FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	Security of IoT Devices Based on ESP32
Student:	Bc. Michal Vácha
Supervisor:	Ing. Jiří Dostál, Ph.D.
Study Programme:	Informatics
Study Branch:	Computer Security
Department:	Department of Information Security
Validity:	Until the end of summer semester 2020/21

Instructions

A typical use case for IoT is collecting telemetry data (temperature, humidity...) and sending them to the cloud for further processing. The security of these devices so far has been given a low priority, which limits their usage in the industry [1].

Analyze current threats related to IoT devices and propose suitable countermeasures.
Describe the ESP32 platform, focusing on its security features.
Assess known vulnerabilities in ESP32.
Compare communication protocols used in IoT (HTTP, MQTT, and COAP) regarding their communication effectiveness, reliability, and security.
Describe various ways of device identification by the server (e.g. by using PKI, PUF...). Choose one method and implement it.
Create an app that collects telemetry data and sends them to a cloud-based backend in a secure way.
Describe the options of security over the air update (OTA) for an app running on ESP32.

References

[1] https://www.bain.com/insights/cybersecurity-is-the-key-to-unlocking-demand-in-the-internet-of-things/

prof. Ing. Róbert Lórencz, CSc.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 13, 2020

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Security of IoT Devices Based on ESP32

Bc. Michal Vácha

Department of Information Security
Supervisor: Ing. Jiří Dostál, Ph.D.

August 4, 2020

Acknowledgements

I want to thank my parents for supporting throughout my studies, my friends for distracting me when working on my thesis, and my supervisor for thoughtful discussions about the IoT security and last-minute advice.

Declaration

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

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

In Prague on August 4, 2020 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Vácha, Michal. *Security of IoT Devices Based on ESP32*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstrakt

Cílem této práce je analyzovat stav zabezpečení ESP32, což je dnes jedna z nejčastěji používaných IoT platforem. Analyzuje současné hrozby pro zařízení IoT, často používané protokoly IoT, dostupné bezpečnostní funkce a známé zranitelnosti platformy ESP32. Součástí této práce je ukázková aplikace, která ukazuje, jak lze pomocí Secure Boot, Flash Encryption, vzdálených aktualizací a vzájemné autentizace přes TLS s využitím HSM modulu ATECC608A vytvořit bezpečné IoT řešení pro vzdálený sběr dat. Poznatky z analýzy a implementace jsou poté shrnuty a diskutovány.

Klíčová slova ESP32, Bezpečnost IoT, Secure Boot, ATECC608A, Flash Encryption, Azure IoT Hub

Abstract

The focus of this thesis is to analyze the state of ESP32 security, which is one of the most commonly used IoT platforms today. It analyzes the current threats to IoT devices, frequently used IoT protocols, available security features, and known vulnerabilities of the ESP32 platform. Part of this thesis is a proof of concept app that shows how Secure Boot, Flash Encryption, OTA updates, and TLS Mutal Authentication using the ATECC608A HSM module can be used to create a secure IoT solution for remote telemetry. Learnings from the analysis and the implementation are then summarized and discussed.

Keywords ESP32, IoT Security, Secure Boot, ATECC608A, Flash Encryption, Azure IoT Hub

Contents

List of Figures

List of Tables

Introduction

Motivation

ESP32 is a highly popular IoT platform and the internet is full of articles and tutorials describing what can be built with it. Weather stations, home security systems, remotely controlled door locks or light switches, robots, or flower watering systems. Anything is possible, it is usually easy to do and affordable. But the security of these solutions is often not given a second thought. While it may seem pointless for a hobby project to deal with security, it is not an exception that IoT devices lacking elementary security measures are released to the market and then exploited in the wild. This thesis describes threats to IoT devices in general, goes deep in ESP32 security features, and presents a proof-of-concept app for the collection of telemetry data.

Goals

This thesis aims to describe the current threats related to IoT devices, describe the security features of ESP32 and its security vulnerabilities, compare communication protocols used in IoT. Based on the analysis create a proof-of-concept app utilizing important security features such as Secure Boot, TLS, and OTA update, implement one of the device identification methods and describe problems encountered during implementation. Summarize and discuss the learnings from both the analysis and the implementation.

Thesis Organization

The content of this thesis is divided into four chapters and a summary. The first chapter describes IoT Security in general, discusses limits of TLS on IoT devices, and how it should be used securely, plus contains the comparison of HTTP, MQTT, and CoAP protocols in the context of IoT. The second chapter

describes the ESP32 Hardware, Software, and basic features of the platform. The third chapter is a deep dive into ESP32 security features and describes their limits. It also contains an analysis of known ESP32 vulnerabilities and what security changes are present in newer ESP devices. The fourth chapter, the practical part, explains the decisions made when creating the proof of concept app and describes what was needed to do to support the external HSM module on ESP32. The summary presents short takeaways from the thesis mentioning all the important points made in the text.

Chapter 1

IoT Devices & Security

This chapter describes IoT Security in general, discusses limits of TLS on IoT devices and how it should be used securely, It also contains a description of various ways IoT devices can be identified, and a comparison of HTTP, MQTT, and CoAP protocols in the context of IoT Devices.

1.1 IoT Security

To get a comprehensive overview of security risks for IoT devices, this section combines an OWASP IoT Top Ten list, a survey on IoT Security, and other papers, plus a book that argues that IoT security is not just about technical problems but also about the human-machine symbiosis.

1.1.1 OWASP IoT Top Ten 2018

In 2018 the OWASP Foundation, known for its Top Ten list of Web Application Security Risks, has published the IoT Top Ten. It describes the most common vulnerabilities of IoT solutions, that developers should know about and avoid. This section discusses each point from the list and presents potential countermeasures unless they are already obvious from the official description.

1.1.1.1 Weak, Guessable, or Hardcoded Passwords

Use of easily brute-forced, publicly available, or unchangeable credentials, including backdoors in firmware or client software that grants unauthorized access to deployed systems [1].

Passwords and other kinds of credentials should never be the same across devices. It ensures that compromise of one device does not automatically lead to compromise of others. If password is user-configurable, the device should check for weak passwords. There is also guidance from NIST recommending

that user-provided passwords should be checked against known data breaches [2].

This vulnerability had been used by the Mirai botnet in 2016, which contained over 600K mostly IoT devices whose credentials were brute-forced [3].

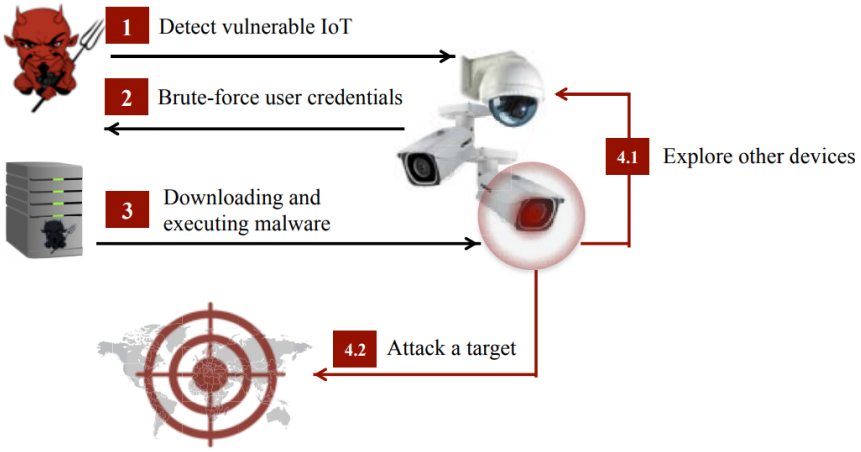

Figure 1.1: Mirai attack process [4]

1.1.1.2 Insecure Network Services

Unneeded or insecure network services running on the device itself, especially those exposed to the internet, that compromise the confidentiality, integrity/authenticity, or availability of information or allow unauthorized remote control [1].

TLS should be used for all communication over the internet. It is not sufficient to just support TLS, but also disable old and vulnerable versions of the protocols, weak ciphersuites, and ensure that certificates are properly validated.

We should also strive to make the attack surface of the device as small as possible. Instead of making the device a server and have clients connect to it, we should consider making the device a client that communicates with a backend which is then used by other clients. This scenario is easier to secure, as we have a single service that is fully managed by us, instead of servers running on individual IoT devices that we may only have limited access to.

1.1.1.3 Insecure Ecosystem Interfaces

Insecure web, backend API, cloud, or mobile interfaces in the ecosystem outside of the device that allows compromise of the device or its related components. Common issues include a lack of authentication/authorization, lacking

or weak encryption, and a lack of input and output filtering [1]. The attacker may also compromise the cloud service communicating with the device. All interactions with these services must be authenticated, audited, and use proper authorization.

Usually, a single centralized service manages devices that are used by multiple customers, so it has to make sure that individual customer can only control their own devices.

In 2017 an unsecured MongoDB in an ecosystem of a smart teddy bear called 'CloudBear' led to a disclosure of passwords and audio messages that were recorded by kids and parents using this toy [5].

1.1.1.4 Lack of Secure Update Mechanism

Lack of ability to securely update the device. This includes lack of firmware validation on device, lack of secure delivery (un-encrypted in transit), lack of anti-rollback mechanisms, and lack of notifications of security changes due to updates [1].

IoT devices should have *Renewable Security*[6], so not only they are secure when released, but we can maintain their security via remote updates as new vulnerabilities may be discovered. This process also has to ensure that non-genuine firmware can not be downloaded to the device and that an attacker cannot force a rollback to an older and vulnerable firmware.

1.1.1.5 Use of Insecure or Outdated Components

Use of deprecated or insecure software components/libraries that could allow the device to be compromised. This includes insecure customization of operating system platforms, and the use of third-party software or hardware components from a compromised supply chain [1].

Firmware developers should use technologies that are supported by their authors and receive regular security updates. It is also important to issue a timely patch of the firmware after a vulnerability in any of the components is patched.

1.1.1.6 Insufficient Privacy Protection

User's personal information stored on the device or in the ecosystem that is used insecurely, improperly, or without permission [1].

Devices that collect information that may identify individuals (PII) have to comply with regulations regarding data protection and ensure that such data do not get disclosed to others without the user's consent. They should also support wiping of such data from the device.

1.1.1.7 Insecure Data Transfer and Storage

Lack of encryption or access control of sensitive data anywhere within the ecosystem, including at rest, in transit, or during processing [1].

Apart from using TLS when data are transmitted, encryption of data stored by the device should also be used. This ensures that even if an attacker manages to dump the device memory, he will not able to extract the secrets and other data from the device. This could become a serious issue especially when the data collected are sensitive - e.g. location data collected by a smartwatch or videos stored inside a camera.

1.1.1.8 Lack of Device Management

Lack of security support on devices deployed in production, including asset management, update management, secure decommissioning, systems monitoring, and response capabilities [1]. As the devices are typically installed in remote unattended environments, we have to be able to remotely manage them over their entire lifetime. This is typically provided by platform as a service (PaaS) IoT solutions, e.g. Azure IoT Hub, AWS IoT Core, mDash IoT Cloud, and others. These services have solutions for device provisioning, remote control, and disabling stolen/lost/broken devices.

1.1.1.9 Insecure Default Settings

Devices or systems shipped with insecure default settings or lack the ability to make the system more secure by restricting operators from modifying configurations [1].

Secure by default or secure by design is a well-known principle of HW and SW design, that mandates the product is designed with security in mind from the beginning and does not require any additional user configuration or additional SW/HW to become secure. Today it has even become one of the rules advocated by the British government [7], and in the future, it may even become a regulatory requirement for IoT devices in the UK.

1.1.1.10 Lack of Physical Hardening

Lack of physical hardening measures, allowing potential attackers to gain sensitive information that can help in a future remote attack or take local control of the device [1].

As IoT devices often operate in unattended autonomous environments [4], we have to consider the possibility that an attacker might have physical access to them and attempts to replace their firmware, extract their secrets, or creates a clone of a device.

To resist this attack, IoT device has to ensure that it only runs genuine code and its memory should be encrypted with a key that is not exportable

from the device. Parts of the device that store cryptographic keys and perform operations with them, typically hardware secure modules (HSM) and cryptographical coprocessors (e.g. ARM TrustZone), should also be resistant to physical tampering and side-channel attacks [6].

1.1.2 Other Threats

Apart from the threats in OWASP IoT Top Ten, a survey on IoT Security done by Neshenko *et.al* [4] mentions two more: *insufficient auditing* and *energy harvesting*.

1.1.2.1 Insufficient Auditing

Many IoT devices lack thorough logging/auditing capabilities and even less often are the logs analyzed by a centralized system (e.g. SIEM). This is especially a problem for devices in the area of physical access control (cameras, card readers, home alarms, wireless locks, etc.). Auditing (and analyzing those logs for anomalies) can be also useful for detecting attempts at circumventing access control and other security measures.

1.1.2.2 Energy Harvesting

This is a denial of service attack for battery-powered IoT devices. These devices are optimized for a long battery life during regular use, but an attacker can choose an operation that requires a lot of power, flood the device with a large number of messages and drain the battery, which also called the *vampire attack* [8][9].

1.1.3 IoT Security vs Computer Security

In the book 'The Internet of Risky Things'[10], the author argues that there is a difference in securing IoT devices and other computers. He argues, that computer security is based on human-machine symbiosis because people care when their computers/phones/servers are broken or not updated. This symbiosis is not there for IoT devices, because there may be just too many of them for us care about, they are cheap, and people do not even know they have to case. Study [11] performed by Canonical has shown, that people expect that the security of IoT devices is a responsibility of the device manufacturer, not theirs.

This non-caring by users extends to other problems such as not using strong credentials for the devices, or not even changing the default ones. Which had a significant impact recently and enabled creation of the Mirai botnet.

He also points out, that IoT devices may easily outlive the company that made them, and then there would be no one to patch those devices in case a

vulnerability is discovered. Even if the company is still in business, it might not be incentivized to produce a security patch for the device, as it is a pure expense from their perspective. Also, many IoT devices are cheap, therefore companies only have small profit margins on them.

1.2 Device Identity

In a web app or mobile app, we typically need the identity of the end-user, that proves it to our service using something he knows (username+password), something he has (USB token or a device paired to their identity) or something he is (biometric credentials). Based on the provided credentials, the user is authenticated. In IoT, we typically want to identify the device communicating with our backend in order to be sure that it is a device manufactured/provisioned by us and not some untrustworthy or potentially cloned device. Two ways of device identification are discussed: PUF and PKI.

1.2.1 PUF Based Device Identity

PUF stands for physically unclonable function, in practice it is an electronic[1] circuit whose output depends on random physical and manufacturing variations, that even the manufacturer cannot control, therefore it is not possible to replicate them. The output of PUF has to be unique and repeatable for the same device, but different between any two devices.

The most commonly used variant of PUF in IoT is SRAM PUF, which uses random differences in SRAM threshold voltages, causing individual memory bits to either prefer 1 or 0 when the SRAM is powered on [12]. This pattern of nonrandom 1s and 0s gives a unique fingerprint of the particular SRAM chip. This fingerprint is then used to identify the device and to derive a secret cryptographic key. The main advantage is the secret key does not have to be stored anywhere in the memory, and if the device gets physically tampered with, its fingerprint changes, and the original key is lost.

For ARM, SoCs with SRAM PUF [13] made by Intristic ID is available and widely used[2]. ESP32 does not come with an integrated PUF, so the only alternative is to use a separate chip that performs cryptographic operations using a key derived from PUF e.g. ChipDNA ECDSA Authenticator made my Maxim Integrated [14].

1.2.2 PKI Based Device Identity

The most common way to identify devices is to use public key infrastructure (PKI). This method is de-facto a standard used by all IoT PaaS offerings

[1] Optical PUFs also exist, but they are not practical to use in the context of IoT devices.

[2] 180 million ARM devices with Intristic ID have been shipped based on a statistic at `https://www.intrinsic-id.com/`.

(Azure IoT Hub, AWS IoT Core, Google Cloud IoT, etc.). Each device has its own certificate that it uses for authentication when communicating with the cloud backend. This certificate can be either based on a key pair generated by backend and then copied (together with the certificate) to the device. Or the device itself can generate the key pair and send a certificate signing request (CSR) to the CA, so a certificate can be issued without the private key ever leaving the device. On ESP32 we can store this private key in device flash memory, ideally with memory encryption and secure boot features turned on. An alternative is using a hardware security module (HSM) to store and generate the key, e.g. Microchip ATECC608A. It has additional security features such as tamper resistance and a TLS library can be configured to use this chip for TLS handshakes, so no computations using the private key are performed on the main CPU. PUF can be also used in creation of the device certificate - we can use a signature, created by the aforementioned ChipDNA ECDSA Authenticator chip, in a CSR that is sent to the CA to issue a certificate whose private key can be only extracted from the PUF.

All the cloud vendors provide a UI to manage all issued device certificates, so if a device gets lost or stolen, it is possible to revoke its certificate and prevent it from communicating with our backend ever again. There also could be a situation when we have to revoke device certificate, but do not want to revoke its access, e.g. if the CA signing certificate gets revoked or it is discovered that there is some problem with our certificates. In order to do this, we have to have a system of remote over the air update in place, that can update certificates on devices. In case the private key is stored on the device, we can either renew a certificate - create a new CSR using the old key pair or re-key it - generate a new keypair and use it to request a new certificate from the CA.

1.2.3 MAC Address

`esp_efuse_mac_get_default` function from ESP-IDF can be used to retrieve the original MAC address of a ESP32 device, that is burned into `BLK0` eFuse during manufacturing. This MAC is unique and works as a unique serial number, but it is not secret and other devices can easily spoof it.

1.3 Communication Protocols for IoT

Unlike the Web, where a single messaging protocol (HTTP) is the standard, there are many protocols currently used in IoT based on various requirements of IoT systems, the most commonly used are HTTP, AMQP, MQTT, CoAP and XMPP [15]. Keeping in mind the subtask of this thesis - implementing a demo app for collecting telemetry data, we can leave out AMQP and XMPP from this comparison. AMQP was designed to ensure reliable and high-performance business transaction and it is not very well suited for

IoT devices due to its higher power/processing/bandwidth requirements [16]. XMPP is originally an instant messaging protocol that uses XML to represent messages, does not have guaranteed delivery (QoS), and runs over TCP, all of which makes it less suitable for IoT, especially when communicating over low power networks [16].

Three protocols: HTTP, MQTT, and CoAP are described and compared in this chapter, focusing on their security features, reliability, and overhead.

1.3.1 HTTP

HTTP and RESTful APIs are the standard for many use cases today - e.g. server to server, mobile app to server and SPA web app to server communication, this often makes HTTP the first protocol people use for communication with IoT devices. Its main advantages are universality (same API/service can be used by multiple different clients), a wide range of available developer resources (tools, frameworks, tutorials, etc.) and widespread availability (e.g. all PaaS IoT solutions from public cloud vendors offer HTTP based API). The main downside is a large overhead due to protocol text-based nature which makes it a less ideal choice for typical IoT apps that periodically transmit a small amount of data [16].

HTTP is typically used over TLS which ensures secure communication. A client can be authenticated by: a) presenting its certificate during TLS handshake, b) using Basic/Digest Authentication in HTTP, c) sending a valid access token in an HTTP header or a URL parameter.

1.3.2 MQTT

MQTT is a Client-Server publish/subscribe messaging protocol that has been invented in 1999 by IBM for communication with sensors on oil pipelines using a satellite connection [17]. Based on the constraints of the 1990's MCUs and slow satellite connection, MQTT had been designed to be simple to implement on the clients and have minimal communication overhead, both of which are also reasons why MQTT is one of the most popular protocols in IoT. It an application layer protocol that is used over TCP or WebSocket.

In MQTT clients connect to a server (called broker) and subscribe to selected topics. When a new message is published in a topic, it is immediately sent by the broker to all subscribed clients as shown in figure 1.2. This enables devices to communicate together in a decoupled way without delay [18]. E.g. a CO_2 sensor can periodically publish its reading into the `office/co2` topic and an actuator attached to a window subscribed to this topic can automatically open the window if it receives a message on `office/co2` topic informing it that CO_2 level is above 500ppm.

Messages (represented by the publish packet shown in figure 1.3) consist of a small 2B long binary header, name of a topic, and a payload. Format of the

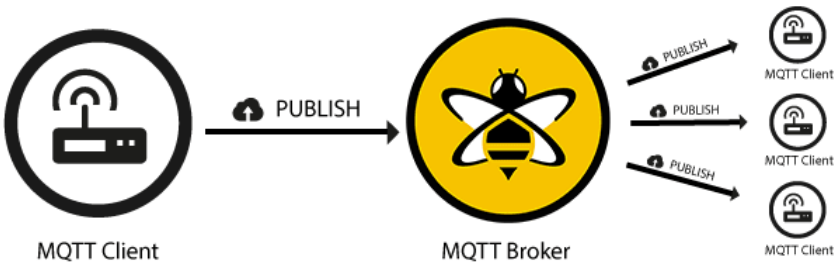

Figure 1.2: MQTT message flow [19].

payload is not defined in the protocol itself, therefore any format suitable for the particular application can be used (e.g. JSON, MessagePack[3], Protocol Buffers[4], etc.). There are three different *Quality of Service* levels that can be set for a message: 1) `QoS-0` - *at most once* - the message is acknowledged to the publisher when the broker receives it 2) `QoS-1` - *at least once* - the message is acknowledged to the publisher if it gets delivered to any subscriber 3) `QoS-2` - *exactly once* - using two-phase commit just once delivery of the message is guaranteed. Messages also can have a *retained flag* set - then the broker stores the latest message in a topic and sends it to a client immediately after it subscribes to the topic. If multiple messages are sent by one publisher, they are guaranteed to arrive in the same order to all subscribers [18]. If clients want to receive messages (with QoS 1 and 2) in subscribed topics even when they are temporarily offline, they can use *persistent sessions* by setting the `cleanFlag` to false when connecting to the broker.

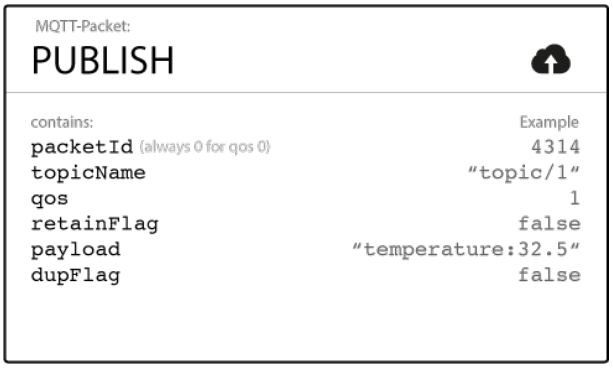

Figure 1.3: MQTT publish packet [19].

[3] `https://msgpack.org/`
[4] urlhttps://developers.google.com/protocol-buffers

1.3.2.1 Security

MQTT supports authentication using username and password fields in a connect packet shown in figure 1.4. Unless MQTT is run on top of TLS, the credentials are transported in plain text and therefore can be intercepted and stolen. Many MQTT brokers (e.g. HiveMQ [20], Aws IoT Core [21], Azure IoT Hub [22]) support authentication using a certificate presented by the client during TLS handshake. A general recommendation is to only use MQTT over TLS [20].

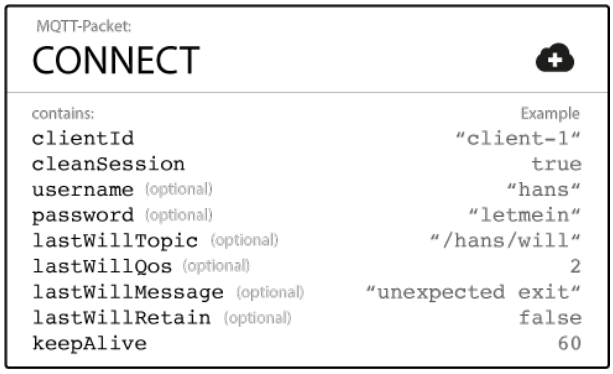

Figure 1.4: MQTT connect packet [19].

1.3.2.2 MQTT 5

All previously described features were for MQTT 3.1.1[5]. MQTT 5 brings support for HTTP/2 as transport layer, timeouts for both individual messages and persistent sessions, extended error codes and error messages, coupling request and response messages together directly on the protocol level, support for challenge-response authentication flow and more as described in [23].

1.3.2.3 MQTT-SN

MQTT-SN is a variant of MQTT for devices that cannot maintain a persistent connection over TCP and communicate over UDP on low power networks such as LoRaWan, BLE, Zigbee, or NB-IoT. Devices using MQTT-SN communicate with an MQTT-SN gateway that implements a regular MQTT client communicating with an MQTT broker. To save bandwidth, topics are represented by numbers instead of strings, subscriptions are permanent by default, DTLS is used instead of TLS, and persistent sessions are used to store messages on the server for clients that are not permanently connected. These

[5]In the protocol itself MQTT 3.1.1 is represented as version 4, that is why the next version is 5.

devices sleep most of the time and only communicate with the broker when they want to publish a new message [24].

1.3.2.4 Support in Public Clouds

As of writing this thesis, only MQTT version 3.1.1 is supported by the widely used public clouds (Azure, AWS, Google) and only a subset of its features is available. None of the mentioned providers supports QoS 2 or retained messages and only AWS and Azure have support for persistent sessions [25][26][27]. None of the mentioned providers support MQTT-SN.

1.3.3 CoAP

Constrained Application Protocol (CoAP) is a protocol developed especially for IoT devices communicating over low power (constrained) networks. It is similar to HTTP and can be used to build RESTful APIs [28]. The main differences are binary headers instead of text headers, UDP instead of TCP and, usage of binary serialization format CBOR[6] instead of JSON. A CoAP to HTTP proxy can be also used for communication between devices using CoAP and a server using HTTP as shown in figure 1.5 [29].

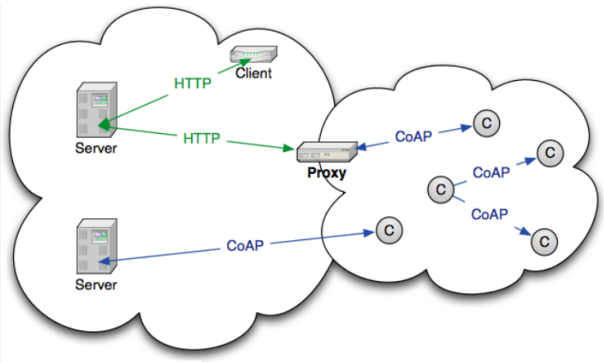

Figure 1.5: CoAP to HTTP proxy. IoT (constrained) devices are marked with "C". [30]

As described in RFC7552[29] apart from HTTP-like features CoAP also supports multicasting (based on UDP multicast), subscribing to resource changes (simplified publish/subscribe), asynchronous messages (when a request is sent to a URI on a sleeping device, it is cached by the proxy and client responds to it when it wakes up), and automatic message retransmission as it has to communicate over UDP instead of TCP. To ensure guaranteed delivery over UDP, CoAP implements *confirmable messages*.

[6] `https://cbor.io/` is similar to *MessagePack*, but easier to implement on power-constrained devices.

	HTTP	**MQTT**	**CoAP**
Architecture	Client/Server	Client/Broker	Client/Server
Semantics	Request/Response	Publish/Subscribe	Request/Response Publish/Subscribe
Transport	TCP	TCP/WebSocket (UDP for MQTT-SN)	UDP
Security	TLS	TLS (DTLS for MQTT-SN)	DTLS
Representation	Text	Binary	Binary
Header Size	Undefined, 10s-100s of bytes	2 Bytes	4 Bytes
Guaranteed Delivery	Yes (TCP)	Yes (with Qos 1 and 2)	Yes (confirmable msg.)
Support in Public Cloud	Yes	Yes	No

Table 1.1: Comparison of HTTP, MQTT and CoAP.

1.3.3.1 Security

CoAP is intended to be used over DTLS to ensure secure communication as no encryption or authentication method is implemented in the protocol itself as described in [29].

1.3.3.2 Support in Public Clouds

As of writing this theses no widely used public cloud offers CoAP as part of their PaaS offerings for IoT. The only possibility is to run a CoAP to HTTP proxy in a VM/Container that communicates with the REST API of the selected platform (Azure IoT Hub, Aws IoT Core, etc.).

1.3.4 Comparison of Communication Protocols

Characteristics of previously described protocols are summarized in table 1.1.

1.3.4.1 Reliability and Guaranteed Delivery

One characteristic that can be slightly ambiguous is *guaranteed delivery* which compares protocol reliability. For TCP based protocols, lost packets get automatically retransmitted, in UDP the protocol itself has to detect and retransmit packets to guarantee delivery. Also for Publish/Subscribe semantics, the sender communicates with a broker and there is no guarantee that there is another client listening for messages or if they are getting discarded. To ensure delivery not just to a broker, but to another client, QoS in MQTT or Confirmable Messages in CoAP can be used.

In general TCP-based protocols are intended to be used on broadband networks (DSL, cable, WiFi, LTE, etc.) and UDP-based which are used in IoT such as MQTT-SN or CoAP are designed specifically for low-power networks such as NB-IOT where TCP is unreliable [31].

1.3.4.2 Communication Efficiency

A communication protocol is more efficient if it has lower overhead – less additional data get transmitted along with the message payload. Intuitively one can expect that HTTP is the least efficient, and MQTT and CoAP are more efficient of the discussed protocols. This has also been shown by surveys [15] and [16] from which the following studies were selected:

When MQTT is compared with CoAP, it has been experimentally shown that CoAP has lower overhead if we ignore lost messages (MQTT QoS 0 and non confirmable CoAP messages) and even if we need guaranteed delivery (MQTT QoS 2 and confirmable CoAP messages)[32].

Another study [33] calculated protocol efficiency (as a ratio of payload bytes exchanged and the number of bytes transmitted) for MQTT and CoAP. It has shown that CoAP is more efficient, but MQTT has lower communication latency if there is packet loss.

A study comparing MQTT, HTTP REST and CoAP [34] has shown that CoAP uses the least amount of bandwidth when transmitting small payloads (10 B to 100 KB), but HTTP is the most efficient for large payloads (1 MB+).

1.3.4.3 Security

All of the protocols support (D)TLS and clients can be authenticated using certificates which should be the preferred choice for IoT devices. If the device has enough processing power (such as ESP32), we can use device certificates for TLS Mutal Authentication to prove the device identity to the server and the backend app running on it. When using plain text authentication (e.g. Basic Authentication in HTTP or username/password in MQTT) we have to keep in mind that an adversary performing an attack can intercept the credentials and then use them to impersonate the compromised device. Salted Challenge-Response Authentication [35] can be used to prove device identity to the server without transmitting the password itself.

1.4 Transport Layer Security (TLS)

TLS is a network protocol used to secure communication over TCP[7]. In IoT, TLS is typically used with HTTP and MQTT protocols.

[7]The only other reliable transport protocol that is widely used with TLS is QUIC [36]. Although it is not an accepted standard yet. `https://quicwg.org/`

TLS provides authentication of both server and client[8] using public-key cryptography, encryption of the data in transit using a shared symmetric key and integrity validation of data using message authentication code (MAC).

1.4.1 Communication Overhead

TLS brings communication overhead, which may be significant for scenarios where IoT devices communicate often, but transmit only a small amount of data, a typical scenario is collecting live data from a sensor (remote telemetry) over HTTP. TLS adds additional two roundtrips as shown in figure 1.6 and approximately 6.5 KB of data transfer [37] for every new connection.

This may not be an issue for a device connected to a broadband network, but low-power networks such as LTE-M and NB-IoT are becoming commonplace and enable new scenarios for IoT devices. These low-power networks come with high latency (10s+ for NB-IoT), low transfer speed (from 10s of kbit/s to 1 MBit/s), and often charge per transferred data (aprox. 10 EUR per 500 MB [38]). In these use cases using TLS 1.3 with 0-RTT Session Resumption, which is described in the following paragraph, will be beneficial. Another possibility is to use UDP based TLS called DTLS (Datagram TLS) [39].

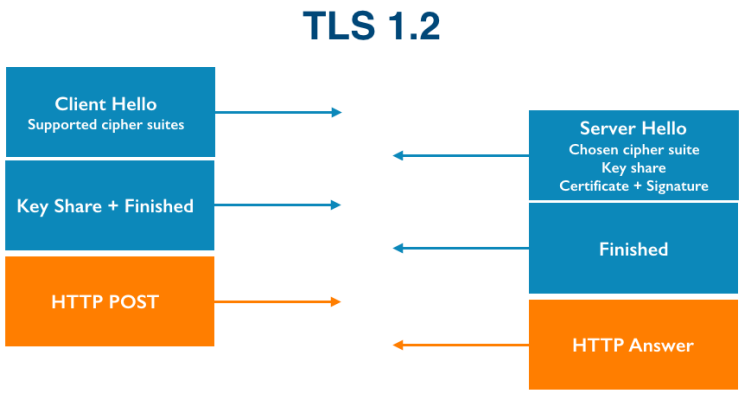

Figure 1.6: TLS 1.2 Handshake [40]

1.4.2 TLS 1.3

TLS 1.3 solves some of the issues regarding communication overhead as its handshake adds only one roundtrip (shown in figure 1.7) instead of two. It also requires *perfect forward secrecy*, which ensures that captured TLS sessions cannot be decrypted even if the server private key is later compromised.

[8] Typically only the server is authenticated, but additional authentication of a client also can be done (called *mutal TLS*).

Cipher suites in TLS 1.3 were reduced when compared to TLS 1.2. RSA is no longer supported, neither is AES in CBC mode, and all other vulnerable algorithms and ciphers were removed (RC4, MD5, SHA-1, 3DES, etc.). The key exchange algorithm is always (EC)DHE and all cipher suites must support AEAD (authenticated encryption). All these changes also mean that TLS 1.3 is incompatible with TLS 1.2 and older.

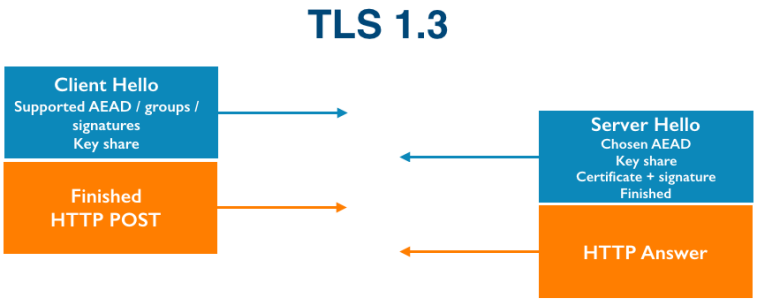

Figure 1.7: TLS 1.3 Handshake [40]

1.4.2.1 0-RTT Session Resumption

TLS 1.3 *0-RTT* Session Resumption (figure 1.8) eliminates the handshake if a client connects again to the same server. This is done using a Client Hello packet that contains Session Ticket (to restore the session) together with data intended to be transferred e.g. an HTTP request.

There is a certain drawback, using 0-RTT is vulnerable against *replay attack* and breaks forward secrecy. The application has to protect against the replay attack, typically by not using session resumptions for HTTP requests that are not idempotent[9], e.g. CloudFlare supports 0-RTT only for GET requests without query parameters [41].

1.4.3 Difficulties with TLS on Constrained IoT Devices

IoT device may not have enough RAM and CPU power to support TlS (aprox. 100+MHz CPU and tens of KB of RAM are required [42]) and if they do then the device might not have enough entropy to generate unpredictable random numbers [43]. Another issue could be battery draining by repeatedly performing expensive cryptographic operations [9]. IoT devices also have constrained persistent memory, so it might not be possible to store multiple CAs certificates, or many TLS cipher suites for future compatibility [42].

[9] an operation is idempotent if the state of the system is the same even if the operation is repeated

Figure 1.8: TLS 1.3 Session Resume [40]

Pre-Shared Key Ciphersuites for Transport Layer Security (TLS-PSK)[18] can be used on such constrained devices in order to save power or even to fit TLS into the computing envelope of such devices. In order to use a pre-shared key, it first has to be securely provided to both ends of the TLS connection (in out case IoT device and a server). The static pre-shared key can be either used for both authentication and data encryption (e.g. in `TLS-PSK-AES128-CBC-SHA256` cipher suite) or only for authentication. In the latter case ephemeral keys established using DHE or ECDHE will be used for encryption (e.g. in `TLS-DHE-PSK-AES128-SHA256` cipher suite) and guaranteeing forward secrecy [44].

1.4.4 The Most Dangerous Code in the World

If servers certificates are not properly validated, it makes the communication vulnerable to man-in-the-middle attacks. This was found to be a common problem in various client libraries in 2012 by Georgiev et al. in their paper *The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software*. They state: "This is exactly the attack that SSL is intended to protect against. It does not involve compromised or malicious certificate authorities, nor forged certificated, nor compromised private keys of legitimate servers. The only class we exploit are logic errors in client-side SSL certificate validation"[45].

The most common problems were: SSL libraries not being *secure by default*, developers not handling correctly/ignoring errors returned by the SSL libraries or application completely disabling certificate validation.

1.4.5 SSL/TLS Certificate Validation

The attacker can poison the DNS records and make the client connect to their server with its own certificate. However, if the client correctly validates it, it will detect that this certificate is invalid for the domain and will not establish a secure connection to the attacker's server.

Two basic steps in certificate validation are *chain-of-trust verification* and *hostname verification*. The certificate also might have been *revoked* (e.g. if its private key has been leaked) or could be used in a disallowed way.

1.4.5.1 Chain-of-trust Verification

As described in [45] during SSL/TLS handshake the server sends its certificate to the client, including all CA certificates used in the chain of signatures from servers certificate to the root CA certificate. The client then starts to validate this *chain-of-trust* by starting with the server certificate and validating that it has been signed by the CA immediately above it. Then continues with the intermediate CA certificates until it reaches the root CA certificate. The root CA certificate is validated against a list of trusted root certificates stored in the device. Each certificate is checked for expiration and whether it is allowed to sing the preceding certificate.

1.4.5.2 Hostname Verification

In order to verify the hostname, the client has to compare the *fully qualified domain name* (FQDM)[10] to the server certificates *SubjectAltName* (SAN) or *CommonName* (CN). SAN should be primarily used according to RFC 2818 [46] and CN is only supported for backward compatibility. Domain name and SAN/CN either has to be matched exactly or SAN/CN can contain a *wildcard* pattern to enable usage of a single certificate for multiple domains (e.g. certificate with CN:*.cvut.cz can be used for cvut.cz, fit.cvut.cz, fa.cvut.cz, etc.).

1.4.5.3 Certificate Revocation

Certificate revocation status can be checked either by ensuring it is not present on CAs *certificate revocation list* or by using a *Online Certificate Status Protocol* (OCSP). The latter is preferred in IoT devices, as revocation lists can become too large for IoT devices to process and can effectively DoS them.

1.4.5.4 Additional Checks

X.509 certificate extensions can be used to constrain how the certificate private key can be used. E.g. use of this key to sign other certificates, limit which CAs/SANs can be signed with the key and other *certificate policies*. These extensions are described in RFC 2527 [47].

[10] Example: `fit.cvut.cz`

Chapter 2

ESP32 Overview

This chapter presents an short overview of the ESP32 platform that is relevant to understand the security features described later. It is based on the official documentation [48] and ESP32 Technical Reference, where more information can be found. [49].

2.1 Hardware

ESP32 uses a System on Chip architecture, integrating CPU(s), RAM, Wi-Fi, Bluetooth, specialized co-processors, and controllers on a single chip.

2.1.1 SoCs, Modules and Devboards

ESP32 is typically used in a form of module e.g. `ESP32-WROOM-32`. It contains the `ESP32-DOWD`[11] SoC, 4MB of flash memory and a PCB antenna. Another often used module is `ESP32-WROVER-B` which contains an additional 8MB of external RAM memory. The module can be connected to other components using its 38 pins. A development board, similar to the one shown in figure 2.1, is typically used for development, prototypes, and hobby projects.

[11] D - Dual-Core, 0 - 0MB of embedded flash, WD - Wi-Fi+BT/BLE

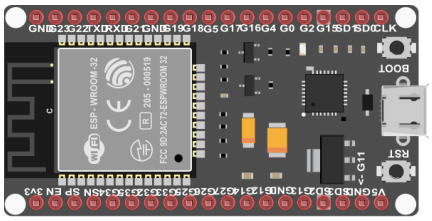

Figure 2.1: ESP32 development board with a `ESP32-WROOM-32` module [50]

2.1.2 SoC

ESP32 SoC shown in figure 2.2 contains:

- Two[12] Xtensa 32-bit Cores (called PRO and APP CPU) @ 240Mhz
- RTC[13] subsystem with Ultra Low Power (ULP) co-processor
- 448 KiB ROM and 530 KiB SRAM
- 8 KiB of FAST RTC SRAM and 8 KiB of SLOW RTC SRAM
- 1 Kb eFuse memory
- *Integrated Radios:* Wi-Fi 802.11/b/g/e/i, Bluetooth 4.2 and Bluetooth Low Energy radios
- *Peripheral Input/Output:* SPI, UART, I2C, Ethernet, ADCs (analog to digital converters), DACs (digital to analog converters), Capacitive Touch Sensors, PWM (pulse with modulation), etc.
- Optional *Embedded Flash* Two chip variants exists with an embedded flash - ESP32-D2WD with 2MiB and ESP32-PICO-D4 with 4 MiB.
- *Cryptographic HW Accelerator* with SHA-256, AES, RSA and RNG

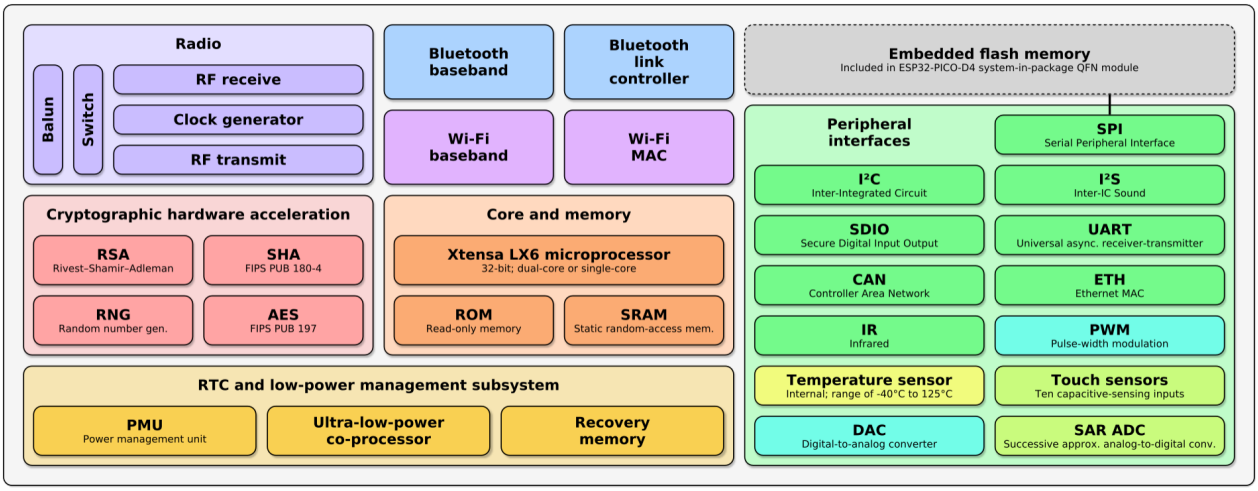

Figure 2.2: ESP32 Block Diagram [51]

[12]There is also a single-core chip: `ESP32-S0WD`, but it is not commonly used.

[13]RTC is used by Espressif as an encompassing name for the entire low power subsystem, it does not refer only to the Real-Time Clock.

2.1.3 Flash Memory

Flash memory can be embedded inside the SoC or connected externally via SPI. Typically an ESP32 module has 2 or 4 MB of external flash memory. More details are explained later in section 2.3.1 about ESP32 memory layout.

2.1.4 Low-power Subsystem

ESP32 can enter a deep sleep mode during which the main CPU and radios are turned off, but ULP co-processor and RTC memory remain powered. They can use timers, internal sensors, or external devices and wake up the main CPU if needed.

2.2 Software

2.2.1 ESP-IDF

Espressif IoT Development Framework (ESP-IDF) is the official development platform for ESP32 and ESP32-S2. It is comprised of an API for ESP32, individual components (e.g. esp-tls, esp-mqtt, etc.) for the platform, and an SDK (build and flash tools, language server, IDE plugins, etc.) used by the developers.

2.2.2 FreeRTOS

ESP-IDF FreeRTOS is a fork of FreeRTOS 8.2. with added support for symmetrical multiprocessing (SMP) on ESP32. There are also changes in Task Scheduling to accommodate for the few edge cases where PRO CPU and APP CPU cores are not interchangeable (e.g. access to RTC Fast memory is only possible for PRO CPU). ESP-IDF FreeRTOS also adds support for per-core idle and tick hooks, ring buffers and contains two features back-ported from newer versions of FreeRTOS: task deletion and static allocations.

ESP-IDF FreeRTOS is a component of ESP-IDF and by default, all apps developed using the official SDK use it.

2.3 Platform Description

This section delves into selected platform features that are relevant from the security perspective or referred to in the rest of the thesis.

2.3.1 Memory Layout

ESP32 is based on the Harvard architecture and uses separate memory for data and instructions. ESP32 SoC contains an internal System ROM, IRAM (Instructions RAM), DRAM (Data RAM), Fast RTC instructions memory,

and slow RTC data memory. Then there are regions of flash memory used to store data (DROM) and instructions (IROM). DRAM can be extended via external RAM.

2.3.1.1 System ROM

ROM Integrated in the SoC is used for the first-stage bootloader and other system components. It is not accessible for applications running on ESP.

2.3.1.2 IRAM (SRAM)

Used for PRO CPU and APP CPU caches, parts of the Wi-Fi component, interrupt handlers, and code that has been explicitly placed there (using `IRAM_ATTR` macro).

2.3.1.3 IROM (Flash)

The code that is not explicitly placed in IRAM or RTC memory is kept in the flash memory. Flash MMU handles the mapping of addresses in an application image to the IROM range. This memory is read-only.

2.3.1.4 DRAM (SRAM)

Nonconstant static data and zero-initialized data are placed here by the linker. Any space that is left is used as heap memory. If Bluetooth or Memory Tracing is used, parts of DRAM become reserved and are not accessible from app code.

2.3.1.5 DROM (Flash)

Constant data except literals folded by the compiler into application code are placed in DROM.

2.3.1.6 External SPI RAM (SRAM)

ESP32 supports up to 8MB (at most 4MB mapped at a time) of external SRAM connected using SPI. SPI RAM extends DRAM and is used for dynamically allocated heap memory.

External RAM uses the same cache regions as flash memory.

2.3.1.7 Fast Instructions RTC Memory (SRAM)

Used for code intended to be run on PRO CPU after waking up from deep sleep mode.

2.3.1.8 Slow Data RTC Memory (SRAM)

Global and static variables used by the ULV co-processor in deep sleep are be placed here.

2.3.1.9 Memory Allocation

When a standard *libc* function to allocate memory (e.g. `malloc`) is invoked, it internally works with the *capability-based heap allocator* that is used by ESP32. It selects the appropriate memory type where the memory block should be allocated. By default calling `malloc` uses `MALLOC_CAP_8BIT` capability, thus it allocates memory in DRAM or external SPI RAM if available.

When an RTOS Task is started, its stack is allocated on the heap placed in the internal DRAM.

2.3.2 Watchdog Timers

There are two watchdog timers (Main and RTC) to monitor the system for faults and glitches. Both of these timers are started when the device boots and during normal operation of the device, they are regularly fed (dismissed). If this does not happen for any reason (app fault, CPU failure, deadlock, etc.) watchdog timer enters the first (of possible four) stages and performs an action associated with it (interrupt, CPU reset, core reset or system reset). If the timer is still not fed within the set time period, it enters the next stage and performs the next associated action. The last configured stage of the RTC timer must perform system reset.

2.3.3 Device Startup

2.3.3.1 Boot

When the device starts only the PRO CPU is active and it first checks whether it was started by wake up from deep sleep (`RTC_CNTL_STORE6_REG` is non-zero) or by powered-on/software reset/watchdog reset. In the first case it validates CRC of RTC memory (stored in `RTC_CNTL_STORE7_REG`) and if it is valid, continues from an address stored in `RTC_CNTL_STORE6_REG`.

In the latter cases or if RTC CRC is invalid, PRO CPU core executes the first-stage bootloader located in ROM memory embedded inside the ESP32 SoC. Bootloader first checks if the device memory should be flashed and switches to the download mode if needed.

If the download mode is not requested, the first-stage bootloader initializes access over SPI to the external flash, based on values stored in eFuse memory. Then it loads the second-stage bootloader into RAM from the address `0x1000` of the external flash memory.

This bootloader reads the partition table located at `0x8000` and tries to find a valid app partition based on the data stored in `ota_data` partition. If

this process fails, it selects the factory partition. For the selected partition, data and code are mapped into RAM (IRAM, DRAM, RTC mem, and IROM). Flash MMU is configured to provide correct mappings from load addresses (stored in code) to addresses in the particular memory type the code is mapped to (IRAM or IROM). The application stored in the selected partition is then started.

2.3.3.2 Application Startup

In the entry point of an ESP-IDF application, the `call_start_cpu0` function, the heap is initialized and APP CPU is started (up until now only PRO CPU had been running). Both cores then execute `start_cpu0` (PRO CPU) and `start_cpu1` (APP CPU) functions. ESP-IDF components are then initialized and FreeRTOS Scheduler is started on both cores. The main task (which runs the `app_main` function that functions as an entry point to the user's code) is created and executed on the APP CPU core.

2.3.3.3 No Bootloader

The second-stage bootloader is a regular app and if it is not needed, the memory at `0x1000` can directly contain an app to be executed after the first-stage bootloader finishes. Such app would have to handle all functionality of the second-stage bootloader (OTA partitions, secure boot, flash encryption, etc.) by itself.

2.3.4 Flash Storage Partitioning

Multiple apps and multiple kinds of data can be stored in ESP32 flash, for this reason, it supports partitioning. A partition has a name, size, memory offset, and one of the following types:
Type: *App*

- `factory` - Used to store an initial firmware flashed to a device during manufacturing.
- `ota_0` ... `ota_15` - Used to store over-the-air installed firmware.
- `test` - Reserved partition for factory test firmware, used if no other valid partition is found. Can be booted by the bootloader only if explicitly enabled.

Type: *Data*

- `ota` - Also referred to as `otadata` - a partition used for firmware selection during boot.

- `phy` - Used to store initialization data of the device physical layer (PHY). By default, this information is stored in firmware, but this partition can be used to configure PHY on per device basis[14] without having to maintain separate firmware versions.
- `nvs` - NVS partition, simple non volatile key-value store as described later in section 2.3.6.
- `nvs_keys` - Used to store AES encryption keys for an encrypted NVS partition.

2.3.5 Factory Partition and Factory Reset

Factory partition is the initial app partition the device boots into. After startup, it can be used to download the latest firmware into an OTA partition and then boot from it.

Factory reset (performed by holding a selected GPIO pin low for 5s) can be used to boot into this partition apart from the first startup. Factory reset can also clear selected data partitions (e.g. NVM or `ota_data`). Note that if a secure version is set in eFuse, performing a factory reset does not alter it.

A typical use case would be to: 1) delete device internal state, 2) delete stored config (e.g. wifi credentials), 3) reflash the possibly corrupted or outdated FW in OTA partition with the latest one.

2.3.6 Non Volatile Storage (NVS)

NVS (Non-volatile storage) is a library in ESP-IDF for storing key-value pairs in a partition of type `data_nvs` of the flash memory. Keys are ASCII strings of a maximum 15 characters and values can be integer types, strings (4000 chars max.), or binary data (blobs). The key-value pairs are stored inside namespaces to prevent key collisions between different components of the system. If there are more than one NVS partitions, they are independent and their namespaces are separate. Storing a key in one NVS partition cannot affect the same key in another partition.

NVS partition can be encrypted as desctibed later in section 3.4.

2.4 Other ESP devices

2.4.1 ESP8266

The predecessor of ESP32 - ESP8266 is a microcontroller with Wi-Fi support. Its main usage is to connect another MCU to a WiFi. It can be used with

[14] One use case is if the end product is used in two markets with different RF regulations (e.g. Japan and the EU). Different configurations for the WiFi module can be then stored in the PHY partition.

custom firmware (and is compatible with Arduino), but as it has only 80Mhz processor and 90KB of RAM it is not suited for advanced IoT applications. Due to its constraints, it does not properly support SSL/TLS and it also does not have any of the security features of ESP32 such as Secure Boot or Flash Encryption.

As a result of these limitations, it cannot be recommended for IoT devices made with security in mind.

2.4.2 ESP32-S2

Espressif has announced ESP32-S2, a successor to the ESP32 with lower power consumption and improved security [52]. In order to reduce power consumption, the CPU is now a single-core RISC-V chip instead of a dual-core CPU based on custom architecture in ESP32, there is less RAM and ROM memory inside the SoC, and Bluetooth support has been dropped. Wi-Fi has a new power-saving mode that automatically turns off the RF transceiver when not needed. ESP32-S2 security features are described later in section 3.12.

Chapter 3

ESP32 Security Features

The largest part of this chapter analyzes what is possible with the ESP32 security features, what are the platform vulnerabilies, and how newer revisions of ESP32 differ from the current version.

3.1 eFUSEs

The ESP32 has a specialized write-once memory called eFuse. This memory is composed of individual eFuse bits, that once set to 1 cannot be reverted back to 0.

ESP32 has in total four eFuse blocks of 256bits each, divided into eight 32 bit registers. The first block `EFUSE_BLK0` is used entirely for system purposes and is not user-programmable. Device MAC address, chip version, CPU frequency, and various flags (e.g. for system encryption, JTAG debugging, console output, etc.) are stored there. Blocks `EFUSE_BLK1` and `EFUSE_BLK2` are used for Flash Encryption and Secure Boot keys. Last block `EFUSE_BLK3` can be used for custom MAC address, or for any other purpose by the application. ESP-IDF also uses a few bits of the last block. The full description of eFuse blocks can be found in section 20 of ESP32 Technical Specification [49].

Following security fields should be set if ESP32 based device is shipped to the end customer to ensure safe operation:

- `efuse_wr_disable` - disables write operations to itself and all other fields in the eFuse memory
- `efuse_rd_disable` - disables read of a subset eFuse fields from SW, HW read is unaffected. E.g. reading of Secure Boot and Flash Encryption key can be disabled using this field.
- `console_debug_disable` - disables BASIC interpreter stored in ROM
- `JTAG_disable` - disables use of JTAG debugging

If the following features are not used by the user app, they can also be disabled by setting eFuse bits:

- `DISABLE_SDIO_HOST` - disables SD card host
- `DISABLE_BT` - disables Bluetooth
- `DISABLE_APP_CPU` - disables APP CPU core (useful when ESP32 is used as a WiFi module only, and another MCU communicates with it using AT commands).

3.2 Secure Boot

Secure Boot ensures that only code signed by the private key of the device manufacturer can run on the device. It can be used without flash encryption if protection from physical access to memory is not required. Note that if an attacker has physical access to an unencrypted flash, he can swap an app image after its signature has been verified and therefore bypass the Secure Boot.

During build app images and partition table data are signed using the Secure Boot private key and the public key is embedded into the second stage bootloader image. These steps can be performed independently - signing can be performed remotely when app is built in a CI/CD pipeline, so the private keys are kept outside of the CI environment.

Secure boot uses two keys:

- *Secure Bootloader Key* - AES 256 key used to calculate the digest of the second-stage bootloader.
- *Secure Boot Signing Key* - ESDSA key pair used for image signing. Its public key is embedded in the second-stage bootloader.

Secure Boot can work in two modes: *One-Time Flash* and *Reflashable*. In One-Time Flash the secure bootloader key is generated by the device on the first boot. In Reflashable mode, a SHA256 hash of the Secure Boot Signing Private Key is used as the Secure Bootloader Key. Because the signing key is available during the build process, a new bootloader image, and the secure boot digest for it, can be generated and flashed to the device.

3.2.1 Secure Boot Process

On the first boot with Secure Boot activated (all app and partition table binaries are signed and the public key is embedded in the second-stage bootloader image), the HW generates an AES 256 key (unless there is a key already present in `BLOCK2` eFuse) and 128 B Initial Vector (IV). The key is then used as device Secure Bootloader Key and stored in read/write protected

`BLOCK2` eFuse. A digest is then calculated from the key, the initial vector, and the bootloader image. The resulting digest is: `(IV | Sha512(AES256(IV | bootloaderImg | 0xFF pading))[0..64])`[15] and it is flashed to `0x0` address of the flash memory. After this process, the secure boot is permanently enabled and `ABS_DONE_0` eFuse is burned. The first stage bootloader will then only load the second stage bootloader if its digest matches with the digest stored at `0x0`.

3.2.2 App Signature Verification

The mechanism used by Secure Boot to validate an app image signature can be used without the bootloader verification. This can be used to protect the device against installing an unsigned firmware but does not forbid bootloader reflashing. If enabled, an app image is verified during OTA (to prevent spoofing of OTA updates) and during boot. Keep in mind, that the signature is verified using a public key embedded inside the bootloader that can be reflashed, so if an attacker has physical access to the device, he will be able to install any firmware he wants after the bootloader is reflashed.

3.2.3 Secure Boot as a Root of Trust

The Secure Boot serves as a Root of Trust for the device, as it creates a verification chain of the code running on the device [53]. First, the first-stage bootloader, stored in the ROM that is implicitly trusted, is loaded. Then it verifies the second-stage bootloader using a stored digest and loads it into the RAM. The second-stage bootloader verifies the partitioning table and selected boot partition using ECDSA and starts the app. Secure Boot ensures integrity and authenticity of the code running on the device, therefore we can trust for example the telemetry data sent by the device to the server.

3.3 Flash Encryption

The main memory of ESP32 can be encrypted using a built-in Flash Encryption. By default, only the bootloader, partition table, and all app partitions are encrypted. Other partitions are only encrypted if they have the encrypt flag set in the partition table. Each 32B block of flash memory is split into two 16B AES blocks that are encrypted using an AES key XORed with the block offset. The same key is used for both 16B blocks of a 32B memory block.

It is possible to either ”burn” the flash encryption key into devices eFuse before it is booted for the first time, or let the device generate the key. If a custom key is used, it is important not to share this key across devices as compromising one device would break the security of others. After the key

[15] `|` marks concatenation and `[0..64]` is subsetting of the first 64 bytes.

is written into the eFuse, it cannot be read again by an app and only the bootloader can read it.

There are two modes - developer and release mode, they differ in ability to reflash the encrypted storage - only developer mode allows it. In the release mode, firmware can be only updated using OTA. In development mode, the flash encryption can also be disabled after it has been enabled.

3.3.1 Flash Encryption Process

The encryption is handled by the second stage bootloader based on the value in `flash_crypt_cnt`. On the first run, the flash memory is unencrypted, the second stage bootloader check if an encryption key is present in the `BLOCK1` eFuse and if not, it generates an AES-256 key and stores it into `BLOCK1`. Then it performs in-place encryption of flash memory. When the encryption is finished, `flash_crypt_cnt` eFuse is updated to mark the storage as encrypted.

3.3.2 Reading Encrypted Data

When reading data stored in the flash memory and mapped into the flash cache (e.g. IROM and DROM), decryption is performed by the flash MMU. The only exception applies to flash regions used by NVS partitions, as they are not encrypted using flash encryption and MMU does not decrypt them. NVS handles its encryption separately as described in section 3.4.

3.3.3 Limitations of Flash Encryption

The flash encryption uses the same key for each successive pair of 16-byte blocks, therefore if the blocks contain the same plain text, their ciphertexts are also the same. This can be potentially used to reveal a secret stored in the flash memory or to fingerprint the device. For storing secrets an encrypted NVS partition should be used as it does not have this deficiency.

3.3.4 Flash Encryption and Secure Boot

If Secure Boot is used without Flash Encryption the flash content may be swapped after it had been verified by secure boot. If Flash Encryption is used without Secure Boot, encrypted partitions can be reflashed only with pre-encrypted binaries. An attacker can also corrupt individual blocks of the flash memory. Non-encrypted partitions can only be written to the device using OTA update or if the Secure Boot is enabled in *Re-flashable* mode.

3.4 NVS Encryption

Because NVS is incompatible with Flash Encryption, it implements its own NVS encryption. Unless NVS encryption is used, the data in NVS can be

read/erased or modified by anyone with physical access to the flash memory.

NVS encryption uses a standard AES-XTS algorithm (same as Bitlocker on Windows 10 [54]). The key is stored in a `nvs_keys` partition in the flash memory, which itself is encrypted using flash encryption. The `nvs_keys` partition can either be generated externally and then flashed to the device, or the keys can be generated by calling `nvs_flash_generate_keys`. For multiple encrypted NVS partitions multiple `nvs_keys` partitions can be used.

3.5 Cryptographic Accelerator

The accelerator embedded in the ESP32 SoC supports:

- AES 128/192/256 (FIPS PUB 197)
- SHA 1/256/384/512 (FIPS PUB 180-4)
- RSA w. keys up to 4096b long (bigint multiplication and exponentiation)
- RNG - TRNG/PRNG

HW accelerated operations are faster than if they were performed on the 240 MHz CPU. The HW implementation also functions as a 'secure default', therefore developers are encouraged to use it instead of depending on external SW libraries of potentially lower quality. The cryptographic accelerator is also used internally by ESP for Flash Encryption and Secure Boot.

3.5.1 Random Number Generator

Random number generator works in two modes - TRNG and PRNG. If radios are active, it uses the signal noise from them as the source of entropy and works as a true random number generator (TRNG). It has been repeatedly shown [55],[56] that numbers generated in this mode pass the DieHarder RNG test suite [57]. If radios are disabled, RNG switches to pseudo-random number generator (PRNG), but the ESP-IDF documentation explicitly warns against using the generator in PRNG mode, as the device does not have enough entropy available [58].

3.6 Over the Air Update (OTA)

Over the air update is a mechanism for updating the firmware of a running device without user intervention. It has to be done in a secure and reliable way, to prevent non-authorized firmware from being loaded into the device and to prevent device bricking when the firmware update is not successful [59]. A standard way to do OTA updates is to have (at least) two partitions for the firmware, so when the device is booted from one partition, the new

firmware can be downloaded into the other. After the firmware is verified and downloaded into the device, the second partition is set as boot partition and after a restart, the device runs the new firmware. On ESP32 this flow is supported by the `esp-ota-https` component.

3.6.1 Automatic App Rollback

To increase realiability, the FW can support automatic rollback, which happens if the new FW does not mark itself as valid (by cancelling the rollback using `esp_ota_mark_app_valid_cancel_rollback()`) or forces a rollback to a previous version by calling `esp_ota_mark_app_invalid_rollback()`.

If during the first boot of a new FW the power goes out or the watchdog timer is not fed, then automatic rollback will take place. Rollback is possible only between the apps with the same security versions. If a new firmware has a higher secure version than the old one and fails to mark itself as valid, the device would not boot again.

3.6.2 Security Version

In order to prevent a downgrade attack (forcing an automatic rollback or forcing a device to install an older firmware than it currently has), it is possible to set `secure_version` on the firmware image. When a firmware with a `secure_version` set is booted by the device, the value of this property is stored in the eFuse memory. After that the bootloader boots only firmware with security version larger or equal to the value stored in eFuse. If this validation fails, the device does not boot and the firmware gets erased from the device memory. Due to the nature of eFuse memory (inability to store 0 in a bit that has been previously set to 1) it is only possible to have 32 different secure versions.

3.7 Device Provisioning

When a customer receives a new device, he typically wants to connect it into his own network and configure it. This is called *device provisioning* and the two often used methods are WiFi (SoftAP) and BLE. In this section, we discuss the advantages and disadvantages of these methods, ESP-IDF *Unified Provisioning* system and one method of device provisioning that should be avoided.

3.7.1 Provisioning Using SoftAP

The Software enabled Access Point (SoftAP) is often used for configuring Wi-Fi in headless (without a browser or a screen) IoT devices. When the device is first started, it creates a temporary Wi-Fi network. The customer downloads

a configuration app provided by the device manufacturer and connects to the temporary Wi-Fi network. In the app, customer enters the credentials of the Wi-Fi network the device should connect to. The device then stops the SoftAP and joins the Wi-Fi network of the customer.

Advantages

- Device acts as AP only for a brief period of time, after that there is no direct communication with the device and all additional configuration can go through the cloud service.
- No additional radio (such as BT/BLE) is required.

Disadvantages

- Challenging to use for users with low technical skills [60] [61].
- Different handling of SoftAP in different mobile OSes, which may be confusing for the end-user and may result in unpleasant user experience. E.g. iOS does not allow an app to change the phone Wi-Fi network.
- Difficult to handle all edge cases of device handover between two WiFi networks as the IoT device may not support SoftAP and being connected to a different network at the same time. ESP32 does not suffer from this shortcoming.

3.7.2 Provisioning Using Bluetooth Low Energy (BLE)

IoT device does a BLE Advertisement and a device nearby (typically smartphones) detects this broadcast and notifies the user to connect to the device using Bluetooth. After the connection is made, the user uses an app provided by the device manufacturer to set up the Wi-Fi network credentials and to configure the device.

Advantages

- Users are familiar with this method as other (non Wi-Fi) devices use it - e.g. smartwatches, toothbrushes, toys, cameras...
- Better user experience as both iOS and Android enable mobile apps to scan and connect with BLE devices from apps.

Disadvantages

- Using BT increases the attack surface of the IoT device as BT is an additional attack vector that can be abused.

- Additional CPU/RAM requirements caused by two radio stacks (BT and WiFi) being active at the same time. E.g. in ESP32 this limits the amount of RAM available to the application running on it.

3.7.3 Unified Provisioning

Unified provisioning is a new feature of ESP-IDF in version 4 [62], it comes with mobile apps for iOS and Android and a library for ESP32 that communicate together. The developer only has to select which provisioning method (WiFi/BLE) he wants to enable for the device, set up a *proof of possession* key, choose a device name, and whether encryption should be used for the credentials exchange. Encryption uses AES-CTR and the key is established using the Diffie-Helmann algorithm on the X25519 elliptic curve.

3.7.3.1 Proof of Possession Key

Proof of possession key is a password that is unique for each device to make sure that only its possessor (and not e.g. his neighbor) can configure the device. This key can be printed on the device case or on its packaging and read using a QR code, so the user does not even need to know about it.

3.7.3.2 Provisioning Flow

When the user downloads one of the mentioned apps, he can see all devices that are in a provisioning mode and selects one of them either by name or by scanning a QR code printed on the device. If he chooses the device by name, he is asked for the proof of possession key. After that, he can select which WiFi he wants the device to connect to and enters its credentials. This setting is then transferred to the device and verified that it is valid. If yes, the device is now in a configured mode and no longer is visible in the provisioning app. If the user wants to repeat the process he typically has to press a dedicated button on the device, that erases the stored configuration and switches the device into the provisioning mode again.

3.7.4 ESP Touch or ESP Smart Config

ESP Touch [63] is based on proprietary protocol made by Texas Instrument called Smart Config[64]. It is used for transferring Wi-Fi credentials to another device by encoding them into the length field of 802.11 frames broadcasted on a Wi-Fi network. Recipient device captures these packets and tries decode the credentials from the length field.

Details of this protocol have never been officially published by Texas Instruments, its description of on Espressif website is very vague, and the implementation has not been open-sourced. Based on the API description of ESP Touch it is clear that it is not possible to encrypt the transferred Wi-Fi

credentials. This issue raises serious questions about the security of this technology and Changyu Li et. al [65] has shown that indeed implementation of Smart Config by Espressif is insecure, does not use encryption, and any device can sniff and decode the credentials being broadcasted.

3.8 Remote Control and Cloud Connection

3.8.1 Amazon AWS IoT Core

Espressif has developed a component for ESP-IDF [66] that wraps the official AWS IoT Device SDK [67]. It supports remote control, device shadow, and integrates the AWS IoT MQTT broker.

The device is authenticated using PKI and connects to the AWS IoT Gateway Endpoint over TLS. The device validates the presented server certificate using an AWS Root certificate embedded in the device firmware. Both of these certificates have by default an expiration date of 30 years.

3.8.2 Azure IoT Hub

Similarly to the component for AWS IoT Core, Espressif provides a component for Azure IoT Core, that wraps the official SDK [68]. It supports all of the main features of the platform - device provisioning[16], cloud messaging (using MQTT), remote control, and device shadow. From device protocols, only HTTP/MQTT is supported and for TLS it uses the esp-tls library.

Devices can be authenticated using PKI, Symmetric Key, or TPM. The Azure CA certificate is embedded in the SDK code and will expire in 2025, so all apps using this library will need to upgrade to a new certificate in the near future.

3.9 ESP-TLS

ESP-TLS is a component of ESP-IDF and brings a common API for working with TLS. The API exposes functionality for server certificate validation (CA cert, CA cert global store, and CN validation), client certificate authentication, support for pre-shared keys (PSK), and ALPN (application protocol negotiation). The API internally uses Mbed TLS or wolfSSL as the underlying SSL/TLS library. Both of these libraries utilize the cryptographic acceleration on ESP32. By default, Mbed TLS is used. Other configuration, e.g. cipher suite or TLS version selection can be set using library-specific configuration constants.

[16] In the context of Azure IoT Hub, device provisioning means the initial configuration of the device using the cloud backend. It has nothing to do with BLE/SoftAP provisioning mentioned in the previous section.

3.9.1 ESP x509 Certificate Bundle

This functionality creates a certificate bundle from all or a subset of Mozilla's NSS root certificates. It is created during build and stored in the flash memory. This bundle can be then used by ESP-TLS to verify server certificates during the TLS handshake.

3.9.2 ATECC608A support

Cryptographic module ATECC608A can be used for authorization in ESP-TLS [69]. Instead of passing a client certificate private key to the library, the library calls the crypto chip to perform all ECDSA operations using this key. It is supported only when MBed TLS is used. This crypto chip is embedded in the `ESP23-WROOM-32SE` module, but it can also be connected using I2C to any other ESP32 module.

3.9.3 WolfSSL

WolfSSL is an SSL/TLS library often used in embedded devices for its small size, low memory requirements, support of latest features, and FIPS compliancy [70]. It is available for free under a GPLv2 license or under a commercial license. Due to the restrictiveness of the GPL license, the source code of solutions based on wolfSSL must be freely available, so for commercial closed-source solutions, a commercial license is required. There is a licensed wolfSSL 4.3 binary provided by Espressif for free even for commercial EPS32 apps, but it comes with certain limitations: omission of TLS 1.3, FIPS-compliant features, and its free updates are guaranteed only until 2021.

3.9.4 Mbed TLS

Mbed TLS is an SLL/TLS library developed by ARM Mbed for IoT devices that has become an industry standard. It is licensed under a permissive Apache 2.0 license and can be used for free even in commercial closed-source products. As of writing this thesis, it does not yet support TLS 1.3. [71].

3.10 Known Vulnerabilities for ESP32

In the last year, an unpatchable vulnerability in ESP32 main security features - Secure Boot and Flash Encryption - has been discovered. Other known vulnerabilities were patched by Espressif in recent versions of ESP-IDF.

3.10.1 CVE-2019-17391 - Fault Injection and eFuse protection

The security of Secure Boot and Flash Encryption depends on the inability to extract their secret keys from `BLK1` and `BLK2` eFuse blocks. This attack work by injecting a 6V glitch to `VDD_CPU` and `VDD_RTC` CPU pins when the eFuse controller is initialized and the keys are read into a buffer for the Flash Controller [72]. It was shown that the glitching enables a readout of the keys in read-protected eFuses. The correct timings of the attack were discovered using Simple Power Analysis but were not described by the author. After the keys are obtained an attacker can use them to sign a non-genuine bootloader and firmware, and to decrypt the content of device flash memory.

According to the author, the attack requires approximately a day of work and equipment worth $500-$1000. There is no mitigation for this vulnerability in the form of a SW patch, it is only fixed in the latest HW revision of ESP32.

3.10.2 CVE-2019-15894 - Fault Injection and Secure Boot

This vulnerability[73] was discovered by the same author as 3.10.1 and it was the first published fault injection vulnerability in ESP32. The attack bypasses the Secure Boot by injecting a short 6V glitch to `VDD_CPU` and `VDD_RTC` pins of the CPU when branching based on the result of bootloader digest verification. A glitch makes the CPU to execute the success branch even if the digest is invalid [74]. According to the author, the attack requires approximately a day of work and equipment worth $500-$1000. This vulnerability is patched in ESP-IDF 3.3.1 and newer.

3.10.3 CVE-2019-12587 - Zero PMK Installation

Zero PMK Installation vulnerability is an attack against the improper implementation of WPA2-Enterprise in ESP32 WiFi stack. It enabled bypassing of authentication if WPA2-Enterprise was used for ESP WiFi in station mode [75]. This vulnerability has been patched in ESP-IDF 3.3+ and also backported to older versions.

3.10.4 CVE-2019-12586 - EAP DoS

This vulnerability has been reported together with the previous one and it allowed the attacker to crash the device by sending `EAP-SUCCESS` before PMK negotiation is completed. The two vulnerabilities were patched together.

3.10.5 CVE-2018-18558 - Secure Boot Bypass

Secure Boot contained a vulnerability where an attacker could create an app binary that would overwrite part of the second-stage bootloader and bypass

the Secure Boot if Flash Encryption was not enabled. It was patched in ESP-IDF 3.1.1 and backported to 3.0.6.

3.11 Security Improvements in ESP32 V3

To address the vulnerability described in section 3.10.1 (CVE-2019-17381), Espressif has released[76] a new revision of ESP32 referred to as V3 or ECO V3 with the following improvements:

- *Secure Boot* now uses standard public-key cryptography (RSA) to verify the second-stage bootloader (previously an AES key was used to calculate its digest). The key is stored in flash and its hash is stored in eFuse [77].
- *Resilience to Physical Fault Injection* The new revision has improved resistance to fault injection attacks to mitigate vulnerability CVE-2019-17381 described in section 3.10.1.

Modules with ESP32 V3 are `ESP32-WROVER-E` and `ESP32-WROOM-32E`.

3.12 Security Improvements in ESP32-S2

As mentioned previously in section 2.4.2 Espressif has announced a new version of ESP32 with most notable changes being reduced power consumption and improved security features.

ESP32-S2 brings the following security improvements when compared to ESP32 [77]:

- *Secure Boot v2* uses standard public-key cryptography (RSA-PSS) instead of custom digest algorithm (described in section 3.3.3) used by ESP32. The public key is stored in eFuse memory.
- *Flash Encryption* now uses AES-XTS, previously this algorithm has only been used to encrypt NVS partitions and flash encryption used a custom AES based mechanism, that had drawbacks mentioned in section 3.2.1.
- *New cryptographical accelerator* with faster performance and support for ECC and AES-GCM. It also supports the ability to sign messages using RSA without letting the app access the private key. This previously required an external chip e.g. Microchip ATECC608A[17].
- *Increased size of eFuse memory* - as RSA keys and signatures are larger than AES keys and SHA-256 hashes, the size of eFuse memory has been increased to 4Kbits.

[17] ATECC608A supports ECDSA (elliptic curve digital signature algorithm) instead of RSA based signatures.

- *Resilience to Physical Fault Injection* The new chip patches vulnerability CVE-2019-17381 described in section 3.10.1 by having improved resistance against fault injection.

Chapter 4

Practical Part

This chapter describes a proof-of-concept app I have created that utilizes the described security features of ESP32. This app implements a typical IoT scenario - collection of live data from a sensor and sending them to a cloud service.

This part contains examples of the app source code. The examples are often simplified and have non-essential parts (such as error checking or variable declaration) omitted for brevity.

4.1 Overview

The app is build using the ESP-IDF platform and communicates with an Azure IoT Hub as the backend service. It collects live data from a DHT-22 temperature and humidity sensor and sends them to the cloud for potential further processing. All communication is secured using TLS with the device

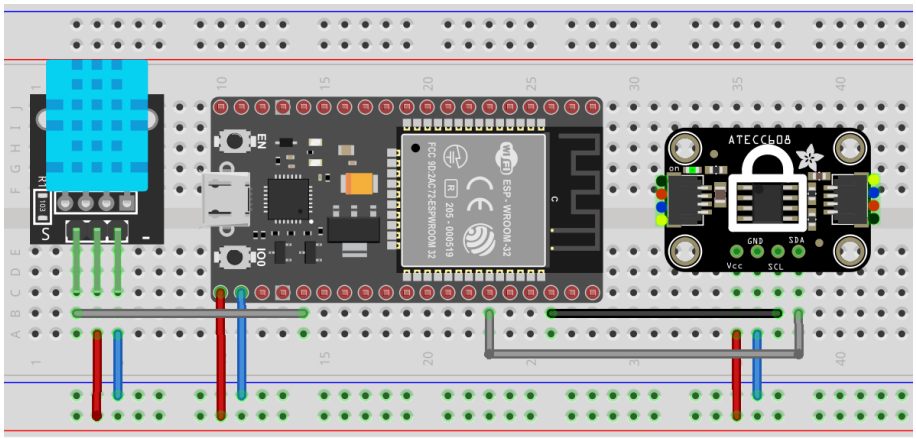

Figure 4.1: DHT22, ATECC608A and ESP32 connected on a breadboard [50].

being authenticated by presenting its X.509 certificate. This certificate is stored in a separate partition of the device storage, so it is not a part of the firmware, which can be then shared among multiple devices. The private key to this certificate is generated and then stored in an ATECC 608A HSM module, connected over I2C, and never leaves this device nor is present in the device memory. The app also uses Secure Boot, NVS Encryption, and Flash Encryption.

4.2 Cloud Provider Selection

I have considered two PaaS cloud services for this thesis - Aws IoT Core and Azure IoT Hub. The services are mostly similar in what they offer (device management, remote control, messaging, etc.) and for both, there are official ports of their SDKs available for the ESP-IDF platform. AWS has a more user-friendly interface and is easier to understand and configure. Azure, on the other hand, has a better component for ESP-IDF which does not suffer, from my experience, with bugs in basic functionalities, is better integrated with the rest of ESP-IDF toolkit, and its Github repository (esp-azure[18]) seems to be more active compared to esp-aws-iot[19]. For these reasons I have picked Azure over AWS for this thesis, even though I found working with AWS IoT Core more pleasant than with the Azure platform.

4.2.1 Device Authentication

In both services, users can add their own CA in order to use certificates signed by this CA for device authentication. AWS also has a built-in CA, so users can send CSRs to it and the devices can use certificates signed by AWS IoT Core CA.

There are however differences in how the certificates are bound to individual devices. In AWS IoT Core a certificate is explicitly bound to a device and a policy. Azure IoT Hub does not have this configurable binding but the certificate has to have *Common Name* matching the device name. Because of this, there is no management of device certificates directly in Azure IoT Core.

This leads to a different behavior for banning/disabling devices - in Azure, if a device is disabled it means that IoT Hub will disable connection from a device with the disabled name in its certificate. In AWS IoT Core it is possible to disable device certificate, in order to prevent it from connecting again, and then attach a different certificate to the device. To achieve the same workflow in Azure, we have to use an additional service - Azure IoT Hub Device Provisioning Service [78].

[18] `https://github.com/espressif/esp-azure`
[19] `https://github.com/espressif/esp-aws-io`

4.2.2 Azure IoT Hub Device Provisioning Service

This service works as a coordinator that assigns individual IoT Hubs (in large solutions there may be multiple instances of IoT Hub e.g. for different regions or customers) and initial device configuration to devices when they connect to Azure for the first time.

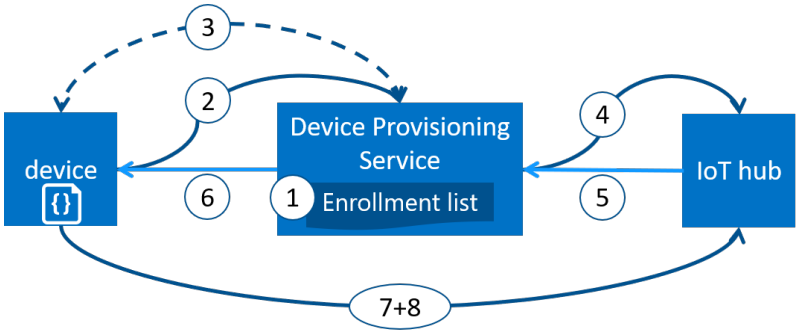

Figure 4.2: Azure Device Provisioning Service flow [78]

The process is shown in figure 4.2. In the first step device credentials (X.509 certificate, symmetric key, or TPM identity) are added to the enrollment list in the Device Provisioning Service (DSP). The device then connects to DSP (step 2) and DSP validates its identity (step 3). In the next step, a device is created in an IoT Hub and device twin (also called device shadow) is populated with the desired initial configuration. After that (step 5) the DSP sends access information to the device (step 6). The device now can communicate directly with the IoT Hub (step 6) and receives the desired state from its twin (step 7).

The credentials used for each enrollment can be individually managed, it is possible to manage individual device certificates with this service. If it is later needed to change the device certificate, the old enrollment with the old certificate can be deleted, a new enrollment with a new certificate created, and the device deleted from IoT Hub. The device can then be re-provisioned using the new certificate.

As this process is too cumbersome for a demo app that should showcase the security features of ESP32, I choose not to use it.

4.3 Device Identity & Authentication

The device is identified using its certificate. This certificate is signed by a self-signed CA for the demonstration purposes of this app. The CA certificate must be uploaded and verified in the Azure IoT Core service first, so the service can validate the certificate presented by the device.

I have decided to store this certificate private key in an external HSM instead of the ESP32 flash because of the vulnerability affecting eFuses and SecureBoot as mentioned before in section 3.10.1.

To work with the ATECC608 I have used an `esp-cryptoauthlib` component which wraps Microchip CryptoAuthLib[20] library as an ESP-IDF component and also contains a utility to provision the module.

The library is intended to be used with ESP32SE and not regular ESP32, but the only modifications I needed for it to run was changing the ports ATECC 608A is connected to in the ESP32 HAL `hal_esp32_i2c.c` as following,

```
#define SDA_PIN 21
#define SCL_PIN 22
```

and changing the I2C bus used to communicate with ATECC module from 2 to 1.

```
ATCAIfaceCfg cfg_ateccx08a_i2c_default = {
    .iface_type                 = ATCA_I2C_IFACE,
    .devtype                    = ATECC608A,
    {
        .atcai2c.slave_address  = 0xC0,
        .atcai2c.bus            = 1,
        .atcai2c.baud           = 400000,
    },
    .wake_delay                 = 1500,
    .rx_retries                 = 20
};
```

When secrets are transported over the I2C bus, they are encrypted using an IO Protection feature of ATECC608A. Unfortunately, its description is not available in the public version of the documentation [79].

4.3.1 ATECC608A Provisioning and Certificate Generation

At first we need to provision the cryptochip and generate a device certificate. This is done using the `secure_cert_mfg.py` script:

```
python secure_cert_mfg.py --fw secure_cert_mfg.bin \
    --signer-cert-private-key ca-key.pem \
    --signer-cert ca-cerr.pem --port COM3
```

The script and the provisioning FW is from the `esp-cryptoauthlib` [21] library and it does the following:

[20] https://github.com/MicrochipTech/cryptoauthlib/
[21] https://github.com/espressif/esp-cryptoauthlib

1. Opens a connection over the serial port (set by `--port`) with the device and loads the specified firmware `secure_cert_mfg.bin` into its RAM (IRAM and DRAM). Note, that this FW is not flashed to the device.

2. Sends a command to the device to generate a new *ECC 256* key-pair using the ATECC module and send back the public key.

3. Sends a command to the device to create a certificate signing request (CSR) signed by the key generated in the previous step. It uses the device MAC as CN in the CSR.

4. The received CSR is then signed by the CA and stored in PEM format in the `device-cert.pem` file.

The provisioning FW is distributed by Espressif only as a binary.

4.3.2 Certificate Storage on ATECC 608A

I have also explored the possibility of storing the device certificate directly in the cryptomodule. In order to do that I needed to customize the provisioning firmware. Fortunately, I have found an older version of it in a fork of the official `esp-idf` repo[22].

I compiled the firmware with the `ESPWROOM32SE` flag. This flag switches the library used for cryptographical actions from `mbedtls` to `cryptoauthlib` which used for communication with the ATECC module.

The module supports certificate storage via the `atcacert_write_cert` function, but because the module has a small amount of memory, the certificate is stored in a compressed form as described in [80]. The problem is that this compression is achieved by storing only a minimum amount of data inside the chip. And storing the rest in a certificate definition and a certificate template in the application. The certificate template is just a byte array, but the certificate definition is a struct in c (parts of both shown below).

```
const uint8_t cert_template[] = {
    0x30, 0x82, 0x01, 0xcc, 0x30, 0x82, 0x01, 0x73,
    0x02, 0x14, 0x24, 0x9f, 0xb7, 0xe7, 0x9d, 0x58,
    ...
}

const atcacert_def_t cert_def = {
    .type                   = CERTTYPE_X509,
    .template_id            = 2,
```

[22] https://github.com/AdityaHPatwardhan/esp-idf/tree/feature/add_example_for_using_atecc608a_with_WROOM32SE/examples/security/atecc608a_ecdsa/secure_cert_mfg/main

```
    .chain_id               = 0,
    .private_key_slot       = 0,
    .sn_source              = SNSRC_PUB_KEY_HASH,
    .cert_sn_dev_loc        = {
        .zone      = DEVZONE_NONE,
    },
    .std_cert_elements      = {
        { // STDCERT_PUBLIC_KEY
            .offset = 315,
            .count  = 64
        },
        ...
    }
    ...
}
```

The certificate definition may slightly change even with a small change in the certificate and this would require building a new firmware with each certificate (in a real-world scenario a unique FW for each device). I found this impractical and hard to work with and decided to store the device certificate in an encrypted NVS partition instead.

4.3.3 Certificate Storage in NVS

The device X.509 certificate is stored in an encrypted NVS partition. In a real-world scenario, this enables decoupling of the app from per-device specific data (in our case the device certificate), so the app FW does not have to be recompiled separately for all devices.

First, the encryption key for the partition is generated:

```
python nvs_partition_gen.py generate-key --keyfile nvs-key.bin
```

The partition is defined by the `certs_nvs.csv` file, which describes its name and where is the device certificate located. This file is then transformed into a binary file (`certs-nvs.bin`) with the specified size (`0x20000`) and encrypted with the key.

```
python nvs_partition_gen.py encrypt --keyfile nvs-key.bin
    certs_nvs.csv certs-nvs.bin 0x20000
```

Now the two partitions can be flashes to the device to their corresponding offsets in the flash memory:

```
esptool.py write_flash --port COM3 0x29000 certs-nvs.bin
esptool.py write_flash --port COM3 0x49000 nvs-key.bin
```

The app reads the certificate during the initialization of the HSM module layer by Azure IoT SDK. It firsts finds the correct partition with the encryption key, parses it, then initializes the `certs` NVS partition, opens it for reading, and reads the PEM certificate stored in it into a string, that is then passed into the *tlsio layer*.

```
esp_partition_t* keys_partition = esp_partition_find_first(
    ESP_PARTITION_TYPE_DATA,
    ESP_PARTITION_SUBTYPE_DATA_NVS_KEYS, "nvs_keys");
nvs_flash_read_security_cfg(keys_partition, &nvs_keys);
nvs_flash_secure_init_partition("certs", &nvs_keys);
nvs_flash_init_partition("certs");
nvs_open_from_partition("certs", "certs", NVS_READONLY, &nvs);
...
nvs_get_str(nvs, "device_cert", buff, &required_size);
```

4.3.4 Connection to Azure

In order to connect with Azure IoT Hub, I had to make small changes to the `esp-azure` component. First, I needed to connect the Azure IoT SDK with the ATECC608a HSM. There is an API [81] for HSMs that store X.509 certificates and the SDK uses this API to load the certificates and associated private keys. As this is not possible, I only implemented the `custom_hsm_get_certificate` method, that returns the certificate stored in the device memory. In the `custom_hsm_get_key` method I return a dummy value, so the checks in the SDK do not fail. Values from these methods then flow into the *tlsio layer* that servers as an abstraction of TLS communication for the SDK and its implementation is platform-specific. ESP32 implementation is in `tlsio_esp_tls.c` which uses the `esp-tls` library. Here the only thing I had to change was setting the `use_secure_element` property and disable loading of the private key.

After that the HSM module can be initialized and then used to authenticate with the IoT Hub:

```
iothub_security_init(IOTHUB_SECURITY_TYPE_X509);
client = IoTHubDeviceClient_LL_CreateFromConnectionString(
    connectionString, MQTT_Protocol)
```

The `connectionString` contains only the IoT Hub hostname and DeviceId, there are no credentials in it when HSM is used.

4.3.5 Validation of Azure TLS Certificate

As earlier described in section 1.4.4, the validation of the server certificate is essential to ensure that the TLS connection is secure, so I was naturally

interested to see how the Azure certificate is validated. As mentioned previously, all TLS operations in Azure IoT SDK are handled by platform-specific libraries, in our case `esp-tls`, that leaves the validation to the underlying TLS library, expecting that either it will validate the certificate against the Microsoft Root CA certificate.

Because the `esp-tls` library is insecure by default, as later describes, the validation has to be explicitly configured by setting the `OPTION_TRUSTED_CERT` option. The validation is handled later in the tlsio HAL and passed to the underlying mbedTLS library.

```
#include "certs.h"
...
IoTHubDeviceClient_LL_SetOption(iotHubClientHandle,
    OPTION_TRUSTED_CERT, certificates);
```

The `certificates` variable comes from `certs.h` file of the SDK and contains all Azure CA certificates used by IoT Hub. There is also an additional build flag `USE_BALTIMORE_CERT` that ensures only the certificate of Azure Global is used and excludes CA certificates of Azure Germany and Azure China from the `certificates` array.

An alternative is to use the *ESP x509 Certificate Bundle* with Mozilla's NSS root certificate store. It can be used in `tlsio_esp_tls.c` in the following way:

```
#include "esp_crt_bundle.h"
static int tlsio_esp_tls_open_async(...){
...
tls_io->esp_tls_cfg.crt_bundle_attach = esp_crt_bundle_attach;
...
}
```

Now the library is correctly configured and it performs Common Name validation and Chain of trust validation (against the CA cert/CA store). Additionally, expiration checking can be enabled by the `MBEDTLS_HAVE_TIME_DATE` constant. Mbed TLS also supports checking of the CA Certificate Revocation List (CRL), if `MBEDTLS_X509_CRL_PARSE_C` is defined. Note that there is currently a bug[23] that makes CRL checking effectively non-functional in Mbed TLS unless `MBEDTLS_HAVE_TIME_DATE` is also defined.

The app uses `OPTION_TRUSTED_CERT` validation and is build with following constants defined: `USE_BALTIMORE_CERT`, `MBEDTLS_HAVE_TIME_DATE`, and `MBEDTLS_X509_CRL_PARSE_C`.

[23] urlhttps://github.com/ARMmbed/mbedtls/pull/3433

4.4 Flash Partitioning

The device flash memory has the following partitions:

- `nvs` - Default NVS storage used by various ESP-IDF componnents (e.g. Unified Provisioning).
- `phy_init` - System partition with settings for the physical layer, described in section 2.3.4.
- `certs` - Encrypted NVS partition containing the device certificate (and potentionally other certificates and secrets).
- `nvs_keys` - Keys for the `certs` partition.
- `ota_data` - System partition with data for OTA.
- `ota_0` - Default OTA partition.
- `ota_2` - Second OTA partition.

In the demo app, this partition table is specified in the `partitions.csv` file, transformed into a binary file during build, and then flashed to the device.

Sometimes it is needed to know the exact offsets of individual partitions, to get them the binary partition table can be transformed back to a .cvv file:

```
python gen_esp32part.py build/partition_table/partition-table.bin
    partitions_w_offset.csv
```

4.5 WiFi Provisioning

For WiFi configuration (also called WiFi provisioning), I used the Unified Provisioning component that is described in section 3.7.3. I choose to use the SoftAP method and not BLE in order to keep the app image smaller and also to try the security features of the provisioning. According to the docs [82] if the `security1` scheme is enabled, the transmitted credentials are encrypted using AES in CTR mode with the key created using ECDH on the *Curve25519*. There is also a SHA256 hash of the *proof of possession key* (a secret that should be unique per device to prevent non-owners from provisioning it) that is XORed to the shared secret established using ECDH to get the final AES-265 shared key.

To keep the app image the same for potentially multiple devices, I have decided to store the proof of possession key in the same encrypted NVS partition as a device certificate.

I had a few issues with Unified Provisioning - first, it made the app image larger than 1M, so I had to change the partitioning table. Second, the key exchange uses Mbed TLS, so I had to enable support for *Curve25519*. Also,

the iOS app provided by Espressif has become buggy and the provisioning process sometimes failed for me.

I was also curious to see how the WiFi credentials are stored and I have found in the documentation [82] that they are stored in the default NVS partition. Unfortunately the implementation of `esp-wifi` is closed-source, so I was not able to check how exactly the credentials are stored, but they can be read in plain text using the `esp_wifi_get_config` function.

```
wifi_config_t wifi;
esp_wifi_get_config(ESP_IF_WIFI_STA, &wifi);
printf("ssid: %s, pswd: %s", wifi.sta.ssid, wifi.sta.password);
```

Based on that I choose to also use encryption for the default NVS encryption to ensure its content cannot be read by dumping the flash memory. And as explained earlier in section 3.4, the NVS partitions are not encrypted by default even if *Flash Encryption* is used.

4.6 Data Collection

To collect some real-world telemetry data, I used the DHT-22 temperature and humidity sensor. To read the values from the sensor I used a `dht22`[24] library that implements the protocol used by DHT-22. The data are collected every 1m and send in JSON to the IoT Hub over MQTT.

```
{
    "deviceId": "01237539495F6120EE",
    "temperature": 27.40,
    "humidity": 40.33
}
```

The received data are routed by the Azure Event Hub to the Azure Table Storage, which stores them. A real-world app we would have apps processing the live messages received by the Event Hub, e.g the received telemetry data could be stored in a time-series database TimeScale[25] which enables efficient querying of the latest data and their automatic aggregation. The Table Storage or similar service would function as a cold-storage, which is cheap, has all the data, but is slow to query.

[24] `https://github.com/gosouth/DHT22`
[25] `https://www.timescale.com/`

4.7 TLS

4.7.1 TLS 1.3 Support on ESP32

TLS 1.3 is not well supported on ESP32. One issue with TLS 1.3 on ESP32 is that non of its supported ciphers (AES GCM/CCM, CHACHA20, (EC)DHE and POLY1305) are HW accelerated on the device and therefore are up to 10 times slower [83]. Mbed TLS also does not yet support TLS 1.3 and while WolfSSL does, it is only available under commercial or restrictive GPL license. But this can change soon, as ECDHE and AES-GCM have HW acceleration on ESP32-S2, and MBed TLS has a prototype implementation of TLS 1.3. [84].

4.7.2 ESP-TLS - Insecure by Default

I have discovered that the `esp-tls` library *does not perform any chain of trust validation* unless explicitly configured, and it does not even produce a security warning. This goes against a basic rule of secure software development - *secure by default*, which is also part of IoT OWASP Top Ten.

The problematic method is `set_client_config` that checks if any of the certificate validation (namely: certificate bundle, global store, and ca certificate) method has been configured and if not, it configures the underlying library (Mbed TLS or wolfSSL) to not perform validation on the received server certificate. This makes the communication vulnerable to the man in the middle attack, therefore any party that can intercept the traffic is also able to decrypt the communication.

Note that this is the exact same problem a famous paper "The most dangerous code in the world" [45] described as being widely present in 2012, that is still present today in a SW stack used by tens of millions devices.

4.7.3 ATECC608A Support in ESP-TLS

As I was working on this thesis, Espressif has published an official integration of ATECC608A into the esp-tls library [85]. So I discarded some of the patches, for communication with the HSM, I had been working on and switched to the official implementation. It requires ESP-IDF 4.2 or newer, which is still a beta version and the following build constants to be set.

- `CONFIG_ATECC608A_TCUSTOM` - Sets the type of ATECC608A module.
- `CONFIG_ESP_TLS_USE_SECURE_ELEMENT` - Builds the `esp-tls` library with support for the module.
- `CONFIG_ATCA_MBEDTLS_ECDSA`, `CONFIG_ATCA_MBEDTLS_ECDSA_SIGN`, and `CONFIG_ATCA_MBEDTLS_ECDSA_VERIFY` - Makes the `mbed-tls` library call the ATECC608A for ECDSA operations.

- `CONFIG_MBEDTLS_ECP_DP_SECP256R1_ENABLED` - The *secp256r1* curve is the only one implemented by ATECC608A.

Then the support can be enabled by setting single property in `esp-tls` configuration:

```
esp_tls_cfg_t cfg = {
    ...
    .use_secure_element = true,
};
```

The constants are usually set correctly by the ESP configuration tool, but that sometimes fails and either the `cryptoauthlib` or the `mbed-tls` library starts to return nondescript errors.

4.7.4 Cipher Suite Selection

I have configured the Mbed TLS library to only use `DHE-RSA`, `ECDHE-RSA`, and `ECDHE-ECDSA` based ciphersuites as they ensure forward secrecy. For symmetric cipher, I have allowed AES in CBC, CCM, and GCM modes, because only AES is HW accelerated on ESP32. For hashing algorithms, only SHA-1 and SHA-256 are allowed. Note that SHA-1 is still considered secure for usage in HMAC-SHA1 as HMAC is not compromised if the weak-collision resistance of the hashing function is compromised [86].

The resulting list of all allowed TLS ciphersuites (obtained by enabling debug logging in Mbed TLS) is in appendix B. It could be further pruned by disallowing AES-CBC encryption that may be considered weak due to padding oracle vulnerabilities (e.g. POODLE) discovered in various libraries implementing it. It is not insecure by itself, as some TLS implementations, e.g. Mbed TLS[26], had never been susceptible to this vulnerability [87].

I have also disabled older TLS 1.0 and 1.1 versions, so only TLS 1.2 is supported by the app. During this I have discovered that disabling TLS 1.1 was broken in `esp-tls` and submitted a pull request to the `esp-idf` repository[27].

For communication with the Azure IoT Hub, the server chooses its preferred cipher suite `TLS-ECDHE-RSA-AES-128-CBC-SHA256` from the list of allowed ones presented by the client. It is the same cipher suite as Edge on Windows 10 or Safari on iOS chooses by default, as shown in a report by SSL labs[28].

[26] When the POODLE attack was published in 2014, Mbed TLS was still called Polar SSL.

[27] https://github.com/espressif/esp-idf/pull/5675

[28] `https://www.ssllabs.com/ssltest/analyze.html?d=esp-thesis.azure-devices.net`

4.8 Secure Over the Air Update

To implement secure over the air update (OTA) I have used *Direct method invocation* from the IoT Hub and `esp-https-ota` library from ESP-IDF. Direct method invocation allows request-response communication between cloud and the device when the device is online. When the `ota` method is invoked with the following payload:

```
{"url": "https://esp32thesisstorage.blob.core.windows.net/
    otafw/thesis-app.bin"}
```

the method callback on the device executes the following code.

```
esp_http_client_config_t config = {
    .url = url,
    .cert_pem = (char *)certificates,
};
return esp_https_ota(&config);
```

It establishes an HTTPS connection, validates the server certificate with Azure *Baltimore* Root CA certificate[29] and downloads the firmware. The library then checks the firmware signature, if secure boot or code signing is enabled, and installs the FW into an OTA partition that is not currently booted. After that a response is returned to the sender, informing him whether the operation was successful. The device then reboots and boots into the new firmware.

4.8.1 Secure Boot and Signed App Verification

To ensure only authorized firmware is used on the device, signed app verification should be used. It is required if Secure Boot is active, but can also be used without it by explicitly enabling it in the project configuration. I have used it to verify that the entire process of OTA update with signed binaries is working correctly without risking bricking the device. First an ECDSA keypair on *secp256r1* curve must be generated and the public key extracted into a .bin file:

```
openssl ecparam -name prime256v1 -genkey -noout
    -out signing_key.pem
python espsecure.py extract_public_key
    --keyfile signing_key.pem signing_pub.bin
```

The public key is then added in the project configuration as the *Secure Boot Signing Key* (even if only Signed App Verification is used) and the private key is used for signing the firmware binary as following:

[29] I choose to store the firmware in Azure Blob Storage, that is why the same Root CA certificate can be used.

```
espsecure.py sign_data --keyfile signing_key.pem thesis-app.bin
```

Now the binary can be uploaded to the Blob Storage and its URL send to the device by invoking the `ota` method. Its signature will be verified before it is written to the flash memory and then every time before it is booted.

4.8.2 Automatic Rollback and Anti-rollback Protection

As described earlier in section 3.6.1, to ensure the correct function of the device, an *Automatic Rollback* functionality can be used to restore the previous firmware if the new OTA firmware is not working correctly on the device. This functionality has to be enabled in the project configuration, which sets the `CONFIG_APP_ROLLBACK_ENABLE` constant. In the proof of concept app I use the code below which calls `esp_ota_mark_app_valid_cancel_rollback()` if the first boot, when `ESP_OTA_IMG_PENDING_VERIFY` is set, of the new firmware is successful.

```
partition = esp_ota_get_running_partition();
res = esp_ota_get_state_partition(partition, &part_state);
if (res != ESP_OK) return;

//Confirm OTA FW if pending
if (part_state == ESP_OTA_IMG_PENDING_VERIFY){
    ESP_ERROR_CHECK(esp_ota_mark_app_valid_cancel_rollback());
}
```

To prevent installing an old firmware and potentially insecure firmware on the device, the anti-rollback protection can be enabled, which disallows booting a firmware with a lower security version than the one stored in an eFuse. I have only tested this feature in the emulation mode, which does not burn the eFuse.

4.8.3 Securing the Firmware Blob Storage

For the example above I have used the Blob Storage in a mode that allows anonymous access, enabling anyone with the link download the particular firmware. The Azure Storage does not support authentication using a client certificate, so either the firmware would have to be downloaded through a proxy service (e.g. an Azure Function with certificate authentication) or a SAS token can be used. The SAS token is part of the URL and allows temporary access to the resource [88]. In our case, the device receives firmware URL with a token that enables the device to download one particular file in the next hour. The token can be even restricted only to the device's public IP. With the SAS token the payload of `ota` method looks like this:

{"url":"https://esp32thesisstorage.blob.core.windows.net/otafw/thesis-app.bin?sp=r&st=2020-08-01T00:05:29Z&se=2020-08-01T01:05:29Z&spr=https&sv=2019-12-12&sr=b&sig=sIBVGwlCB483ly956jv78\%2FybM\%2BzrPYIqQuBmz8NJx2o\%3D"}

The main advantage is that the app does not have to be modified in order to work with the SAS token authentication.

4.8.4 OTA Update Using Device Twin

An alternative to the direct method would be using *device twin* with the new firmware URL being passed as the desired state. After the firmware is updated, the device would confirm the update by sending this URL as a reported state. The advantage of this method is compatibility with bulk device provisioning using Azure IoT Hub Device Provisioning Service. The disadvantage is that this is an indirect (asynchronous) method and there is no built-in feature to report back errors that occurred during OTA.

4.9 Secure Boot and Flash Encryption

With the proof-of-concept app, my intention was to test these features, and not to create a production-ready device, as I have used both features in a development mode only.

I have used Secure Boot in *reflashable mode* with `SECURE_BOOT_INSECURE` option, so JTAG would not be disabled, and reflashing the device via UART would still be possible. The main difference between these settings and *Signed App Verification* is that the bootloader digest is also being verified.

For Flash Encryption I have used the *Development mode*, which again enables reflashing and even disabling the Flash Encryption after it has been enabled. I choose that the key would be generated by the device on the first run, so the binaries are flashed unencrypted to the device, which then encrypts them.

For a production device, both Secure Boot and Flash Encryption must be enabled in the *production mode*, which also disables features like JTAG debugging and UART ROM download. Otherwise, the device would be basically insecure.

Even with both Secure Boot and Flash Encryption being vulnerable to a fault injection attack, they still prevent less sophisticated attacks, so I believe there is still value in using them, but should not be used with the expectation of protecting secrets on the device or protecting against non-genuine firmware.

4.10 List of keys

All the cryptographic keys are used to secure the app are listed in the table 4.1.

Keep in mind, that all the keys that are generated outside of the device should not be reused between multiple devices and should be securely stored. If the keys are reused, one compromised key can leave multiple devices vulnerable.

Purpose	**Type**	**Key Storage**	**Generated By**
Device Identity & Authentication	ECC 256 (secp256r1)	ATECC 608A (Private key) Flash memory (Certificate)	ATECC 608A
Bootloader Digest	AES 256	eFuse, external	External (PC)
Secure Boot App Signing	ECC 256 (secp256r1)	External (Private key) Flash memory (Public key)	External (PC)
Certs NVS Partition	AES 256	Flash memory	External (PC)
Flash Encryption	AES 256	eFuse	ESP32

Table 4.1: List of cryptographic keys used by the device.

4.10.1 OWASP Top 10 IoT and Countermeasures

Table 4.2 shows all OWASP Top 10 IoT, which were discussed in the first chapter, and countermeasures that were taken to prevent them in the proof-of-concept app. The other risks to IoT devices discussed were energy harvesting and insufficient auditing. The first one does only apply to battery-powered devices, which this thesis did not explore. And for insufficient auditing, there just are not any practical solutions available - none of the PaaS IoT services offer log collect solution. Also, the ESP-IDF framework is not written with auditing in mind, so it was not implemented in the proof-of-concept app.

OWASP Top 10 IoT	**Countermeasures**
Weak, Guessable, or Hardcoded Passwords	Only unchangable password is the proof of posession key. It is not shared between devices and has sufficient lenght.
Insecure Network Services	No network services run on the device except when WiFi provisioning is active. In standard operation the device is only a client, not a server.
Insecure Ecosystem Interfaces	All IoT Hub APIs use encrypted communication and require authentication.
Lack of Secure Update Mechanism	Secure OTA update.
Use of Insecure or Outdated Components	Latest versions of components with no known vulnerabilities are used. OTA updates are supported.
Insufficient Privacy Protection	WiFi credentials are stored in an NVS encrypted partition. No other PII are stored on the device.
Insecure Data Transfer and Storage	TLS 1.2 is used for all communication. No data are stored on the device itself.
Lack of Device Management	Azure IoT Hub is used for device management.
Insecure Default Settings	There are no user settings that have impact on security.
Lack of Physical Hardening	Cryptographic keys are stored in a tamper-resistant HSM.

Table 4.2: OWASP IoT Top Ten and counter measures used by the app.

Chapter 5

Summary & Discussion

In this chapter, all the main points of the analysis and learning from creating of the proof-of-concept app are summarized and discussed.

5.1 Communication Protocols

- MQTT should always be used with TLS, as it is not an encrypted protocol.
- MQTT is the right choice for most IoT solutions.
- CoAP or MQTT-SN can be used for IoT devices communicating over LPWAN.
- TLS has large overhead, especially if reconnection is frequent and sessions tickets are not used.
- TLS 1.3 support is not yet supported on ESP32 and PaaS solutions.

MQTT has rightfully become the standard protocol in the IoT world, it is efficient, secure if used with TLS, reliable, and supported by all the PaaS services for IoT by major public cloud vendors. Unfortunately, the MQTT brokers in PaaS offerings only support a subset of MQTT features. Where MQTT is not usable (e.g. on LPWAN networks such as SigFox or NB-IoT) an MQTT-SN bridge or CoAP protocol can be used to achieve reliable cloud-to-device and device-to-cloud communication.

Using TLS brings overhead of two handshakes and approximately 6KB unless session tickets or 0-RTT connection resumption is used. TLS 1.3, which implements 0-RTT, is not yet well supported by the ESP32 and neither by PaaS offerings for IoT solutions.

5.2 Device identification & Authentication

- Devices should be identified using PKI, ideally in combination with an HSM module.
- Usage of PUF is not as widely used on ESP32 as it is on ARM.

I have found that the only widely used solution for ESP32 device identification is by using PKI. The device private key can be stored in an HSM chip or an external PUF. I have used an external HSM by Microchip that performs ECDSA operations with a private key that never leaves the HSM. This key is then used to sign a CSR passed to a CA which is trusted by the cloud service. The devices use the issued certificate for authentication to the service.

ESP32 does not have a PUF solution integrated in the SoC as many ARM MCUs have and there are no known solutions with good support for ESP.

5.3 Secure Boot and Flash Encryption are Vulnerable

- Secure Boot and Flash Encryption can be circumvented using fault injection attack.
- Secure Boot cannot be used as a Root of Trust for the device.
- Flash Encryption itself is insufficient to protect secrets stored on the device.
- Use ESP32-V3 or ESP32S2 when available, consider using HSM for cryptographic keys.

ESP32 is vulnerable to a fault injection attack which can be used to circumvent *Secure Boot* and *Flash Encryption*, so the device can be reflashed with untrusted code and its memory dumped and decrypted. Therefore additional security measures should be taken to preserve the confidentiality of secrets stored on ESP32. One possibility is to use an HSM such as ATECC608a I used in this thesis or ESP32SE which has this HSM directly in its module. This vulnerability has been patched in newer HW revision ESP32-V3 and in ESP32S2, which should be used for new designs. Unfortunately, none of them were available as a DevKit in retail stores when writing this thesis.

5.4 ESP-TLS Library is Insecure by Default

- ESP-TLS makes the device vulnerable to MiTM attack unless properly configured.

- Disabling TLS 1.1 is broken in the current version.
- Consider using Mbed TLS without the ESP-TLS wrapper.

I have discovered, that the ESP-TLS library does not validate the certificate presented by the server unless it is explicitly configured to do so. This behavior does not cause any warnings to the developer, potentially leaving many applications vulnerable to MitM attack on TLS by accident. This vulnerability is also inherited by other libraries using ESP-TLS such as ESP-MQTT or ESP-Azure. I've also discovered that disabling TLS 1.1 in ESP-TLS does not disable it in the underlying library. I've fixed it and it has been merged into `esp-idf` master. My recommendation for future projects would be to use MBed TLS directly and do not rely on potentially buggy ESP-TLS wrapper.

5.5 Great HW and SW Support

- ESP32 is a popular platform with great support.
- Official SDK supports many features, but the quality is sometimes lacking.
- Most of the platform is open-sourced.

ESP32 is very popular and has many libraries and HW drivers available, made both by Espressif and community. Espressif's libraries are usually available as open-source software and the official documentation is also well-made. What is sometimes lacking is the quality and reliability of apps and tools as even the official ones crash, or stop working.

5.6 Over the Air Update

- Should verify the firmware signature.
- Should support rollback and anti-rollback protection.
- Use the official library in ESP-IDF.

The official library for over the air update is simple to use and supports all the required features such as code signature verification, an automatic rollback in case the new FW malfunctions, and also protection against installing an old and potentially insecure firmware.

5.7 Proof of Concept App

- Uses ATECC 608A HSM to store device private key.
- Implements certificate-based authentication with Azure IoT Hub.
- Implements secure over the air update.
- Wifi Configuration using a mobile app is easy to implement, but the app sometimes crashes.
- Shows how Secure Boot, NVS, and Flash Encryption can be used.

The proof of concept app was developed to try the previously described security feature. There were just small issues when integrating the ATECC 608A crypto chip, that has been added to ESP-IDF recently. I've also encountered crashes when using the official app for Unified Provisioning, which is used to connect the device to a WiFi.

Conclusion

The thesis first described the issues of IoT security in general and then moved to security features of the platform that may be used to prevent them. Known vulnerabilities of the ESP32 platform were described and countermeasures presented in the practical part. Main communication protocols and ways of device identification were analyzed. In the end, a proof-of-concept app has been created utilizing the described features, working around their limits, and even discovering two more issues in the TLS library. It communicates with Azure IoT Hub and authorizes using a key stored in ATECC608 HSM. The app has also implemented secure over the air update and secure wifi provisioning.

The learnings from the analysis and making the proof-of-concept app were summarized and discussed. To pick just a few: MQTT is an efficient choice for IoT, and secure if used over TLS, Secure Boot and Flash Encryption are vulnerable on ESP32, ESP-TLS library is not secure by default, HSM should be used for storing secret keys, and over the air update is easy to do and supports all the required features.

As can be seen, all the goals of the thesis were accomplished and a few more things were done on top of the original assignment.

Open Questions & Future Work

Some issues were mentioned in this thesis, but not addresses in-depth as they were out of its scope. One of these issues is privacy and how to make privacy-aware solutions based on ESP32. Another is the management of PKI infrastructure and how to handle certificate rotation/revocation in IoT infrastructure efficiently. The thesis also touched the area of security on constrained IoT devices, that may not have enough power to run TLS or maintain a reliable TCP connection. The next interesting area is how do we solve the issue of devices outliving the companies that made them, and therefore potentially being forever insecure. I believe all of these are interesting questions and could be areas of future study.

Bibliography

[1] OWASP Internet of Things Project - OWASP. https://wiki.owasp.org/index.php/OWASP_Internet_of_Things_Project#tab=Main, (Accessed on 08/02/2020).

[2] Grassi, P. A.; Fenton, J. L.; et al. NIST special publication 800-63b: digital identity guidelines. *Enrollment and Identity Proofing Requirements. url: https://pages. nist. gov/800-63-3/sp800-63a. html*, 2017.

[3] Antonakakis, M.; April, T.; et al. Understanding the mirai botnet. In *26th {USENIX} security symposium ({USENIX} Security 17)*, 2017, pp. 1093–1110.

[4] Neshenko, N.; Bou-Harb, E.; et al. Demystifying IoT security: an exhaustive survey on IoT vulnerabilities and a first empirical look on internet-scale IoT exploitations. *IEEE Communications Surveys & Tutorials*, volume 21, no. 3, 2019: pp. 2702–2733.

[5] Internet of Things Teddy Bear Leaked 2 Million Parent and Kids Message Recordings. https://www.vice.com/en_us/article/pgwean/internet-of-things-teddy-bear-leaked-2-million-parent-and-kids-message-recordings, (Accessed on 08/03/2020).

[6] Hunt, G.; Letey, G.; et al. The seven properties of highly secure devices. *tech. report MSR-TR-2017-16*, 2017.

[7] Secure by Design - GOV.UK. https://www.gov.uk/government/collections/secure-by-design, (Accessed on 08/02/2020).

[8] Vasserman, E. Y.; Hopper, N. Vampire attacks: Draining life from wireless ad hoc sensor networks. *IEEE transactions on mobile computing*, volume 12, no. 2, 2011: pp. 318–332.

[9] Trappe, W.; Howard, R.; et al. Low-energy security: Limits and opportunities in the internet of things. *IEEE Security & Privacy*, volume 13, no. 1, 2015: pp. 14–21.

[10] Smith, S. *The Internet of Risky Things: Trusting the devices that surround us.* "O'Reilly Media, Inc.", 2017.

[11] Who should bear the cost of IoT security: consumers or vendors? — Ubuntu. `https://ubuntu.com/blog/who-should-bear-the-cost-of-iot-security-consumers-or-vendors`, (Accessed on 08/03/2020).

[12] Zhao, S.; Zhang, Q.; et al. Providing root of trust for ARM TrustZone using on-chip SRAM. In *Proceedings of the 4th International Workshop on Trustworthy Embedded Devices*, 2014, pp. 25–36.

[13] Anchoring Arm TrustZone with SRAM PUF - TrustZone for Armv8-M blog - TrustZone for Armv8-M - Arm Community. `https://community.arm.com/developer/ip-products/processors/trustzone-for-armv8-m/b/blog/posts/anchoring-trustzone-with-sram-puf`, (Accessed on 07/15/2020).

[14] DS28E38 DeepCover® Secure ECDSA Authenticator with ChipDNA PUF Protection - Maxim Integrated. `https://www.maximintegrated.com/en/products/embedded-security/secure-authenticators/DS28E38.html`, (Accessed on 07/15/2020).

[15] Naik, N. Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP. In *2017 IEEE international systems engineering symposium (ISSE)*, IEEE, 2017, pp. 1–7.

[16] Dizdarević, J.; Carpio, F.; et al. A survey of communication protocols for internet of things and related challenges of fog and cloud computing integration. *ACM Computing Surveys (CSUR)*, volume 51, no. 6, 2019: pp. 1–29.

[17] IBM Podcasts - MQTT. `https://www.ibm.com/podcasts/software/websphere/connectivity/piper_diaz_nipper_mq_tt_11182011.pdf`, (Accessed on 07/05/2020).

[18] Eronen, P.; Tschofenig, H. Pre-Shared Key Ciphersuites for Transport Layer Security (TLS). RFC 4279, RFC Editor, December 2005, `http://www.rfc-editor.org/rfc/rfc4279.txt`. Available from: `http://www.rfc-editor.org/rfc/rfc4279.txt`

[19] MQTT Essentials - All Core Concepts explained. `https://www.hivemq.com/mqtt-essentials/`, (Accessed on 07/05/2020).

[20] TLS/SSL - MQTT Security Fundamentals. https://www.hivemq.com/blog/mqtt-security-fundamentals-tls-ssl/, (Accessed on 07/11/2020).

[21] X.509 client certificates - AWS IoT. https://docs.aws.amazon.com/iot/latest/developerguide/x509-client-certs.html, (Accessed on 07/11/2020).

[22] Understand Azure IoT Hub security — Microsoft Docs. https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-devguide-security#supported-x509-certificates, (Accessed on 07/11/2020).

[23] MQTT 5 Essentials. https://www.hivemq.com/mqtt-5/, (Accessed on 07/05/2020).

[24] Ian Craggs, D. O., Simon Johnson. MQTT-SN: MQTT for the Internet of Smaller Things, 2020, oASIS Open Standards - MQTT-SN Subcommittee. Available from: https://www.youtube.com/watch?v=Cvt7LoAXau0

[25] MQTT - AWS IoT. https://docs.aws.amazon.com/iot/latest/developerguide/mqtt.html, (Accessed on 07/05/2020).

[26] Understand Azure IoT Hub MQTT support — Microsoft Docs. https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-mqtt-support, (Accessed on 07/05/2020).

[27] Requirements — Cloud IoT Core Documentation — Google Cloud. https://cloud.google.com/iot/docs/requirements, (Accessed on 07/05/2020).

[28] GOTO 2017 • An Intro to IoT Protocols: MQTT, CoAP, HTTP & WebSockets • A. Almeida & J. Berciano - YouTube. https://www.youtube.com/watch?v=s6ZtfLmvQMU, (Accessed on 07/11/2020).

[29] Shelby, Z.; Hartke, K.; et al. The Constrained Application Protocol (CoAP). RFC 7252, RFC Editor, June 2014, http://www.rfc-editor.org/rfc/rfc7252.txt. Available from: http://www.rfc-editor.org/rfc/rfc7252.txt

[30] CoAP Video Tutorial - Internet of Things - Internet of Things - Arm Community. https://community.arm.com/iot/b/internet-of-things/posts/coap-video-tutorial, (Accessed on 07/11/2020).

[31] Wirges, J.; Dettmar, U. Performance of TCP and UDP over Narrowband Internet of Things (NB-IoT). In *2019 IEEE International Conference on Internet of Things and Intelligence System (IoTaIS)*, 2019, pp. 5–11.

[32] Sarafov, V. Comparison of iot data protocol overhead. In *Proceedings of the Seminars of Future Internet (FI) and Innovative Internet Technologies and Mobile Communication (IITM)*, volume 720, 2018.

[33] Thangavel, D.; Ma, X.; et al. Performance evaluation of MQTT and CoAP via a common middleware. In *2014 IEEE ninth international conference on intelligent sensors, sensor networks and information processing (ISSNIP)*, IEEE, 2014, pp. 1–6.

[34] Tandale, U.; Momin, B.; et al. An empirical study of application layer protocols for IoT. In *2017 International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS)*, IEEE, 2017, pp. 2447–2451.

[35] Salted Challenge Response Authentication Mechanism - Wikipedia. `https://en.wikipedia.org/wiki/Salted_Challenge_Response_Authentication_Mechanism`, (Accessed on 07/13/2020).

[36] QUIC, a multiplexed stream transport over UDP - The Chromium Projects. `https://www.chromium.org/quic`, (Accessed on 05/13/2020).

[37] TLS overhead - netsekure rng. `http://netsekure.org/2010/03/tls-overhead/`, (Accessed on 05/14/2020).

[38] Pricing - The 1NCE IoT Flat Rate explained — 1NCE - IoT SIM. `https://1nce.com/en/pricing/`, (Accessed on 05/14/2020).

[39] Kothmayr, T.; Schmitt, C.; et al. DTLS based security and two-way authentication for the Internet of Things. *Ad Hoc Networks*, volume 11, no. 8, 2013: pp. 2710–2723.

[40] TLS 1.3 is going to save us all, and why IoT is still insecure. `https://blog.cloudflare.com/why-iot-is-insecure/`, (Accessed on 05/14/2020).

[41] Introducing Zero Round Trip Time Resumption (0-RTT). `https://blog.cloudflare.com/introducing-0-rtt/`, (Accessed on 05/14/2020).

[42] David Brown, L. IoT TLS: Why It's Hard, 2018, open Source Summit + Embedded Linux Conference & OpenIoT Summit Europe 2018. Available from: `https://www.youtube.com/watch?v=C7snRkLbIWM`

[43] Researchers Exploit Low Entropy of IoT Devices to Break RSA Certificates - IEEE Spectrum. `https://spectrum.ieee.org/tech-talk/telecom/security/low-entropy-iot-internet-of-things-devices-security-news-rsa-encryption`, (Accessed on 05/13/2020).

[44] When to use Pre Shared Key (PSK) Cipher Suites - wolfSSL. https://www.wolfssl.com/when-to-use-pre-shared-key-psk-cipher-suites-2/, (Accessed on 07/05/2020).

[45] Georgiev, M.; Iyengar, S.; et al. The most dangerous code in the world: validating SSL certificates in non-browser software. In *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 38–49.

[46] Rescorla, E. HTTP Over TLS. RFC 2818, RFC Editor, May 2000, http://www.rfc-editor.org/rfc/rfc2818.txt. Available from: http://www.rfc-editor.org/rfc/rfc2818.txt

[47] Chokhani, S.; Ford, W. Internet X.509 Public Key Infrastructure Certificate Policy and Certification Practices Framework. RFC 2527, RFC Editor, March 1999, http://www.rfc-editor.org/rfc/rfc2527.txt. Available from: http://www.rfc-editor.org/rfc/rfc2527.txt

[48] ESP-IDF Programming Guide - ESP32 - — ESP-IDF Programming Guide documentation. https://docs.espressif.com/projects/esp-idf/en/latest/esp32/, (Accessed on 07/17/2020).

[49] Systems, E. ESP32 Technical Reference Manual V4.1. https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf, 2019, (Accessed on 05/11/2020).

[50] Fritzing Parts. https://fritzing.org/parts/, (Accessed on 05/25/2020).

[51] Commons, W. File:Espressif ESP32 Chip Function Block Diagram.svg — Wikimedia Commons the free media repository. 2020, [Online; accessed 24-May-2020]. Available from: https://commons.wikimedia.org/w/index.php?title=File:Espressif_ESP32_Chip_Function_Block_Diagram.svg&oldid=417508187

[52] Espressif Announces the Release of ESP32-S2 — Espressif Systems. https://www.espressif.com/en/news/espressif-announces-%E2%80%A8esp32-s2-secure-wi-fi-mcu, (Accessed on 05/24/2020).

[53] Securing the IoT: Part 2 - Secure boot as root of trust - Embedded.com. https://www.embedded.com/securing-the-iot-part-2-secure-boot-as-root-of-trust/, (Accessed on 07/16/2020).

[54] BitLocker (Windows 10) - Microsoft 365 Security — Microsoft Docs. https://docs.microsoft.com/en-us/windows/security/information-protection/bitlocker/bitlocker-overview, (Accessed on 05/25/2020).

[55] ESP32 Arduino: Random Number Generation – techtutorialsx. https://techtutorialsx.com/2017/12/22/esp32-arduino-random-number-generation/, (Accessed on 05/11/2020).

[56] RNG Quality (Dieharder Test Results) - ESP32 Forum. https://www.esp32.com/viewtopic.php?t=12622, (Accessed on 05/11/2020).

[57] Brown, R. G. Robert G. Brown's General Tools Page. http://webhome.phy.duke.edu/~rgb/General/dieharder.php, (Accessed on 05/11/2020).

[58] Miscellaneous System APIs - ESP32 - — ESP-IDF Programming Guide latest documentation. https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/system.html#_CPPv410esp_randomv, (Accessed on 05/11/2020).

[59] Updating firmware reliably - Embedded.com. https://www.embedded.com/updating-firmware-reliably/, (Accessed on 08/01/2020).

[60] Wikipedia contributors. SoftAP — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=SoftAP&oldid=954025876, 2020, [Online; accessed 11-May-2020].

[61] The Challenges of Soft AP: What Goes Wrong and Why. https://blog.cirrent.com/challenges-soft-ap, (Accessed on 05/11/2020).

[62] Unified Provisioning - ESP32 - — ESP-IDF Programming Guide latest documentation. https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/provisioning/provisioning.html, (Accessed on 05/11/2020).

[63] ESP-Touch Resources — Espressif Systems. https://www.espressif.com/en/products/software/esp-touch/resources, (Accessed on 05/11/2020).

[64] CC3100 Provisioning Smart Config - Texas Instruments Wiki. https://processors.wiki.ti.com/index.php/CC3100_Provisioning_Smart_Config, (Accessed on 05/11/2020).

[65] Li, C.; Cai, Q.; et al. Passwords in the air: Harvesting wi-fi credentials from smartcfg provisioning. In *Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, 2018, pp. 1–11.

[66] espressif/esp-aws-iot: AWS-IoT SDK as an ESP-IDF component. https://github.com/espressif/esp-aws-iot, (Accessed on 05/11/2020).

[67] aws/aws-iot-device-sdk-embedded-C: SDK for connecting to AWS IoT from a device using embedded C. https://github.com/aws/aws-iot-device-sdk-embedded-C, (Accessed on 05/11/2020).

[68] Azure/azure-iot-sdk-c: A C99 SDK for connecting devices to Microsoft Azure IoT services. https://github.com/Azure/azure-iot-sdk-c, (Accessed on 08/03/2020).

[69] ESP-TLS - ESP32 - — ESP-IDF Programming Guide latest documentation. https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/protocols/esp_tls.html#atecc608a-secure-element-with-esp-tls, (Accessed on 07/21/2020).

[70] wolfCrypt FIPS 140-2 Information — wolfSSL Embedded SSL/TLS Library. https://www.wolfssl.com/license/fips/, (Accessed on 05/13/2020).

[71] Comparison of TLS implementations - Wikipedia. https://en.wikipedia.org/wiki/Comparison_of_TLS_implementations, (Accessed on 05/13/2020).

[72] Pwn the ESP32 Forever: Flash Encryption and Sec. Boot Keys Extraction - LimitedResults. https://limitedresults.com/2019/11/pwn-the-esp32-forever-flash-encryption-and-sec-boot-keys-extraction/, (Accessed on 08/01/2020).

[73] CVE - CVE-2019-15894. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-15894, (Accessed on 08/01/2020).

[74] Pwn the ESP32 Secure Boot - LimitedResults. https://limitedresults.com/2019/09/pwn-the-esp32-secure-boot/, (Accessed on 08/01/2020).

[75] Security advisories about Zero PMK installation and beacon crash — Espressif Systems. https://www.espressif.com/en/Security_advisories_about_Zero_PMK_installation_and_beacon_crash, (Accessed on 08/01/2020).

[76] ESP32 Fault Injection Vulnerability - Impact Analysis — Espressif Systems. https://www.espressif.com/en/news/ESP32_FIA_Analysis?position=13&list=fZxdQ9z_aBYeVOwDFgYTMhhva5QUMshYyNCoG5dU4z0, (Accessed on 05/25/2020).

[77] ESP32-S2 — Security Features - The ESP Journal - Medium. https://medium.com/the-esp-journal/esp32-s2-security-improvements-5e5453f98590, (Accessed on 05/24/2020).

[78] Service concepts in Azure IoT Hub Device Provisioning Service — Microsoft Docs. https://docs.microsoft.com/en-us/azure/iot-dps/concepts-service, (Accessed on 07/26/2020).

[79] ATECC608A CryptoAuthentication Device Summary Data Sheet. http://ww1.microchip.com/downloads/en/DeviceDoc/ATECC608A-CryptoAuthentication-Device-Summary-Data-Sheet-DS40001977B.pdf, (Accessed on 07/30/2020).

[80] ATECC CryptoAuthentication Device Family - Compressed Certificate Definition. http://ww1.microchip.com/downloads/en/AppNotes/Atmel-8974-CryptoAuth-ATECC-Compressed-Certificate-Definition-ApplicationNote.pdf, (Accessed on 07/27/2020).

[81] azure-iot-sdk-c/using_custom_hsm.md at master · Azure/azure-iot-sdk-c. https://github.com/Azure/azure-iot-sdk-c/blob/master/provisioning_client/devdoc/using_custom_hsm.md#hsm-x509-api, (Accessed on 07/29/2020).

[82] Network Configuration — ESP-Jumpstart documentation. https://docs.espressif.com/projects/esp-jumpstart/en/latest/networkconfig.html#nvs-persistent-key-value-store, (Accessed on 07/31/2020).

[83] wolfSSL Espressif Support - wolfSSL. https://www.wolfssl.com/docs/espressif/, (Accessed on 07/21/2020).

[84] Any plans for TLS 1.3 support? · Issue #508 · ARMmbed/mbedtls. https://github.com/ARMmbed/mbedtls/issues/508#issuecomment-601258457, (Accessed on 07/21/2020).

[85] ESP-TLS - ESP32 - — ESP-IDF Programming Guide latest documentation. https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/protocols/esp_tls.html#atecc608a-secure-element-with-esp-tls, (Accessed on 07/29/2020).

[86] Bellare, M. New proofs for NMAC and HMAC: Security without collision-resistance. In *Annual International Cryptology Conference*, Springer, 2006, pp. 602–619.

[87] PolarSSL is not vulnerable to POODLE-against-TLS - Tech Updates - Mbed TLS (Previously PolarSSL). https://tls.mbed.org/tech-updates/blog/polarssl-not-vulnerable-to-poodle-against-tls, (Accessed on 07/31/2020).

[88] Grant limited access to data with shared access signatures (SAS) - Azure Storage — Microsoft Docs. https://docs.microsoft.com/

en-us/azure/storage/common/storage-sas-overview, (Accessed on 08/01/2020).

Appendix A

Acronyms

AES Advanced Encryption Standard

ALPN Application Layer Protocol Negotiation

AMQP Advanced Message Queuing Protocol

API Application Programmable Interface

AP Access Point

AWS Amazon Web Services

BLE Bluetooth Low Energy

BT Bluetooth

CA Certificate Authority

CBC Cipher Block Chaining

CN Common Name (in a certificate)

CRC Cyclic Redundancy Check

CVE Common Vulnerabilities and Exposures

CoAP Constrained Application Protocol

DDoS Distributed Denial of Service

DHE Diffie-Hellman Ephemeral

DMA Direct Memory Access

DNS Domain Name Server

DSA Digital Signature Algorithm

DTLS Datagram Transport Layer Security

ECC Elliptic Curve Cryptography

ECDHE Elliptic Curve Diffie-Hellman Ephemeral

ECDSA Elliptic Curve Digital Signature Algorithm

EEA European Economic Area

ESP-IDF Espressif IoT Development Platform

FIPS Federal Information Processing Standard

FQDM Fully Qualified Domain Name

FW Firmware

GPIO General Purpose Input/Output

HAL Hardware Abstraction Layer

HMAC Keyed-hash Message Authentication Code

HSM Hardware Security Module

IV Initial Vector (e.g. in AES)

IoT Internet of Things

LPWAN Low Power Wide Area Network

MAC Media Access Control Address

MCU Micro-controller

MMU Memory Management Unit

NIST National Institute of Standards and Technology

NVS Non-volatile Storage

OCSP Online Certificate Status Protocol

OTA Over the Air (e.g. update)

OWASP Open Web Application Security Project

PaaS Platform as a Service

PAL Platform Abstraction Layer

PCB Printed Circuit Board

PHY Physical layer

PKI Public Key Infrastructure

PRNG Pseudo Random Number Generator

PR Public Relations

PSK Pre-Shared Key

PUF Physically Unclonable Function

QoS Quality of Service

RAM Random Access Memory

RFC Request For Comments

RF Radio-Frequency

RNG Random Number Generator

ROM Read Only Memory

RTC Real Time Control

RTOS Real Time Operating System

RTT Round Trip Delay Time

SAN Subject Alternative Name (in a certificate)

SDK Software Development Kit

SIEM Security Information and Event Management

SMP Symmetrical Multiprocessing

SPI Serial Peripheral Interface

SSL Secure Socket Layer

SoC System on Chip

TLS Transport Layer Security

TRNG True Random Number Generator

ULP Ultra Low Power

URI Uniform Resource Identifier

VM Virtual Machine

XMPP Extensible Messaging and Presence Protocol

Appendix B

Allowed TLS Ciphersuites

- TLS-ECDHE-ECDSA-WITH-AES-256-CCM
- TLS-DHE-RSA-WITH-AES-256-CCM
- TLS-DHE-RSA-WITH-AES-256-CBC-SHA256
- TLS-ECDHE-ECDSA-WITH-AES-256-CBC-SHA
- TLS-ECDHE-RSA-WITH-AES-256-CBC-SHA
- TLS-DHE-RSA-WITH-AES-256-CBC-SHA
- TLS-ECDHE-ECDSA-WITH-AES-256-CCM-8
- TLS-DHE-RSA-WITH-AES-256-CCM-8
- TLS-ECDHE-ECDSA-WITH-AES-128-GCM-SHA256
- TLS-ECDHE-RSA-WITH-AES-128-GCM-SHA256
- TLS-DHE-RSA-WITH-AES-128-GCM-SHA256
- TLS-ECDHE-ECDSA-WITH-AES-128-CCM
- TLS-DHE-RSA-WITH-AES-128-CCM
- TLS-ECDHE-ECDSA-WITH-AES-128-CBC-SHA256
- TLS-ECDHE-RSA-WITH-AES-128-CBC-SHA256
- TLS-DHE-RSA-WITH-AES-128-CBC-SHA256
- TLS-ECDHE-ECDSA-WITH-AES-128-CBC-SHA
- TLS-ECDHE-RSA-WITH-AES-128-CBC-SHA
- TLS-DHE-RSA-WITH-AES-128-CBC-SHA

- TLS-ECDHE-ECDSA-WITH-AES-128-CCM-8
- TLS-DHE-RSA-WITH-AES-128-CCM-8

Appendix C

Contents of enclosed CD

- `thesis` ..... the directory containing this thesis
- `src` ..... the directory with the proof-of-concept app
  - `main` ..... the directory with source codes of the app
  - `components` .... the directory with forked components used by the app
    - `esp-cryptoauthlib` ..... ATECC 608 component
      - `esp_cryptoauth_utility` ..... provisioning utility and firmware
- `utils` ..... the directory with utilities mentioned in the practical part