Freie Universität Berlin
Fachbereich Informatik und Mathematik
Institut für Informatik

Masterarbeit

Automatisierte, netzwerkbasierte und sichere Bereitstellung von Firmware Updates für smarte Systeme

Lars Hochstetter

lars.hochstetter@fu-berlin.de

Gutachter	Prof. Dr.-Ing. Jochen Schiller
	Dr.-Ing. Achim Liers
Betreuer	Dr. rer. nat. Enrico Köppe

Zusammenfassung

Das Internet of Things umfasst mehrere Milliarden Geräte mit Verbindung zum Internet, auch smarte Systeme genannt. Im Betrieb führen smarte Systeme Firmware aus, welche am Ende der Produktion in den Flash-Speicher des smarten Systems geflasht wird. Da smarte Syteme häufig an schwer zugänglichen Orten und in großen Stückzahlen eingesetzt werden, stellen Firmware Updates eine besondere Herausforderung dar. Um die Firmware zu aktualisieren, werden sogenannte OTA Updates verwendet. Dabei wird das Update über das Internet auf das smarte System übertragen und anschließend im Flash-Speicher gespeichert. OTA Updates stellen aber auch ein Ziel für Angreifer da. In dieser Arbeit wird ein Update-System entwickelt, welches den gesamten Prozess von der Kompilierung bis zur Verteilung von Firmware Updates für ein einzelnes smartes System automatisiert. Besonderes Augenmerk liegt dabei auf der Absicherung des Update-Prozesses gegen Angriffe und die Robustheit gegen Stromausfälle. Zuerst wird ein Modell des Update-Systems entworfen und mit dem CIA Modell Anforderungen an die Sicherheit des Update-Systems und des Update-Prozesses formuliert. Anschließend wird eine Bedrohungsanalyse mit dem STRIDE Bedrohungsmodell und dem Elevation of Privilege Kartenspiel durchgeführt. Für die gefundenen Bedrohungen werden, wo es sinnvoll ist, Schutzmaßnahmen ausgewählt und in das Update-System und den Update-Prozess integriert.

Das modellierte Update-System wird exemplarisch implementiert, wobei bevorzugt Open-Source-Komponenten eingesetzt werden. Als smartes System kommt der ESP32 der Firma Espressif Systems zum Einsatz, dessen Firmware auf Basis des esp-idf SDKs entwickelt wird. Der Quellcode wird mit Gitlab verwaltet und jede neue Version des Quellcodes wird automatisch in einem Docker Container kompiliert und anschließend über einen nginx Webserver verteilt.

Bei der Implementierung der Schutzmaßnahmen werden bevorzugt aktiv genutzte und weit verbreitete Lösungen gewählt, während die Eigenentwicklung von kryptographischen Algorithmen und Verfahren möglichst vermieden wird. Weiterhin werden die Vorgaben und Empfehlungen des esp-idf SDKs und die Unterstützung durch die kryptographischen Rechenbeschleuniger des ESP32 berücksichtigt.

Neben Angriffen stellen Stromausfälle während des Update-Prozesses eine Gefahr dar, da sie das smarte System in einem funktionsunfähigen Zustand bringen können. Als Schutzmaßnahme werden zwei Partitionen im Flash-Speicher angelegt, die je eine funktionsfähige Kopie der Firmware enthalten. Bei einem Update wird eine der Partitionen überschrieben, um im Falle eines Stromausfalls während des Update-Prozesses auf die andere Partition zurück wechseln zu können und so die Funktionsfähigkeit zu erhalten. Es werden schließend die allgemeine Skalierbarkeit auf mehrere smarte Systeme und die Überwachung des Update-Prozesses skizziert. Einige weitere Vorschläge zielen auf das implementierte Update-System ab, unter anderem die Update-Verteilung in Mesh Netzwerken, die Optimierung des Energieverbrauchs des Update-Prozesses und die Verwendung anderer Übertragungsprotokolle.

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Ausarbeitung mit dem Titel „Automatisierte, netzwerkbasierte und sichere Bereitstellung von Firmware Updates für smarte Systeme" selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Zitate sind als solche durch Anführungszeichen und Quellenangabe gekennzeichnet. Diese Arbeit wurde bisher weder in dieser noch in einer ähnlichen Form einer anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Kleinmachnow, den 24.07.2020 _____

Lars Hochstetter

Danksagung

Ich danke folgenden Personen, die mich im Rahmen dieser Arbeit unterstützt haben:
Prof. Dr.-Ing. Jochen Schiller, Dr.-Ing. Achim Liers und Dr. rer. nat. Enrico Köppe für die weitreichende Betreuung dieser Arbeit.
Meinen Eltern, Luca Keidel, David Bohn, Tobias Schülke, meinen Freunden und Kommilitonen für das Korrekturlesen, Vorschläge, Kritik und Zusprache.

Inhaltsverzeichnis

1. Einführung

Das Internet of Things (IoT) kann als globale Infrastruktur der digitalen Gesellschaft verstanden werden, die fortschrittliche Dienstleistungen durch die Vernetzung von physischen und digitalen Dingen ermöglicht. Diese Dinge, nachfolgend smarte Systeme genannt, sind unter anderem Alltagsgegenstände, Geräte und Sensoren, die nicht selbstverständlich als Computer angesehen werden. Studien prognostizieren für das Jahr 2020 26 bis 50 Milliarden eingesetzte smarte Systeme [1].

1.1. Die Funktionsweise smarter Systeme

Ein Teil dieser smarten Systeme basiert auf Mikrocontrollern, die über angeschlossene Sensoren Messwerte erfassen, verarbeiten und diese dann über ein Netzwerk versenden. Abbildung 1 stellt ein solches smartes System schematisch dar: Im Mikrocontroller werden CPU, RAM, Flash-Speicher und unterschiedliche Input-Output (IO) Schnittstellen in einem System-on-a-Chip (SoC) kombiniert. Speziell für den IoT Kontext werden auch Funkschnittstellen wie Wi-Fi oder Bluetooth integriert. Mikrocontroller zeichnen sich weiterhin durch eine hohe Energieeffizienz, geringe physische Größe und hohe Robustheit aus. Dies resultiert allerdings in geringerer Performance beim direkten Vergleich mit herkömmlichen Computern: Die Taktfrequenz der CPU beträgt gewöhnlich 100 bis 200 MHz, der RAM ist nur wenige Kilobyte und der Flash-Speicher nur wenige hundert Kilobyte bis Megabyte groß. Daher werden zusätzlich spezielle Rechenbeschleuniger für rechenintensive Anwendungen wie beispielsweise Kryptographie integriert.

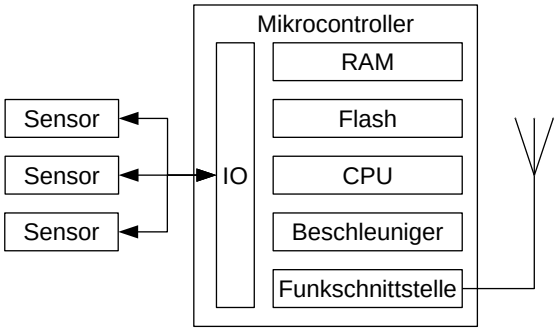


Abbildung 1: Smartes System

Im Einsatz führt ein Mikrocontroller Software, sogenannte Firmware, aus. Je nach Komplexität der Aufgabenstellung besteht diese nur aus anwendungsspezifischem Code. Die Firmware kann aber auch ein für Mikrocontroller optimiertes Betriebssystem enthalten, das die effiziente Verwaltung und Bearbeitung unterschiedlicher Aufgaben vereinfacht.

Da keine Software komplett frei von Fehlern oder Sicherheitslücken ist, kann es nötig werden, die Firmware des smarten Systems im aktiven Einsatz zu aktualisieren. Auch eine Erweiterung der Funktionalität oder das Anpassen der Kommunikationsprotokolle kann ein Update begründen [1].

Hier entsteht ein Problemfeld: Die Firmware wird durch sogenanntes Flashing (auch Flash Vorgang) während der Produktion in den Flash-Speicher eines Mikrocontrollers geschrieben. Für das Flashing wird ein sogenannter Programmer verwendet, welcher entweder in den Mikrocontroller integriert ist oder extern durch ein Kabel an den Mikrocontroller angeschlossen wird. Logistisch wäre es eine praktisch unlösbare Herausforderung, potentiell mehrere Milliarden smarte Systeme für ein Update mit Programmern zu verbinden. Zusätzlich erschwerend kommt hinzu, dass smarte Systeme häufig an schwer zugänglichen Stellen eingesetzt werden oder der Mikrocontroller so tief in das smarte System integriert ist, dass das Verbinden mit dem Programmer bedingt durch das Systemdesign praktisch unmöglich ist.

1.2. OTA Updates

Eine mögliche Lösung stellen Over the Air (OTA) Updates dar: Hier wird die Funkschnittstelle eines smarten Systems genutzt, um das Firmware Update zu empfangen. Da die benötigte Funkinfrastruktur schon durch die anwendungsbedingte Kommunikation mit den smarten Systemen gegeben ist, können so auch große Zahlen schwer erreichbarer smarter Systeme mit Updates versorgt werden. Abbildung 2 zeigt das Zusammenspiel vom Flashing der initialen Firmware während der Produktion und der Verteilung der Firmware Updates über das Internet an aktiv eingesetzte smarte Systeme: Die Entwickler des smarten Systems erstellen eine neue Firmware Version. Diese wird bei der Produktion eines neuen smarten Systems über einen Programmer geflasht. Das smarte System kann dann in den aktiven Einsatz überführt werden. Dieselbe Firmware Version wird auch an einen Update Server geschickt. Über diesen können bereits eingesetzte smarte Systeme die neue Firmware Version als Update beziehen.

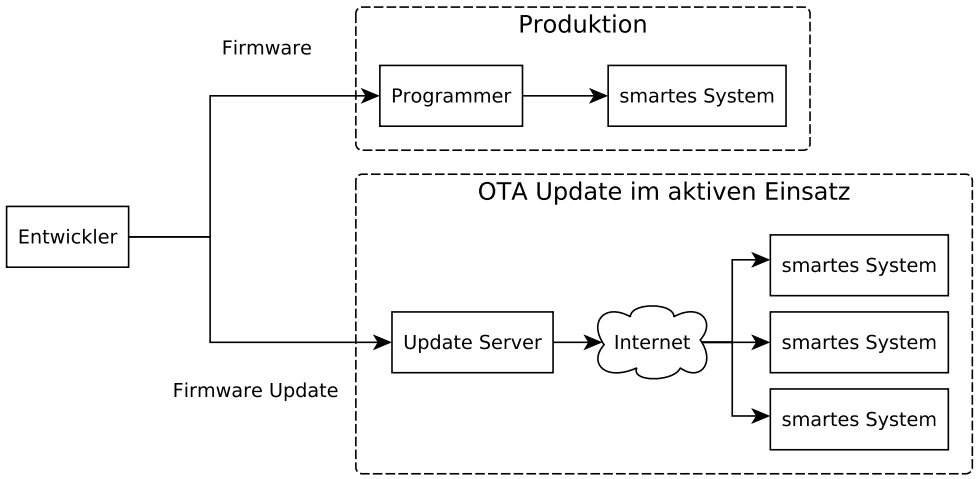


Abbildung 2: Zusammenspiel von Flashing und OTA Updates

1.3. Bisherige Arbeiten

OTA Updates kommen nicht nur im IoT zum Einsatz: Smartphone Hersteller nutzen OTA Updates, um Betriebssysteme wie Android und iOS sowie die darauf laufenden Anwendungen zu aktualisieren [2]. Auch in der Automobilbranche finden OTA Updates immer stärkeren Einsatz: 2019 nutzten moderne Autos durchschnittlich über 100 Electronical Control Units (ECU). Prognosen für das Jahr 2020 gehen von bis zu 300 ECUs pro Auto aus. ECUs werden beispielsweise für das Infotainment System oder die Steuerung der Klimaanlage aber auch sicherheitskritische Systeme wie das Antiblockiersystem (ABS) eingesetzt. Die Firmware der ECUs umfasst bis zu 100 Millionen Zeilen Code [3, S. 14].

Aufgrund der Übertragung des Updates über ein potentiell vertrauensunwürdiges Netzwerk wie dem Internet stellen OTA Updates aber auch ein Einfallstor für Angreifer dar: Durch ein eingeschleustes, manipuliertes Update kann das anvisierte smarte System übernommen oder in seiner Funktion gestört werden. Da smarte Systeme in großer Stückzahl und größtenteils gleicher Konfiguration eingesetzt werden, kann ein Angreifer durch die Übernahme zum Beispiel schnell ein potentes Botnetz aufbauen oder im Falle eines Autos das ABS lahmlegen und so schwere Verkehrsunfälle verursachen.

Zahlreiche Systeme wurden entwickelt, um OTA Updates abgesichert zu verteilen. Von diesen wird eine kleine Auswahl kurz vorgestellt:
Daimi et al. [4] präsentieren ein OTA System, welches Firmware Updates für Autos realisiert. Dabei kann das Update sowohl in der Produktion, beim Händler als auch beim Kunden selbst übertragen werden. Die Sicherheit ergibt sich aus der Verwendung von Zertifikaten, die es einerseits erlauben, die Authentizität der Firmware sicherzustellen, als auch eine sichere Kommunikation zwischen den beteiligten Parteien zu ermöglichen.
Idrees et al. [5] stellen ein generisches System vor, welches ebenfalls für Autos ausgelegt wurde. Zusätzlich zu softwarebasierten Schutzmaßnahmen wird dedizierte Hardware verwendet, die beispielsweise die sichere Speicherung von sicherheitskritischen Daten erlaubt.
Lethaby [6] präsentiert ein OTA Update System für smarte Systeme basierend auf dem TI SimpleLink Mikrocontroller. Die Verteilung der Updates erfolgt über Amazon Web Services (AWS) IoT. Die Firmware basiert auf dem Betriebssystem Amazon FreeRTOS und dem SimpleLink Software Development Kit (SDK). Dies erlaubt den Entwicklern, sich auf die eigentliche Funktionalität zu konzentrieren, da die OTA Update Funktionalität und Absicherung durch das Betriebssystem und SDK umgesetzt werden.
Cesanta [7] bietet mit mDash und mongoose OS ähnlich zu AWS IoT und Amazon FreeRTOS eine cloudbasierte Plattform und ein Betriebssystem, welche im Zusammenspiel OTA Updates ermöglichen. Anders als das von Lethaby präsentierte System werden eine Reihe Mikrocontroller unterschiedlicher Hersteller unterstützt.
Den angesprochenen Update-Systemen fehlt aber die Fähigkeit, die Firmware automatisch aus dem Quellcode zu kompilieren, sobald die Entwickler eine neue Version freigeben. Hier ist das Update-System von Marco Franke [8] zu nennen: Zwar kompiliert es automatisch den Quellcode zu Firmware und verteilt anschließend das Update, es ist aber nur geringfügig abgesichert.

1.4. Gliederung der Arbeit

Ziel dieser Arbeit ist es, ein Update-System zu realisieren, welches zuerst die Firmware automatisch aus Quellcode erstellt und anschließend die Firmware sicher an ein smartes System verteilt. Dabei orientiert sich der inhaltliche Aufbau der Arbeit an dem Four-Step Framework [9, S. xxviii, xxix] aus Adam Shostacks Buch "Threat modeling: Designing for security" [9]. Zwar wurde das Framework ursprünglich zur Strukturierung des Threat Modeling Prozesses geschaffen, aber die auszuführenden Schritte sind an die der Softwareentwicklung und dem Einsatz von Software angelehnt:

1. *Model the system you're building, deploying, or changing.*
2. *Find threats using the modeled system.*
3. *Address the found threats.*
4. *Validate your work for completeness and effectiveness.*

In jedem Schritt wird eine Frage beantwortet, die für eine erste Orientierung bei der Umsetzung genutzt wird [9, S. xxviii, xxix]:

1. *What are you building?*
2. *What can go wrong with it once it's built?*
3. *What should you do about those things that can go wrong?*
4. *Did you do a decent job of analysis?*

Die Arbeit unterteilt sich in einen theoretischen und einen praktischen Teil. Der theoretische Teil gliedert sich dann wie folgt, wobei die Kapitel für die einfachere Orientierung mit der jeweiligen Fragestellung und dem zugehörigen Schritt eingeleitet werden: Der theoretische Teil beginnt mit dem Entwurf des Update-Systems und der damit verbunden Abläufe. Es wird zusätzlich eine Reihe funktionaler und nicht-funktionaler Anforderungen erstellt, die auf eine hohe Sicherheit bei der Verteilung der Updates abzielen. Anschließend daran wird eine Bedrohungsanalyse durchgeführt, um die Gefahren während des Update-Prozesses zu ermitteln. Die gefundenen Gefahren werden dann priorisiert und adressiert. Der theoretische Teil wird durch eine Analyse abgeschlossen, die untersucht, ob die vorhergehenden Schritte vollständig bearbeitet wurden.

Im praktischen Teil der Arbeit wird auf den gewonnenen Erkenntnissen des theoretischen Teils ein Update-System implementiert. Als smartes System wird der ESP32 Mikrocontroller der Firma Espressif Systems verwendet. Der ESP32 verfügt über zahlreiche Funktionseinheiten, die in Abbildung 3 dargestellt sind: Besonderes Interesse gilt der integrierten Wi-Fi Funkschnittstelle und den Rechenbeschleunigern für kryptographische Operationen, da diese die Grundlage für abgesicherte OTA Updates bilden. Abschließend folgen ein Ausblick auf zukünftige Erweiterungen des Update-Systems und ein Fazit.

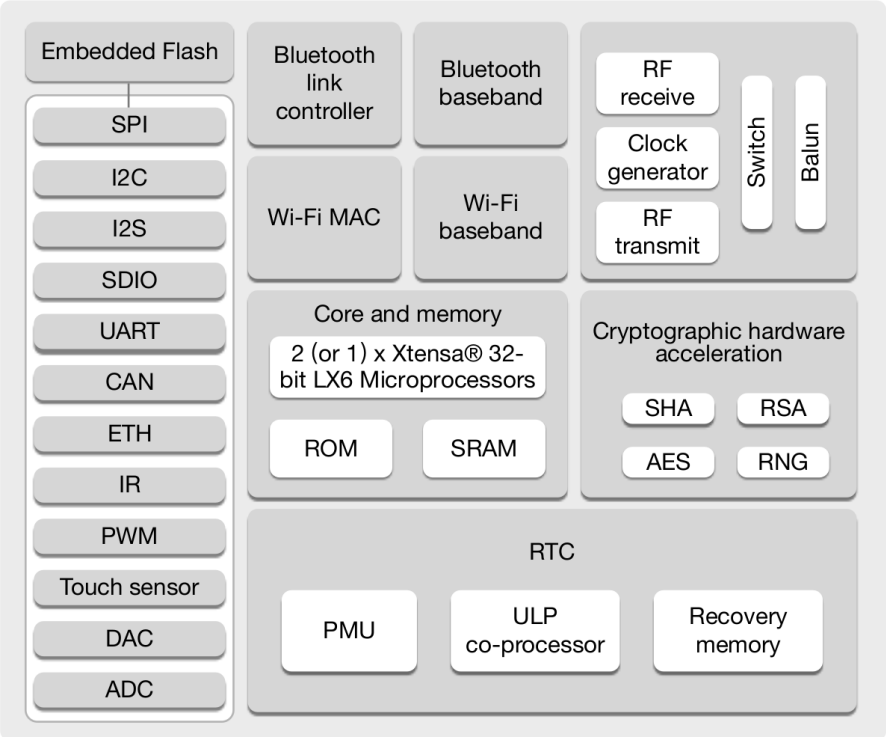


Abbildung 3: Der ESP32 dargestellt als Blockdiagramm [10, S. 12]

Der entstandene Quellcode kann unter `https://git.imp.fu-berlin.de/lhochstetter/master-thesis` eingesehen werden.

2. Zielsetzung

> *What are you building? - Model the system you're building, deploying, or changing.*

Im Rahmen dieser Arbeit wird ein Update-System für die automatisierte, netzwerkbasierte und sichere Bereitstellung von Firmware Updates für smarte Systeme realisiert. Zuerst wird das Systems modelliert und die Arbeitsschritte für die Erstellung und Verteilung eines OTA Updates aufgeführt. Das Update-System wird bei der Erstellung und Verteilung auf ein einzelnes smartes System beschränkt. Abbildung 4 stellt das Modell dar:

1. Die Entwickler laden eine neue Version des Quellcodes in eine Quellcodeverwaltung hoch.

2. Die neue Version des Quellcodes wird automatisch auf einen Server, den sogenannten Build Server, geladen und kompiliert.

3. Abhängig davon, ob sich das smartes System schon im Einsatz befindet, ergeben sich zwei unterschiedliche Zweige, wie die Firmware auf das smarte System gelangt:

   a) Die Firmware wird über einen Programmer am Ende der Produktion geflasht. Das smarte System wird dann in den aktiven Einsatz überführt.

   b) Alternativ wird die kompilierte Firmware als Update an einen weiteren Server, den sogenannten Deploy Server, geschickt. Von dort aus wird das Update dann durch das smarte System heruntergeladen und geflasht. Das smarte System führt nach dem Flashen des Updates einen Neustart in die aktualisierte Firmware durch.

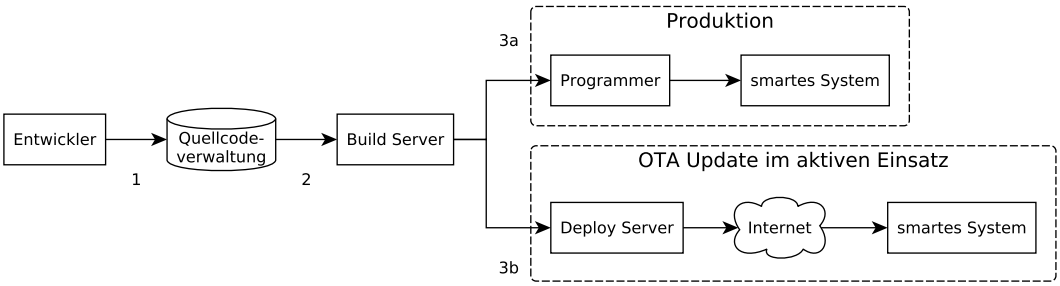


Abbildung 4: Modell für die Verteilung von OTA Updates an das smarte System

Das Übertragen der Updates geschieht über ein potentiell nicht vertrauenswürdiges, fehleranfälliges Netzwerk. Der gesamte Update-Prozess muss daher so robust gestaltet werden, dass auftretende Übertragungsfehler erkannt und behoben werden. Gezielte Manipulationen durch Dritte müssen erkannt und bestenfalls von vornherein verhindert werden.
Um diese Anforderungen besser handhaben zu können, wird ein Modell für den Datenzugriff aus dem Bereich der Informationssicherheit genutzt: die CIA (Confidentiality Integrity Availability) triad [11, S. 5ff]. Nachfolgend werden die deutschen Begriffe Vertraulichkeit für Confidentiality, Integrität für Integrity und Verfügbarkeit für Availability verwendet und die Abkürzung CIA triad als CIA Modell übersetzt. Abbildung 5 stellt das CIA Modell dar.

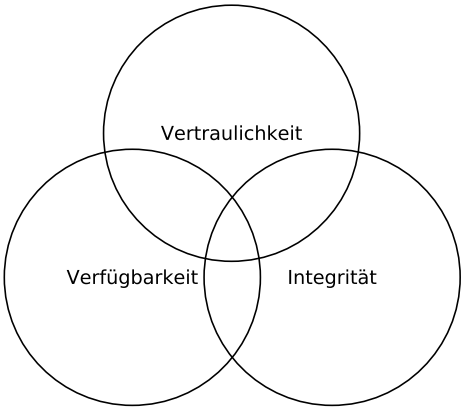


Abbildung 5: CIA Modell [11, S. 5]: Vertraulichkeit sagt aus, dass nur autorisierte Personen auf die betreffenden Daten zugreifen können. Integrität sagt aus, dass die betreffenden Daten vertrauenswürdig, korrekt und unverändert sind. Verfügbarkeit sagt aus, dass auf die betreffenden Daten zuverlässig zugegriffen werden kann.

Im Kontext des Update-Systems werden diese Eigenschaft wie folgt definiert:

- Unter Vertraulichkeit wird hier verstanden, dass nur das eigene smarte System ein Update herunterladen kann. Zusätzlich wird das Update und im weiteren auch die im Flash gespeicherte Firmware vor den Zugriffen unbefugter Dritter geschützt.
- Integrität bedeutet, dass ein Update korrekt und unverändert auf das smarte System übertragen und dort gespeichert wird. Zusätzlich kann das smarte System durch geeignete Maßnahmen die Vertrauenswürdigkeit des Updates validieren.

- Verfügbarkeit hat mehrere Bedeutungen:
    1. Das smarte System kann zuverlässig auf ein auf dem Deploy Server zwischengespeichertes Update zugreifen, wenn ein Update verfügbar ist.
    2. Fehlgeschlagene Updates oder Angriffe auf den Update-Prozess beeinträchtigen die Funktionsfähigkeit des smarten Systems nicht.
    3. Wenn es zu einer Einschränkung der Funktionsfähigkeit oder einer Unterbrechung des Update-Prozesses kommt, kann ein fehlgeschlagenes oder manipuliertes Update auf einen früheren, funktionsfähigen Zustand zurückgesetzt und der Update-Prozess erneut gestartet werden.

Zusätzlich werden folgende nicht-funktionale Anforderungen bei der Umsetzung des zu entwickelnden Update-Systems berücksichtigt:

- Das Update-System soll weitestgehend automatisch und idealerweise ohne menschliche Einflussnahme arbeiten. Damit soll zum einen der menschliche Fehler minimiert und zum anderen das schnelle Ausrollen potentiell sicherheitskritischer Updates realisiert werden.
- Das Update-System soll aus wenigen Komponenten bestehen. Dies reduziert die Komplexität von Wartungen und Anzahl von Abhängigkeiten, wenn eine Komponente ausgetauscht werden soll. Ferner reduziert dies die mögliche Angriffsfläche.
- Die einzelnen Komponenten des Update-Systems sollen gut getestete, weit verbreitete und idealerweise quell-offene (Standard-)Komponenten sein. Besonders im Bereich der Kryptographie sollen Eigenentwicklungen vermieden werden. Durch die weite Verbreitung und damit verbundene hohe Rate an Feldtests kann besser gewährleistet werden, dass die einzelnen Komponenten keine gravierenden Sicherheitslücken, Schwachstellen oder funktionalen Fehler aufweisen. Ferner kann bei weit verbreiteten Komponenten eine schnellere Fehlerbehebung angenommen werden. Der offene Quellcode schließlich ermöglicht eigene Sicherheitsaudits, Anpassungen und Fehlerbehebungen der Komponenten.

3. Bedrohungsanalyse

What can go wrong with it once it's built? - Find threats using the modeled system.

Im vorangegangenen Kapitel wurde das CIA Modell genutzt, um Anforderungen an die Sicherheit des Update-Prozesses zu formulieren. Damit die Anforderungen eingehalten werden, wird eine Bedrohungsanalyse für das Update-System durchgeführt. Durch die Bedrohungsanalyse werden Bedrohungen gefunden, die die Anforderungen verletzen, was im Weiteren die Planung und Implementierungen von Schutzmaßnahmen erlaubt. Dabei liegt der Fokus auf Bedrohungen, die von außen auf das Update-System wirken. Vor der Analyse werden noch die Begriffe Threat (Bedrohung), Vulnerability (Schwachstelle) und Attack (Angriff) nach Kohnfelder et Garg [12] definiert:

Bedrohung Eine Bedrohung beschreibt das Eintreten eines potentiell bösartigen Ereignisses, welches einen unerwünschten Einfluss auf die Ressourcen des Systems hat.

Schwachstelle Eine Schwachstelle ist eine unglückliche Eigenschaft eines Systems, die das Eintreten einer Bedrohung möglich macht.

Angriff Als Angriff wird ein Vorgang bezeichnet, bei dem ein böswilliger Akteur (Angreifer) eine Schwachstelle ausnutzt und so eine Bedrohung für das System darstellt.

3.1. STRIDE

Im Rahmen dieser Arbeit wird das S.T.R.I.D.E. Modell, nachfolgend STRIDE, von Kohnfelder et Garg [12] für die Bedrohungsanalyse verwendet. STRIDE wurde als Bedrohungsmodell ausgewählt, da es viele unterschiedliche Bedrohungen abdeckt, einfach über das Elevation of Privilege [9, S. 206 ff.] Kartenspiel[1] angewendet werden kann und das in Abbildung 4 gezeigte Modell nur geringfügig für die Bedrohungsanalyse angepasst werden muss. Bei umfangreicheren Anpassungen besteht die Gefahr, dass Eigenschaften des Update-Systems nicht genau übertragen werden können und somit Bedrohungen unentdeckt bleiben.

[1] Elevation of Privilege Kartenspiel https://www.microsoft.com/en-us/download/details.aspx?id=20303, zuletzt aufgerufen: 10.06.2020

STRIDE wurde ursprünglich von der Microsoft Security Task Force entwickelt, um unterschiedliche Arten von Bedrohungen noch während der Designphase von Microsoft Produkten zu entdecken und entsprechend frühzeitig beheben zu können [12]. Die durch STRIDE abgedeckten Bedrohungen werden nachfolgend mit einem Beispiel erklärt. Zusätzlich wird ein Beispiel für eine Schwachstelle gegeben, die die jeweilige Bedrohung verursachen kann. Für die Beispiele werden die klassischen Namen [9, S. 341ff.] für die beteiligten Parteien verwendet.

S **Spoofing** [9, S. 62]: Der Angreifer gibt vor eine, andere Identität als die eigene zu besitzen. Wenn es keine Möglichkeit zur Authentifizierung einer Partei gibt, kann diese Schwachstelle für einen Angriff ausgenutzt werden.

Eve kann die Identität von Bob annehmen und Alice anschließend eine Nachricht mit der gefälschten Identität von Bob schicken. Alice geht bei Erhalt der Nachricht davon aus, dass diese von Bob kommt.

T **Tampering** [9, S. 62]: Der Angreifer kann gespeicherte Daten oder Daten während der Übertragung oder Nutzung manipulieren. Manipulationen umfassen das Neuerzeugen, Löschen und Verändern von Daten. Das Fehlen von einer Integritätsprüfung und der Authentifizierung des Absenders erlauben die Ausnutzung dieser Schwachstelle für einen Angriff.

Eve kann die Nachrichten zwischen Bob und Alice abfangen, diese verwerfen, in Teilen verändern oder selbstständig eigene Nachrichten erzeugen und diese an Alice oder Bob schicken. Alice und Bob gehen bei Erhalt der Nachricht dann davon aus, dass diese in ihrer vorliegenden Form so durch den Absender verschickt wurde.

R **Repudiation** [9, S. 63]: Der Angreifer streitet die Durchführung oder die Verantwortung an der Durchführung einer Operation ab. Dies gilt analog für die Nichtdurchführung. Ermöglicht wird dies durch die fehlende Aufzeichnung von durchgeführten Operationen, dem Zeitpunkt der Durchführung und dem verantwortlichen Auslöser, oder dem Nichtdurchführen.

Eve schickt eine Nachricht an Alice, kann dann aber aufgrund der fehlenden Erfassung aller verschickten Nachrichten abstreiten, je diese Nachricht geschickt zu haben.

I **Information Disclosure** [9, S. 63]: Der Angreifer greift auf Informationen zu, auf die er nicht zugreifen darf. Dies ist möglich, wenn die Informationen nicht geschützt übertragen und gespeichert werden.

Alice oder Bob gewähren im einfachsten Fall wissentlich und willentlich Eve Zugriff auf ihre Kommunikation, wenn ihnen der Wert der kommunizierten Informationen nicht bewusst ist. Eve kann auch auf die Nachrichten zugreifen, wenn Alice oder Bob diese frei zugänglich im Klartext speichern. Schließlich kann Eve die Nachrichten während der Übertragung abfangen, den Inhalt lesen und die Nachrichten dann an den eigentlichen Empfänger weiterleiten.

D **Denial of Service (DoS)** [9, S. 63]: Der Angreifer verbraucht gezielt die Ressourcen, die für die Bereitstellung eines Dienstes benötigt werden. Die verursachenden Schwachstellen kann ein zu freizügiges Ressourcenmanagement sein, welches dem Angreifer erlaubt zu viele Systemressourcen anzufordern. Der Angreifer kann auch externe Komponenten verwenden (Distributed Denial of Service, DDoS), um mit vielen Anfragen an das eigene System dessen Ressourcen zu blockieren.

Eve schickt eine große Menge einfacher Nachrichten an Alice oder Bob, deren Bearbeitung den Empfänger daran hindern auf andere Nachrichten zu reagieren. Alternativ kann Eve auch das Kommunikationsmedium mit Nachrichten fluten und so die Kommunikation erheblich stören oder komplett unterbinden.

E **Elevation of Privilege** [9, S. 63]: Der Angreifer verschafft sich zusätzliche Rechte und führt Operationen aus, die er mit seinen ursprünglichen Rechten nicht ausführen durfte. Ein fehlendes oder schwaches Rechtemanagement ermöglicht die Ausnutzung dieser Schwachstelle für einen Angriff.

Eve verschafft sich unbefugten Zugriff auf den Nachrichtenspeicher von Alice oder Bob und kann so die ausgetauschten Nachrichten lesen.

3.2. Analyse des Update-Systems

In Abbildung 6 wird das für die Analyse mit STRIDE angepasste Modell gezeigt: Die gestrichelten Linien stellen sogenannte Trust Boundaries [9, S. 5, 6] dar und die Pfeile die Datenflüsse zwischen den einzelnen Komponenten. Trust Boundaries beschreiben, welcher Akteur den betreffenden Teil eines Systems kontrolliert. Wenn ein Datenfluss eine Trust Boundary überschreitet, verlassen die Daten die Kontrolle eines Akteurs und treten potentiell in den Einflussbereich eines anderen Akteurs ein. Dabei können Eigenschaften der Daten wie Vertraulichkeit, Integrität und Verlässlichkeit verloren gehen, da diese in der betretenen Trust Boundary nicht garantiert werden. Für die Analyse sind also die Datenflüsse interessant, die von der Trust Boundary eines Akteurs in die eines anderen verlaufen.

Für das modellierte Update-System aus Abbildung 4 wurden folgende Trust Boundaries festgelegt, die in Abbildung 6 dargestellt sind: Die erste Trust Boundary (Entwicklung) umfasst die Entwickler, die Quellcodeverwaltung, der Build Server und das Flashing der initialen Firmware über einen Programmer auf das smarte System sowie die Datenflüsse $1 - 4$. Diese Trust Boundary beschreibt einen idealisierten, gesicherten Bereich, der vor jeglichen Angriffen geschützt und somit nicht Teil dieser Bedrohungsanalyse ist.

Die zweite, übergeordnete Trust Boundary (Update) umfasst die zuvor genannten Komponenten und den Deploy Server. Damit wird ausgedrückt, dass der Deploy Server als solcher unter der Kontrolle der Entwickler steht und ein neues Firmware Update sicher vom Build Server auf den Deploy Server übertragen werden kann (Datenfluss 5). Der Deploy Server besitzt aber auch eine Schnittstelle zum Internet hin und kann somit Ziel von Angriffen von außen werden.

Das smarte System besitzt eine eigene Trust Boundary (Einsatz), welche das smarte System im aktiven Einsatz sowie die unmittelbare Umgebung umfasst: Zwar kontrollieren die Entwickler den Update-Prozess, aber die Umgebung in der das smarte System eingesetzt wird, hat unmittelbaren Einfluss auf die Sicherheit. Auch die Übertragung des Updates von Deploy Server auf smartes System (Datenfluss 6) entzieht sich der Kontrolle der Entwickler, da die Übertragung über das Internet stattfindet. Für das Internet wird keine einheitliche Trust Boundary angenommen, da es ein Netzwerk aus Netzwerken ist, die durch unterschiedliche Akteure betrieben werden.

Mögliche Angriffsziele und damit auch Bestandteil der Bedrohungsanalyse sind der Deploy Server, das smarte System und der Datenfluss 6 vom Deploy Server zum Internet und vom Internet zum smarten System.

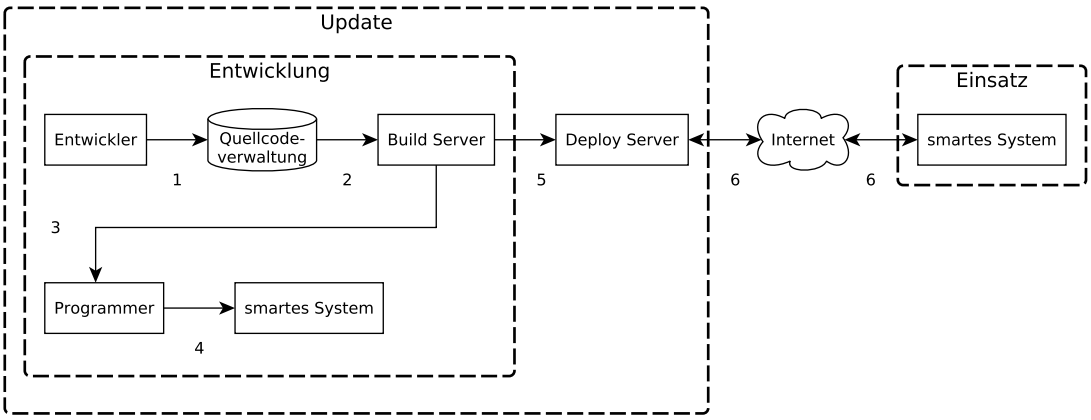


Abbildung 6: Trust Boundaries des Update-Systems

Für die Bedrohungsanalyse selbst wird das angesprochene Elevation of Privilege Kartenspiel verwendet: Die einzelnen Karten sind mit je einer Bedrohung beschriftet. Die Karten werden nacheinander gezogen. Es wird versucht, die jeweilige Bedrohung auf das Update-System anzuwenden. Die Bedrohungen, die angewendet werden konnten, werden in der folgenden Liste gesammelt. Der Listenindex setzt sich dabei aus dem zur Bedrohungsgruppe zugehörigen Buchstaben im STRIDE Akronym und einem fortlaufenden Zähler pro Bedrohungsgruppe zusammen.
Zusätzlich zu den aufgelisteten Bedrohungen kommt noch die Bedrohung durch einen Stromausfall während des Update-Prozesses: Das smarte System befindet sich potentiell in einem nicht sicheren Zustand, aus dem die Funktionalität nicht ohne weiteres wiederhergestellt werden kann. Ein smartes System in diesem Zustand wird Brick genannt. Für einen Stromausfall wird angenommen, dass dieser nicht durch einen Angreifer herbeigeführt wird, sondern Teil der Umwelt ist, in der das smarte System zum Einsatz kommt.

S-1: Ein Angreifer kann sich auf den durch den Deploy Server genutzten zufälligen Port oder Socket zwischenschalten.

S-2: Ein Angreifer kann sich anonym auf den Server verbinden, da angenommen wird, dass die Authentifizierung auf einer höhreren Ebene erfolgt.

S-3: Ein Angreifer kann die Identität des Deploy Servers annehmen, da diese nicht auf dem smarten System gespeichert ist und bei erneuter Verbindungsaufnahme nicht auf Übereinstimmung geprüft wird.

S-4: Ein Angreifer kann eine nicht authentifizierte (und nicht verschlüsselte) Verbindung zum Deploy Server oder smartem System aufbauen.

T-1: Ein Angreifer kann ein Update erneut und unbemerkt verteilen, da es keine Sequenznummern oder Zeitstempel gibt.

T-2: Ein Angreifer kann in den Speicher des Deploy Servers schreiben, in dem das Update zwischengespeichert wird.

T-3: Ein Angreifer kann das Update bei der Übertragung verändern, da keine Integritätsprüfung existiert.

T-4: Ein Angreifer kann Daten im Speicher des smarten Systems durch einen Programmer verändern.

T-5: Ein Angreifer kann Code auf dem smarten System durch einen Programmer ausführen.

R-1: Ein Angreifer kann die Durchführung von Updates abstreiten und es existiert keine Möglichkeit das Gegenteil zu beweisen.

R-2: Das Update-System hat keine Logs.

I-1: Ein Angreifer kann das Update vom Deploy Server oder smarten System auslesen, da dieses nicht verschlüsselt gespeichert wird.

I-2: Ein Angreifer kann als ”man in the middle” (MITM) agieren, weil keine Authentifizierung der Endpunkte einer Netzwerkverbindung stattfindet. Bei einem MITM Angriff leitet der Angreifer die Kommunikation über sich um, in dem er beispielsweise die jeweiligen Identitäten der Kommunikationspartner imitiert.

I-3: Ein Angreifer kann die gesamte Kommunikation zwischen Deploy Server und smartem System mitschneiden, da diese nicht verschlüsselt ist.

D-1: Ein Angreifer kann ohne sich selbst zu authentifizieren das smarte System / den Deploy Server temporär unerreichbar oder unbenutzbar machen.

D-2: Ein Angreifer kann ohne sich selbst zu authentifizieren das smarte System dauerhaft unerreichbar oder unbenutzbar machen.

E-1: Hier wurden keine entsprechenden Bedrohungen gefunden oder die gefundenen Bedrohungen wurden bereits in den vorhergehenden Kategorien aufgeführt.

4. Konzeptionelle Grundlagen für die Umsetzung

> *What should you do about those things that can go wrong? - Address the found threats.*

In diesem Kapitel werden unterschiedliche Schutzmaßnahmen vorgestellt, um den im vorangegangenen Kapitel gefundenen Bedrohungen zu begegnen. Da aber jede implementierte Schutzmaßnahme selbst Ziel von Angriffen [9, S 342 ff.] werden kann, muss entschieden werden, ob die Umsetzung der Schutzmaßnahme sinnvoll ist. Dies wird vom Aufwand und Nutzen der Schutzmaßnahme, aber auch dem Aufwand für einen Angreifer, die zugrundeliegende Schwachstelle für einen Angriff auszunutzen, abhängig gemacht. Anschließend werden die umzusetzenden Schutzmaßnahmen mit ihrer Funktion und dem Einsatzort im Update-System vorgestellt und der Ablauf des Update-Prozesses entsprechend angepasst.

4.1. Kryptographische Grundlagen

Viele mögliche Schutzmaßnahmen basieren auf Verfahren aus der Kryptographie. Die benötigten Grundlagen und Begriffe werden nachfolgend vorgestellt und erklärt. Für die Erklärungen werden die klassischen Namen der Akteure [9, S 341 ff.] verwendet.

4.1.1. Verschlüsselung

Jason Andress [11, S. 70] beschreibt Verschlüsselung als ein Teilgebiet der Kryptographie. Verschlüsselung wird genutzt, um die Vertraulichkeit von Daten (Nachrichten) während der Übertragung und Speicherung zu sichern. Dazu werden die unverschlüsselten Daten (Klartext) so verändert, dass die verschlüsselten Daten (Ciphertext, Geheimtext) keine Rückschlüsse auf die unverschlüsselten Daten erlauben. Der Geheimtext kann durch Entschlüsselung wieder in den ursprünglichen Klartext übertragen werden.
Die Ver- und Entschlüsselung der Daten findet dabei durch einen kryptographischen Algorithmus statt. Dieser nutzt einen oder mehrere Schlüssel um die Daten zu ver- oder entschlüsseln. Abhängig davon ob ein oder mehrere Schlüssel verwendet werden, wird in symmetrische oder asymmetrische Verschlüsselung unterschieden:
Bei der symmetrischen Verschlüsselung [9, S 334-336] wird derselbe Schlüssel verwendet, um eine Nachricht zu ver- und entschlüsseln. Die verschlüsselte Nachricht hat dabei dieselbe Länge wie die unverschlüsselte. Symmetrische Verschlüsselung stellt die Vertraulichkeit von Nachrichten sicher. Alice und Bob verschlüsseln ihre Nachrichten symmetrisch. Eve kann diese nicht lesen, solange sie den symmetrischen Schlüssel nicht kennt.

Alice und Bob müssen sich aber auf einen symmetrischen Schlüssel einigen, was hier nur unverschlüsselt möglich ist. Eve kann die Nachricht mit dem verwendeten symmetrischen Schlüssel abfangen und so die Kommunikation von Alice und Bob entschlüsseln.

Bei der asymmetrischen Verschlüsselung [9, S 334-337] wird ein Schlüsselpaar verwendet, um eine Nachricht zu ver- und entschlüsseln. Das Schlüsselpaar besteht aus einem privaten und einem öffentlichen Schlüssel. Der private Schlüssel muss dabei geheimgehalten werden, während der öffentliche Schlüssel beliebig oft verteilt werden kann.

Abhängig davon mit welchem Schlüssel eine Nachricht verschlüsselt wird, kann asymmetrische Verschlüsselung genutzt werden, um eine Nachricht zu verschlüsseln oder zu signieren. Die Signatur von Nachrichten wird in Kapitel 4.1.3 genauer betrachtet.

Um asymmetrisch verschlüsselt zu kommunizieren, werden die eigenen Nachrichten mit dem öffentlichen Schlüssel des Empfängers verschlüsselt. Da nur der Empfänger im Besitz des zum öffentlichen Schlüssel zugehörigen privaten Schlüssels ist, kann nur dieser die Nachrichten entschlüsseln.

Alice und Bob besitzen je ein Schlüsselpaar bestehend aus einem privatem und einem öffentlichem Schlüssel. Alice nutzt Bobs öffentlichen Schlüssel zur Verschlüsselung einer Nachricht an Bob. Nur Bob kann diese mit seinem privaten Schlüssel entschlüsseln. Selbst wenn Eve im Besitz von Bobs öffentlichen Schlüssel ist, kann sie die Nachricht von Alice nicht entschlüsseln, da dies nur mit Bobs privatem Schlüssel möglich ist. Dies funktioniert analog für Nachrichten von Bob an Alice. Solange Bob und Alice ihre privaten Schlüssel geheim halten und Eve keinen Zugriff auf diese hat, kann Eve die asymmetrisch verschlüsselten Nachrichten nicht entschlüsseln und lesen.

Asymmetrische Verschlüsselung hat den Nachteil, langsamer zu arbeiten als symmetrische Verschlüsselung. Daher werden asymmetrische und symmetrische Verschlüsselung meistens gemeinsam verwendet: Mithilfe asymmetrischer Verschlüsselung wird ein symmetrischer Schlüssel vereinbart, der für die eigentliche Kommunikation verwendet wird. Alice und Bob verschlüsseln zu Beginn ihrer Kommunikation ihre Nachrichten asymmetrisch. Sie einigen sich auf einen symmetrischen Schlüssel, den sie für die eigentliche Kommunikation verwenden. Alle folgenden Nachrichten werden dann mit diesem symmetrischen Schlüssel verschlüsselt. Eve kann die asymmetrisch verschlüsselten Nachrichten und damit den symmetrischen Schlüssel nicht lesen, weil ihr die jeweiligen privaten Schlüssel zur Entschlüsselung fehlen. Eve kann auch die folgenden symmetrischen Nachrichten nicht entschlüsseln, da sie nicht im Besitz des verwendeten symmetrischen Schlüssels ist.

4.1.2. Hash Funktionen

Hash Funktionen [9, S 335, 337, 338] erzeugen aus einer beliebig langen Eingabe einen Hash mit fester Länge. Hash Funktionen sind sogenannte Einwegfunktionen: Es ist nicht möglich von einem Hash auf die ursprüngliche Eingabe zu schließen. Hashes stellen somit die Vertraulichkeit der Eingabe sicher [11, S. 79, 80].
Eine weitere Eigenschaft [11, S. 79, 80] von Hash Funktionen ist, dass unterschiedliche Eingaben unterschiedliche Hashes erzeugen. Da Hashes aber eine feste Länge haben, kann es zu sogenannten Kollisionen kommen, bei denen zwei unterschiedliche Eingaben auf denselben Hash abgebildet werden. Kollisionen treten aber sehr selten auf, weshalb Hashes genutzt werden, um die Integrität einer Nachricht abzusichern: Vor der Übertragung wird der Hash der Nachricht berechnet und mit der Nachricht übertragen. Der Empfänger berechnet bei Empfang den Hash der Nachricht mit derselben Hash Funktion und vergleicht den berechneten und den empfangenen Hash. Wenn beide Hashes übereinstimmen, ist die Nachricht mit hoher Wahrscheinlichkeit fehlerfrei übertragen worden.
Alice berechnet vor dem Abschicken den Hash ihrer Nachricht. Sie schickt Hash und Nachricht an Bob. Bob kann den Hash der empfangenen Nachricht mit derselben Hash Funktion wie Alice berechnen und mit dem übersendeten Hash vergleichen. Wenn die Hashes übereinstimmen, ist mit hoher Wahrscheinlichkeit die Nachricht nicht verändert worden. Der Hash selbst muss aber verschlüsselt übertragen werden, da Eve die Nachricht sonst abfangen, verändern und den Hash neuberechnen kann. Bob geht bei Erhalt der Nachricht davon aus, dass diese so von Alice verschickt wurde, da die Hashes übereinstimmen.

4.1.3. Digitale Signaturen

Die Authentizität einer Nachricht kann durch die Verwendung einer digitalen Signatur sichergestellt werden. Wie oben angesprochen kann asymmetrische Verschlüsselung zum Signieren von Nachrichten verwendet werden [9, S 335, 337, 338]. Dafür wird die zu signierende Nachricht mit dem privaten Schlüssel des Absenders verschlüsselt (signiert). Jeder, der im Besitz des öffentlichen Schlüssels des Absenders ist, kann die Nachricht entschlüsseln (Signatur prüfen). Da das Entschlüsseln der Nachricht nur mit dem öffentlichen Schlüssel des Absenders funktioniert, kann so der Absender authentifiziert werden.

Alice verschlüsselt (signiert) ihre Nachricht mit ihrem privaten Schlüssel. Bob kann mit Alice öffentlichem Schlüssel die Nachricht entschlüsseln (Signatur prüfen). Da die Nachricht nur mit Alice öffentlichem Schlüssel wieder korrekt entschlüsselt werden kann, kann sich Bob sicher sein, dass die Nachricht von Alice kommt. Eve kann mit ihrem eigenen privaten Schlüssel zwar eine Nachricht signieren und diese an Bob mit dem gefälschten Absender Alice schicken, aber Bob erkennt dies, da die Nachricht nicht mit Alice öffentlichem Schlüssel entschlüsselt werden kann.
Eine digitale Signatur kann auch durch die Kombination von asymmetrischer Verschlüsselung und Hashes erstellt werden [11, S. 80]: Anstelle der gesamten Eingabe wird nur der Hash der Eingabe signiert. Da Hashes immer eine feste Länge haben, kann der Hash zumeist schneller asymmetrisch verschlüsselt (signiert) werden als die eigentliche, beliebig lange Eingabe.
Ausgehend von dem vorigen Beispiel nutzt Alice ihren privaten Schlüssel und erstellt den zugehörigen Fingerabdruck durch das Signieren des Hashes der Nachricht an Bob. Eve kann zwar die Nachricht verändern und den Fingerabdruck mit ihrem eigenen privaten Schlüssel neu erstellen, aber Bob erkennt aufgrund des Fingerabdrucks, der nicht zu Alice öffentlichem Schlüssel passt, dass die Nachricht nicht von Alice stammen kann.

4.1.4. Public Key Infrastructure

Mit den bereits vorgestellten Verfahren kann vertraulich kommuniziert werden. Die Identität des Kommunikationspartners kann aber nicht verifiziert werden. Ein Angreifer kann seinen eigenen öffentlichen Schlüssel unter anderem Namen verteilen und so die Identität einer anderen Partei annehmen.
Um die Identität einer Partei mit einem öffentlichen Schlüssel (und damit Schlüsselpaar) zu verbinden werden digitale Zertifikate genutzt. Das Zertifikat (cert) umfasst Informationen zur eindeutigen Identifikation der Partei und den öffentlichen Schlüssel der Partei [11, S. 81]. Es gibt eine Partei, die die Zertifikate verwaltet. Die sogenannte Certificate Authority (CA) signiert die Zertifikate und authentifiziert so die Partei als Besitzer des öffentlichen Schlüssels [9, S 340, 341]. Die CA kann auch Zertifikate sperren. Gesperrte Zertifikate werden in Zertifikatsperrlisten (Certificate Revocation List, CRL) eingetragen, welche dann an die anderen Parteien verteilt werden.
Abbildung 7 zeigt das Zusammenspiel aus Zertifikaten, CA und kommunizierenden Parteien: Alice generiert ein Schlüsselpaar (1) und schickt eine Zertifikatsignierungsanforderung (cert req) an die CA (2). Die CA stellt ein Zertifikat für Alice öffentlichen Schlüssel aus und schickt das Zertifikat der CA sowie das Zertifikat von Alice an sie zurück (3).

Alice will verschlüsselt mit Bob kommunizieren und schickt ihr Zertifikat. Bob besitzt ein eigenes Schlüsselpaar mit Zertifikat und das Zertifikat der CA. Mithilfe des CA Zertifikats kann er das das Zertifikat von Alice validieren (4). Bob schickt sein eigenes Zertifikat an Alice. Alice validiert Bobs Zertifikat analog (5). Alice und Bob haben einander authentifiziert und können nun einen symmetrischen Schlüssel für die folgende Kommunikation austauschen (6).

Die entstandene Infrastruktur zur Authentifizierung von Parteien über Zertifikate wird Public Key Infrastructure (PKI) genannt.

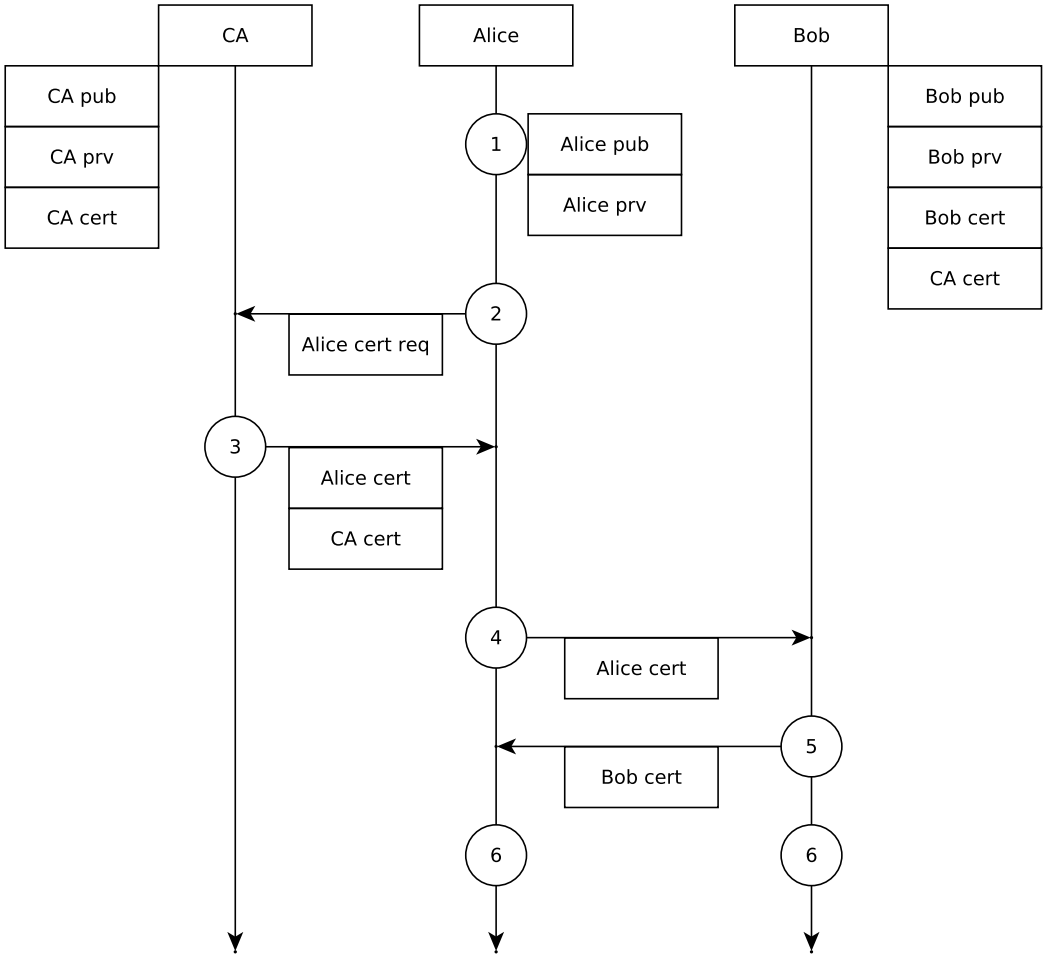


Abbildung 7: Zusammenspiel der Komponenten einer PKI

4.2. Umgang mit Bedrohungen und Auswahl von Schutzmaßnahmen

Vor der Implementierung von Schutzmaßnahmen muss abgewogen werden, ob der Nutzen einer Schutzmaßnahme den Aufwand der Umsetzung rechtfertigt. Der Umgang mit Bedrohungen gestaltet sich dementsprechend flexibel [9, S. 12, 13]:

Bedrohung entschärfen Es werden Maßnahmen ergriffen (Mitigation), die ein Ausnutzen der Bedrohung für einen Angriff erschweren und die Auswirkungen der Bedrohung abschwächen.

Bedrohung entfernen Durch das Entfernen von Funktionen werden die damit verbundenen Bedrohungen entfernt oder abgeschwächt. Allerdings muss abgewogen werden, ob das Entfernen der betreffenden Funktionen möglich ist oder ein anderer Weg zur Bekämpfung der Bedrohung eingeschlagen werden sollte.

Bedrohung verschieben Schwachstellen, die eine Bedrohung im eigenen System hervorrufen, können außerhalb des eigenen Einflussbereiches liegen. Die Abwehr der Bedrohung wird daher an die Partei übertragen, die für den fraglichen Teil, der die Bedrohung hervorruft, zuständig ist.

Risiko akzeptieren Das Risiko, dass eine Schwachstelle für einen Angriff ausgenutzt wird, wird bewusst in Kauf genommen und keine Schutzmaßnahme implementiert. Dies kann vor allem dann sinnvoll sein, wenn die Schwachstelle für einen Angriff nur schwierig ausgenutzt werden kann oder die vorhergehenden Optionen zu aufwendig in der Umsetzung sind.

4.2.1. Absicherung gegen Stromausfälle

Damit Stromausfälle während des Update-Prozesses und allgemein fehlgeschlagene Updates das smarte System nicht bricken können, wird ein Mechanismus benötigt, der das smarte System bei Bedarf wiederherstellen kann. Im embedded Bereich [13] wird dieses Problem durch die Partitionierung des Flash Speichers gelöst: Dabei wird die Funktionalität auf mehrere Partitionen aufgeteilt, um im Falle eines fehlgeschlagenen Updates zumindest einen Rollback Mechanismus funktionsfähig zu erhalten. Eine Partition wird fix als erstes bei jedem (Re-)Boot ausgeführt. Diese Partition enthält einen Bootloader.

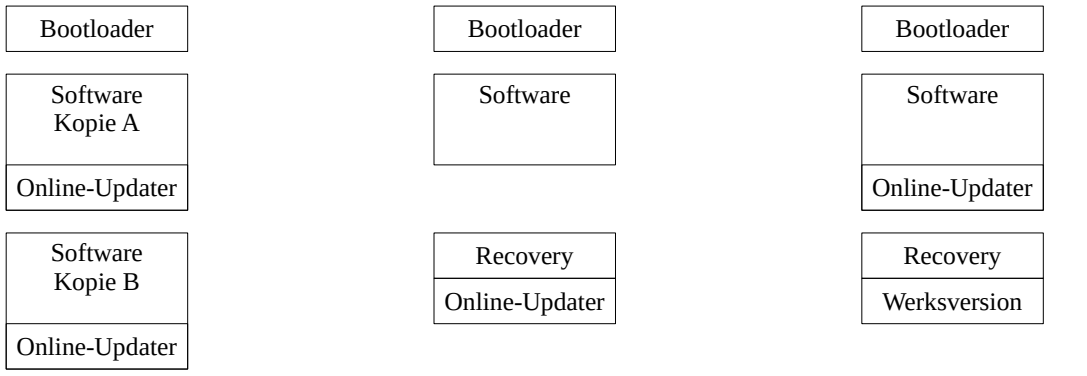
Der Bootloader führt basierend auf seiner Konfiguration eine weitere Partition aus, die die eigentliche Firmware enthält. Es gibt unterschiedliche Varianten für die Partitionierung, die in Abbildung 8 dargestellt und nachfolgend beschrieben werden.

Beim A/B Ansatz, dargestellt in Abbildung 8a, werden zwei Firmware Partitionen verwendet, Software Kopie A und Software Kopie B genannt, die je eine vollständige Version der Firmware enthalten. In der Konfiguration des Bootloaders ist eine Firmware Partition als aktiv, die andere als passiv markiert. Der Bootloader führt die aktive Firmware Partitionen aus. Wenn es ein Update gibt, wird die Updatefunktionalität der Firmware, Online-Updater genannt, ausgeführt. Diese lädt das Firmware Update in die passive Partition herunter, markiert diese in der Konfiguration als aktive Partition und die aktuell aktive Partition als passive und startet das smarte System neu. Der Bootloader liest dann die aktualisierte Konfiguration und führt die zuvor passive jetzt aktive Partition aus. Wenn das Update fehlschlägt, erfolgt der Rollback in die zuvor aktive jetzt passive Partition. Die Vorteile sind die hohe Robustheit, Einfachheit und die Möglichkeit den Online-Updater als Teil der Firmware zu aktualisieren. Der Nachteil ist der von den vorgestellten Varianten größte Speicherbedarf.

Der Recovery für Update Ansatz, dargestellt in Abbildung 8b, hat im Vergleich zum A/B Ansatz nur eine Firmware Partition. Wenn ein Update verfügbar ist, bootet das smarte System in die Recovery Partition, welche eine Minimalversion der Firmware und den Online-Updater enthält. Dieser lädt dann die neue Firmware Version in die Software Partition herunter. Anschließend bootet das smarte System in die aktualisierte Firmware. Wenn das Update fehlgeschlagen ist, findet ein Rollback in die Recovery Partition statt. Der Online-Updater wiederholt das Update solange, bis es gelingt. Der Vorteil dieses Ansatzes ist der geringste Speicherverbrauch im Vergleich zu den anderen vorgestellten Varianten. Die Nachteile sind, dass die Wiederherstellung einen Internetzugang benötigt, durch potentiell mehrere Wiederholungen einige Zeit bis zur einer erfolgreichen Wiederherstellung vergeht und der Online-Updater nicht einfach aktualisiert werden kann.

Mit dem Recovery für Werksversion Ansatz, dargestellt in Abbildung 8c, werden einige Nachteile des Recovery für Update ausgeglichen, allerdings ist der Speicherbedarf im Vergleich größer: Ähnlich dem A/B Ansatz ist der Online-Updater in der Firmware enthalten, der zuerst das Update herunterlädt, zwischenspeichert und anschließend in die Recovery Partition bootet. Die Minimalfirmware führt dann das Update durch. Wenn das Update fehlschlägt nutzt die Minimalfirmware die Werksversion Partition zur Wiederherstellung des smarten Systems: Die Werksversion Partition enthält die Version der Firmware zum Zeitpunkt der Auslieferung des smarten Systems in komprimierter Form.

Zur Wiederherstellung wird diese in die Firmware Partition entpackt. Somit wird die Funktion des smarten Systems zumindest auf einem älteren Stand wiederhergestellt. Anschließend kann der Update-Prozess wiederholt werden.
Aus den drei vorgestellten Möglichkeiten wird der A/B Ansatz aufgrund seiner größeren Robustheit und Flexibilität gewählt: Im Falle eines Rollbacks kann ohne Zwischenschritte auf die letzte funktionsfähige Firmware zugegriffen werden und der Online-Updater kann als Teil der Firmware aktualisiert werden.



(a) A/B

(b) Recovery für Update

(c) Recovery für Werksversion

Abbildung 8: Unterschiedliche Partitionierung für OTA Updates [13]

4.2.2. Absicherung gegen Spoofing Bedrohungen

Die Spoofing Bedrohungen S-1 bis S-4 sind auf die fehlende Authentifizierung der Verbindungsendpunkte zurückzuführen. Im Modell des Update-Systems wurde davon ausgegangen, dass nur das eigene smarte System und der eigene Deploy Server miteinander kommunizieren. Um den Spoofing Bedrohungen zu begegnen wird ein Authentifizierungsverfahren implementiert, das dem Deploy Server und dem smarten System vor Beginn des Update-Prozesses die gegenseitige Authentifizierung erlaubt. Das Authentifizierungsverfahren wird dabei so ausgewählt, dass es idealerweise keine neuen angreifbaren Datenflüsse oder Komponenten in das bestehende Update-System einbringt. Damit soll verhindert werden, dass das Authentifizierungsverfahren selbst zu einer Schwachstelle für das Update-System wird.
Für die Auswahl eines entsprechenden Verfahrens werden die Erkenntnisse der Studie von El-hajj et al. [14] zu im IoT eingesetzten Authentifizierungstechniken genutzt.

Die Studie unterteilt das IoT in drei Schichten, die in Abbildung 9 dargestellt sind: Die Schicht Perception, nachfolgend Wahrnehmung, besteht aus den smarten Systemen. Diese nehmen die für die Anwendung relevanten Messwerte auf. Die aufgenommenen Messwerte werden durch die Schicht Network, nachfolgend Netzwerk, empfangen, verarbeitet und gespeichert. Die Schicht Application, nachfolgend Anwendung, greift auf die Daten der Schicht Netzwerk zu, um Dienste nach den Anforderungen der Nutzer bereitzustellen.
Im Rahmen dieser Arbeit entspricht die Schicht Wahrnehmung dem smarten System und die Schicht Netzwerk dem Deploy Server. Die Schicht Anwendung wird hierbei nicht betrachtet, da der Update-Prozess möglichst ohne menschliche Zuarbeit ablaufen soll.

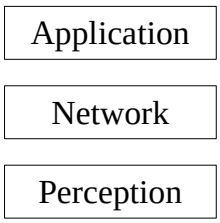


Abbildung 9: Unterteilung des IoT in drei Schichten [14, S. 3]

Abhängig von der Schicht werden unterschiedliche Anforderungen an die verwendeten Authentifizierungsmaßnahmen gestellt, was in unterschiedlichen Umsetzungen resultiert. Diese wurden hinsichtlich ihrer Merkmale durch El-hajj et al. kategorisiert. Abbildung 10 stellt die einzelnen Kategorien dar. Diese werden nachfolgend kurz umrissen: Die Kategorie IoT Layer gibt die Schicht des angesprochenen drei Schichten Modells an, in der die Authentifizierung abläuft.
Für die Authentifizierung selbst können unterschiedliche Faktoren wie die Identität und der Kontext genutzt werden. Dabei ist die Identität einer Partei eine Information, mit der sich eine Partei gegenüber einer anderen Partei authentifizieren kann. Zum Einsatz kommen häufig Hashes und symmetrische sowie asymmetrische kryptographische Algorithmen. Die Authentifizierung über den Kontext basiert auf physischen (biometrischen) Merkmalen und dem Verhalten eines Menschen.
Die Architektur der Authentifizierung kann verteilt oder zentralisiert aufgebaut sein: Bei der verteilten Authentifizierung authentifizieren die kommunizierenden Parteien einander direkt. Wenn die Authentifizierung zentralisiert erfolgt, gibt es eine dritte Partei, der alle anderen Parteien vertrauen. Diese verteilt und verwaltet die Zugangsdaten für die Authentifizierung.

Unabhängig davon, ob die Authentifizierung zentralisiert oder verteilt erfolgt, kann die Authentifizierung hierarchisch oder flach erfolgen. Bei der hierarchischen Authentifizierung wird der Authentifizierungsprozess durch mehrere, aufeinander aufbauende Schichten durchgeführt, während bei der flachen Authentifizierung keine solche Schichten existieren.

Abhängig vom Ablauf findet die Authentifizierung unterschiedlich umfangreich statt: Bei der ein Weg Authentifizierung authentifiziert nur die erste Partei die zweite, während die zweite Partei die erste nicht authentifiziert. Bei der zwei Wege Authentifizierung authentifizieren beide Parteien einander. Schließlich gibt es die drei Wege Authentifizierung, bei der eine dritte Partei die anderen Parteien authentifiziert und diese bei der gegenseitigen Authentifizierung unterstützt.

Die Authentifizierung kann zuvor erzeugte Token (Daten) nutzen, die die Identifizierung einer Partei erlauben. Wenn keine Token verwendet werden, bieten sich für die Authentifizierung Zugangsdaten wie Nutzername und Passwort an.

Die Authentifizierung kann auch physische Eigenschaften der Hardware implizit oder die Hardware selbst explizit einsetzen: Bei der impliziten Verwendung der Hardware kommen die physischen Eigenschaften der Hardware wie True Random Number Generator (TRNG) oder Physical Unclonable Function (PUF) zum Einsatz. Bei der expliziten Verwendung wird zusätzliche Hardware, Trusted Platform Module (TPM) genannt, verwendet. Das TPM speichert und verarbeitet die Schlüssel für die Authentifizierung.

Aus den zuvor vorgestellten Kategorien und Merkmalen wurde mit den in Kapitel 2 definierten Anforderungen folgendes System zur Authentifizierung abgeleitet: Da der menschliche Einfluss möglichst minimiert werden soll, kann die Authentifizierung nur auf den Schichten Wahrnehmung und Netzwerk erfolgen. Zusätzlich kann bei den Faktoren nur die Identität genutzt werden, da bei einer Authentifizierung über den Kontext Eigenschaften und Verhalten eines Menschen verwendet werden. Um den Aufbau der Authentifizierung einfach zu halten, wird eine flache, zentralisierte Architektur umgesetzt. Mit der zentralen Partei bietet sich eine drei Wege Authentifizierung an, die durch die Einschränkung auf den Faktor Identität Token zur Identifizierung erstellt, verwaltet und an die weiteren Parteien verteilt. Auf die Nutzung dedizierter Hardware wird verzichtet, um die die Kosten gering, das Hardwaredesign einfach und die Angriffsfläche klein zu halten. Wo möglich werden aber die Rechenbeschleuniger des smarten Systems für kryptographische Operationen genutzt.

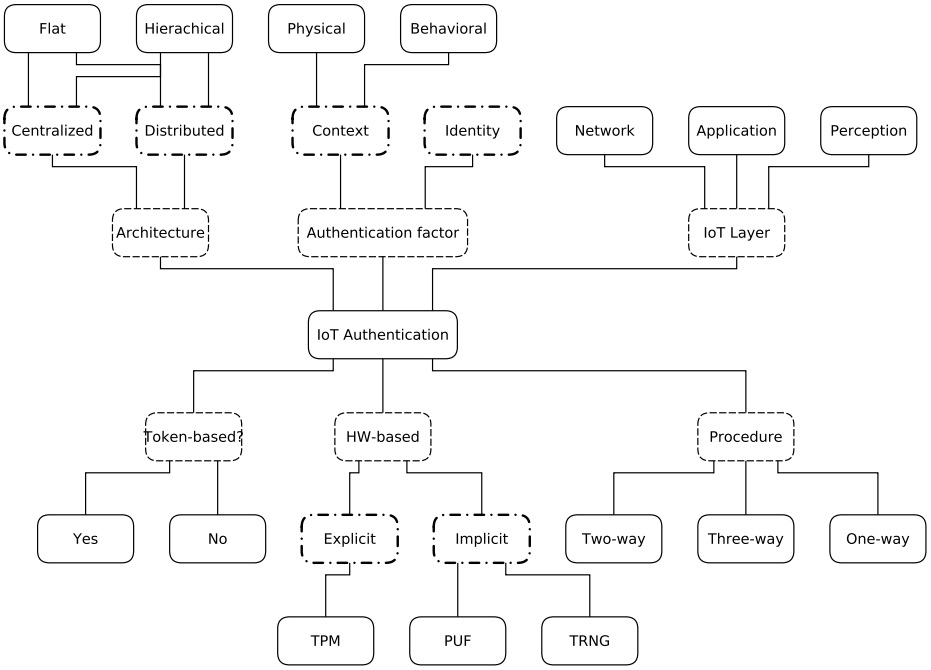


Abbildung 10: Überblick über IoT Authentifizierungsverfahren [14, S. 7]

Das beschriebene Authentifizierungsverfahren gleicht einer PKI. Das Update-System und der Update-Prozess werden dementsprechend erweitert: Eine CA wird als Komponente hinzugefügt und die Verbindung zwischen Deploy Server und smartem System wird entsprechend authentifiziert. Die PKI wird so ausgelegt, dass die CA keine Internetverbindung benötigt, um so die Angriffsfläche zu minimieren.

4.2.3. Absicherung gegen Tampering Bedrohungen

Die Tampering Bedrohung T-1 wird durch die Nutzung einer Versionsnummer abgewehrt. Bei der Kompilation wird für jede neue Firmware und jedes neue Update eine eindeutige Versionsnummer auf dem Build Server erzeugt. Diese wird vor dem Herunterladen des Updates geprüft und der Update-Prozess nur dann fortgesetzt, wenn die Versionsnummer höher als die der aktuell eingesetzten Firmware ist. Die Versionsnummer wird durch Verschlüsselung vor Manipulation geschützt.

Die Tampering Bedrohung T-2 wird nicht direkt abgewehrt: Das Update wird digital signiert. So können Veränderungen am Update oder ein eingeschleustes Update erkannt werden. Um Manipulationen am Update weiter zu erschweren, wird dieses symmetrisch verschlüsselt. Beide Schutzmaßnahmen werden nach der Kompilation auf dem Build Server durchgeführt. Die digitale Signatur muss vor der symmetrischen Verschlüsselung erstellt werden, da die Signatur sonst keinen Bezug mehr zum Update hat. Die digitale Signatur und der symmetrische Schlüssel müssen verschlüsselt an das smarte System übertragen werden, um diese vor Manipulation zu schützen.
Die Tampering Bedrohung T-3 kann durch dieselben Maßnahmen wie T-2 abgewehrt werden.
Die Tampering Bedrohungen T-4 und T-5 setzen Hardwarezugriff auf das smarte System voraus. Der Angreifer kann dann einen Programmer an das smarte System anschließen und auf den Flash-Speicher zugreifen. Die Schutzmaßnahmen gleichen denen von T-2: Die Firmware wird auf dem Build Server digital signiert und das smarte System so konfiguriert, dass es vor dem Ausführen der Firmware die Signatur dieser überprüft. Der Flash-Speicher des smarten Systems wird verschlüsselt. Eine Löschung oder Veränderung der Firmware kann zwar nicht vollständig verhindert, aber erschwert werden.
Beide Schutzmaßnahmen setzen allerdings Hardwareunterstützung seitens des smarten Systems voraus: Das smarte System muss in der Lage sein, die Signatur der Firmware noch vor der Ausführung dieser zu prüfen. Weiterhin muss die Flash-Verschlüsselung transparent für das smarte System erfolgen, da sonst unverschlüsselter und somit angreifbarer Code benötigt wird, der die Firmware vor der Ausführung entschlüsselt.

4.2.4. Absicherung gegen Repudiation Bedrohungen

Das Risiko, das von den Repudiation Bedrohungen R-1 und R-2 ausgeht, wird akzeptiert. Die Schutzmaßnahme [9, S. 153] bestünde im Erfassen der Operationen mit Zeitstempeln (Logging) während eines Updates (Updatestart und -ende, etwaige Fehlermeldungen). Das Logging kann dabei zentralisiert oder verteilt erfolgen.
Beim zentralisierten Logging kommunizieren der Deploy Server und das smarte System die einzelnen Phasen des Update-Prozesses an einen dedizierten Logging Server. Dies fügt dem Update-System allerdings angreifbare Komponenten und Datenflüsse hinzu, da der Logging Server eine Schnittstelle zum Internet braucht, um mit dem Deploy Server und smarten System zu kommunizieren. Dadurch vergrößert sich die Angriffsfläche.

Beim verteilten Logging führen der Deploy Server und das smarte System je einen eigenen Log, welche dann einzeln abgefragt und zu einem Log kombiniert werden. Hier besteht die Gefahr, dass der Angreifer bei Zugriff auf den Deploy Server oder das smarte System den Log entsprechend manipulieren kann.
Das Akzeptieren des Risikos ist hier sinnvoll, da es genau ein einziges smartes System gibt. Ein Ausfall aufgrund eines fehlgeschlagenen oder manipulierten Updates wird mit hoher Wahrscheinlichkeit durch die ausbleibende funktionsgebundene Kommunikation bemerkt.
Wenn aber mehrere hundert oder tausend smarte Systeme eingesetzt werden, stellen die gefundenen Repudiation Bedrohungen eine ernsthafte Gefahr für das Update-System dar, da das Erkennen und Erfassen von funktionsunfähigen smarten Systemen aufgrund der großen Anzahl unvergleichbar schwieriger ist.

4.2.5. Absicherung gegen Information Disclosure Bedrohungen

Die Information Disclosure Bedrohungen I-1 und I-3 werden durch die symmetrische Verschlüsselung des Updates abgewehrt [9, S. 155]. Der verwendete symmetrische Schlüssel muss, wie bei T-2 und T-3 bereits angesprochen, verschlüsselt auf das smarte System übertragen werden.
Die Information Disclosure Bedrohung I-2 wird durch das Authentifizierungsverfahren, das S-1 bis S-4 abwehrt, mit abgedeckt.

4.2.6. Absicherung gegen DoS Bedrohungen

Bei den Bedrohungen D-1 und D-2 ist zu beachten, dass hier nur die Kommunikation zwischen Deploy Server und smartem System und die Kommunikation des smarten Systems mit dem restlichen Internet betrachtet wird. Ausfälle bedingt durch manipulierte oder fehlgeschlagene Updates werden nicht betrachtet, da die bereits ausgewählten Schutzmaßnahmen die Manipulation des Updates verhindern und ein fehlgeschlagenes Update zurückgesetzt und erneut angestoßen werden kann.
Die Bedrohungen der Kategorie Denial of Service D-1 und D-2 liegen außerhalb des Bereichs der eigenen Einflussnahme und können somit nur an die jeweiligen Netzbetreiber abgegeben werden: Einige Cloud Services und Webhosting Angebote beinhalten entsprechende Schutzmaßnahmen gegen netzwerkbasierte DoS / DDoS Angriffe [9, S. 157].

4.3. Erkennen, dass ein neues Update existiert

Um den Update-Prozess zu starten, muss das smarte System erkennen, dass ein neues Update zur Verfügung steht. Dafür bieten sich der „Pull“ und der „Push“ Ansatz an:
Beim „Pull“ Ansatz fragt das smarte System periodisch den Deploy Server nach neuen Updates. Wenn ein neues Update vorliegt, wird dieses entsprechend heruntergeladen. Der Vorteil besteht in der Einfachheit, da nur das smarte System den Deploy Server kennen muss. Weiterhin stellen Eigenheiten der Netzwerktopologie wie NAT Netzwerke oder Firewalls ein geringeres Problem dar. Der Nachteil liegt im höheren Energieaufwand, da das smarte System periodisch aus dem Schlafmodus aufwachen muss, um den Deploy Server anzufragen. Bis ein neues potentiell sicherheitskritisches Update das smarte System erreicht kann einige Zeit vergehen. Wenn mehrere smarte Systeme den Deploy Server anfragen gleicht dies einem DDoS Angriff, folglich müssen die smarten Systeme hinsichtlich der Abfrageintervalle koordiniert werden.
Beim „Push“ Ansatz benachrichtigt der Deploy Server das smarte System wenn ein neues Update vorliegt, damit dieses es herunterladen kann. Die Vor- und Nachteile sind genau umgekehrt denen des „Pull“ Ansatzes: Das smarte System kann länger im Energiesparmodus bleiben und wird durch den Deploy Server bei einem neuen Update geweckt. Neue Updates können schneller verteilt werden. Auch der Zeitpunkt, wann welches smarte System für ein Update benachrichtigt wird, kann durch den Deploy Server koordiniert werden. Allerdings steigt dadurch der Verwaltungsaufwand auf dem Server, da dieser alle smarten Systeme kennen muss und der Verbindungsaufbau durch den Deploy Server von Firewalls blockiert werden kann. Auch eine dynamische Zuweisung der IP Adresse eines smarten Systems kann die Benachrichtigung erschweren.
Da das Update-System nur, wie in Kapitel 2 festgelegt, für ein smartes System Updates bereitstellt, wird hier der einfachere „Pull“ Ansatz verwendet. Wenn das Update-System allerdings auf mehr smarte Systeme ausgedehnt werden soll, müssen die Ansätze abermals evaluiert werden.

4.4. Erweiterung des Update-Systems und des Update-Prozesses

Das Authentifizierungsverfahren benötigt als neue Komponente eine CA. Abbildung 11 stellt das Update-System mit CA dar. Die Zertifikate und Schlüsselpaare werden innerhalb der Trust Boundary „Entwicklung“ erstellt. Das Zertifikat und Schlüsselpaar für das smarte System wird auf den Build Server übertragen, um dort in die initiale Firmware integriert zu werden. Das Zertifikat und Schlüsselpaar für den Deploy Server wird einmalig auf diesen übertragen. Dieser Datenfluss ist gleich dem Datenfluss von Build Server auf Deploy Server abgesichert.

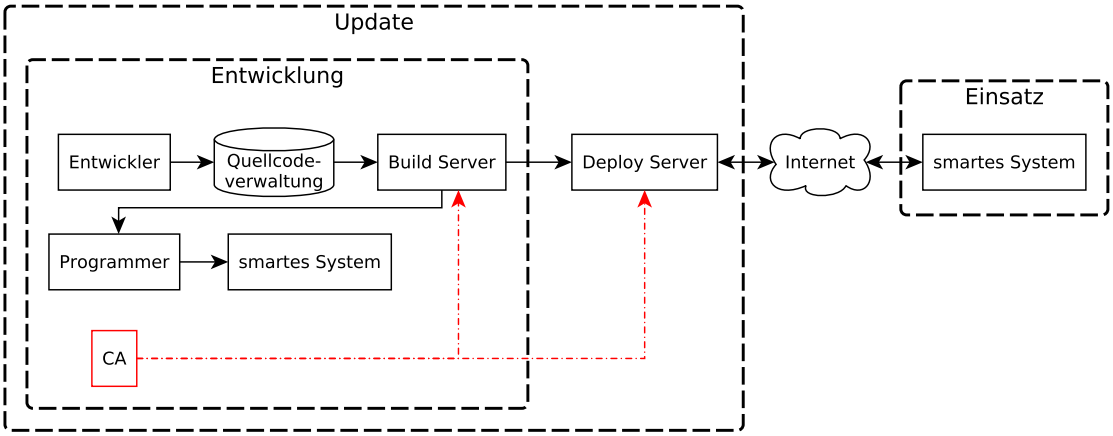


Abbildung 11: Update-System mit CA

Der Update-Prozess muss für die Integration der Schutzmaßnahmen angepasst werden. Abbildung 12 stellt den angepassten Update-Prozess dar: Die Firmware wird auf dem Build Server kompiliert, der öffentliche Schlüssel des Build Server integriert und mit der nächsten Versionsnummer versehen. Anschließend wird das Update mit dem privaten Schlüssel des Build Servers digital signiert und symmetrisch verschlüsselt. Der symmetrische Schlüssel wird für jedes Update neu erstellt, um zu verhindern, dass ein Angreifer durch statistische Analysen auf den Schlüssel schließen kann. Versionsnummer, digitale Signatur und symmetrischer Schlüssel werden in einer Datei, der sogenannten Meta Datei, gebündelt. Diese wird anschließend asymmetrisch mit dem öffentlichen Schlüssel des smarten Systems verschlüsselt (1). Update und Meta Datei werden auf den Deploy Server übertragen (2).

Das smarte System baut periodisch eine authentifizierte Verbindung mit dem Deploy Server auf (3) und lädt die Meta Datei herunter (4). Die heruntergeladene Meta Datei wird mit dem privaten Schlüssel des smarten Systems entschlüsselt und die enthaltene Versionsnummer mit der Versionsnummer der aktuell eingesetzten Firmware verglichen. Wenn die Versionsnummer gleich oder älter ist, bricht das smarte System den Update-Prozess ab. Wenn die Versionsnummer neuer ist, wird der Update-Prozess fortgesetzt (5). Das smarte System lädt das Update herunter (6). Das Update wird mit dem symmetrische Schlüssel aus der Meta Datei entschlüsselt. Die Signatur wird geprüft. Wenn diese korrekt ist, wird das Update in die inaktive der Firmware Partitionen im Flash-Speicher geschrieben. Ansonsten wird das Update verworfen und der Update-Prozess zurückgesetzt (7). Nach gelungenem Update wird der Bootpointer auf die Partition mit dem Update umgesetzt und das smarte System führt einen Neustart in die neue Firmware durch (8).

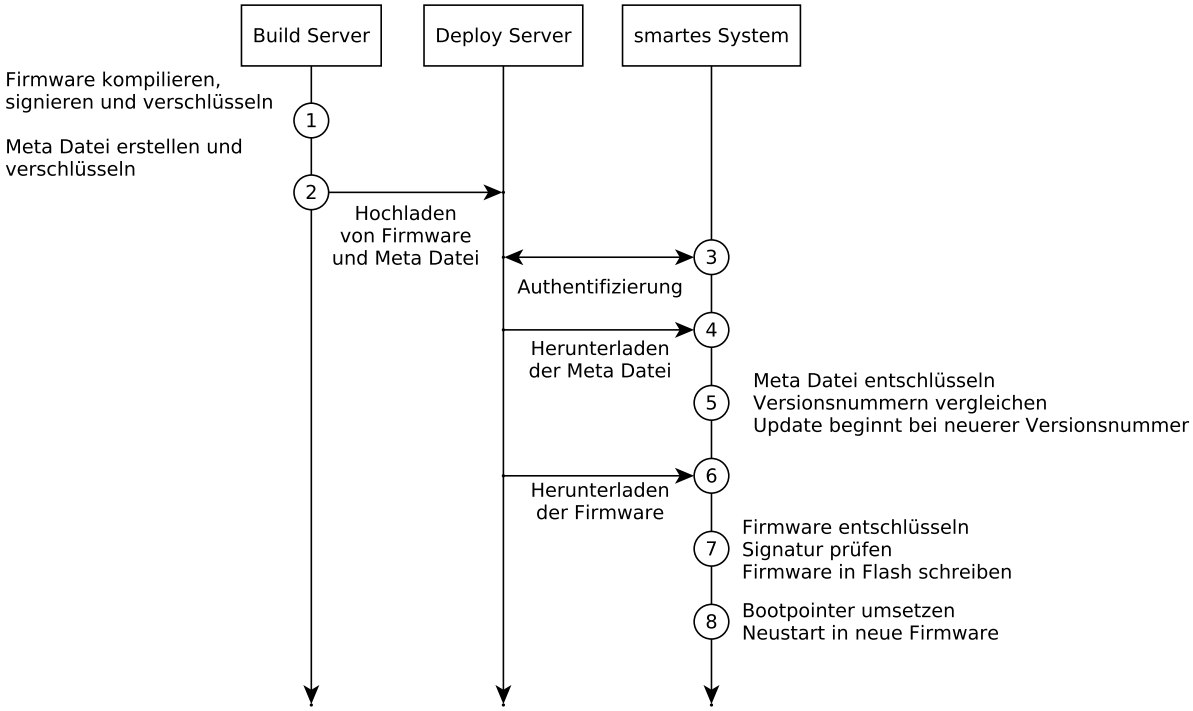


Abbildung 12: Der Update-Prozess mit Schutzmaßnahmen

5. Analyse der Schutzmaßnahmen des Update-Systems

> *Did you do a decent job of analysis? - Validate your work for completeness and effectiveness.*

Der letzte Schritt im theoretischen Teil ist die Analyse des Update-Systems und der Schutzmaßnahmen. Für die Analyse wird die von Shostack vorgeschlagene Checkliste [9, S. 27] verwendet. Allerdings muss beachtet werden, dass aufgrund des hohen Abstraktionsgrades des entworfenen Update-Systems Bedrohungen, die auf Schwachstellen in konkreten Hardware- und Softwarekomponenten basieren, nicht entdeckt werden können. Bei der Umsetzung in Kapitel 6 wird daher auf die für relevant befundenen Schwachstellen in den angreifbaren Komponenten hingewiesen.

5.1. Diagramming

1. Can we tell a story without changing the diagram?
2. Can we tell that story without using words such as "sometimes" or "also"?

In Kapitel 4.4 werden die vorgenommenen Veränderungen am in Kapitel 2 entworfenen Update-System vorgestellt. Der gesamte Update-Prozess und die beteiligten Komponenten können mit den angepassten Diagrammen beschrieben werden, ohne dabei die Diagramme nochmals anpassen oder Sonderfälle getrennt ansprechen zu müssen.

3. Can we look at the diagram and see exactly where the software will make a security decision?
4. Does the diagram show all the trust boundaries, such as where different accounts interact? Do you cover all UIDs, all application roles, and all network interfaces?

Während dem in Kapitel 2 entworfenen Update-System Trust Boundaries und grundlegende Sicherheit fehlten, wurden Trust Boundaries für die Bedrohungsanalyse in Kapitel 3 und Schutzmaßnahmen in Kapitel 4 hinzugefügt. Da das Update-System sehr abstrakt und ohne konkrete Hardware und Software entworfen wurde, können hier keine Aussagen bezüglich möglicher Implementierungen getroffen werden.

5. Does the diagram reflect the current or planned reality of the software?

Das Update-System wurde nur auf das Bereitstellen von Updates für ein einziges smartes System ausgelegt. Die Prozesse zur Erstellung und Verteilung von Updates sind einfach genug in Struktur und Ausführung, um bei Bedarf auf mehrere smarte Systeme skaliert zu werden. Die Ausnahme ist das Erkennen eines neuen Updates durch die smarten Systeme. Hier müssen, wie angesprochen, die Ansätze erneut evaluiert werden.

6. Can we see where all the data goes and who uses it?

7. Do we see the processes that move data from one data store to another?

Sämtliche Datenflüsse des Update-Prozesses sind aus den Diagrammen in Kapitel 4.4 ersichtlich. Alle Datenflüsse haben eine eindeutige Quelle und eine eindeutige Senke. Der Datenfluss mit der Quelle Build Server hat eine Besonderheit: Dieser hat zwei Senken, wobei die eine der Programmer ist und die andere der Deploy Server. Die bidirektionalen Datenflüsse zwischen Deploy Server, Internet und smartem System können für die jeweilige Richtung als ein Datenfluss dargestellt werden: Deploy Server → Internet → smartes System und analog umgekehrt.

5.2. Threats

1. Have we looked for each of the STRIDE threats?

Durch die Verwendung des Elevation of Privilege Kartenspiels wurden die einzelnen Bedrohungskategorien systematisch abgearbeitet. Bis auf die Elevation of Privilege Kategorie konnten für jede Kategorie Bedrohungen gefunden werden.

2. Have we looked at each element of the diagram?

3. Have we looked at each data flow in the diagram?

Die Anzahl an zu betrachtenden Datenflüssen und Komponenten wurde durch die Annahmen die für die einzelnen Trust Boundaries eingeschränkt: So werden die Komponenten und Datenflüsse der Trust Boundary „Entwicklung“ als unangreifbar angenommen. Die einzigen angreifbaren Komponenten und Datenflüsse sind somit der Deploy Server, das smarte System und der Datenfluss zwischen diesen.

5.3. Validating Threats

1. Have we written down or filed a bug for each threat?

Das Ergebnis des Elevation of Privilege Kartenspiels wurde in der Bedrohungsanalyse in Kapitel 3 in Form der gefundenen Bedrohungen festgehalten.

2. Is there a proposed/planned/implemented way to address each threat?

Der Umgang mit den gefundenen Bedrohungen wurde in Kapitel 4.2 vorgestellt: Für die Bedrohungen der Kategorien Spoofing, Tampering und Information Disclosure werden Schutzmaßnahmen implementiert. Die Implementierung von Maßnahmen gegen DoS Bedrohungen werden an die Netzbetreiber abgegeben, da sie außerhalb des eigenen Einflussbereiches liegen. Das von Repudiation Bedrohungen ausgehende Risiko wurde akzeptiert. Das Risiko wurde als zu gering bewertet, um eine Schutzmaßnahme zu implementieren.

3. Do we have a test case per threat?

4. Has the software passed the test?

Es wurden keine Testfälle pro Bedrohung aufgestellt, da lediglich eine theoretische Betrachtung möglich wäre. Das korrekte Verhalten der Implementierung wird durch gezielte Falscheingaben überprüft.

6. Umsetzung des Update-Systems

Das in den vorangegangenen Kapiteln entworfene Update-System wird in diesem Kapitel implementiert. Als smartes System wird der ESP32 von Espressif Systems genutzt. Zuerst wird die Basis der Firmware und die Umsetzung der Schutzmaßnahmen vorgestellt und anschließend die Komponenten des Update-Systems.

6.1. Firmwareentwicklung und Schutzmaßnahmen

Für die Programmierung des ESP32 wird das esp-idf Software Development Kit (SDK) von Espressif Systems genutzt. Dieses wird in Version v3.3.1-stable zur Entwicklung der eigenen Firmware genutzt. Das SDK umfasst mehrere Code Beispielen zu OTA Updates[2] und die Kryptographiebibliothek mbed TLS[3], die die Nutzung der Rechenbeschleuniger für kryptographische Operationen des ESP32 vereinfacht.
Der Quellcode des „Native OTA“ Beispiels[4] wird als Grundlage für die Entwicklung der Firmware verwendet und entsprechend mit Schutzmaßnahmen erweitert. Das Beispiel implementiert bereits einige Schutzmaßnahmen: Das Update wird über das Hypertext Transfer Protocol Secure (HTTPS) heruntergeladen. Die Verbindung wird durch Transport Layer Security (TLS) verschlüsselt und es findet eine Authentifizierung des Servers durch den ESP32 statt. Die Firmware enthält eine Versionsnummer. Bei einem Update wird zuerst nur der Teil, der Firmware der die Versionsnummer enthält, heruntergeladen und mit der Versionsnummer der aktuell ausgeführten Firmware verglichen. Wenn die Versionsnummer des Updates neuer ist wird das Update fortgesetzt und ansonsten der Update-Prozess zurückgesetzt.
Der Flash-Speicher des ESP32 ist in einer Mischform aus dem „A/B“ Ansatz aus Abbildung 8a und dem „Recovery für Werksversion“ Ansatz aus Abbildung 8c partitioniert: Es existieren zwei Partitionen, die die eigentliche Firmware enthalten sowie eine Partition, die die Werksversion mit Online-Updater enthält.

[2] OTA Beispiele https://github.com/espressif/esp-idf/tree/v3.3.1/examples/system/ota, zuletzt aufgerufen: 10.06.2020
[3] mbed TLS https://tls.mbed.org/, zuletzt aufgerufen: 10.06.2020
[4] „Native OTA“ Beispiel https://github.com/espressif/esp-idf/tree/v3.3.1/examples/system/ota/native_ota_example, zuletzt aufgerufen: 10.06.2020

Da das zugrundeliegende Beispiel bereits Versionsnummern, Transportverschlüsselung und Authentifizierung des Deploy Servers implementiert, sind noch folgende Schutzmaßnahmen zu implementieren: Partitionierung des Flash-Speichers nach dem „A/B“ Ansatz, symmetrische Verschlüsselung der Firmware, asymmetrische Verschlüsselung der Meta Datei, Signieren der Firmware, Flash-Verschlüsselung und die Authentifizierung des ESP32.
Es folgt die Auflistung der ausgewählten kryptographischen Algorithmen. Die Auswahl folgt, wo möglich, den Vorgaben oder Empfehlungen des esp-idf SDK. Zusätzlich werden die Algorithmen bevorzugt gewählt, die durch mbed TLS und die Rechenbeschleuniger des ESP32 unterstützt werden. Damit werden Eigenentwicklungen im Bereich Kryptographie vermieden.

- Der Advanced Encryption Standard (AES) [15] wird für die symmetrische Verschlüsselung genutzt. AES arbeitet auf Datenblöcken von 128 Bit (16 Byte) Länge und verwendet Schlüssel mit 128, 192 und 256 Bits Länge. AES wird in folgenden Arbeitsmodi im Update-System eingesetzt: Die Firmware wird mit AES-CBC verschlüsselt. Die Flash-Verschlüsselung erfolgt mit AES-ECB und AES-XTS. Der ESP32 verfügt über einen Rechenbeschleuniger für AES und die Verwendung von AES-ECB und AES-XTS wird durch die Hardware des ESP32 und das esp-idf SDK vorgegeben [16].

- Der Algorithmus von Rivest, Shamir und Adleman (RSA) [17] wird für die asymmetrische Verschlüsselung eingesetzt. RSA wird genutzt, um die Meta Datei zu verschlüsseln. Der ESP32 verfügt über einen Rechenbeschleuniger für RSA.

- Für das Signieren der Firmware wird der Deterministic Elliptic Curve Digital Signature Algorithm (Deterministic ECDSA) mit der elliptischen Kurve NIST256p (auch prime256v1 oder secp256r1) und SHA256 als Hashfunktion verwendet [18]. Deterministic ECDSA ist ein asymmetrisches Verfahren. Die SHA256 Hashfunktion [19] arbeitet auf 512 Bit langen Blöcken und berechnet einen 256 Bit langen Hash. Wenn die Größe der Eingabe nicht ein Vielfaches von 512 ist, so wird die Eingabe entsprechend und automatisch aufgefüllt. Die Verwendung von Deterministic ECDSA wird durch das esp-idf SDK vorgegeben [18].

- HTTPS wird für die Transportverschlüsselung und gegenseitige Authentifizierung beibehalten. Für die Authentifizierung werden Zertifikate für den Deploy Server und ESP32 benötigt.

Für die Erstellung der von den Schutzmaßnahmen verwendeten Schlüsseln und Zertifikaten werden folgende Werkzeuge eingesetzt:

- OpenSSL[5] ($v.1.1.1d$) wird hier eingesetzt, um die RSA und AES Schlüssel zu erstellen und ist ein quelloffenes[6], weit verbreitetes Kryptographie und SSL Toolkit.
- easy-rsa 3.0.6[7] wird nachfolgend noch genauer vorgestellt und dient der Erstellung der Zertifikate für den Deploy Server und ESP32.
- Die Werkzeuge espsecure[8] und nvs_partition_gen[9] werden für die Erstellung von Schlüsseln für durch das esp-idf SDK empfohlene Schutzmaßnahmen genutzt. Einige dieser Schlüssel werden mit espefuse[10] in die E-Fuses des ESP32 eingebrannt und können danach nicht mehr verändert werden.

6.1.1. Partitionierung des Flash-Speichers

Der Flash-Speicher nach dem „A/B“ Ansatz aus Abbildung 8a partitioniert, um den Flash-Speicher besser ausnutzen zu können. Abbildung 13 stellt die Partitionierung dar: Der Bootloader wird als erstes ausgeführt und nutzt die Informationen aus Partitionstabelle und OTA Pointer (Ptr) Partition, um eine der beiden Partitionen OTA 1 oder OTA 2 auszuführen. Diese enthalten die eigentliche Firmware. Zusätzlich gibt es die Non-Volatile Storage (NVS) Partition, die die WLAN Zugangsdaten, den privaten Schlüssel des ESP32 und die für HTTPS erforderlichen Schlüssel und Zertifikate des ESP32 enthält. Der Anhang A.4 enthält Details zur Partitionierung des Flash-Speichers.

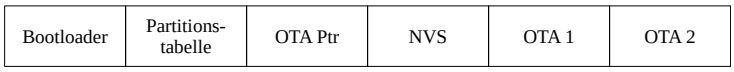


Abbildung 13: Partitionierung des Flash Speichers

[5] OpenSSL https://www.openssl.org/, zuletzt aufgerufen: 10.06.2020
[6] OpenSSL Quellcode https://github.com/openssl/openssl, zuletzt aufgerufen: 10.06.2020
[7] easy-rsa 3.0.6 https://github.com/OpenVPN/easy-rsa/releases/tag/v3.0.6, zuletzt aufgerufen: 10.06.2020
[8] espsecure https://github.com/espressif/esptool/wiki/espsecure, zuletzt aufgerufen: 10.06.2020
[9] nvs_partition_gen https://docs.espressif.com/projects/esp-idf/en/v3.3.1/api-reference/storage/nvs_partition_gen.html, zuletzt aufgerufen: 10.06.2020
[10] espefuse https://github.com/espressif/esptool/wiki/espefuse, zuletzt aufgerufen: 10.06.2020

6.1.2. Signieren der Firmware

Die Firmware wird mit dem espsecure Werkzeug aus dem esp-idf SDK auf dem Build Server signiert [18]: Der SHA256 Hash der Firmware wird berechnet und anschließend mit dem privaten ECDSA Schlüssel signiert. Die erstellte Signatur ist 68 Byte lang: 4 Byte sind Versionsinformationen und die restlichen 64 Byte die eigentliche Signatur des Firmware Updates. Die Signatur wird dann hinten an die Firmware angehängt. Der öffentliche ECDSA Schlüssel wird in die Firmware integriert, damit die Signatur eines Updates nach dem Herunterladen geprüft werden kann. Wenn die Signatur nicht korrekt ist, wird die heruntergeladene Firmware nicht gebootet und der Update-Prozess zurückgesetzt. Das genutzte ECDSA Schlüsselpaar wird mit dem espsecure Werkzeug aus dem esp-idf SDK erzeugt.
Wenn aber ein Angreifer mit Hardwarezugriff einen Programmer mit dem ESP32 verbindet, kann dieser so manipulierte Firmware flashen. Damit dies erkannt und die eingeschleuste Firmware nicht ausgeführt wird, kann der Bootloader die Signatur der Firmware vor der Ausführung überprüfen [18]. Für die Überprüfung kommt derselbe öffentliche Schlüssel zum Einsatz wie bei der Überprüfung nach dem Herunterladen. Wenn die Firmware nicht oder falsch signiert ist, verfällt der Bootloader nach dem ersten Anlauf in einen Bootloop, da er wiederholt versucht die Firmware mit falscher Signatur zu booten, bei der Signaturprüfung scheitert und anschließend neustartet.
Um den Bootloop zu brechen, wird die Rollback Funktion [20] genutzt: Im Updatefall mit signierter Firmware wird der Bootpointer auf die Partition mit der neu heruntergeladenen Firmware umgesetzt und markiert, damit der erste Start dieser Firmware überwacht wird. Wenn der Bootloader die Firmware zum ersten Mal ausführt, muss die Firmware die Markierung entfernen und die eigene als funktionsfähig markieren. Die Firmware sollte in diesem Fall auch einen Selbsttest ausführen, um zu verhindern, dass ein Fehler zu einem späteren Zeitpunkt den ESP32 funktionsunfähig macht.
Wenn ein Angreifer eigene Firmware ohne korrekte Signatur flasht und den Bootpointer manuell umsetzt, scheitert die Überprüfung und der Bootloader startet den ESP32 neu ohne die Firmware auszuführen. Der Bootloader prüft nach dem ersten erfolglosen Start die Markierung und erkennt, dass diese nicht entfernt wurde. Die Partition mit dem Update mit falscher Signatur wird dann final als fehlerhaft markiert und die Partition mit der älteren Firmware gebootet.

Ein Angreifer kann aber den Bootloader selbst austauschen, um die Überprüfung der Signatur zu umgehen. Der Bootloader wird daher mit dem Secure Boot Feature [18] vor diesem Austausch geschützt. Ähnlich der Signatur der Firmware wird hier eine Signatur für den Bootloader erzeugt: Diese wird aus einem AES Schlüssel, einem Initialization Vector und dem Inhalt des Bootloaders selbst erzeugt und an den Anfang des Flash-Speichers geschrieben. Der AES Schlüssel wird mit dem espsecure Werkzeug erzeugt: Der SHA256 Hash des privaten ECDSA Schlüssels wird als AES Schlüssel verwendet [18]. Der AES Schlüssel wird dann mit dem espefuse Werkzeug in die E-Fuses eingebrannt und kann damit nur einmal festgelegt werden.
Alternativ kann der Schlüssel auch intern durch den ESP32 erzeugt werden, allerdings besteht dann keine Möglichkeit mehr, um den Bootloader zu aktualisieren, da auf den AES Schlüssel für die Erstellung der Signatur nicht zugegriffen werden kann.
Die Dokumentation[11] des esp-idf SDKs enthält Details zur Verwendung und Konfiguration des Signierens der Firmware und des Secure Boot Features.

6.1.3. Symmetrische Verschlüsselung der Firmware

Die Firmware wird mit AES im Cipher Block Chaining (CBC) Modus [21, S. 230] auf dem Build Server verschlüsselt. AES-CBC nutzt zusätzlich zum symmetrischen Schlüssel einen zufällig erzeugten Initialization Vector (IV), um zu verhindern, dass durch die den Daten unterliegende Struktur Rückschlüsse auf die Daten gezogen werden können. Der AES Schlüssel ist 256 Bit lang und wird für jedes Update mit OpenSSL neu erstellt. Wenn die Größe der Firmware nicht durch 16 restlos teilbar ist, werden manuell Nullbytes an die Firmware angehängt, bis die Größe restlos durch 16 teilbar ist. Vor der Auffüllung mit Nullbytes muss die Größe der Firmware gespeichert werden, um nach dem Herunterladen der Firmware durch den ESP32 die Nullbytes wieder entfernen zu können. Der AES-CBC Schlüssel, IV und die Firmware Größe werden in der Meta Datei gespeichert. Der Anhang A.6.3 enthält Details zur Erzeugen des AES-CBC Schlüssels und IV und der Verschlüsselung der Firmware.

[11] Signatur der Firmware und Secure Boot https://docs.espressif.com/projects/esp-idf/en/v3.3.1/security/secure-boot.html, zuletzt aufgerufen: 10.06.2020

6.1.4. Asymmetrische Verschlüsselung der Meta Datei

Die Meta Datei wird auf dem Build Server mit dem öffentlichen RSA Schlüssel des ESP32 mit RSA verschlüsselt. Der RSA Schlüssel ist 2048 Bit (=256 Byte) lang und wird einmalig vor der Auslieferung des ESP32 mit OpenSSL erstellt. Hierbei ist zu beachten, dass nur die Anzahl an Byte verschlüsselt werden kann, die kleiner oder gleich mit der Schlüssellänge ist: Die Meta Datei darf maximal eine Größe von 245 Byte haben, da 11 Byte durch etwaige Header und Padding des verwendeten Public-Key Cryptography Standards (PKCS) #1 v1.5 belegt werden [22]. Der Anhang A.6.2 enthält Details zur RSA Schlüsselerstellung und Verschlüsselung.

6.1.5. Flash-Verschlüsselung

Die Flash-Verschlüsselung des ESP32 verwendet AES im Electronic Code Book (ECB) Modus [16]: AES-ECB wird durch die Hardware des ESP32 vorgegeben und kann nicht verändert werden. Von der Verwendung des ECB Modus wird aber grundsätzlich abgeraten [9, S. 336]. Da hier jeder Datenblock mit dem gleichen Schlüssel verschlüsselt wird, entsteht bei Datenblöcken mit gleichem Klartext derselbe Geheimtext. Ein Angreifer kann durch die Analyse der verschlüsselten Datenblöcke Rückschlüsse auf die unverschlüsselten Datenblöcke ziehen [21, S. 228 - 230]. Daher werden bei der Flash-Verschlüsselung des ESP32 zwei 16 Byte Datenblöcke zu einem 32 Byte Datenblock zusammengefasst und mit einem einzigartigen Schlüssel verschlüsselt. Der einzigartige Schlüssel wird aus dem festen Schlüssel für die Flash-Verschlüsselung und der Position des 32 Byte Datenblocks im Flash berechnet (key tweak).
Der feste Schlüssel für die Flash-Verschlüsselung kann wahlweise durch den ESP32 selbst oder extern erzeugt und dann auf den ESP32 übertragen werden. Der Schlüssel wird mit dem espefuse Werkzeug in die E-Fuses eingebrannt und kann damit nur einmal festgelegt werden. Wenn der Schlüssel durch den ESP32 erzeugt wird, gibt es keine Möglichkeit diesen auszulesen und Flash-Speicher von außen zu entschlüsseln. Hier wird der Schlüssel extern mit dem espsecure Werkzeug aus dem esp-idf SDK erzeugt, um den Flash-Speicher im Falle eines Bricks entschlüsseln und erneut flashen zu können.
Die NVS Partition kann nicht durch die Flash-Verschlüsselung verschlüsselt werden [23]. Daher muss die NVS Partition separat mit AES-XTS verschlüsselt werden. AES-XTS ist für die Verschlüsselung von Sektor oder Block basierten Speichern optimiert [24]. Die Nutzung von AES-XTS wird durch das esp-idf SDK vorgegeben. Die NVS Verschlüsselung setzt aktive Flash-Verschlüsselung voraus, da der AES-XTS Schlüssel in einer eigenen Partition im Flash gespeichert wird.

Die Schlüssel Partition kann wiederum durch die Flash-Verschlüsselung verschlüsselt werden. Der AES-XTS Schlüssel wird mit dem nvs_partition_gen Werkzeug aus dem esp-idf SDK erzeugt [23].
Die Dokumentation[12] des esp-idf SDKs enthält Details zur Verwendung und Konfiguration der Flash-Verschlüsselung.

6.1.6. Anpassung des Update-Prozesses

Der in Kapitel 4.4 vorgestellte Update-Prozess muss aufgrund der Hardwarelimitationen des ESP32 und der Vorgaben und Empfehlungen des esp-idf SDKs angepasst werden: Der RAM des ESP32 ist in vielen Fällen nicht groß genug, um ein vollständiges Firmware Update zwischenzuspeichern. Daher wird die Firmware in mehreren 1024 Byte großen Blöcken heruntergeladen, entschlüsselt und in den Flash-Speicher geschrieben. Beim letzten Block müssen, wenn vorhanden, die Nullbytes nach der Entschlüsselung wieder entfernt werden.
Durch die Integration der Versionsnummer und digitalen Signatur in die Firmware werden diese nicht mehr als Teil der Meta Datei übertragen. Bei der Überprüfung, ob ein neues Update vorliegt, muss daher der erste Block der Firmware zusätzlich zur Meta Datei heruntergeladen werden. Der Block wird dann entschlüsselt und die Versionsnummer entsprechend ausgelesen und mit der der aktiven Firmware verglichen.
Die digitale Signatur wird erst geprüft, nachdem das Update vollständig heruntergeladen, entschlüsselt und in den Flash-Speicher geschrieben wurde und nicht wie vorgesehen gleich nach den Entschlüsselung.

[12] Flash-Verschlüsselung https://docs.espressif.com/projects/esp-idf/en/v3.3.1/security/flash-encryption.html, zuletzt aufgerufen: 10.06.2020

6.2. Komponenten des Update-Systems

Abbildung 14 stellt das Update-System mit den einzelnen Komponenten schematisch dar. Die verwendeten Komponenten werden nachfolgend vorgestellt.

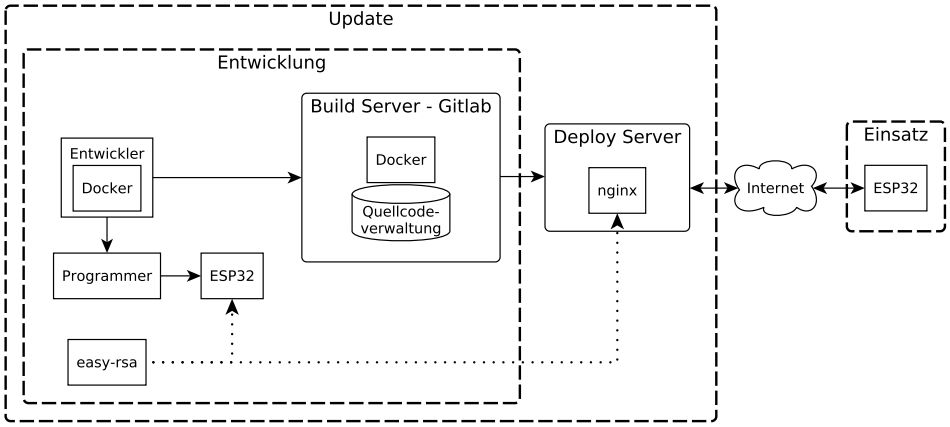


Abbildung 14: Komponenten des Update-Systems

6.2.1. Programmer und Auslieferung des ESP32

Das hier verwendete Entwicklerkit auf Basis des ESP32-WROOM32 Moduls integriert einen Programmer auf der Platine. Vor der Auslieferung werden die auf dem ESP32 benötigten Schlüssel und Zertifikate mit den WiFi Zugangsdaten in der NVS Partition zusammengefasst. Die NVS Partition wird vor dem Flashen separat mit dem AES-XTS Schlüssel verschlüsselt. Die Schlüssel für Secure Boot und Flash-Verschlüsselung werden erstellt und in die E-Fuses eingebrannt. Anschließend wird der Bootloader mit Signatur geflasht. Schließlich werden die Partitionstabellen mit den weiteren Partition und der initialen Firmware geflasht.

6.2.2. Quellcodeverwaltung und Build Server

Die Software Gitlab wird als Quellcodeverwaltung und Build Server verwendet. Gitlab ist eine vollständig integrierte Plattform für Softwareentwicklung. Für die Build Server Funktion wird Gitlabs Continuous Integration / Continuous Deployment beziehungsweise Continuous Delivery (CI/CD, CICD) Pipeline genutzt. Die Pipeline automatisiert wiederkehrende Schritte der Software Entwicklung und minimiert so den menschlichen Fehler. Die einzelnen Pipeline Stufen werden in einer Konfigurationsdatei hinterlegt [25].

Die entwickelte Pipeline besteht aus drei Stufen: *build*, *secure* und *deploy*. In der *build* Stufe wird die Firmware kompiliert. In der *secure* Stufe werden die Schutzmaßnahmen auf das Update angewendet. In der *deploy* Stufe werden Update und Meta Datei per scp auf den Deploy Server geladen.
Für die lokale Entwicklung und in der Pipeline wird ein Docker Container genutzt, um den Compiler für den ESP32, das esp-idf SDK und die durch das SDK benötigten Programme und Bibliotheken zu kapseln. So kann eine konsistente Entwicklungsumgebung realisiert, Kompatibilitätsprobleme beim Bauen der Firmware vermieden und die Anzahl an unterschiedlichen Softwarekomponenten reduziert werden.
Docker [26] ist eine Plattform, die Programme und eine Laufzeitumgebung für Verwaltung, Erzeugung und Ausführung von Containern bietet. Ein Docker Container ist eine lauffähige Instanz eines Docker Images. Ein Docker Image wird aus einem Dockerfile erzeugt, der eine Reihe Anweisungen enthält, wie das Docker Image aufzubauen ist.
Gitlab verwaltet die Docker Images in einer Registry und instantiiert bei der Verwendung in einer Pipeline automatisch den Docker Container [27]. Abbildung 15 zeigt die entwickelte Pipeline und das Zusammenspiel mit dem Docker Container.

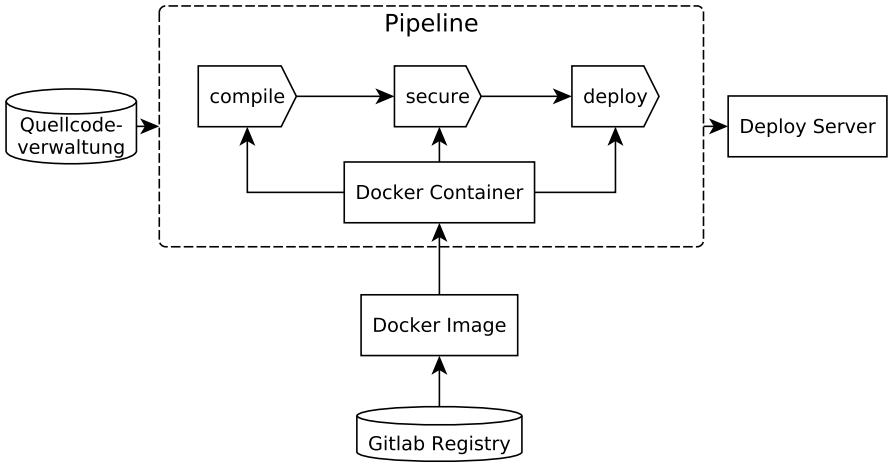


Abbildung 15: Die Gitlab Pipeline mit Docker Integration

Der Anhang A.2 enthält Details zur CICD Pipeline und der Anhang A.1 enthält Details zum Dockerfile.

Abbildung 16 stellt die Reihenfolge der Schutzmaßnahmen innerhalb der Pipeline dar: In Schritt (1) in der *build* Stufe werden der öffentliche Schlüssel zum Signieren der Firmware (FSS pub) und die Versionsnummer in die Firmware geschrieben. Die Firmware wird dann an die *secure* Stufe der Pipeline übergeben und in Schritt (2) mit dem privaten Schlüssel zum Signieren der Firmware (FSS prv) signiert. Schritt (3) füllt die Firmware mit Nullbytes auf, wenn die Größe der Firmware nicht durch 16 teilbar ist und schreibt die Größe des Updates in die Meta Datei. Danach wird in Schritt (4) der einmalige AES-CBC Schlüssel mit IV für das Update generiert und das Update damit verschlüsselt. Der AES-CBC Schlüssel mit IV ebenfalls in der Meta Datei gespeichert. Die Meta Datei wird schließlich in Schritt (5) mit dem öffentliche RSA Schlüssel (RSA pub) des ESP32 verschlüsselt.

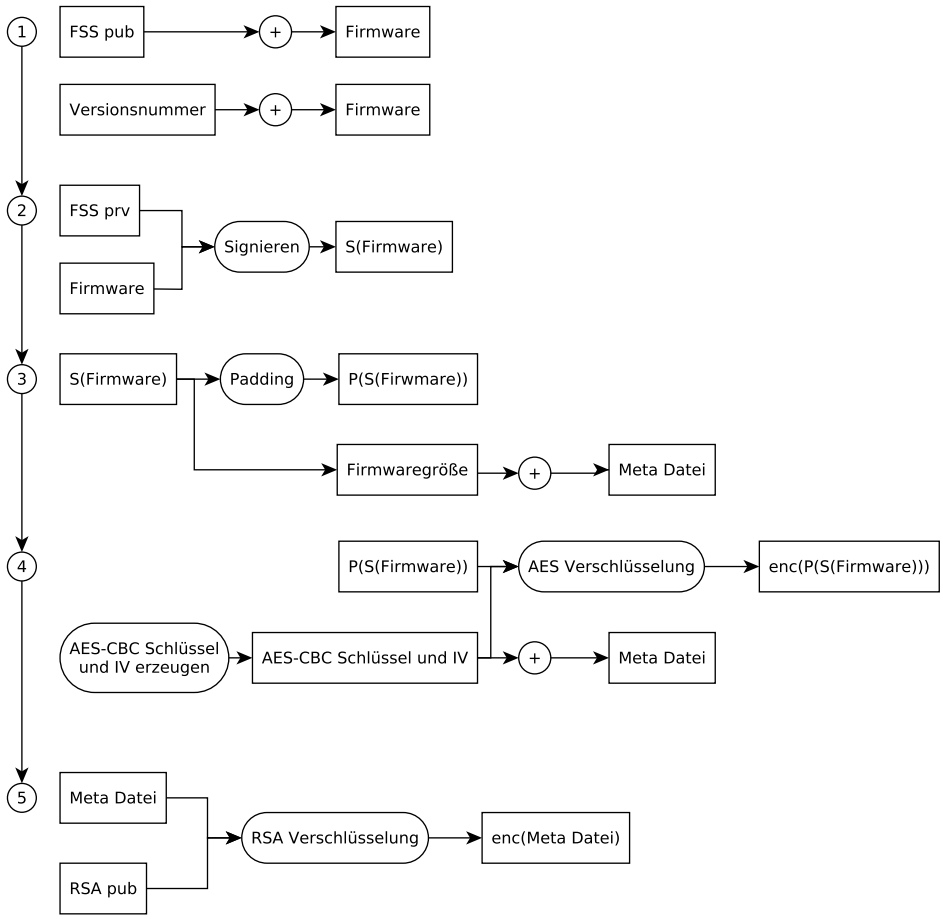


Abbildung 16: Die Anwendung der Schutzmaßnahmen in der Pipeline

6.2.3. Deploy Server

Das esp-idf Beispiel verwendet für die Bereitstellung der Updates den bei OpenSSL mitgelieferten, generischen TLS Server. Dieser kann zwar die Authentifizierung des ESP32 durchführen, bietet aber nicht die gewünschte Flexibilität für zukünftige Erweiterungen. Für die Verteilung der Updates wird daher der nginx[13] Webserver verwendet. Dieser kann einfach konfiguriert werden und unterstützt HTTPS.
Die Konfiguration lenkt alle versuchten HTTP Verbindungen auf Port 80 auf den TLS Port 443 um und lässt somit nur verschlüsselte Verbindungen zu. Verbindungsversuche durch einen Client ohne passendes Zertifikat werden verweigert. Der Anhang A.3 enthält Details zur Konfiguration von nginx.

6.2.4. easy-rsa PKI

Die PKI wird mit easy-rsa 3.0.6 aufgesetzt. esay-rsa zeichnet sich durch einfache Bedienbarkeit und frei verfügbaren Quellcode aus. Die Erstellung der Schlüssel und Zertifikate ist ohne Netzwerkzugang möglich, sodass nur die Übertragung auf die Zielsysteme geschützt werden muss. Die easy-rsa PKI wird durch die Erstellung des CA Zertifikats initialisiert. Anschließend können einmalig Zertifikatsignierungsanforderungen für den Deploy Server und den ESP32 erstellt und signiert werden. Nach der Bestätigung der Zertifikatsignierungsanforderung werden der private und der öffentliche Schlüssel des Deploy Servers zusammen mit dem CA Zertifikat auf den Deploy Server geladen. Dies geschieht analog für den ESP32. Der Anhang A.6.1 enthält Details zur Verwendung von easy-rsa.

[13] nginx https://www.nginx.com/, zuletzt aufgerufen: 10.06.2020

6.3. Update-System-spezifische Schwachstellen

Der Sicherheitsforscher LimitedResults hat mehrere Angriffe auf die Kryptographiebibliothek mbed TLS [28], die Kryptographiehardwarebeschleuniger [29], die Secure Boot Funktion [30] und Flash-Verschlüsselung [31] des ESP32 gefunden. Die Angriffe basieren auf Voltage Fault Injection: Hierbei wird gezielt die Spannung innerhalb des ESP32 manipuliert, um beispielsweise den Leseschutz des E-Fuse Controllers zu umgehen und so den Schlüssel für Secure Boot oder Flash-Verschlüsselung auszulesen. Der Angreifer kann mit diesen Angriffen sämtliche umgesetzte Schutzmaßnahmen auf dem ESP32 außer Kraft setzen, die Firmware vollständig auslesen und entschlüsseln sowie eigene Firmware unerkannt auf den ESP32 aufspielen.
Ein Angriff dieser Art setzt allerdings voraus, dass der Angreifer Hardwarezugriff hat und die EMV Abschirmung des hier verwendeten ESP32-WROOM32 Moduls ablötet. Dieser Angriff hat zwar katastrophale Folgen, aber das tatsächliche Schadenspotential wird als sehr gering eingestuft. Der Aufwand für den Angreifer ist sehr hoch und speziell bei mehreren hunderten oder tausenden ESP32 übersteigt der (logistische) Aufwand für den Angriff den Nutzen für den Angreifer.
Espressif Systems hat Hinweise zu den Angriffen auf Secure Boot [32] und Flash-Verschlüsselung [33] veröffentlicht: So wurden bereits Mitigationen in das esp-idf SDK als auch den Boot ROM neuerer ESP32 Chip Revisionen implementiert. Weiterhin empfiehlt Espressif Systems keine gemeinsamen Schlüssel für mehrere ESP32 zu verwenden.
Als zusätzliche Schutzmaßnahmen können manipulationssichere Gehäuse verwendet werden. Alternativ kann der ESP32 den Zustand des Gehäuses überwachen und bei unsachgemäßer Öffnung einen Alarm auslösen. Dabei muss allerdings sichergestellt werden, dass der Angreifer nicht einfach die Stromzufuhr unterbrechen kann oder aber der ESP32 über eine unabhängige Energiequelle verfügt.

7. Ausblick

Während der Umsetzung des Update-Systems wurden einige Ideen für Erweiterungen entwickelt. Diese werden hier vorgestellt und die Integration in das bestehende Update-System skizziert. Einige Ideen basieren auf dem modellierten Update-System aus Kapitel 4, andere basieren auf dem umgesetzten Update-System aus Kapitel 6.

7.1. Skalierung des Update-Systems

Das entwickelte Update-System kann nur ein smartes System mit OTA Updates versorgen. Der Vorteil der Skalierbarkeit von OTA Updates bleibt ungenutzt. Das Update-System kann wie in Kapitel 5.1 angesprochen erweitert werden.
Jedes smarte System benötigt eine Reihe an Schlüsseln und Zertifikaten. Diese werden in Verbindung mit einer eindeutigen ID für jedes smarte System in der Datenbank „Schlüssel“ gespeichert. Die Datenbank wird durch einen dedizierten Kryptographie Server „Krypto“ verwaltet, der auch die CA beinhaltet. Die Schlüssel der smarten Systeme werden dem Build Server zur Verfügung gestellt, damit dieser bei einer neuen Firmware Version die jeweiligen Meta Dateien und Firmware verschlüsseln kann. Der Build Server kompiliert und signiert die Firmware einmalig. Für jedes smarte System wird eine Kopie der signierten Firmware mit einem eigenen symmetrischen Schlüssel verschlüsselt. Die Meta Datei wird mit dem jeweiligen asymmetrischen Schlüssel des smarten Systems aus der Datenbank verschlüsselt. Die Firmware und die zugehörige Meta Datei werden in der Datenbank „Firmware“ zwischengespeichert.
Durch die Skalierung müssen, wie in Kapitel 4.3 angesprochen, auch der „Push“ und der „Pull“ Ansatz erneut bewertet und angepasst werden.
Beim bisherigen „Pull“ Ansatz überträgt das smarte System jetzt zusätzlich die eigene ID, wenn es den Deploy Server nach einem neuen Update fragt. Der Deploy Server ruft dann die entsprechende Meta Datei und, im Falle eines Updates, die Firmware aus der Datenbank „Firmware“ ab. Durch die größere Anzahl an smarten System, die den Deploy Server anfragen, könnte dieser aber überlastet werden. Eine Lösung wäre der Einsatz kommerzieller Cloud Services, die dynamisch auf erhöhte Belastungen reagieren können. Dies käme auch der Robustheit gegen DoS Bedrohungen zugute. Allerdings sind viele Angebote proprietär.

Beim „Push“ Ansatz melden sich die smarten Systeme beim Deploy Server mit ihrer ID an. Dieser erfasst dabei auch IP Adresse und Port des smarten Systems. Um zu verhindern, dass das smarte System unbemerkt eine neue IP Adresse zugewiesen bekommt, aktualisiert es regelmäßig die Anmeldung beim Deploy Server. Wenn ein neues Update vorliegt, kann der Deploy Server die smarten Systeme entsprechend benachrichtigen. So kann auch die Belastung des Deploy Servers besser reguliert werden, da durch die Benachrichtigungen der smarten Systeme gesteuert werden kann, wann und wie viele smarte Systeme ein Update herunterladen. Ähnlich dem „Pull“ Ansatz können kommerzielle Cloud Services verwendet werden, um die Belastung zu verteilen und mehr Updates gleichzeitig durchzuführen.

Der „Push“ Ansatz bietet sich jetzt eher an als der „Pull“ Ansatz, da durch die unterschiedlichen IDs bereits erfasst wird, welche smarten Systeme für ein Update infrage kommen. Weiterhin wird der Aufwand, die Anmeldung beim „Push“ Ansatz zu aktualisieren, geringer eingeschätzt als das Herunterladen der Meta Datei beim „Pull“ Ansatz.

Abbildung 17 stellt die Erweiterungen dar. Die Sicherheitsannahmen für die hinzugekommenen Datenflüsse und Komponenten werden durch die Trust Boundaries kenntlich gemacht. Der gestrichelte Datenfluss vom Krypto Server zum Deploy Server stellt das einmalige Übertragen des Zertifikates dar.

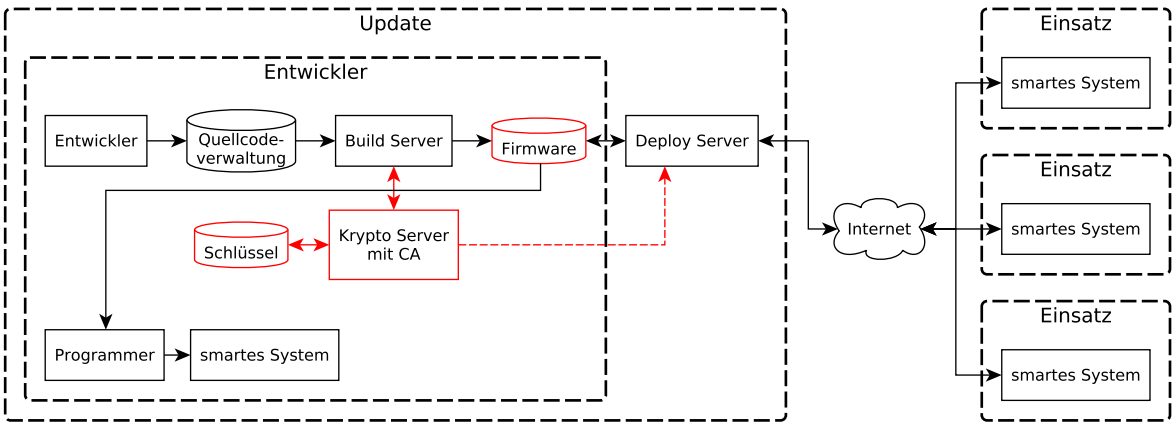


Abbildung 17: Skalierung des Update-Systems

7.2. Überwachung des Update-Prozesses

Durch die Skalierung erhöht sich das Risiko durch Repudiation Bedrohungen, welches in Kapitel 4.2.4 noch akzeptiert wurde. Um den Repudiation Bedrohungen zu begegnen, wird der Update-Prozess zentralisiert überwacht: Ein dedizierter Logging Server sammelt von Programmer, Build Server, Krypto Server, Deploy Server und jedem eingesetzten smarten System unterschiedliche Parameter und schreibt diese in die Datenbank „Log“. Die Absicherung des Logging Servers geschieht gleich der des Deploy Servers. Abbildung 18 stellt die Erweiterungen dar. Die Sicherheitsannahmen für die hinzugekommenen Datenflüsse und Komponenten werden durch die Trust Boundaries kenntlich gemacht. Der gestrichelte Datenfluss vom Krypto Server zum Logging Server stellt das einmalige Übertragen des Zertifikates dar.
Für die Überwachung werden folgende zu überwachende Parameter vorgeschlagen:

- Kompilation der Firmware: Zeitpunkt, Versionsnummer der Firmware, Signatur der Firmware, unverschlüsselte Firmware / Referenz auf den Quellcode in der Quellcodeverwaltung
- Hinzufügen / Entfernen eines smarten Systems: Zeitpunkt (Schlüsselerstellung und Flashing getrennt), ID des smarten Systems
- Update: Zeitpunkt (Anfrage / Anmeldung / Aktualisieren der Anmeldung durch smartes System, Beginn, Fertigstellung), ID des smarten Systems, Versionsnummer der Firmware vor / nach Update, etwaige Fehlermeldungen

Die Zeitstempel können genutzt werden, um Abweichungen bei der normalen Updatedauer zu erkennen. Diese können ein Hinweis auf beispielsweise einen DoS Angriff liefern. Die Erfassung der IDs erlaubt eine Eingrenzung der betroffenen smarten Systeme.

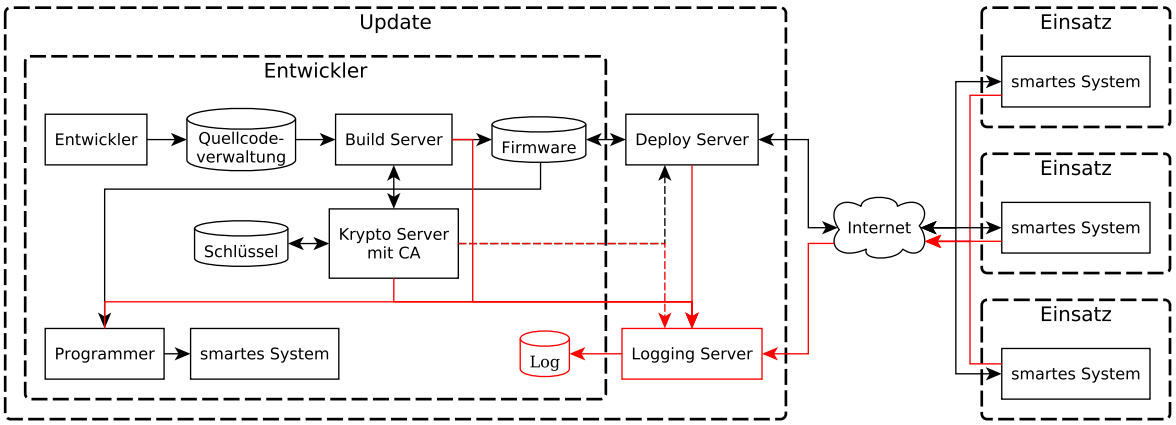


Abbildung 18: Überwachung des Update-Systems

7.3. ESP32 spezifische Vorschläge

Die folgenden Vorschläge beziehen sich konkret auf das in Kapitel 6 umgesetzte Update-System.

7.3.1. Erweiterung und Optimierung der Updatefunktionalität

Der Bootloader, die Partitionstabelle und die NVS Partition können nicht aktualisiert werden. Eine entsprechende Erweiterung des Update-Systems in diesem Bereich ist sinnvoll, um beispielsweise die Zertifikate und Schlüssel auf dem ESP32 oder den Bootloader zu aktualisieren.
Bei einem Firmware Update wird aktuell die gesamte Firmware heruntergeladen. Hier könnten die Unterschiede zur vorigen Version ermittelt werden und ein Update erstellt werden, was lediglich die Teile der Firmware aktualisiert, die sich verändert haben.
Ein weiteres nicht betrachtetes Feld ist der Energieverbrauch der OTA Updates in Kombination mit den Absicherungsmaßnahmen: Potentielle Einsparungen könnten sich durch Variieren der Größe der Zwischenpuffer auf dem ESP32 ergeben.
Auch die Aufteilung der einzelnen Aufgaben, Download eines Blocks, Entschlüsseln und in den Flashspeicher schreiben, auf eigene FreeRTOS Tasks könnte effizienter sein: Zwar würde mehr Strom benötigt werden, da die Hardware kurzzeitig stärker ausgelastet wird, aber die Verarbeitung wäre auch schneller abgeschlossen, was die Zeit bis zum Eintritt in tiefere Energiesparmodi verkürzen würde.

7.3.2. Nutzung anderer Schnittstellen, WLAN Mesh und Übertragungsprotokolle

Da die OTA API des esp-idf SDKs von der physischen Schnittstelle und des Protokolls unabhängig ist, können auch andere Übertragungsarten untersucht werden. Der ESP32 bietet einen Ethernet Media Access Controller (MAC) [10, S. 10]: Dieser benötigt zwar zusätzliche, externe Komponenten um an ein bestehendes Netzwerk angeschlossen zu werden [10, S. 31], kann dann aber mit der zusätzlichen Umsetzung des Power over Ethernet (PoE) Standards auch zur Spannungsversorgung des ESP32 genutzt werden [34]. Denkbar wäre dann die Verwendung des ESP32 als WLAN Router für andere IoT Geräte: Mit der WLAN Mesh Fähigkeit des ESP32 könnte ein Ad Hoc Netzwerk aufgespannt werden, welches dann die Verteilung von OTA Updates ermöglicht.

Espressif Systems bietet mit dem esp-mdf SDK[14] eine Grundlage, die bereits über einen eigenen OTA Updatemechanismus „Mupgrade“ [35] für ESP32 im Mesh verfügt. „Mupgrade“ ist aber hinsichtlich seiner Funktionsweise inkompatibel zu dem in dieser Arbeit implementierten Update-System: Der ESP32, der als Wurzel des Mesh Netzwerks agiert, lädt das Update herunter und verteilt dieses dann im Mesh. Folglich müsste entweder „Mupgrade“ angepasst werden oder das Update für jeden einzelnen Meshknoten müsste als normaler Datenverkehr durch das Mesh Netzwerk geroutet werden, was aber weniger effizient wäre.

Das Übertragungsprotokoll MQTT[15] könnte den Push Ansatz attraktiv machen: MQTT nutzt zur Verteilung von Nachrichten das Publish-Subscribe Prinzip. Dabei melden (subscribe) sich mehrere MQTT Clients (ESP32) bei einem MQTT Broker (Deploy Server) für unterschiedliche Themen (topic) an. Je nach Einstellungen für das jeweilige Thema wird die Nachricht dann an alle Clients verteilt (publish). Abbildung 19 stellt die Kommunikation über MQTT da: Um OTA Updates zu erhalten, melden sich die ESP32 beim Deploy Server für Nachrichten des „Update“ Themas an (1). Der Deploy Server schickt bei einem neuen Update (2) eine Nachricht an die ESP32 (3). Diese erwachen aus tieferen Energiesparmodi und laden das Update herunter (4, 5).

Eine Veränderung an der Authentifizierung oder Verschlüsselung der Verbindung ist nicht nötig, da MQTT ebenfalls TLS nutzt.

[14] esp-mdf SDK https://docs.espressif.com/projects/esp-mdf/en/latest/get-started/, zuletzt aufgerufen: 10.06.2020
[15] MQTT http://mqtt.org/, zuletzt aufgerufen: 10.06.2020

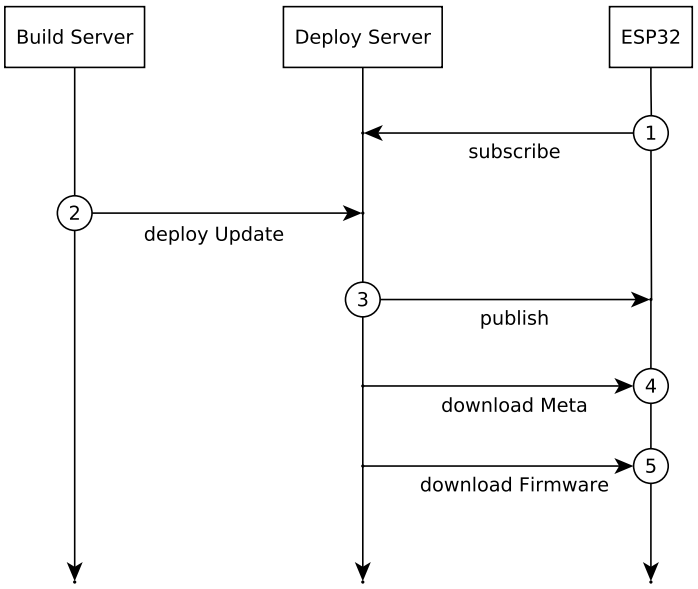


Abbildung 19: Ablauf eines Updates mit MQTT

7.4. Weitere Ansätze

Während der Recherche wurden einige interessante Ansätze gefunden, die als Grundlage für weitere Entwicklungen genutzt werden können.
Die Studie von Halder et al. [3] gibt einen Überblick über unterschiedliche Ansätze in der Automobilbranche. Abbildung 20 zeigt die untersuchten Kategorien von Schutzmaßnahmen. Besonders der Blockchain und Steganographie Ansatz sind hierbei hervorzuheben, da diese die Übertragung und Absicherung eines Updates anders als die anderen Ansätze realisieren.
Der Blockchain Ansatz nutzt ein Overlay Netzwerk, um die Updates dezentral zu verteilen. Eine Blockchain wird zur sicheren Speicherung und Verteilung der Updates genutzt. Es existiert eine Proof-of-concept Implementierung, die besser als eine Zertifikat basierte Architektur funktionieren soll und als Basis für eigene Entwicklungen verwendet werden kann.
Der Steganographie Ansatz verwendet zur Verschlüsselung der Firmware eine modifizierte Version des RSA Algorithmus. Die verschlüsselte Firmware wird durch Steganographie entlang der Ecken eines Cover Bildes versteckt. Die Kommunikation wird so vor einem Angreifer versteckt [9, S. 339]. Dieser Ansatz hat aber einen geringeren Datendurchsatz bei der Installation eines Updates.

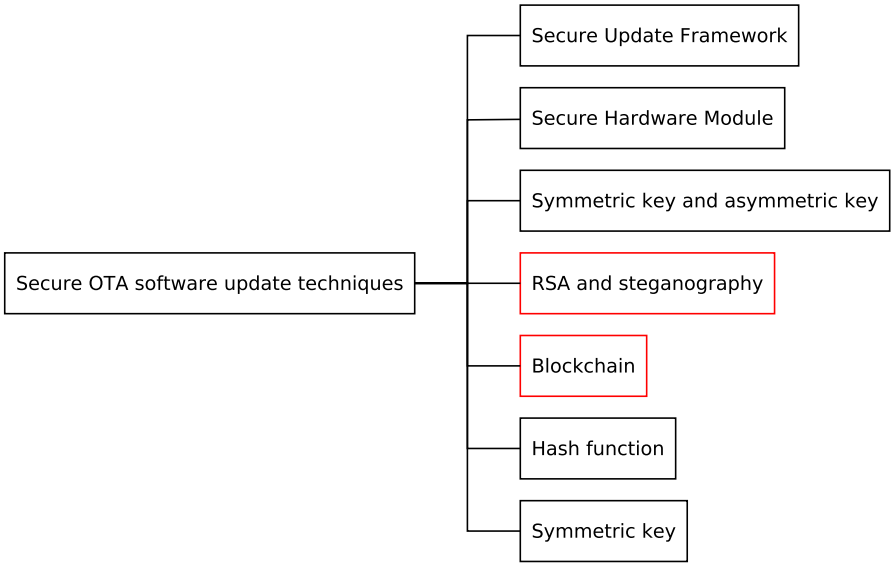


Abbildung 20: Überblick über Verfahren zur Absicherung von OTA Software Updates [3, S. 11]

Marawaha [36] entwickelte in seiner Masterarbeit ein System zur Verteilung von OTA Updates an einen Bluetooth Low Energy (BLE) fähigen Mikrocontroller. Die besondere Schwierigkeit liegt hier in der Verbindung des Internets mit BLE. Um die Updates über das Internet verteilen zu können, wurde das 6LoWPAN (IPv6 over Low power Wireless Personal Area Network) Protokoll verwendet. Ein Gateway übernahm die Vermittlung zwischen BLE und dem Internet.
Für die Überwachung wird der OMA-LwM2M (Lightweight machine-to-machine von OMA SpecWorks) Standard[16] verwendet. Die Übertragung der Firmware Updates erfolgt über CoAP (Constrained Application Protocol).
Gore et al. [37] entwickelten ein System für ein Mesh Netzwerk aus ESP8266 mit einem Raspberry Pi als Gateway. Zwar wurde hier der Vorgänger des ESP32 verwendet, aber die zugrundeliegenden Ideen hinsichtlich Verteilung der Updates im Mesh Netzwerk und die Absicherung der Übertragung sollten übertragbar sein.

[16] OMA-LwM2M https://omaspecworks.org/what-is-oma-specworks/iot/lightweight-m2m-lwm2m/, zuletzt aufgerufen: 10.06.2020

8. Fazit

In dieser Masterarbeit wurde ein Update-System für die automatisierte, netzwerkbasierte und sichere Bereitstellung von Firmware Updates für ein einzelnes smartes System entwickelt. Dabei erstellt das Update-System ausgehend vom Quellcode ein Firmware Update und verteilt dieses dann abgesichert über das Internet an das smarte System. Der gesamte Prozess läuft vorwiegend automatisch ab.
Als Grundlage wurden zuerst Modelle des Update-Systems und des Update-Prozesses erstellt und mit dem CIA Modell Anforderungen an die Vertraulichkeit, die Integrität und die Verlässlichkeit des Update-Prozesses formuliert.
Um die Sicherheit gewährleisten zu können, wurde eine Bedrohungsanalyse mit dem STRIDE Bedrohungsmodell und dem Elevation of Privilege Kartenspiel durchgeführt. In allen Kategorien außer der Elevation of Privilege Kategorie wurden Bedrohungen gefunden.
Für die gefundenen Bedrohungen, mit Ausnahme der Repudiation Bedrohungen, wurden Schutzmaßnahmen ausgewählt und in das Update-System integriert. Das Risiko der Bedrohungen der Repudiation Kategorie wurde akzeptiert, da die entsprechende Schutzmaßnahme die Angriffsfläche des Update-Systems vergrößert hätte.
Das Modell des Update-Systems wurde auf Vollständigkeit analysiert. Dabei wurde festgestellt, dass es keine unklaren Bereiche gab. Allerdings konnten, begründet durch den hohen Abstraktionsgrad des Modells, nur allgemeine Aussagen getroffen werden. Dies zeigte sich speziell bei den Schutzmaßnahmen, da hier einige Anpassungen bei der Implementierung nötig waren.
Das modellierte Update-System wurde für das smarte System ESP32 umgesetzt. Bei der Umsetzung wurde primär auf weit verbreitete Open-Source-Komponenten gesetzt. Der Quellcode wurde mit Gitlab verwaltet. Über die Docker Integration von Gitlab wurde der Quellcode in einem Docker Container automatisch kompiliert. Das Firmware Update wurde auf einen nginx Webserver geladen und von dort durch den ESP32 heruntergeladen. Die zuvor ausgewählten Schutzmaßnahmen wurden, wo möglich, den Empfehlungen und Vorgaben des verwendeten esp-idf SDKs entsprechend umgesetzt. Weiterhin wurde auf die Unterstützung der kryptographischen Algorithmen und Verfahren durch die Rechenbeschleuniger des ESP32 geachtet.

Während der Umsetzung wurden durch den Sicherheitsforscher LimitedResults Schwachstellen in der Hardware des ESP32 entdeckt, die mehrere der implementierten Schutzmaßnahmen wirkungslos machen. Entsprechende Gegenmaßnahmen wurden in spätere Hardwarerevisionen durch den Hersteller integriert. In dieser Masterarbeit wurde zwar ein anfälliger ESP32 verwendet, aber der Aufwand für einen Angriff wurde als zu hoch eingeschätzt, als dass eine reelle Gefahr von diesem ausginge.
Schließlich wurden eine Reihe von Vorschlägen für die Erweiterung des bestehenden Update-Systems entwickelt, namentlich die Skalierung auf mehrere smarte Systeme und die Überwachung des Update-Prozesses.

A. Anhang

A.1. Dockerfile

Listing 1 zeigt den Dockerfile, welches die Grundlage für das Docker Image bildet: In Zeile 1 wird das zugrundeliegende Docker Image *debian:buster-slim* spezifiziert. Zeilen 4, 12, 17, 21 und 26 führen Befehle innerhalb des Images aus: Zeile 4 installiert die benötigten Bibliotheken und Programme. Zeile 12 lädt den Cross Kompiler herunter und richtet diesen ein. Zeile 17 checkt das esp-idf SDK v3.3.1 aus. Zeile 21 installiert weitere vom SDK benötigte Python Programme und Bibliotheken. Zeile 26 setzt die Rechte für die /esp-idf/ Bibliothek so um, dass jeder die Inhalte dieses Ordners lesen, schreiben und ausführen können. Dies ist nötig, da Docker standardmäßig mit root Rechten läuft und somit sämtliche erzeugte Dateien ausschließlich root gehören. Zeilen 29 bis 31 schließlich setzen eine Umgebungsvariablen mit Pfaden auf das SDK, den Kompiler und den aktuellen Ordner.

A.2. Gitlab CICD Pipeline Konfiguration

Die .gitlab-ci.yml Datei enthält die Konfiguration der Gitlab CICD Pipeline im YAML[17] Format. Listing 2 zeigt die .gitlab-ci.yml Datei. Mit *image* wird das Docker Image ausgewählt, das in der Pipeline verwendet wird. *stages* listet die Pipeline Stufen in der Reihenfolge der Abarbeitung auf. Die einzelnen Stufen *build*, *secure* und *deploy* werden mit *stage* definiert. Die Befehle nach *before-script* und *script* beschreiben, was in der jeweiligen Pipeline Stufe getan werden soll. *artifacts* schließlich die Ergebnisse einer Pipeline Stufe zu speichern und einer zweiten zur Verfügung zu stellen.
Damit Zugangsdaten, Schlüssel und Zertifikate nicht im Klartext in der Konfigurationsdatei oder dem Repository hinterlegt werden müssen, werden diese als Variablen[18] angelegt. In der Pipeline kann über *$Variablenname* auf diese zugegriffen werden. Die genutzten Variablen und ihre Funktion werden in Tabelle 1 aufgeführt.

[17] https://docs.gitlab.com/ee/ci/yaml/
[18] https://docs.gitlab.com/ee/ci/variables/

```
1  FROM debian:buster-slim
2
3  # install required packages
4  RUN apt-get -qq update \
5          && apt-get -y dist-upgrade \
6          && apt-get install --no-install-recommends -y openssh-client build-
               essential gcc git wget make libncurses-dev flex bison gperf python
               python-pip python-setuptools python-serial python-cryptography python-
               future \
7          && apt-get -y  autoremove \
8          && apt-get autoclean \
9          && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*
10
11 # download the cross compiler
12 RUN wget https://dl.espressif.com/dl/xtensa-esp32-elf-linux64-1.22.0-80-g6c4433a
       -5.2.0.tar.gz \
13         && tar -xzf xtensa-esp32-elf-linux64-1.22.0-80-g6c4433a-5.2.0.tar.gz -C /
               opt \
14         && rm xtensa-esp32-elf-linux64-1.22.0-80-g6c4433a-5.2.0.tar.gz
15
16 # download the ESP IDF source code
17 RUN git clone -b v3.3.1 --recursive https://github.com/espressif/esp-idf.git \
18         && cd /esp-idf/ \
19         && git submodule update --init --recursive
20
21 RUN pip install wheel \
22         && wget https://raw.githubusercontent.com/espressif/esp-idf/v3.3.1/
               requirements.txt \
23         && /bin/bash -c "/usr/bin/python -m pip install -r requirements.txt" \
24         && rm requirements.txt
25
26 RUN chmod -R a+rwx /esp-idf/
27
28 # set required paths
29 ENV IDF_PATH /esp-idf/
30 ENV PATH /opt/xtensa-esp32-elf/bin:$PATH
31 ENV PWD /app/
```

Listing 1: Dockerfile

```
image: $DOCKER_IMAGE

stages:
  - build
  - secure
  - deploy

build:
  stage: build
  script:
    - make -j$(nproc) app
  artifacts:
    paths:
    - build

secure:
  stage: secure
  script:
    - $IDF_PATH/components/esptool_py/esptool/espsecure.py sign_data --keyfile
        $FSK_PRV build/app-template.bin
    - echo "sha=$(sha256sum build/app-template.bin | cut -d' ' -f1)" > build/
        aes_key
    - ./padder.sh
    - openssl enc -aes-256-cbc -nosalt -pbkdf2 -iter 1000 -p -pass pass:`dd if=/
        dev/urandom count=32 bs=1 status=none | base64` -in build/app-template.bin
         -out build/app-template.bin.enc >> build/aes_key
    - openssl rsautl -in build/aes_key -out build/aes_key.enc -certin -inkey
        $AES_SIGNING_CERT -verify -encrypt -pkcs
    - mv build/aes_key.enc build/aes_key
  artifacts:
    paths:
    - build

deploy:
  stage: deploy
  before_script:
    - eval $(ssh-agent -s)
    - echo "$SSH_PRIVATE_KEY" | tr -d '\r' | ssh-add -
    - mkdir -p ~/.ssh
    - chmod 700 ~/.ssh
    - echo "$SSH_KNOWN_HOSTS" > /root/.ssh/known_hosts
    - chmod 644 ~/.ssh/known_hosts
  script:
    - scp build/aes_key $DEPLOY_USER@$DEPLOY_SERVER:$DEPLOY_DIR/key
    - scp build/app-template.bin.enc $DEPLOY_USER@$DEPLOY_SERVER:$DEPLOY_DIR/
        firmware.bin
  artifacts:
    paths:
    - build
```

Listing 2: .gitlab-ci.yml Konfigurationsdatei

Name	Typ	Funktion
DOCKER_IMAGE	Variable	enthält das zu verwendende Docker Image
DEPLOY_DIR	Variable	enthält den Zielordner auf dem Deploy Servers für scp
DEPLOY_SERVER	Variable	enthält die Adresse des Deploy Servers für scp
DEPLOY_USER	Variable	Nutzer für scp mit Zugriffsrechte auf den durch $DEPLOY_DIR beschriebenen Ordner auf dem durch $DEPLOY_SERVER beschriebenen Deploy Server
SSH_KNOWN_HOSTS	Variable	Inhalt der known_hosts Datei für scp
SSH_PRIVATE_KEY	Variable	SSH Schlüssel für scp
FSK_PUB	File	öffentlicher ECDSA Schlüssel für das Signieren der Firmware
FSK_PRV	File	privater ECDSA Schlüssel für das Signieren der Firmware
RSA_PUB	File	öffentlicher RSA Schlüssel für das Verschlüsseln der Meta Datei

Tabelle 1: In der CICD Pipeline verwendete Variablen

A.3. nginx Konfiguration

Listing 3 zeigt die nginx Konfiguration. Der erste server Block leitet alle HTTP Anfragen nach HTTPS um. Der zweite server Block enthält die Konfiguration für HTTPS Verbindungen. Zeilen 13, 14 und 16 enthalten den Pfad zum Zertifikat und privaten Schlüssel des Deploy Servers, dem Zertifikat der CA und Zeile 15 aktiviert die Client Authentifizierung.

A.4. Anpassen der Flash Partitionstabelle

Listing 4 zeigt die angepasste Partitionstabelle: Hier wurde die *factory* Partition entfernt und der freigewordene Speicher auf die *ota* Partition aufgeteilt. Auch die Positionen und Größen der anderen Partitionen wurde angepasst, um den vorhandenen Speicher effizienter zu nutzen, allen voran die NVS Partition, da diese diverse Zertifikate und Schlüssel enthält. Auch die Position der Partitionstabelle wurde angepasst, um mehr Platz für etwaige Updates des Bootloaders zu schaffen.

A.5. NVS Partition

Der Inhalt der NVS Partition wird in einer CSV Datei festgelegt, welche dann durch das nvs_partition_gen Werkzeug in eine Binärdatei übersetzt wird. Listing 5 zeigt die verwendete CSV Datei: Zeile 1 gibt die Bedeutung der einzelnen Werte an. Zeile 2 definiert den Namespace „config“, in dem sich die darauf folgenden Daten befinden. Zeilen 3 und 4 stellen die Wi-Fi Zugangsdaten dar, welche als nullterminierte Strings angelegt werden. Zeilen 5 bis 8 umfassen die Zertifikate und Schlüssel, welche als Dateipfad angegeben werden und durch nvs_partition_gen in der NVS Partition als Binärdaten angelegt werden. Hierbei ist zu beachten, dass die Dateien mit keinem Nullterminator in die Partition geschrieben werden und bei der Verwendung als C String in der Firmware manuell mit einem Nullterminator versehen werden müssen.

```
server {
        listen          80;
        server_name     DEPLOY_SERVER_URL;
        return          301 https://DEPLOY_SERVER_URL$request_uri;
}

server {
        listen          443 ssl default_server;
        listen          [::]:443 ssl default_server;
        server_name     DEPLOY_SERVER_URL;
        root            /var/www/DEPLOY_SERVER_URL;
        index           index.html;
        ssl_certificate         /root/certs/DEPLOY_SERVER_URL/DEPLOY_SERVER_URL.
            crt;
        ssl_certificate_key     /root/certs/DEPLOY_SERVER_URL/DEPLOY_SERVER_URL.
            key;
        ssl_verify_client       on;
        ssl_client_certificate  /root/certs/DEPLOY_SERVER_URL/ca.crt;
}
```

Listing 3: nginx Konfigurationsdatei für den Server

```
# Name,   Type, SubType , Offset  , Size,  Flags
otadata,  data, ota     ,         , 0x002000
phy_init, data, phy     ,         , 0x001000
nvs_key,  data, nvs_keys,         , 0x001000
nvs,      data, nvs     , 0x020000, 0x020000
ota_0,    app,  ota_0   , 0x040000, 0x1E0000
ota_1,    app,  ota_1   ,         , 0x1E0000
```

Listing 4: angepasste Flash Partitionierungstabelle

```
key,type,encoding,value
config,namespace,,
SSID,data,string,"$WIFI_SSID"
PASS,data,string,"$WIFI_PASSWORD"
CA_CERT,file,binary,/keys/ca.crt
RSA_PRV,file,binary,/keys/rsa_prv.pem
HTTPS_CRT,file,binary,/keys/esp32.crt
HTTPS_KEY,file,binary,/keys/esp32.key
```

Listing 5: CSV Datei mit Inhalten der NVS Partition

A.6. Erstellen von Schlüsseln und Zertifikaten

A.6.1. CA, HTTPS Zertifikate und Schlüssel

Listing 6 zeigt die Befehle, die für das Initialisieren der PKI und der Schlüssel- und der Zertifikatserstellung für die CA genutzt werden.
Listing 7 erstellt die kryptographischen Schlüsselpaare, Zertifizierungsanfragen und Zertifikate für ESP32 und Deploy Server. Hierbei ist zu beachten das die vollständige URL des Deploy Servers als *Common Name* gesetzt werden muss. Weiterhin werden die Zertifikate nicht mit Passwörtern abgesichert, da das esp-idf SDK dies nicht unterstützt.

A.6.2. RSA

Listing 8 zeigt die Erstellung des 2048 Bit langen, privaten RSA Schlüssel *rsa_prv.pem* mit OpenSSL. Anschließend wird der öffentliche Schlüssel *rsa_pub.pem* aus dem privaten Schlüssel extrahiert.
Listing 9 zeigt den Befehl für die RSA Verschlüsselung: Die zu verschlüsselnde Datei (*$FILE_TO_ENCRYPT*) wird mit dem öffentlichen Schlüssel *$RSA_PUB* verschlüsselt (*$ENCRYPTED_FILE*). Die anderen Optionen geben Auskunft über das Format des Schlüssels.

A.6.3. AES-CBC

Der AES-CBC Schlüssel und IV wird mit OpenSSL erstellt. Hierbei ist zu beachten, dass der Wert für den *-iter* Parameter größer gewählt werden sollte[19]. Da es im esp-idf SDK keine Möglichkeit zur Verwendung eines Passwort Salts gibt, werden 32 Byte aus */dev/random* gelesen, nach base64 kodiert und so ein zufälliges Passwort erstellt. Listing 10 zeigt den Befehl für die AES-CBC Verschlüsselung, wobei der Schlüssel generiert, die zu verschlüsselnde Datei (*$FILE_TO_ENCRYPT*) verschlüsselt (*$ENCRYPTED_FILE*) und der Schlüssel dann in die Schlüsseldatei (*$KEY_FILE)* geschrieben wird.

[19] https://www.openssl.org/docs/man1.1.1/man1/enc.html

```
$ ./easyrsa init-pki
$ ./easyrsa build-ca
```

Listing 6: Initialisierung der PKI und CA

```
$ ./easyrsa gen-req esp32 nopass
$ ./easyrsa sign-req client esp32
$ ./easyrsa gen-req DEPLOY_SERVER_URL nopass
$ ./easyrsa sign-req server DEPLOY_SERVER_URL
```

Listing 7: Erzeugen von kryptographischen Schlüsselpaaren, Zertifizierungsanfragen und Zertifikaten

```
$ openssl genrsa -out rsa_prv.pem 2048
$ openssl rsa -in rsa_prv.pem -pubout -out rsa_pub.pem
```

Listing 8: RSA Schlüsselerstellung

```
$ openssl rsautl -in $FILE_TO_ENCRYPT -out $ENCRYPTED_FILE -keyform PEM -pubin -
    inkey $RSA_PUB -verify -encrypt -pkcs
```

Listing 9: RSA Verschlüsselung

```
$ openssl enc -aes-256-cbc -nosalt -pbkdf2 -iter 1000 -p -pass pass:‘dd if=/dev/
    urandom count=32 bs=1 status=none | base64‘ -in $FILE_TO_ENCRYPT -out
    $ENCRYPTED_FILE >> $KEY_FILE
```

Listing 10: Erstellung des AES-CBC Schlüssels und IV und Verschlüsselung

Literatur

[1] Working paper updating firmware of embedded systems in the internet of things. Technical report, International Working Group on Data Protection in Telecommunications, 2017.

[2] O. C. Novac and M. Novac and C. Gordan and T. Berczes and G. Bujdosó. Comparative study of Google Android, Apple iOS and Microsoft Windows Phone mobile operating systems. In *2017 14th International Conference on Engineering of Modern Electric Systems (EMES)*, pages 154–159, June 2017.

[3] Subir Halder, Amrita Ghosal, and Mauro Conti. Secure OTA software updates in connected vehicles: A survey. *CoRR*, abs/1904.00685, 2019.

[4] Kevin Daimi, Mustafa Saed, Scott Bone, , and Muhammad Rizwan. Securing Vehicle ECUs Update Over the Air. 2016.

[5] Sabir Idrees, Hendrik Schweppe, Yves Roudier, Marko Wolf, Dirk Scheuermann, and Olaf Henniger. Secure automotive on-board protocols: A case of over-the-air firmware updates. pages 224–238, 03 2011.

[6] Nick Lethaby. A more secure and reliable OTA update architecture for IoT devices. 2018.

[7] Mongoose OS - reduce IoT firmware development time up to 90%. https://mongoose-os.com/. Last visited: 10.06.2020.

[8] Automatische Firmware-Updates für Microcontroller mit Gitlab und PlatformIO. https://byte-style.de/2018/01/automatische-updates-fuer-microcontroller-mit-gitlab-und-platformio/. Last visited: 10.06.2020.

[9] Adam Shostack. *Threat modeling: Designing for security*. John Wiley & Sons, 2014.

[10] Espressif Systems. *ESP32 Series Datasheet*. Version 2.7.

[11] Jason Andress. *The basics of information security: understanding the fundamentals of InfoSec in theory and practice*. Syngress, 2014.

[12] Loren Kohnfelder and Praerit Garg. The threats to our products. *Microsoft Interface, Microsoft Corporation*, 33, 1999.

[13] Willi Flühmann and Sebastian Gerstl. Automatisches Firmware-Update für Embedded-Linux. https://www.embedded-software-engineering.de/automatisches-firmware-update-fuer-embedded-linux-a-833310/. Last visited: 10.06.2020.

[14] Mohammed El-hajj, Ahmad Fadlallah, Maroun Chamoun, and Ahmed Serhrouchni. A survey of internet of things (iot) authentication schemes. *Sensors*, 19(5):1141, Mar 2019.

[15] NIST FIPS Pub. 197: Advanced encryption standard (aes). *Federal information processing standards publication*, 197(441):0311, 2001.

[16] Espressif Systems. Flash Encryption. https://docs.espressif.com/projects/esp-idf/en/v3.3.1/security/flash-encryption.html. Last visited: 10.06.2020.

[17] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[18] Espressif Systems. Secure Boot. https://docs.espressif.com/projects/esp-idf/en/v3.3.1/security/secure-boot.html. Last visited: 10.06.2020.

[19] FIPS PUB. Secure hash standard (shs). *Fips pub*, 180(4), 2012.

[20] Espressif Systems. Over The Air Updates (OTA). https://docs.espressif.com/projects/esp-idf/en/v3.3.1/api-reference/system/ota.html. Last visited: 10.06.2020.

[21] Alfred J Menezes, Jonathan Katz, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 1996.

[22] Rsa encryption maximum data size. https://tls.mbed.org/kb/cryptography/rsa-encryption-maximum-data-size. Last visited: 10.06.2020.

[23] Espressif Systems. Non-volatile storage library. https://docs.espressif.com/projects/esp-idf/en/v3.3.1/api-reference/storage/nvs_flash.html. Last visited: 10.06.2020.

[24] IEEE 1619 Security in Storage Working Group et al. Ieee p1619/d19: Draft standard for cryptographic protection of data on block-oriented storage devices, 2007.

[25] Introduction to CI/CD with GitLab. https://docs.gitlab.com/ee/ci/introduction/. Last visited: 10.06.2020.

[26] Docker overview. https://docs.docker.com/get-started/overview/. Last visited: 10.06.2020.

[27] Using Docker images. https://docs.gitlab.com/ee/ci/docker/using_docker_images.html. Last visited: 10.06.2020.

[28] LimitedResults. Pwn MBedTLS on ESP32: DFA Warm-up. https://limitedresults.com/2019/05/pwn-mbedtls-on-esp32-dfa-warm-up/. Last visited: 10.06.2020.

[29] LimitedResults. Pwn the ESP32 crypto-core. https://limitedresults.com/2019/08/pwn-the-esp32-crypto-core/. Last visited: 10.06.2020.

[30] LimitedResults. Pwn the ESP32 Secure Boot. https://limitedresults.com/2019/09/pwn-the-esp32-secure-boot/. Last visited: 10.06.2020.

[31] LimitedResults. Pwn the ESP32 Forever: Flash Encryption and Sec. Boot Keys Extraction. https://limitedresults.com/2019/11/pwn-the-esp32-forever-flash-encryption-and-sec-boot-keys-extraction/. Last visited: 10.06.2020.

[32] Espressif Systems. Espressif Security Advisory Concerning Fault Injection and Secure Boot (CVE-2019-15894). https://www.espressif.com/en/news/Espressif_Security_Advisory_Concerning_Fault_Injection_ Last visited: 10.06.2020.

[33] Espressif Systems. Security Advisory concerning fault injection and eFuse protections (CVE-2019-17391). https://www.espressif.com/en/news/Security_Advisory_Concerning_Fault_Injection_and_eFus Last visited: 10.06.2020.

[34] Helga Hansen. Internet der dinge: Olimex bringt neue esp32-boards. https://www.heise.de/make/meldung/Internet-der-Dinge-Olimex-bringt-neue-ESP32-Boards-4123706.html. Last visited: 10.06.2020.

[35] Espressif Systems. Mupdgrade. https://docs.espressif.com/projects/esp-mdf/en/latest/api-guides/mupgrade.html. Last visited: 10.06.2020.

[36] Manas Marawaha. IoT Firmware Management: Over the Air Firmware Management for Constrained Devices using IPv6 over BLE. Master's thesis, University of Dublin, Trinity College, 2017.

[37] Shreya Gore and Shraddha Kadam and Shraddha Mallayanmath and Shruti V Jadhav. Review on Programming ESP 8266 with Over the Air Programming Capability. 2017.

Abbildungsverzeichnis

Tabellenverzeichnis