen und Updates vom Server herunter und verwaltet die installierten Anwendungen in Form von Kind-Prozessen. Die vorgeschlagene Architektur wurde in einer Python-Implementierung umgesetzt und auf einem Raspberry Pi Einplatinencomputer eingesetzt. Schließlich wird die entwickelte Lösung bezüglich ihrer Benutzbarkeit und Sicherheit evaluiert.
Insgesamt konnte festgestellt werden, dass durch die Berücksichtigung von Sicherheit während des gesamten Entwicklungsprozesses und den Einsatz standardisierter Sicherheitsmaßnahmen eine Anwendungen sicher entwickelt werden kann, ohne kryptografische Verfahren selbst zu programmieren. So kann ein aufwendiger und zugleich fehleranfälliger Arbeitsschritt eingespart werden.

# Abstract

Despite the strong growth of the IoT industry, IoT management systems are usually developed without the necessary security measures. This creates profound dangers for companies, private individuals, as well as society in general.
In this thesis the dangers for an IoT management solution are analyzed and techniques for safeguarding such systems are elucidated. Especially the security of the update process as well as the security of the online management systems will be addressed. Subsequently, the architecture of a secure management solution is proposed with the help of the frameworks *The Update Framework (TUF)* and *Django*. The architecture comprises a server that allows users to install and configure applications on the IoT device via a web interface. Furthermore, it distributes software updates and configuration data to the clients. The client-side implementation downloads configurations and updates from the server and manages the installed applications as child processes. The proposed architecture is created in a Python implementation and deployed on a Raspberry Pi single-board computer. Finally, the developed solution is evaluated regarding its usability and security.
Overall, it was established that such an application can be developed securely using standardised security measures and prioritising safety throughout the entire development process, without being required to program cryptographic procedures oneself. This way a costly and fault-prone work step can be avoided.

# Inhaltsverzeichnis

# Kapitel 1

# Einleitung

## 1.1 Motivation

Das Internet of Things (IoT, deutsch: Internet der Dinge) gewinnt sowohl für Privatnutzer als auch in Unternehmen an Bedeutung. So bewerten die Hälfte der Unternehmen die Relevanz von IoT-Themen schon jetzt als sehr hoch, in zwei bis drei Jahren prognostizieren über zwei Drittel der Unternehmen eine hohe Relevanz der IoT. In der Wirtschaft wird IoT hauptsächlich in der Qualitätskontrolle sowie in der vernetzten Produktion (Industrie 4.0) eingesetzt [6].
Das Wachstum der Branche wird allerdings von Bedenken bezüglich der Sicherheit solcher Geräte gebremst. Für Privatnutzer ist vor allem die Sicherheit des Heimnetzwerks bedroht. Unsichere IoT-Geräte bieten Angreifern einen Weg in das Heimnetzwerk und damit zu sensiblen Daten. Unternehmen fürchten Hackerangriffe, die zu Produktionsausfällen führen können. Auch die Möglichkeit der Industriespionage stellt ein Risiko dar[6].
Es bestehen aber noch weitreichendere Gefahren. So können durch die Infektion vieler energieverbrauchsrelevanter IoT-Geräte besonders gefährliche Botnetze erstellt werden. Mit diesen Netzen lassen sich Spitzen im Energieverbrauch erzeugen, um das Stromnetz durch überlastung anzugreifen.
Doch warum sind Sicherheitsbedenken bei IoT größer als in anderen Bereichen? Bei herkömmlichen IT-Produkten wie Computer oder Smartphone lassen sich Sicherheitslücken gewöhnlich durch das Installieren von Sicherheitsupdates beheben. Da im IoT-Umfeld selten große Betriebssysteme wie Windows oder Android eingesetzt werden, müssen Sicherheitslücken von Geräteherstellern oder Administratoren behoben werden. Ein weiteres Problem ist die begrenzte Leistungsfähigkeit der IoT-Hardware. Sie erschwert den Einsatz der Sicherheitssoftware von Drittanbietern. Auch eine Trennung der Geräte vom Internet ist keine Lösung, da diese häufig online verwaltet werden. Von Web-Diensten erhalten die IoT-Geräte ihre Konfiguration, die ihre Arbeitsweise beeinflusst. Auch in

diesem Arbeitsschritt ist es wichtig, die Korrektheit dieser Konfiguration sicherzustellen, um ein ordnungsgemäßes Verhalten der Geräte gewährleisten zu können.

## 1.2 Problemstellung

Wie kann die Sicherheit von IoT-Geräten erhöht werden? Momentan gibt es zwei große Schwachstellen bei IoT-Geräten. Eine davon ist das Fehlen von sicherheitskritischen Aktualisierungen der Software. Die andere Schwachstelle liegt in den Online-Verwaltungsplattformen, die häufig ohne die nötigen Sicherheitsvorkehrungen entwickelt werden. Ziel dieser Bachelorarbeit ist es, diese zwei Probleme zu lösen.
Im Folgenden wird eine Anwendung entwickelt, die sowohl das sichere Laden von Softwareaktualisierungen als auch Konfigurationen aus dem Internet ermöglicht. Die zu entwickelnde Anwendung besteht aus zwei Komponenten:

1. Ein auf dem IoT-Gerät laufenden Programm, das selbstständig eine sichere Verbindung zu einem Webserver aufbaut und von dort Aktualisierungs- und Konfigurationsdaten lädt.
2. Ein Webserver, der die Updates bereitstellt und eine Weboberfläche für das Konfigurieren der IoT-Geräte besitzt.

## 1.3 Gliederung

Dieses Kapitel beinhaltet neben einer Einführung in die Umsetzung sowie Notwendigkeit dieser Arbeit eine Übersicht über andere wissenschaftliche Beiträge in diesem Gebiet. In Kapitel 2 werden sichere Methoden für die Verteilung von Software-Updates sowie die Entwicklung von Webanwendungen vorgestellt. Als Nächstes wird anhand dieser Informationen eine Architektur für ein IoT-Managementsystem vorgeschlagen und bezüglich ihrer Sicherheit überprüft. Kapitel 4 beschäftigt sich mit der Umsetzung der Architektur mithilfe der Frameworks Django und TUF. Außerdem wird auf den Betrieb des Management-Servers eingegangen. Im 5. Kapitel findet eine Evaluierung der Arbeit bezüglich der Einsetzbarkeit aus Sicht eines IoT-Entwicklers sowie Endbenutzers statt. Das letzte Kapitel liefert einen Ausblick über mögliche Erweiterungen dieser Arbeit. Außerdem wird darin das Ergebnis der Bachelorarbeit zusammengefasst.

## 1.4 Andere Arbeiten

In [11] wird die Architektur eines cloudbasierten IoT-Managementsystems beschrieben. Die Architektur sieht eine Trennung zwischen Plattform und Anwendung vor. Der Besitzer einer IoT-Anwendung kann die Services des Plattformbetreibers kaufen, um seine

Anwendung zu verwalten, sodass er sich keine Gedanken über die Sicherheit oder die Infrastruktur machen muss. Ein weiterer interessanter Punkt der Arbeit ist die Verteilung kryptografischer Schlüssel auf die Geräte. Um Schlüssel nicht auf jedes Gerät laden zu müssen, wird mittels einer *Physical unclonable function (PUF)* ein Schlüssel anhand der physischen Eigenschaften des Prozessors erzeugt. Die von Schrijen et al. präsentierte Lösung eignet sich, um IoT im industriellen Umfeld zu verwalten. Vor allem die PUF-generierten Schlüssel vereinfachen die Roll-out-Phase. Nicht jede IoT-Anwendung kann sich ein derart komplexes Verwaltungssystem leisten. In dieser Arbeit wird eine einfachere, wenn auch nicht so skalierbare Alternative vorgestellt.

Auch mit der Blockchain-Technologie wird an Lösungen zur Verwaltung großer Mengen von IoT-Geräten gearbeitet. Blockchain ermöglicht eine gut skalierbare, dezentrale Verwaltung von Milliarden von Geräten. Aufgrund der geringen Leistung von IoT-Geräten sind diese jedoch häufig nicht in der Lage, Teil einer Blockchain zu sein. In [9] wird eine Architektur beschrieben, in der IoT-Geräte nicht direkt in der Blockchain integriert sind, sondern über Management-Hubs mit dem Netzwerk verbunden sind. Diese Hubs übersetzen die für die Kommunikation von IoT-Geräte spezifischen CoAP-Nachrichten in für die Blockchain verständliche JSON-RCPs. Anhand der dazugehörigen „proof-of-concept“-Implementierung wurde festgestellt, dass sich die vorgestellte Architektur gut für die Verwaltung sehr vieler Geräten eignet. In dieser Arbeit wird ein Ansatz ohne zusätzliche Management-Hubs verfolgt. So kann ein einzelnes Gerät flexibler eingesetzt werden.

Die Arbeit von [10] beschäftigt sich nicht mit technischen Details von IoT-Management, sondern mit der Architektur des kompletten Lifecycles. Ziel ist es, ein dem App-Store von Smartphones ähnliches Konzept zu entwickeln. Das Konzept sieht vor, dass App-Entwickler einen Teil ihrer Einnahmen an den Store-Betreiber abgeben. Außerdem werden die dafür nötigen Leistungsmerkmale der Infrastruktur ermittelt. In dieser Arbeit wird ebenfalls ein App-Store-Konzept vorgestellt, das jedoch nicht mit den großen Smartphone-App-Stores vergleichbar ist. Vielmehr bietet es die Möglichkeit, einen App-Store innerhalb einer Organisation oder für einen bestimmten Gerätetypen anzubieten.

Ein anderer Ansatz für die Sicherung von IoT-Geräten sind Manufacturer Usage Descriptions (MUD)[19]. Die Funktionsweise von MUD beruht auf der Tatsache, dass IoT-Geräte für einen bestimmten Zweck hergestellt wurden und deshalb nur bestimmte Kommunikationsmuster aufweisen. Kommunikation, die nicht diesem Muster entspricht, ist somit bösartig und kann von der Firewall blockiert werden. Welche Kommunikation zulässig ist, wird in einer vom Gerätehersteller bereitgestellten Datei definiert. In [1] wird

von Andalibi et al. eine Erweiterung zu MUD vorgeschlagen, die einige Anpassungen vornimmt, um MUD im „fog computing“ [27] Umfeld einzusetzen. Durch die Einführung einer „peak request rate“ wird die Anzahl der Verbindungen pro Sekunde limitiert. So kann verhindert werden, dass IoT-Geräte Teil eines Botnetzes für DDos-Angriffe werden. Feraudo et al. [5] analysiert den aktuellen Stand von Entwicklungen in diesem Gebiet und entwickelt eine zusätzliche nutzerfreundliche Schnittstelle. Mit der Schnittstelle können Änderungen an der vom Hersteller bereitgestellten Spezifikation durchgeführt werden, um diese an individuelle Anforderungen anpassen zu können.

Auch in [8] wird eine netzbasierte IoT-Sicherheitslösung vorgeschlagen. Da Endbenutzern meist die nötige Kompetenz fehlt, ihr Heimnetzwerk selbst zu administrieren, bietet die dort vorgeschlagene Lösung Sicherheit als Service via Fernwartung. Das *Intrusion Detection System(IDS)* nutzt *Software-defined Networking(SDN)*, um Fernwartung sowie die Isolation von bösartigen Geräten vornehmen zu können. Netzwerkkommunikation wird bei Nobakht et al. nicht mittels Regeln, sondern anhand eines davor trainierten Klassifikators gefiltert. Netzbasierte Ansätze haben im Gegensatz zu der in dieser Arbeit vorgestellten Lösung den Nachteil, dass sie zusätzliche Hardware benötigen oder der Router die Filterung unterstützen muss.

Die Arbeit von [7] verfolgt einen Ansatz, der ebenfalls das Verhalten des IoT-Geräts überwacht, jedoch keine zusätzliche Hardware benötigt. Dafür wird auf dem IoT-Gerät ein nur 1.4 MB großer Agent installiert. Der Agent überwacht laufende Prozesse mittels des Linux /proc filesystem sowie Netzwerkkommunikation mit dem Modul iptables. Treten dabei Abweichungen von einer Whitelist auf, greift der Agent ein. Der von Maloney et al. präsentierte Ansatz kann leicht mit vielen verschiedenen Geräten verwendet werden, bietet allerdings nur eine Abwehr von Angriffen. Die in dieser Arbeit entwickelte Lösung bietet neben Sicherheitsfeatures auch die Möglichkeit Software und Konfiguration des IoT-Geräts zu aktualisieren.

# Kapitel 2

# Grundlagen

Das Gebiet der IT-Sicherheit reicht von organisatorischen Maßnahmen über sichere Programmierung bis hin zu hoch mathematischen Verschlüsselungsalgorithmen. Um die Relevanz der einzelnen Bestandteile beurteilen zu können, ist ein Verständnis grundlegender Begriffe erforderlich. Zum Verstehen dieser Arbeit werden im Folgenden Grundlagen sicherer Softwareaktualisierungen sowie Anwendungen im Internet erläutert.

## 2.1 Sichere Updates

Wie bereits in der Einleitung festgestellt wurde, entstehen die Sicherheitslücken von IoT-Geräten häufig durch fehlende Updates der Systeme. Auch das Bundesamt für Sicherheit in der Informationstechnik (BSI) weist auf diese Problematik hin [21]. Um die Sicherheit eines IoT-Geräts zu gewährleisten, ist es wichtig, die Software aktuell zu halten. Ebenfalls wichtig ist es, den Prozess des Aktualisierens sicher zu gestalten. Gelingt es einem Angreifer Schadsoftware als Aktualisierung zu tarnen, kann er die volle Kontrolle über das Gerät übernehmen und zukünftig seine eigenen Aktualisierungen installieren.

### 2.1.1 Paketverwaltung

Ein gängiges System für das Verteilen von Softwareaktualisierungen sind Paketverwaltungsysteme wie apt [20]. Paketverwalter bestehen meist aus zwei Komponenten: einem Paket Repository und einem Paket Installationsprogramm [3].

**Pakete:** Als Pakete werden Archive bezeichnet, in denen alle Dateien einer Software verpackt sind. Außerdem besitzen viele Pakete zusätzlich eingebettete Metadaten, in denen allgmeine Informationen über das Paket, sowie Referenzen zu zusätzlicher Software, die für dieses Paket notwendig ist (Requirements), aufgelistet sind[2].

**Paket Installationsprogramme:** Das Installationsprogramm ist der Teil der Paketverwaltung, die auf einem Client installiert ist. Wird auf einem Client ein Installations- beziehungsweise Updateprozess gestartet, installiert das Installationsprogramm die für dieses Paket notwendige Software. Dieser Prozess wird *dependency resolution* genannt. Anschließend wird die gewünschte Software installiert.

**Paket Repositories:** Bei Paket Repositories handelt es sich meist um HTTP [28] oder FTP [26] Server, von denen Clients Pakete sowie Metadaten beziehen. Anhand der Metadaten weiß der Client, welche Pakete mit welchen Voraussetzungen vorhanden sind. Des Weiteren existiert eine Root-Metadaten-Datei. In dieser werden die Orte, sowie Hashwerte der anderen Metadaten-Dateien gespeichert. Je nach Paketverwaltung werden Pakete, Metadaten oder Root-Metadaten mittels asymmetrischer Kryptografie signiert [2].

### 2.1.2 Angriffe auf Paketverwaltungssoftware

Da Paketverwaltungssoftware für ihre Funktionalität Superuser(root)-Rechte benötigt, ist sie ein beliebtes Ziel für Angriffe. Im Folgenden wird erklärt, welche Angriffe existieren und wie diese verhindert werden können.

Bedrohungsmodell In dieser Arbeit wird ein Bedrohungsmodell genutzt, in dem ein Angreifer Anfragen des Installationsprogramms unterbrechen und diese selbst dann beantworten kann. Dies kann mittels eines Man-In-The-Middle-Angriffs realisiert werden oder indem der Client einen falschen Server kontaktiert (z.B. durch DNS-Cache Poisoning). Daraus ergibt sich folgendes Bedrohungsmodell:

- Der Angreifer kann dem Client willkürliche Dateien liefern.
- Der Angreifer weiß im vorraus nicht, welches Paket der Client anfordert.
- Der Angreifer besitzt keinen Schlüssel, um Pakete oder Metadaten zu signieren.
- Der Angreifer hat Zugang zu alten Paketen und Metadaten.
- Der Angreifer kennt Schwachstellen in alten Paketen. Diese kann er zum Beispiel aus Änderungsprotokollen entnehmen.
- Der Angreifer kennt keine Schwachstellen in aktuellen Paketen.
- Die Wurzel-Metadaten enthalten ein Ablaufdatum.

Angriffe Aus diesem Bedrohungsmodell ergeben sich einige Angriffe, die gegen einen Client durchgeführt werden können. Die Auswirkung dieser Angriffe variiert vom Absturz bis hin zur Übernahme des Client-Rechners. Jeder dieser Angriffe wurde bereits erfolgreich durchgeführt [2].

**Willkürliches Paket** Der Angreifer liefert an Stelle des gewünschten ein selbst erstelltes Paket.

**Replay-Angriff** Der Angreifer liefert eine alte, korrekt signierte Version eines Pakets. So installiert der Client ein altes unsicheres Paket.

**Freeze-Angriff** Funktioniert analog zum Replay-Angriff. Der Angreifer liefert jedoch keine alten Pakete, sondern lediglich alte Metadaten. Der Client wird somit nicht über Aktualisierungen informiert und bleibt so auf einem alten, potentiell unsicheren Stand.

**Fremde Abhängigkeiten** Der Angreifer fügt weitere Voraussetzungen zu den Metadaten hinzu. Diese werden durch das Installationsprogramm zusätzlich installiert. So kann er unsichere Pakete auf dem System installieren.

**Endlose Daten** Bei diesem Angriff liefert der Angreifer als Antwort auf eine Anfrage einen nicht endenden Strom an Daten. Dies kann den Speicher des Clients auffüllen und das System zum Absturz bringen.

### 2.1.3 Sicherheitstechniken für Paketverwaltungen

Um Paketverwaltungssoftware sicher zu gestalten, ist es wichtig, vor dem Installieren von Software aus dem Internet deren Authentizität sicher zu stellen. Dazu werden Hashfunktionen sowie digitale Signaturen aus der asymmetrischen Kryptografie angewendet. Die Sicherheitssysteme verschiedener Paketverwaltungssoftwares unterscheiden sich vor allem darin, welche Dateien signiert werden. Wie die im vorherigen Abschnitt beschriebenen Angriffe verhindert werden können, wird im Folgenden erläutert.

**Maximalgröße für Downloads** Durch das Hinzufügen einer Grenze für die maximal zulässige Download-Größe lässt sich ein „Endlose Daten“ Angriff verhindern. Trotz seiner Simplizität nutzen nicht alle Paketverwalter dieses Verfahren [2].

**Root-Metadaten-Signaturen** Durch das Signieren der Root-Metadaten werden alle auf Metadaten-Manipulation basierenden Angriffe (fremde Voraussetzungen, willkürliches Paket) verhindert.

**Zeitstempel in den Metadaten** Um Replay-Angriffe abzuwehren, kann ein Zeitstempel in die Root-Metadaten geschrieben werden. Wenn neue Wurzel-Metadaten heruntergeladenen werden, kann somit überprüft werden, ob die heruntergeladenen Daten neu sind.

**Verfallsdaten/Gültigkeit von Metadaten** Freeze-Angriffe können durch das Hinzufügen eines Ablaufdatums für die Root-Metadaten verhindert werden. So kann der Client erkennen, wenn ihm eine alte Version der Metadaten von einem Angreifer geliefert wird.

## 2.2 Sicherheit von Webanwendungen

Bei einer Webanwendung handelt es sich um eine nach dem Client-Server-Modell vorgehende Anwendung die das Hypertext Transfer Protocol (HTTP) nutzt. Auf dem Client, bei dem es sich häufig um einen Internetbrowser oder eine App auf einem mobilen Endgerät handelt, werden meist nur Daten angezeigt bzw. eingegeben. Die Verarbeitung und Sicherung der Daten findet, nachdem eine Übertragung mittels HTTP geschehen ist, auf einem Webserver statt. Da Webanwendungen heutzutage in fast allen Bereichen, unter anderem dem Finanz-, Gesundheits- und IoT-Sektor, eingesetzt werden, ist es essentiel, diese gegen Angriffe zu schützen. Angreifer erbeuten häufig sensible Daten wie Kreditkarteninformationen, Krankendaten oder Firmengeheimnisse. Das Open Web Application Security Project (OWASP) ist eine Non-Profit-Organisation mit dem Ziel, die Sicherheit von Webanwendungen zu erhöhen. Dazu stellt es Software sowie Informationen zur Entwicklung sicherer Webanwendungen zur Verfügung. Der nachfolgende Abschnitt präsentiert für diese Arbeit relevante Informationen des OWASP.

Anwendungssicherheitsanforderungen Um eine sichere Webanwendung zu erstellen, ist es wichtig, vorher zu definieren, was „sicher“ im Falle einer speziellen Anwendung bedeutet. [24]

Anwendungssicherheitsarchitektur Anstatt Sicherheit nachträglich in eine Anwendung oder API einzubauen, ist es kosteneffektiver, diese schon beim Design zu beachten [24].

Standardisierte Sicherheitsmaßnahmen Die Entwicklung starker und anwendbarer Sicherheitsmaßnahmen ist nicht trivial. Standardisierte Sicherheitsmaßnahmen vereinfachen die Entwicklung sicherer Anwendungen oder APIs. Viele moderne Frameworks enthalten heute schon standardmäßig effektive Sicherheitsprüfungen für Autorisierung, Validierung, CSRF-Schutz etc [24].

### 2.2.1 Sicherheitsrisiken von Webanwendungen

Ein wichtiges Projekt des OWASPs sind die OWASP Top 10. Sie sind eine Zusammenstellung der 10 größten Sicherheitsrisiken von Webanwendungen. Diese werden aus Daten hunderter Firmen und Sicherheitsexperten ermittelt. Die Risiken, die für diese Arbeit die größte Relevanz haben, werden im Folgenden erklärt.

Injection Injection-Schwachstellen, wie beispielsweise SQL-, OS- oder LDAP-Injection, treten auf, wenn nicht vertrauenswürdige Daten von einem Interpreter als Teil eines Kommandos oder einer Abfrage verarbeitet werden. Ein Angreifer kann Eingabedaten

dann so manipulieren, dass er nicht vorgesehene Kommandos ausführen oder unautorisiert auf Daten zugreifen kann.

Fehler bei der Authentifizierung Anwendungsfunktionen, die im Zusammenhang mit Authentifizierung und Session-Management stehen, werden häufig fehlerhaft implementiert. Dies erlaubt es Angreifern, Passwörter oder Session-Token zu kompromittieren oder die entsprechenden Schwachstellen so auszunutzen, dass sie die Identität anderer Benutzer vorübergehend oder dauerhaft annehmen können.

Fehler in der Zugriffskontrolle Häufig werden die Zugriffsrechte für authentifizierte Nutzer nicht korrekt um- bzw. durchgesetzt. Angreifer können entsprechende Schwachstellen ausnutzen, um auf Funktionen oder Daten zuzugreifen, für die sie keine Zugriffsberechtigung haben. Dies kann Zugriffe auf Konten anderer Nutzer sowie auf vertrauliche Daten oder aber die Manipulation von Nutzerdaten, Zugriffsrechten etc. zur Folge haben.

Nutzung von Komponenten mit bekannten Schwachstellen Komponenten wie Bibliotheken, Frameworks etc. werden mit den Berechtigungen der zugehörigen Anwendung ausgeführt. Wird eine verwundbare Komponente ausgenutzt, kann ein solcher Angriff zu Datenverlusten bis hin zu einer Übernahme des Systems führen. Applikationen und APIs, die Komponenten mit bekannten Schwachstellen einsetzen, können Schutzmaßnahmen unterlaufen und so Angriffe mit schwerwiegenden Auswirkungen verursachen. [24]

### 2.2.2 Sicherheitstechniken bei Webanwendungen

Um vertrauliche Informationen über das geteilte Medium Internet zu übermitteln, wurden verschiedene Techniken entwickelt, die Vertraulichkeit und Authentizität der Kommunikation gewährleisten.

Hypertext Transfer Protocol Secure Bei unverschlüsselten Verbindungen ist es Angreifern möglich, die komplette Kommunikation mitzuhören. Werden vertrauliche Informationen wie Kreditkarteninformationen oder Firmengeheimnisse übermittelt, ist das problematisch. Das Hypertext Transfer Protocol Secure (HTTPS) ist eine Erweiterung des HTTP Protokolls um *Transport Layer Security (TLS)*. Bei TLS handelt es sich um ein hybrides Verschlüsselungsverfahren. Damit ist es möglich verschlüsselt zu kommunizieren, ohne im Voraus einen Schlüssel zu teilen. Auch bei nicht vertraulichen Informationen wie beispielsweise für Wikipedia wird HTTPS eingesetzt. Hier wird die Privatsphäre des Nutzers geschützt. Durch die Verschlüsselung ist von außen nur sichtbar, dass der Nutzer auf Wikipedia zugegriffen hat, nicht aber, welche Informationen er recherchiert hat.

Digitale Zertifikate HTTPS ermöglicht eine Verbindung, ohne einen im Voraus geteilten Schlüssel zu sichern. Es garantiert jedoch nicht, dass es sich beim Kommunikationspartner um den handelt, für den er sich ausgibt. Aus diesem Grund werden digitale Zertifikate benötigt. Zertifikate werden von vertrauenswürdigen Zertifizierungsstellen für Anbieter von Internetdiensten ausgestellt und signiert, so lässt sich ihre Echtheit überprüfen. Ist das Zertifikate korrekt, so ist der Kommunikationspartner der, für den er sich ausgibt.

# Kapitel 3

# Architektur

Das folgende Kapitel beschreibt die Architektur des IoT-Managementsystems. Schon bei der Entwicklung der Architektur war die Sicherheit des Systems ein entscheidender Faktor. So ist diese von Anfang an Teil des Systems, und wird nicht erst nachträglich hinzugefügt .

## 3.1 System

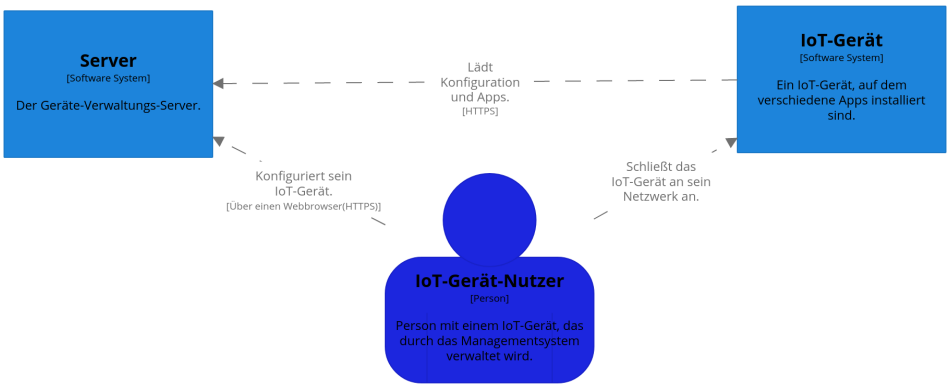


**Abbildung 3.1:** Das Gesamtsystem

Das IoT-Managementsystem besteht aus zwei Komponenten, einem öffentlich erreichbaren Server und einem Client auf dem IoT-Gerät. Abbildung 3.1 bietet einen Überblick der Komponenten und deren Interaktion im System. Jedes Gerät ist mit einem Benutzerkonto auf dem Webserver verbunden. So können sich das Gerät und der Benutzer am Server authentifizieren. Der Server besitzt ein Webinterface über das Benutzer auf einen Appstore zugreifen können. Jener bietet ihm wiederum die Möglichkeit,

Apps auf dem IoT-Gerät zu installieren und zu konfigurieren. Ziel des Systems ist es, Inbetriebnahme und Wartung des IoT-Geräts möglichst einfach zu halten. Um das Gerät einzuschalten, genügt es das Gerät an das Netzwerk anzuschließen und mit Strom zu versorgen. Software-Aktualisierungen und Konfigurationsdaten werden automatisch von dem Server geladen. Die Kommunikation mit dem Server findet über eine verschlüsselte Verbindung statt, um das System vor Angriffen zu schützen.

## 3.2 Server

Der Server ist in drei Container (Kontext, in dem Code ausgeführt oder Daten gespeichert werden) eingeteilt. Der *Django*-Web-Server [15] ermöglicht es Benutzern, ihre Geräte über den Browser zu konfigurieren. Über die *Representational State Transfer(REST)*-Schnittstelle können diese Konfigurationen von den Geräten bezogen werden. Aktualisierungen der Apps werden über ein Software-Repository verteilt. Zusätzlich befindet sich auf dem Server eine Datenbank- instanz, um die Konfigurationen der Benutzer zu persistieren. In Abbildung 3.2 werden die Beziehungen der Komponenten untereinander und deren Kommunikation nach außen veranschaulicht.

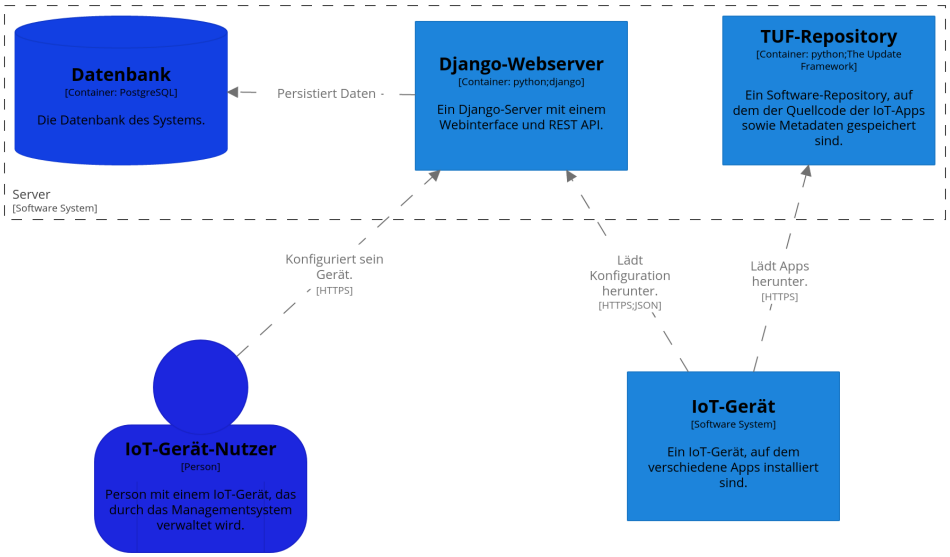


**Abbildung 3.2:** Containerdiagramm des Servers

### 3.2.1 Django-Webserver

Bei Django handelt es sich um ein in Python [18] verfasstes, quelloffenes Framework für die Entwicklung von Webanwendungen. Django eignet sich besonders für die Entwicklung

datenbanklastiger Systeme wie zum Beispiel einem Appstore. Außerdem besitzt es eine Vielzahl eingebauter Funktionen wie eine Benutzerverwaltung, ein REST-Framework sowie viele Sicherheitsfeatures [15].
In Django werden verschiedene Komponenten als „App“ bezeichnet. Für das Erstellen beziehungsweise Verteilen der Konfiguration ist jeweils eine App zuständig (siehe Abbildung 3.3).

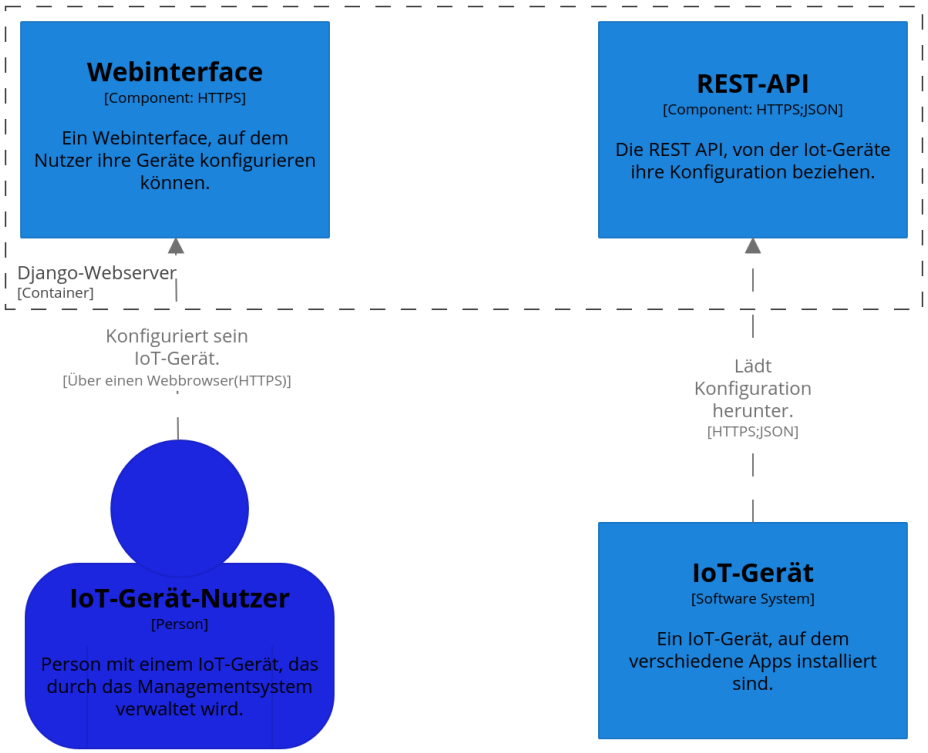


**Abbildung 3.3:** Komponentendiagramm des Django-Servers

Webinterface Das Webinterface bietet angemeldeten Benutzern die Möglichkeit, Apps aus einer Liste verfügbarer Anwendungen zu installieren. Für jede installierte App gibt es außerdem eine Maske, über die ein Benutzer sie konfigurieren kann. Die Verbindung zwischen dem Webbrowser des Benutzers und dem Server ist mit TLS verschlüsselt, um die Vertraulichkeit der Konfiguration sowie der Nutzerdaten zu schützen.

REST API Die REST API bildet die Schnittstelle zwischen dem IoT-Gerät und dem Webserver. Die Anfragelogik der API ist zweistufig aufgebaut: In der ersten Stufe fragt das IoT-Gerät ab, welche Apps vom Benutzer installiert wurden sowie unter welcher Adresse

sich das Software-Repository befindet. In der zweiten Stufe werden für installierte Apps die relevanten Einstellungen vom Server heruntergeladen und der App zur Verfügung gestellt. Für die Entwicklung der REST API wurde das Django REST Framework verwendet.

**Sicherheit** Das Django-Framework besitzt einige Sicherheitsfunktionen, die in dieser Arbeit genutzt werden. In seiner Standardeinstellung bietet Django Schutz gegen *Cross-site-scripting (XSS)*, *Cross-site-request-forgery (CSRF)*, *SQL-injection* sowie *Clickjacking*. Zusätzlich werden mit der Erweiterung *django-session-timeout* Sessions nach 5 Minuten Inaktivität terminiert.

### 3.2.2 TUF-Repository

Das *TUF-Repository* ist ein auf The Update Framework basierendes Software-Repository, das den App-Quellcode in Form von Paketen an die Geräte verteilt.

**The Update Framework (TUF)** TUF ist ein von Justin Cappos im Secure Systems Labs der New York University entwickeltes Framework, um sichere Softwareupdates zu ermöglichen. Ziele sind flexibel und einfach die Sicherheit vorhandener und neuer Software-Updater zu erhöhen und die Auswirkungen eines Schlüsselverlusts zu minimieren. TUF erreicht dies, indem es überprüfbare Aufzeichnungen über den Zustand des Software-Repositories bereitstellt. Die Aufzeichnungen enthalten unter anderem eine Liste vertrauenswürdiger kryptografischer Schlüssel, Hashwerte der Pakete, signierte Metadaten und deren Versionsnummern und Ablaufdaten. Mithilfe dieser Informationen kann die Authentizität einer Aktualisierung überprüft werden. Somit ist TUF gegen die in 2.1.2 beschriebenen Angriffe sicher. Alle zusätzlichen Metadaten werden ausschließlich von TUF verarbeitet und müssen nicht von der eigentlichen Anwendung beachtet werden. In [4] wird auf die Funktionsweise von TUF sowie die anderer Paketverwaltungssoftwares genauer eingegangen.

**Repository Aufbau** Die Aufteilung des Repositories in die vier Komponenten *Keystore-Verzeichnis*, *Repository-Verzeichnis*, *TUF-Signer* und *HTTPS-Server* wird in 3.4 veranschaulicht. Der Hauptbestandteil des Repository ist der TUF-Signer. Mit dessen Hilfe können Metadaten für Pakete erstellt und signiert werden. Dazu benötigt dieser das Paket und die kryptografischen Schlüssel aus dem Keystore-Verzeichnis. Die Pakete und signierten Metadaten werden in einem Verzeichnis gespeichert, von dort verteilt sie ein HTTPS-Server an die Clients.

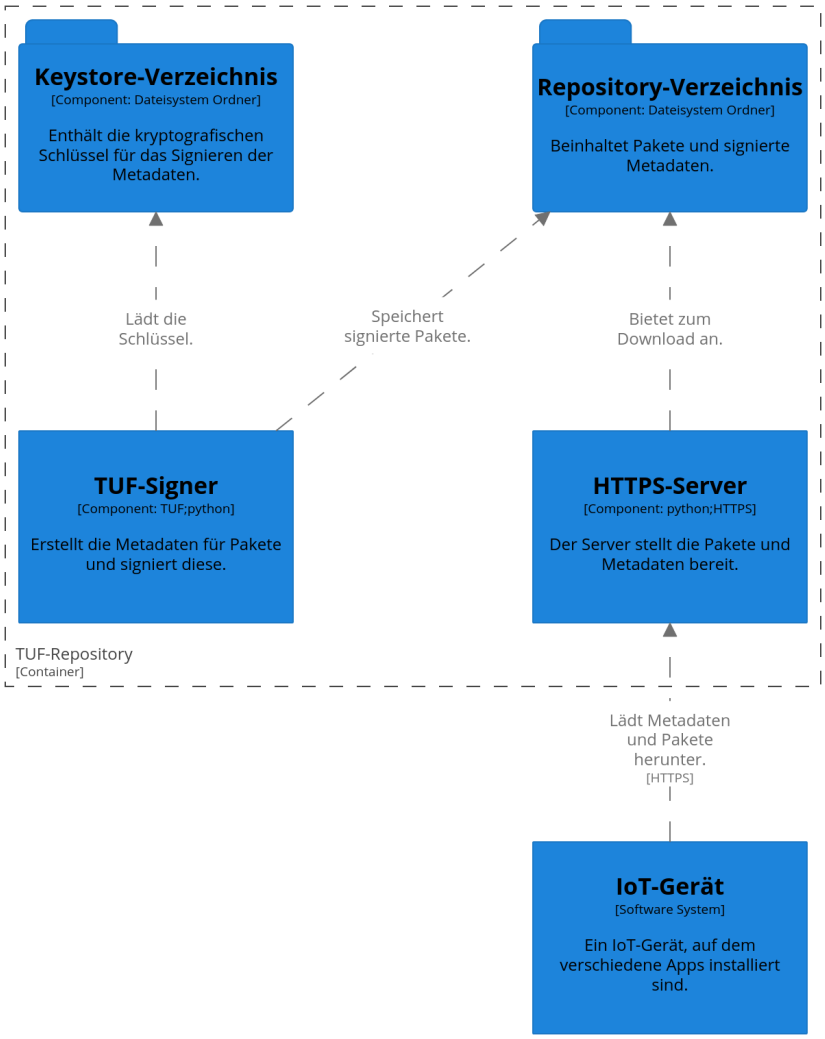


**Abbildung 3.4:** Komponentendiagramm des Repositories

Sicherheit TUF bietet einen state-of-the-art Schutz der Authentizität der Pakete. Zusätzlich wird mittels HTTPS die Vertraulichkeit der Verbindung geschützt. Da die Sicherheit des TUF-Repositories von der Sicherheit des Servers abhängt auf dem die Schlüssel gespeichert sind, muss dieser auch ausreichend geschützt sein. Des Weiteren ergibt es Sinn den Root-Schlüssel, der selten benötigt wird, nicht auf dem Server zu speichern, sondern zum Beispiel auf einem verschlüsselten USB-Stick. Mithilfe des Root-Schlüssels kann im Falle des Verlusts eines anderen Schlüssels dessen Gültigkeit entzogen werden.

## 3.3 Client


**Abbildung 3.5:** Containerdiagramm des Clients

Der Client hat die Aufgabe, die von dem Benutzer konfigurierten Apps herunterzuladen und anschließend auszuführen (siehe Abbildung 3.5). Da das Gerät durch den Nutzer nicht gewartet werden soll, befindet sich zusätzlich zur App-Verwaltung ein *Supervisor* [13] Prozess-Überwachungs-System auf dem Gerät.
Supervisor ist ein Prozess-Überwachungs-System, mit dem Prozesse beim Hochfahren gestartet werden können. Zusätzlich übernimmt das System das Logging von Ereignissen sowie Fehlern. Im Falle des Absturzes eines überwachten Prozesses kann Supervisor diesen neu starten.

### 3.3.1 App-Verwaltung

Die App-Verwaltung bezieht in regelmäßigen Abständen Information von der REST API, welche Apps auf dem Gerät installiert sein sollen. Muss eine neue App installiert werden, wird diese über den TUF-Client aus dem TUF-Repository bezogen. Der TUF-Client, eine mit dem TUF-Framework entwickelte Komponente, übernimmt den Download sowie die Überprüfung der Metadaten. Sobald das Paket bereit steht, wird dieses von der Installationskomponente entpackt und in Form eines Kind-Prozesses gestartet. Wird die App von dem Benutzer wieder deinstalliert, beendet das Programm den Kind-Prozess und löscht alle Dateien der App. Benötigt die App zusätzlich zum Quellcode andere Abhängigkeiten, werden diese von der App-Verwaltung aus dem Internet bezogen. Die Abhängigkeiten werden hiefür vom App-Entwickler in die Metadaten der App eingetragen. Das IoT-Managementsystem unterstützt Abhängigkeiten aus apt [20] und pip [25]. Abbildung 3.6 bietet eine Übersicht über die Beziehungen der Komponenten.

## 3.4 Risikoanalyse

Eine Risikoanalyse untersucht und bewertet Gefahren für eine Organisation, ein Projekt oder ein System. Die folgenden Seiten analysieren Gefahren für das IoT-Managementsystem

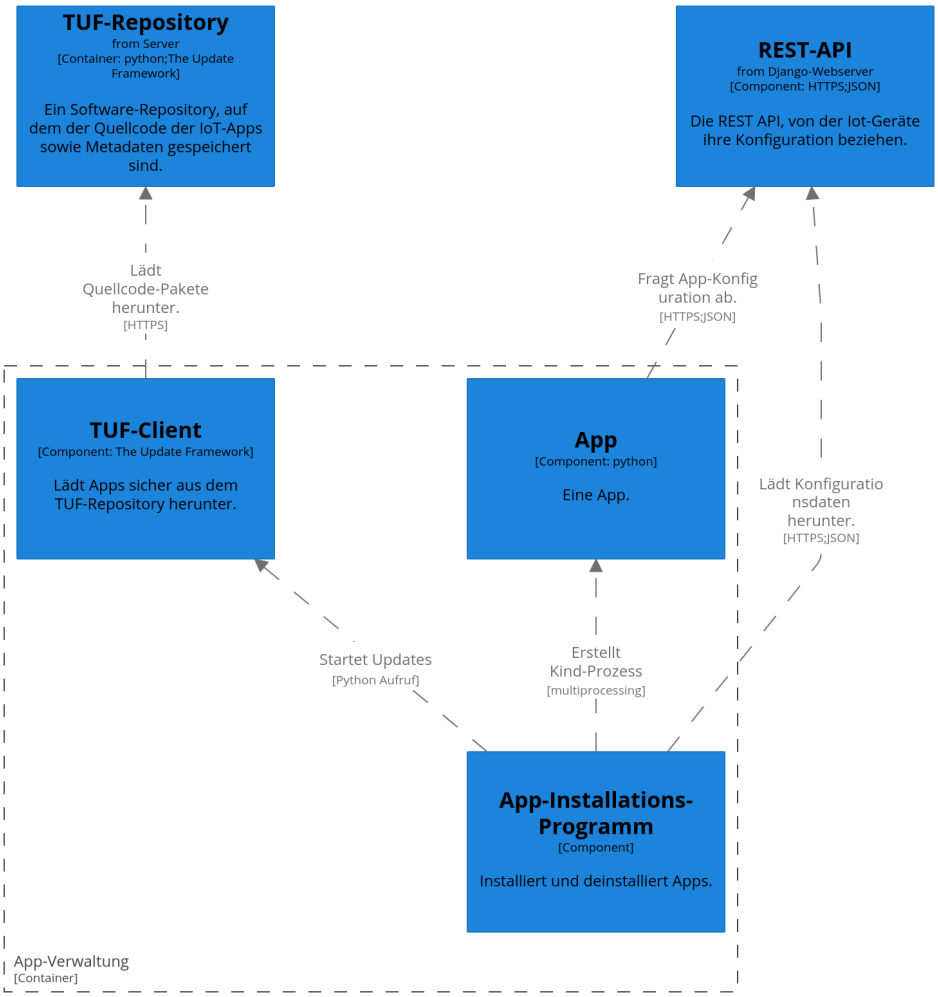


**Abbildung 3.6:** Die Komponenten der App-Verwaltung

und bewerten diese. In Abschnitt 5.3.1 wird auf Maßnahmen, die zur Eindämmung dieser Risiken ergriffen werden sowie vorhandene RESTrisiken eingegangen.

Sicherheitsanforderungen Eine IoT-Managementlösung ist sicher, wenn ein Angreifer diese nicht missbrauchen kann, um an vertrauliche Informationen zu gelangen oder ein Gerät zu kompromittieren, und so zum Beispiel Spam-Mails zu verschicken.

STRIDE Bei STRIDE [22] handelt es sich um ein 1999 von Loren Kohnfelder und Praerit Garg bei Microsoft entwickeltes Modell zur Entdeckung von IT-Sicherheitsrisiken. Gefährdungen werden hierbei in die Kategorien „spoofing of user identity“, „tampering with data“, „repudiability“, „information disclosure“, „denial of service“ und „elevation

of privilege“ eingeteilt.

### 3.4.1 Django Webserver

Der Django Server ist ein typischer Webserver. Als solcher ist er frei im Internet erreichbar und ist dementsprechend auch Risiken ausgesetzt. In diesem Abschnitt werden die den Webserver betreffenden Risiken erläutert.

Spoofing of user identity Bei Spoofing-Angriffen meldet sich ein Angreifer mittels valider Anmeldeinformationen eines anderen Nutzers an und versucht daraufhin, Schaden zu verursachen. Diese Anmeldeinformationen kann der Angreifer über Phishing, Abhören unverschlüsselter Kommunikation oder durch Ausprobieren erhalten. Des Weiteren werden sowohl zwischen dem Webbrowser des Benutzers als auch dem IoT-Gerät und dem Server Zugangsdaten ausgetauscht. Erhält ein Angreifer Benutzerdaten, kann er sich als dieser ausgeben.

Tampering with data Die unerlaubte Manipulation von Daten ist sowohl mithilfe eines Hackerangriffs auf den Server als auch auf dem Übertragungsweg durch einen Man-in-the-Middle-Angriff eine Gefahr für den Webserver.

Repudiability Abstreitbarkeit beziehungsweise fehlende Nachvollziehbarkeit stellen eine Gefahr für alle Systeme dar, in denen Änderungen vorgenommen werden können. Wird die Konfiguration einer App ohne das Wissen des Besitzers geändert, entstehen Gefahren für die Sicherheit seiner Infrastruktur.

Information disclosure Die Offenlegung privater Informationen ist eine Gefahr, von der die meisten IT-Systeme betroffen sind. Bei dieser Arbeit sind vor allem die Zugangsdaten der Benutzer und Konfigurationsdaten der Apps zu schützen. Im Falle der Apps entsteht der Schutzbedarf durch die häufig sensiblen Daten, die von IoT-Anwendungen verarbeitet werden. Zu nennen sind hier beispielsweise Kamerabilder, Gesundheitsdaten oder Firmengeheimnisse.

Denial of Service Ein DoS-Angriff beansprucht so viele Ressourcen eines Systems, dass dieses nicht mehr seiner eigentlichen Arbeit nachkommen kann. Der Webserver des Systems ist wie jeder Webserver gegen diese Angriffe in Form endloser Anfragen anfällig, falls der Angreifer genug Ressourcen besitzt.

### 3.4.2 TUF-Repository

Für das TUF-Repository bestehen andere Gefahren als für den Webserver. Das Repository ist zwar ebenfalls frei im Internet erreichbar, bietet jedoch weniger Angriffsfläche, da es keine Interaktion mit dem Nutzer erlaubt. Daten werden hier lediglich für Machine-to-Machine(M2M)-Kommunikation bereitgestellt.

Tampering with data Daten, die vom Repository an das Gerät übermittelt werden, können durch einen Man-in-the-Middle-Angriff verändert werden. Außerdem sind die Daten auf dem Server durch Hackerangriffe bedroht.

Information disclosure Der Quellcode der Apps ist auf dem Server öffentlich einsehbar. Angreifer können mit dieser Information Angriffe planen. Auch die Übertragung über das Internet kann von Angreifern mitgeschnitten werden.

Denial of Service Das Repository ist von DoS-Angriffen in gleichem Maße betroffen wie der Webserver. Auch die Client-Komponente des TUF-Frameworks ist von DoS-Angriffen bedroht (vlg. Abschnitt 2.1.2).

# Kapitel 4

# Implementierung

Dieses Kapitel beschäftigt sich mit der Umsetzung der im vorherigen Kapitel beschriebenen Architektur. Insbesondere wird hier auf das App-Konzept, als auch auf die Anbindung der verwendeten Frameworks Django und TUF eingegangen. Für die Umsetzung wurde die Programmiersprache Python3 gewählt.

## 4.1 App-Konzept

Das App-Konzept ist eine während der Implementierung des Clients entstandene Idee, um das IoT-Managementsystem einfach und flexibel an verschiedene Anwendungsfälle anpassen zu können. Eine in Python entwickelte Anwendung kann ohne großen Aufwand in das Managementsystem integriert werden. Zusätzlich zum App-Quellcode für das IoT-Gerät muss vom Entwickler nur eine Online-Maske für die Eingabe von Einstellungen der App bereitgestellt werden. Der dabei entstandene Mehraufwand ist im Vergleich mit einer eigenständigen Entwicklung sehr gering und aufgrund der vielen Vorteile des Managementsystems zu vernachlässigen.

### 4.1.1 Struktur

Eine App der IoT-Managementlösung hat folgende Bestandteile:

**main.py** Von diesem Skript aus wird die Anwendung geladen und beendet.

**metadata.json** In dieser Datei werden Informationen über die App gespeichert.

**Weitere Ressourcen** Der Entwickler kann die App um beliebig viele weitere Dateien ergänzen. Dabei kann es sich zum Beispiel um weitere Skripte oder Daten (XML,JSON) handeln.

**config.json** Diese Datei muss nicht vom App-Entwickler bereitgestellt werden. Sie wird nach der Installation von der App-Verwaltung zyklisch bereitgestellt und enthält

die aktuelle, durch den Benutzer auf dem Server erstellte, Konfiguration.

Für die Speicherung auf dem Server und Übertragung werden alle Dateien der App außer config.json zu einer Zipdatei (einem Paket) komprimiert. Auf dem IoT-Gerät wird diese entpackt. Der so entstandene Ordner dient der App als Arbeitsverzeichnis.

### 4.1.2 Technische Anforderungen

Eine Anwendung muss bestimmte Voraussetzungen erfüllen, um von der IoT-Managementlösung verwaltet werden zu können. Genauso wie das restliche System werden auch Apps in Python3 geschrieben. Sind aus technischen Gründen andere Sprachen notwendig können, diese über das Linux-Betriebssystem mittels Bash-Aufrufen angebunden werden. Der Entwickler kann die App größtenteils frei gestalten. Er muss sich lediglich an folgende Rahmenbedingungen halten:

**Startpunkt** Jede App muss eine main.py Datei enthalten. Aus dieser wird die `run()`-Funktion durch die App-Verwaltung bei Start der App aufgerufen. Von hier aus kann der Entwickler seine eigenen Funktionen aufrufen (vlg. Listing 1, Zeile 12).

**Deinstallation** Das IoT-Gerät sollte nach der Deinstallation von Apps wieder in seinen Ursprungszustand versetzt werden. Vor der Deinstallation sendet die App-Verwaltung der App das Signal `SIGTERM`. In dem dazugehörigen Handler kann der Entwickler den Code zum Zurücksetzen des Systems implementieren (vlg. Listing 1, Zeile 7).

```
1  import signal
2  import subprocess
3  import sys
4
5  CONFIG_FILE = 'config.json'
6
7  def handler(signum, frame):
8      #Code hier wird vor dem Deinstallieren der App ausgeführt.
9      #Eventuell vorgenommene Änderungen am System sollten hier
10     #Rückgängig gemacht werden.
11     sys.exit()
12 def run():
13     signal.signal(signal.SIGTERM, handler)
14     #Registriert Handler für das Deinstallieren.
15     #App-Code hier.-----------------------------------------------
```

**Listing 1:** Template für main.py

**Abhängigkeiten von Bibliotheken** Heutzutage benötigen die meisten Anwendungen Programmbibliotheken, um ausgeführt werden zu können. Um App-Entwickler diese Werkzeuge zur Verfügung zu stellen, bietet das IoT-Managementsystem die Möglichkeit, Abhängigkeiten von apt [20] und pip [25] zu laden. Dazu muss lediglich in den Metadaten des Pakets der Name der entsprechenden Bibliothek eingetragen werden (vgl. Listing 2).

**Konfiguration** In der metadata.json Datei (vlg. Listing 2) muss außerdem die URL angegeben sein, unter welcher die Einstellungen der App geladen werden können.

```
{
    "app_name": "App1V1",
    "apt": [],
    "pip": [],
    "config_url" : "/rest/iptables-conf/"
 }
```

**Listing 2:** metadata.json

## 4.2 Einbindung des TUF-Frameworks

Das TUF-Framework (TUF) ist sowohl server- als auch clientseitig für den Updateprozess zuständig. TUF besitzt sowohl eine Kommandozeilen- als auch eine Python-Schnittstelle, letztere bietet einen größeren Funktionsumfang und lässt sich auch besser in eine Python-Anwendung integrieren. Aus jenen Gründen wurden für diese Arbeit zwei Python-Skripte entwickelt, die TUF an das restliche System anbinden.

### 4.2.1 Grundlegende Funktionsweise

Das TUF-Framework bietet Funktionalität an, mit dessen Hilfe Metadaten erstellt, signiert und auch überprüft werden können. Dazu werden kryptografische Schlüssel benötigt. Diese können zu Beginn eines Projekts mit dem Konsolenaufruf `repo.py --init` generiert werden. TUF erstellt daraufhin die Ordner „tufclient“, „tufrepo“ und „tufkeystore“. Darin befinden sich die für Client und Server notwendigen Metadaten und die kryptografischen Schlüssel. Um eine neue Datei (target) auf dem Server bereitzustellen, muss diese in den Ordner `/tufrepo/targets` des des Servers kopiert werden. Zusätzlich werden die Metadaten angepasst und signiert. Um die „targets“ und Metadaten dem Client zur Verfügung zu stellen, wird der gesamte Ordner „tufrepo“ über einen HTTP-Server veröffentlicht. Der Client kann den Ordner herunterladen und die Authentizität

anhand der Metadaten überprüfen. In Abbildung 4.1 wird der gesamte Prozess beispielhaft mithilfe der Kommandozeilen-Schnittstelle veranschaulicht. Um den Updateprozess automatisiert durchführen zu können, wurden zwei Python-Skripte entwickelt.


**Abbildung 4.1:** Beispielhafter Aktualisierungs-Ablauf mit TUF-CLI

### 4.2.2 Server

Für das Hinzufügen und Entfernen von Dateien zu/aus dem Repo-Verzeichnis und den entsprechenden Änderungen der Metadaten wurde das Skript *myRepo.py* entwickelt. Das Skript unterstützt die Funktionen `add_conf(file)` und `delete_conf(file)`. Parameter, wie der Pfad des Ordners, in dem sich die kryptografischen Schlüssel befinden, können über Konstanten am Anfang des Skripts eingestellt werden. So bietet das Skript eine komfortable und erweiterbare Alternative zu der Kommandozeilen-Schnittstelle. Der eigentliche HTTPS-Server des Repositories wurde mit der Python-Bibliothek *http.server* sowie dem Webserver *nginx* [16] entwickelt. Um die Sicherheit der Clients im Fall eines

erfolgreichen Angriffs auf den Server zu erhalten, sollten die Schlüssel nicht auf dem Server gespeichert werden. Wird der komplette Prozess auf einem sicheren Computer (z.B. ohne Internet) durchlaufen und nur der Inhalt des Ordners „tufrepo“ nachträglich auf den HTTPS-Server kopiert, bleiben die Schlüssel sicher. Ebenfalls sicher ist es, den Prozess auf dem Server durchzuführen, die Schlüssel allerdings nicht dort zu speichern. Stattdessen können diese auf einem USB-Stick mitgebracht werden.

### 4.2.3 Client

Die clientseitige Implementierung der TUF-Komponente wird anders als das Repository nicht von einem Menschen, sondern automatisiert aufgerufen. Darum wurde hierfür die Funktion `update_conf(filename, host)` entwickelt. Die Funktion versucht die gegebene Datei von dem Host herunterzuladen. Wurde die Datei manipuliert oder treten sonstige Fehler auf, werden Exceptions geworfen, die von der App-Verwaltung des IoT-Geräts verarbeitet werden können. Ist eine neue Version des „targets“ verfügbar, wird diese für die App-Verwaltung erreichbar gesichert und der Rückgabewert *True* zurückgegeben. Ist keine neue Version vorhanden, wird *False* zurückgegeben.

## 4.3 Django-Webserver

Die Verwendung eines Frameworks für die Entwicklung des Webservers bringt neben einem geringeren Implementierungsaufwand noch weitere Vorteile. So rät die OWASP zur Nutzung von Frameworks: „Die Entwicklung starker und anwendbarer Sicherheitsmaßnahmen ist nicht trivial. Standardisierte Sicherheitsmaßnahmen vereinfachen die Entwicklung sicherer Anwendungen oder APIs.“ [24]. Auch aus der Perspektive eines App-Entwicklers bringt Django Vorteile mit sich. Um die Konfigurationsmaske einer App zu entwickeln, muss der Entwickler sich nicht in den Quellcode dieser Arbeit einarbeiten. Es genügt grundlegende Django-Kenntnisse sowie ein Verständnis für das Datenmodells des Webservers zu besitzen.

### 4.3.1 Datenmodell

Django ist ein stark datenbankgestütztes Framework. Die Definition des Datenmodells findet in Python-Dateien statt, so kann auch ohne SQL-Kenntnisse ein Datenmodell entworfen werden. Das Framework beinhaltet hierfür verschiedene Klassen, mit denen aus Python-Objekten ein relationales Datenmodell erstellt wird. Für den App-Store muss das IoT-Gerät sowie die IoT-App modelliert werden. Ein Benutzer-Datenmodell ist bereits in Django vorhanden. Das Modell des App-Stores ist einfach gehalten. Das Gerätemodell (siehe Listing 3, Zeile 14) hat eine UUID zur Identifikation, ein verknüpftes

Benutzerkonto und eine Liste installierter Apps.

```
1  from django.contrib.auth.models import User
2  from django.db import models
3
4
5  class IoTApplication(models.Model):
6      id = models.fields.IntegerField(
7          primary_key=True, auto_created=True)
8      name = models.CharField(max_length=50)
9      version = models.IntegerField()
10
11     def __str__(self):
12         return str(self.name)
13
14
15 class Device(models.Model):
16     uuid = models.IntegerField(primary_key=True)
17     owner = models.ForeignKey(User, null=True,
18                               on_delete=models.CASCADE)
19     applications = models.ManyToManyField(
20         IoTApplication, blank=True, related_name='App')
21
22     def __str__(self):
23         return 'device: ' + str(self.uuid)
```

**Listing 3:** Datenmodelldefinition in models.py

Das Modell einer App (siehe Listing 3, Zeile 5) besteht aus deren Name, der Id sowie Versionsnummer. Diese Informationen werden sowohl zur Anzeige im App-Store als auch von der App-Verwaltung des IoT-Geräts benötigt.

### 4.3.2 Webinterface

Über das Webinterface kann der Benutzer Apps auf seinem IoT-Gerät IoT-Gerät installieren sowie konfigurieren. Bevor der Benutzer Änderungen vornehmen kann, muss er sich zuerst authentifizieren. Der dafür notwendige Code ist bereits im Framework enthalten und muss lediglich in das Projekt eingebunden werden. Nach einer erfolgreichen Anmeldung hat der Benutzer Zugriff auf einen Überblick der verfügbaren Apps. Diese kann er über ein Eingabefeld auf seinem Gerät installieren. Für jede installierte App erscheint

ein Eintrag in der Seitenleiste, über diesen wird der Benutzer zu der entsprechenden Konfigurationsmaske weitergeleitet (vlg. Abbildung 4.2 dort ist *FIREWALL-IPTABLES* installiert und *Another-App* verfügbar). Django generiert HTML-Seiten aus HTML-Templates und Python-Funktionen, sogenannte „views“. In einer „view“ kann auf die Datenbank sowohl lesend als auch schreibend zugegriffen und darüber hinaus Berechnungen ausgeführt werden. Ein „view“ liefert als Ergebnis eine HTML-Seite, die aus den berechneten respektive gelesenen Werten und einem Template zusammengebaut wurde.

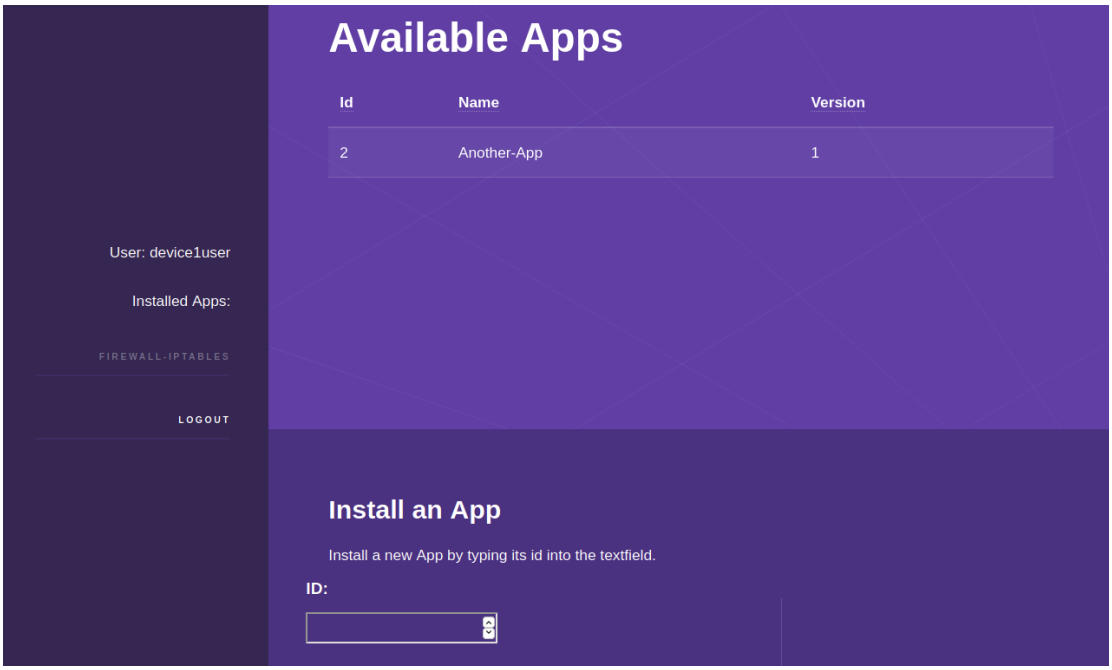


**Abbildung 4.2:** Das Webinterface

### 4.3.3 REST-API

Die REST-API bildet das Gegenstück zum Webinterface. Von Benutzern erstellte Konfigurationen, können hier von den Geräten im JSON-Format (vgl. Listing 5) heruntergeladen werden. Für die Entwicklung der API wurde das *Django REST Framework* [14] verwendet. Mit dem Framework können Daten und ihre Beziehungen zueinander serialisiert und, wie für Django üblich, als „view“ unter einer URL verfügbar gemacht werden. Auch Module, die den Zugriff auf die API kontrollieren, sind Teil des Frameworks. Damit wurde ein Berechtigungskonzept (Listing 4, Zeile 2) entwickelt, mit dem nur der Benutzer des IoT-Geräts dessen Konfiguration abfragen kann. Die API des IoT-Managementsystems besteht aus einer API mit allgemeinen Geräteinformationen (Stage1, siehe Listing 4), auf der die Geräte eine Liste aktuell installierter Anwendungen erhalten, sowie App

spezifischen APIs, die pro App entwickelt werden. Darüber kann das IoT-Gerät App-spezifische Informationen abfragen oder Sensordaten in die Datenbank schreiben.
Für die Klassen IoTApplication und Device des Datenmodells wurden entsprechende Serialisierer entwickelt, die die zu serialisierenden Felder definieren. In der „view“ werden die serialisierten Daten zusätzlich mit *JavaScript Object Signing and Encryption (JOSE)* [12] symmetrisch signiert, so kann das IoT-Gerät die Authentizität der Konfiguration prüfen.

```
1       class Stage1DeviceDetail(APIView):
2       permission_classes = [ownPermissions.IsOwner,
3                 permissions.IsAuthenticated]
4       def get_object(self, pk):
5           try:
6               return Device.objects.get(pk=pk)
7           except Device.DoesNotExist:
8               raise Http404
9
10      def get(self, request, pk, format=None):
11          device = self.get_object(pk)
12          self.check_object_permissions(self.request, device)
13          serializer = Stage1DeviceSerializer(device)
14          data = serializer.data
15          data = sign(data)
16          return Response(data)
```

**Listing 4:** Mit dem REST Framework erstellte View

```
{ "uuid": 12,
  "host": "https://dns-rpz.cs.hm.edu:8001",
  "applications": [
    {
      "id": 1,
      "name": "Firewall-iptables",
      "version": 1
    }
  ]
}
```

**Listing 5:** Beispielhafte Konfiguration eines Geräts im JSON-Format

## 4.4 App-Verwaltung

Bei der App-Verwaltung handelt es sich um eine auf dem IoT-Gerät laufende Endlosschleife, die zyklisch die aktuelle Konfiguration des Geräts abruft und entsprechend Apps über den TUF-Client herunterlädt und diese verwaltet.

Um verschiedene Apps zentral verwalten zu können, kommt die Python-Bibliothek multiprocessing [17] zum Einsatz. Mit multiprocessing kann ähnlich wie mit Threads Code parallel abgearbeitet werden. Apps als Prozesse und nicht als Threads abzubilden, bringt einige Vorteile. So entsprechen Prozesse konzeptionell einer App stärker als Threads. Durch die Unabhängigkeit von Prozessen besteht im Gegensatz zu Threads ein höheres Maß an Sicherheit. Außerdem können Prozessverwaltungstechniken wie Signale eingesetzt werden. Diese Funktion wird bei der Deinstallation von Apps genutzt.

Wenn eine neue App installiert wird, muss diese erst mit TUF heruntergeladen, entpackt, und während der Laufzeit in die Anwendung importiert werden. Anschließend wird die neue importierte Funktion gestartet (vlg. Listing 6). Das beim Entpacken entstandene Verzeichnis dient der App fortan als Arbeitsverzeichnis, in dem sie Dateien ablegen kann. Außerdem wird dort zyklisch die aktuelle Konfiguration der App hinterlegt. Bei der Deinstallation der App wird das Verzeichnis wieder gelöscht.

```
def _importer(name, root_package=False, relative_globals=None, level=0):
    return __import__(name, locals=None,  # locals has no use
                      globals=relative_globals,
                      fromlist=[] if root_package else [None],
                      level=level)

def start_app(app_dirname):
    try:
        import_statement = '{}.main'.format(app_dirname)
        app = _importer(import_statement, root_package=True)
        p = Process(target=app.main.run, args=())
        p.start()
    except:
        logging.error("Error Starting " + app_dirname)
        return
    return p
```

**Listing 6:** Funktionen zum Import und Start einer App

### 4.4.1 Prozessüberwachung

Um eine einfache Bedienung des IoT-Geräts zu ermöglichen, wurde Supervisor, ein System zur Prozessüberwachung, eingesetzt. Supervisor besteht aus zum einen *Supervisord*, einem Daemon-Prozess, der andere Programme als Kind-Prozesse laufen lässt, zum anderen *Supervisorctl*, einem Client-Programm, das den Status der Supervisord-Kind-Prozesse kontrolliert und mitloggt. Um die App-Verwaltung zu überwachen, muss auf dem IoT-Gerät Supervisor installiert werden. Dies kann zum Beispiel über die Paketverwaltung apt geschehen. Außerdem muss unter `/etc/supervisord.conf` ein Eintrag für den Startbefehl der Appverwaltung hinterlegt werden (Listing 7). Dort können zusätzlich Einstellungen bezüglich Logging und dem Startverhalten der App eingetragen werden. Um die App-Verwaltung bei Einschalte des Geräts mitzustarten, wurde die Einstellung autostart gewählt. Zusätzlich kann mit *autorestart* und *startretries* festgelegt werden, wie oft die App-Verwaltung im Falle eines Absturzes neu gestartet werden soll. Diese Einstellungen sind wichtig, da das Gerät ohne menschliche Interaktion lange und stabil laufen soll.

```
[program:IoT_Client]
command= python3 /home/client/client_main.py
directory=/home/client
autostart=true
autorestart=true
startretries=5
stderr_logfile=/var/log/iot/iot.err.log
stdout_logfile=/var/log/iot/iot.out.log
user=root
```

**Listing 7:** supervisord.conf auf dem IoT-Gerät

## 4.5 Betrieb

Um den Django- und TUF-Server zu betreiben, wurde von der Hochschule München ein Debian-Server zur Verfügung gestellt. Dort werden mittels nginx die für das IoT-Managementsystem nötigen Dienste angeboten. Das für HTTPS-Verbindungen nötige X.509-Zertifikat wurde von der kostenlosen Zertifizierungsstelle *Let's Encrypt* ausgestellt. Damit können sowohl die Verbindungen zwischen dem Django-Server und dem Webbrowser des Benutzers als auch zwischen dem IoT-Gerät und dem Repository verschlüsselt werden. Mit nginx wird der Django Server auf dem, für HTTPS üblichen Port 443 angeboten. Zusätzlich wurde eine Weiterleitung von Port 80 auf 443 eingerichtet, um

HTTP-Anfragen umzuleiten. Das TUF-Repository wird auf dem Port 8001 betrieben (siehe Listing 8).
Alternativ zu dem oben beschriebenen Betrieb in der Cloud ist auch ein Betrieb des Servers vor Ort möglich. So könnten beispielsweise alle Geräte auf einem Firmengelände verwaltet werden.

```
upstream django {
    server unix:///home/ll/git/Secure_WEbserver/mysite.sock; # for a file socket
}
server{ #HTTP -> HTTPS Umleitung
    listen 80;
    server_name dns-rpz.cs.hm.edu;
    return 301 https://dns-rpz.cs.hm.edu$request_uri;
}
server {#HTTPS Server
    listen      443 ssl;
    server_name dns-rpz.cs.hm.edu;
    ssl_certificate /etc/letsencrypt/live/dns-rpz.cs.hm.edu/fullchain.pem;
    ssl_certificate_key  /etc/letsencrypt/live/dns-rpz.cs.hm.edu/privkey.pem;
    charset     utf-8;
    client_max_body_size 75M;
    # Django media
    location /media  {
        alias /home/ll/git/Secure_WEbserver/templates;  #Django project media files

    location /static {
        alias /home/ll/git/Secure_WEbserver/static; #Django project static files
    }

    location / {
        uwsgi_pass  django;
        include     /home/ll/git/Secure_WEbserver/uwsgi_params; #the uwsgi file
        }
    }
server{ #TUF-Repo
        listen 8001 ssl;
        server_name dns-rpz.cs.hm.edu;
        ssl_certificate /etc/letsencrypt/live/dns-rpz.cs.hm.edu/fullchain.pem;
        ssl_certificate_key  /etc/letsencrypt/live/dns-rpz.cs.hm.edu/privkey.pem;
        location / {
                proxy_pass http://localhost:8000/;
        }

}
```

**Listing 8:** Die Nginx-Konfiguration auf dem Server

# Kapitel 5

# Evaluierung

Das Ziel dieser Arbeit ist, eine IoT-Managementlösung zu entwickeln, die sowohl für App-Entwickler als auch Endbenutzer leicht zu verwenden ist und ein hohes Maß an Sicherheit garantiert. Inwieweit das entwickelte System diesen Ansprüchen genügt, wird in diesem Kapitel überprüft. Bei der Hardware handelt es sich um einen Debian 10-Server der Hochschule München und ein Raspberry Pi 4 Model B, das als IoT-Gerät fungiert.

## 5.1 Integration einer neuen App

Für die Evaluierung des App-Konzepts wurde das Linux Firewall Programm *iptables* gewählt. Mit iptables können Regeln für ein- und ausgehende Netzwerkpakete erstellt werden. Durch die Integration in das Managementsystem können die Firewall-Regeln über ein Webinterface erstellt werden und müssen nicht über die Kommandozeile des Raspberry Pi eingegeben werden.

### 5.1.1 Iptables-Firewall

Der erste Schritt der Entwicklung ist das Programmieren einer Anwendung, die die gewünschte Aufgabe erfüllt. In diesem Fall ist die Aufgabe das Erstellen von Netzwerkregeln. Wie in Abschnitt 4.1 erklärt wird, übermittelt das Managementsystem die Einstellungen über eine JSON-Datei. Das sollte bereits bei der Entwicklung berücksichtigt werden, so müssen später weniger Anpassungen vorgenommen werden.
Listing 9 zeigt den Quellcode der iptables-Firewall. Darin werden zuerst alte Konfigurationen gelöscht (Zeile 8,9), anschließend wird für jede Einstellung der entsprechende Befehl erstellt und ausgeführt (Zeile 10-19).

```
1  def execute_config():
2      """Loads a iptables configuration from config.json
3       and executes it."""
4      script_dir = os.path.dirname(__file__)
5      with open(os.path.join(script_dir,'config.json')) as json_file:
6          data = json.load(json_file)
7          data = data["ip_rule_set"]
8          _call('echo \"firewall App running\"\n')
9          _call('iptables -F\n')
10         _call('iptables -X\n')
11         for date in data:
12             command = "iptables -A {} ".format(date['chain'])
13             if date['packet_type'] != "all":
14                 command += "-p {} ".format(date['packet_type'])
15
16             if date['destination_port'] > 0:
17                 command += "--dport {} ".format(date['dest_port'])
18             command += "-j {}\n".format(date['action'])
19             try:
20                 _call(command) #/bin/bash call
21             except subprocess.CalledProcessError as error:
22                 logging.error("Error applying firewall rule."
23                     .format(error.stderr))
```

**Listing 9:** Die Iptables-Funktion

### 5.1.2 Anpassungen

Nach der Entwicklung und dem Testen der App kann diese durch wenige Anpassungen in das IoT-Managementsystem eingebunden werden. Dazu muss Listing 9 um eine `run()` Methode zum Starten und einen Handler für die Deinstallation erweitert werden (vlg. Listing 10).

```
def handler(signum, frame):
    logging.critical("uninstall iptables firewall")
    _call('iptables -F\n')  # delete all firewall configurations
    sys.exit()
```

```
def run():
    signal.signal(signal.SIGTERM, handler)
    while 1:
        execute_config()
        time.sleep(20)
```

**Listing 10:** Anpassungen an das App-Konzept

### 5.1.3 Anbindung an den Appstore

Um die App über das Webinterface konfigurieren zu können, muss noch eine entsprechende Maske für Django programmiert werden. Dazu wird, wie es für Django typisch ist, ein Modell der Daten erstellt (Listing 11). Außerdem werden „views“ und dazugehörige HTML-Templates benötigt. Mithilfe der „view“ kann ein Benutzer die Konfiguration des IoT-Geräts erstellen. Zuletzt wird mit dem REST-Framework eine Schnittstelle programmiert, über die das Gerät die Konfiguration herunterladen kann.

```
class IPRule(models.Model):
    id = models.AutoField(primary_key=True)
    device = models.ForeignKey(Device, on_delete=models.CASCADE,
                    related_name='ip_rule')
    chain = models.CharField(max_length=6)  # INPUT/OUTPUT
    destination_port = models.IntegerField()
    packet_type = models.CharField(max_length=10) #TCP,UDP,..
    action = models.CharField(max_length=15)  # Drop,..
```

**Listing 11:** Modell der Iptables-Daten

### 5.1.4 Fazit

Die Anpassungen, die vorgenommen werden müssen, um eine App in das Managementsystem zu integrieren sind überschaubar. Für die in Abschnitt 5.1.2 vorgenommenen Änderungen genügen einfache Python-Kenntnisse. Für die Anpassungen aus Abschnitt 5.1.3 werden grundlegende Kenntnisse des Django-Frameworks benötigt. Abhängig von der Komplexität der Maske und der Erfahrung des Entwicklers sollten die oben aufgeführten Anpassungen innerhalb mehrerer Stunden durchführbar sein.

## 5.2 Inbetriebnahme des IoT-Geräts

Ein Ziel der Entwicklung ist es, die Inbetriebnahme für den Endbenutzer so einfach wie möglich zu gestalten. Um ein mit der Client-Software ausgestattetes IoT-Gerät in Betrieb zu nehmen, muss der Benutzer dieses lediglich mit Strom und einer Netzwerkverbindung, wie in Abbildung 5.1 dargestellt, versorgen. Daraufhin fährt dieses selbstständig hoch und startet anschließend die App-Verwaltung und alle installierten Apps.
Dieser Vorgang ist sehr einfach und kann auch von technisch unversierten Personen durchgeführt werden können. Stromversorgung über Ethernet würde das Einschalten weiter vereinfachen, dann müsste nur noch ein Kabel eingesteckt werden.
Die anschließende Konfiguration des Geräts über das Webinterface ist mit der Bedienung einer herkömmlichen Website vergleichbar.

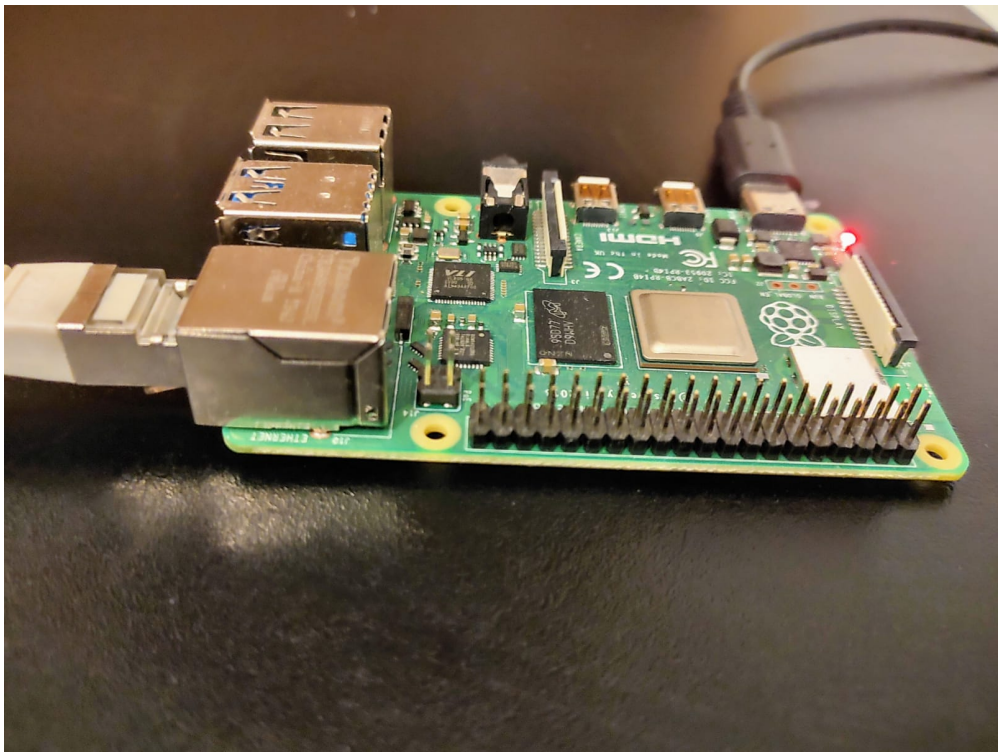

**Abbildung 5.1:** Ein Raspberry Pi 4 Model B mit USB-C Stromversorgung und Ethernet-Kabel

## 5.3 Sicherheit

### 5.3.1 Sicherheitsanalyse

Der folgende Abschnitt diskutiert die Sicherheit dieser Arbeit sowie vorhandene Restrisiken. Grundlage dafür bildet die Risikoanalyse aus Abschnitt 3.4.

Spoofing of user identity Der Webserver speichert Passwörter durch Salt und Hashfunktionen geschützt und auch der Übertragungsweg zum Server ist durch HTTPS sicher. Es besteht jedoch weiterhin ein Risiko, falls der Benutzer ein unsicheres Passwort wählt oder dieses unsicher speichert. So kann ein Angreifer Zugangsdaten erhalten, um das System anzugreifen. Durch Richtlinien für das erstellen sicherer Passwörter kann das Risiko verringert werden.

Tampering with data Die Integrität der Daten des Webservers wird durch das X.509-Zertifikat geschützt. Die Daten des Repositories werden zusätzlich durch die Signaturen des TUF-Frameworks gesichert. Insgesamt sind die Daten des Systems gut gegen Manipulation geschützt. Verschafft sich ein Angreifer jedoch Zugang zum Server, kann dieser die Daten des Webservers beliebig verändern. Um den Quellcode des Repositories zu ändern, benötigt er zusätzlich die kryptografischen Schlüssel.

Repudiability Änderungen der Konfiguration durch den Webserver werden geloggt und bleiben so nachvollziehbar. Im Falle eines erfolgreichen Angriffs auf den Hostrechner des Servers könnten Konfigurationsdaten und Logdaten durch einen Angreifer geändert werden. Den Hostrechner zu schützen, liegt außerhalb des Umfangs dieser Arbeit.

Information Disclosure Der Quellcode der Apps liegt frei einsehbar auf dem Server. Angreifer können das nutzen, um Angriffe zu planen. Genauso kann quelloffene Software von jedermann auf Fehler überprüft werden. Die Vertraulichkeit von Informationen wird auf dem Übertragungsweg mittels HTTPS geschützt. So kann ein Angreifer nur schwer erkennen, welche Apps auf einem IoT-Gerät installiert sind.

Denial of Service Da sich der Client in seinem Einsatzgebiet häufig hinter einer Firewall befindet und durch TUF zusätzlich vor Endloser-Daten-Angriffen geschützt ist, ist das Risiko eines DoS-Angriffs sehr gering. Der Server allerdings kann durch DDoS Angriffe kurzzeitig außer Betrieb gesetzt werden. Ist der Server nicht erreichbar, erhält das IoT-Gerät keine zwar Aktualisierungen mehr , kann seiner Arbeit aber weiterhin nachkommen. Dies wäre nur kritisch, wenn ein Angreifer dies macht, um eine eigentlich geschlossene Angriffslücke durch einen Update-Stopp offen zu halten.

Elevation of privilege Die Django-Nutzerverwaltung ist ein sicheres System. Den Hostrechner gegen „elevation-of-privilege“-Angriffe zu schützen, ist nicht Teil dieser Arbeit.

### 5.3.2 Sicherheitstest

Zusätzlich zur theoretischen Sicherheitsanalyse wurden mit dem Tool OWASP ZAP Penetrationstests getätigt. Dabei wurde gegen Webserver ein automatisierter Scan im *ATTACK Mode* durchgeführt. Das Ergebnis zeigte bis auf Warnungen mit geringem Risiko aufgrund fehlender Header keine Gefährdungen. Auch der Quellcode des Webservers und des Clients wurden mit dem statischen Codeanalysetool Pylint auf Fehler überprüft. Die durchgeführten Tests sind nicht mit einem professionellen Penetrationstest vergleichbar, kombiniert liefern sie jedoch eine verlässliche Aussage über die Sicherheit des Systems.

# Kapitel 6

# Zusammenfassung

## 6.1 Ausblick

In dieser Arbeit wurden viele Grundlagen einer IoT-Managementlösung geschaffen. Aufgrund des limitierten Umfangs dieser Arbeit gibt es jedoch noch einige Änderungen, die vorgenommen werden sollten, bevor das IoT-Managementsystem effektiv eingesetzt werden kann.

Das TUF-Framework bietet ein weitreichendes Berechtigungskonzept. In dieser Arbeit wird allerdings nur eine einfache Variante gewählt. Sinnvoll wäre es, jedem App-Entwickler einen Schlüssel zuzuweisen, mit dem er den Quellcode seiner App signieren kann. Der *Python Package Index* (PyPI), ein Software-Repository für die Programmiersprache Python, nutzt TUF mit einem vergleichbaren Schlüssel-Konzept [23].

Damit das IoT-Gerät funktioniert, müssen Quellcode, kryptografische Schlüssel und eine Benutzerkennung darauf geladen werden. Bei einer größeren Anzahl an Geräten ist diese Arbeit, welche manuell zu erledigen ist, mit entsprechendem Aufwand verbunden. Diesen Schritt möglichst weit zu automatisieren, ist vor einem umfangreicheren Einsatz notwendig.

Um die Daten eines Benutzers auch im Falle der Kompromittierung des Servers zu schützen. kann das Managementsystem um „client-side encryption“ erweitert werden. Dabei wird die Konfiguration des Benutzers vor der Übermittlung an den Server mit dem Passwort des Benutzers verschlüsselt. Das IoT-Gerät, das das gleiche Passwort besitzt, kann diese Konfiguration nach Erhalt wieder entschlüsseln und verwenden.

## 6.2 Zusammenfassung

Die fortlaufende Technologisierung unseres Alltags und das damit einhergehende Wachstum der IoT-Branche erleichtert unser Leben in vielen Bereichen. Werden IoT-Geräte nicht ausreichend geschützt, entstehen für Unternehmen, Privatpersonen und die gesamte Gesellschaft viele neue Gefahren. In dieser Arbeit wurde der aktuelle Stand der Technik bezüglich der Administration solcher Geräte untersucht. Dabei konnte festgestellt werden, dass mithilfe kryptografischer Signaturen sowie einer durchdachten Vorgehensweise Updates sicher verteilt werden können. Dabei sollte, genauso wie bei der Entwicklung von Webanwendungen, auf standardisierte Sicherheitsmaßnahmen zurückgegriffen werden.

Daraufhin wurde aufgrund der gesammelten Informationen die Architektur eines IoT-Managementsystems vorgeschlagen, welche automatisierte Software-Updates für IoT-Geräte ermöglicht. Außerdem kann jedem Gerät eine individuelle Konfiguration übermittelt werden. Die Architektur umfasst eine client- und eine serverseitige Komponente. Der Server verwaltet die IoT-Geräte mithilfe des Django-Frameworks. Um Software-Updates sicher auf die Geräte zu verteilen, wird das TUF-Framework genutzt. Da die eingebundenen Frameworks die nötigen Sicherheitsvorkehrungen besitzen, mussten diese nicht selbst implementiert werden. So konnte ein komplizierter und fehleranfälliger Entwicklungsschritt vereinfacht werden. Um das Managementsystem flexibel an verschiedene Anwendungsfälle anpassen zu können, wurde ein App-Store-Konzept entwickelt. Die Integration einer neuen IoT-Anwendung erfordert lediglich leichte Anpassungen. Danach kann sie über den Server an alle verbundenen IoT-Geräte verteilt werden.

Die anschließende Evaluierung betrachtete die Anwendung aus verschiedenen Gesichtspunkten. Die Entwicklung und Integration einer exemplarischen IoT-App zeigten, dass dafür grundlegende Python-Kenntnisse ausreichen. Die Inbetriebnahme des IoT-Geräts erfordert lediglich das Einstecken zweier Kabel und sollte so jedem möglich sein. Die Sicherheit der Anwendung wurde aus zwei verschiedenen Gesichtspunkten betrachtet. Zum einen wurden mit einer nach dem STRIDE-Modell durchgeführten Sicherheitsanalyse Risiken für das System theoretisch bewertet. Zum anderen wurde der Webserver mithilfe praktischer Tests auf Schwachstellen überprüft.

Insgesamt liefert diese Arbeit eine Grundlage für die Entwicklung neuer Managementlösungen für IoT-Geräte und bietet eine Übersicht aktueller Entwicklungen im Bereich sicherer Software-Updates sowie dem Internet der Dinge.

# Abbildungsverzeichnis

# Quellenverzeichnis

## Literatur


[1] Vafa Andalibi, DongInn Kim und Jean Camp. „Throwing MUD into the FOG:Defending IoT and Fog by expanding MUD to Fog network“. *2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19)* (2019) (siehe S. 3).

[2] Justin Cappos u. a. „A look in the mirror: Attacks on package managers“. In: 2008 (siehe S. 5–7).

[3] Justin Cappos u. a. „Package management security“. *University of Arizona* (2008) (siehe S. 5).

[4] Justin Cappos u. a. „Survivable Key Compromise in Software Update Systems“ (2010) (siehe S. 14).

[5] Angelo Feraudo u. a. „SoK: Beyond IoT MUD Deployments – Challenges and Future Directions“ (siehe S. 4).

[6] IDG. *Studie "Internet of Things 2019/2020"*. Sep. 2019 (siehe S. 1).

[7] Matthew Maloney u. a. „Cyber Physical IoT Device Management Using a Lightweight Agent“. *2019 International Conference on Internet of Things (iThings)* (2019) (siehe S. 4).

[8] Mehdi Nobakht, Vijay Sivaraman und Roksana Boreli. „A Host-Based Intrusion Detection and Mitigation Framework for Smart Home IoT Using OpenFlow“. *2016 11th International Conference on Availability, Reliability and Security* (2016) (siehe S. 4).

[9] Oscar Novo. „Blockchain Meets IoT: an Architecture for ScalableAccess Management in IoT“. *JOURNAL OF INTERNET OF THINGS CLASS FILES, VOL. 14,* (2018) (siehe S. 3).

[10] Marc-Oliver Pahl. „Multi-Tenant IoT Service Managementtowards an IoT App Economy“ (siehe S. 3).

[11] Geert-Jan Schrijen, Georgios Selimis und Jan Jaap Treurniet. „Secure Device Management for the Internet of Things“ (siehe S. 2).

## Software

[12] Inc. Activision Publishing. *JOSE*. 2014. URL: https://jose.readthedocs.io/en (siehe S. 27).

[13] AgendalessConsulting und Contributors. *Supervisor*. 2004. URL: http://supervisord.org/ (siehe S. 16).

[14] Tom Christie. *Django REST framework*. URL: https://www.django-rest-framework.org/ (siehe S. 26).

[15] DjangoSoftwareFoundation. *Django*. 2005. URL: https://www.djangoproject.com/ (siehe S. 12, 13).

[16] Inc. F5. *nginx*. URL: https://www.nginx.com/ (siehe S. 23).

[17] Python. *multiprocessing — Process-based parallelism*. URL: https://www.nginx.com/ (siehe S. 28).

[18] PythonSoftwareFoundation. *Python*. 2001. URL: https://www.python.org/ (siehe S. 12).

## Online-Quellen

[19] E. Lear et al. *Manufacturer Usage Description (MUD)*. URL: https://tools.ietf.org/html/rfc8520 (besucht am 10. 07. 2020) (siehe S. 3).

[20] APT. *Advanced Packaging Tool*. URL: http://apt-rpm.org/ (besucht am 04. 06. 2020) (siehe S. 5, 16, 22).

[21] BSI. *Ops.1.1.3 patch- und änderungsmanagement*. 4. Juni 2020. URL: https://www.bsi.bund.de/DE/Themen/ITGrundschutz/ITGrundschutzKompendium/bausteine/OPS/OPS_1_1_3_Patch-_und_%C3%84nderungsmanagement.html (besucht am 04. 06. 2020) (siehe S. 5).

[22] Loren Kohnfelder und Praerit Garg. *The threats to our products*. URL: https://adam.shostack.org/microsoft/The-Threats-To-Our-Products.docx (besucht am 12. 06. 2020) (siehe S. 17).

[23] Trishank Karthik Kuppusamy, Vladimir Diaz und Justin Cappos. *PEP 480 – Surviving a Compromise of PyPI: The Maximum Security Model*. URL: https://www.python.org/dev/peps/pep-0480/ (besucht am 12. 07. 2020) (siehe S. 37).

[24] OWASP. *OWASP Top 10 -2017Die 10 kritischsten Sicherheitsrisiken für Webanwendungen(Deutsche Version 1.0)*. URL: https://wiki.owasp.org/images/9/90/OWASP_Top_10-2017_de_V1.0.pdf (besucht am 08. 06. 2020) (siehe S. 8, 9, 24).

[25] pip-development-team. *pip installs packages*. URL: pip-installer.org (besucht am 04. 06. 2020) (siehe S. 16, 22).

[26] Wikipedia. *File transfer protocol*. URL: https://de.wikipedia.org/wiki/File_Transfer_Protocol (besucht am 04. 06. 2020) (siehe S. 6).

[27] Wikipedia. *Fog computing*. URL: https://en.wikipedia.org/wiki/Fog_computing (besucht am 10. 07. 2020) (siehe S. 4).

[28] Wikipedia. *Hypertext transfer protocol*. URL: https://de.wikipedia.org/wiki/Hypertext_Transfer_Protocol (besucht am 04. 06. 2020) (siehe S. 6).