Preprints.org

Article

Not peer-reviewed version

Secure Local Communication Between Browser Clients and Resource-Constrained Embedded IoT Devices

Christian Schwinne * and Jan Pelzl

Posted Date: 19 December 2025

doi: 10.20944/preprints202510.1808.v2

Keywords: browser; inter-window messaging; JavaScript; network security; IoT

Preprints.org is a free multidisciplinary platform providing preprint service that is dedicated to making early versions of research outputs permanently available and citable. Preprints posted at Preprints.org appear in Web of Science, Crossref, Google Scholar, Scilit, Europe PMC.

Copyright: This open access article is published under a Creative Commons CC BY 4.0 license, which permit the free download, distribution, and reuse, provided that the author and preprint are cited in any reuse.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 December 2025 doi:10.20944/preprints202510.1808.v2

Disclaimer/Publisher's Note: The statements, opinions, and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions, or products referred to in the content.

Article

Secure Local Communication Between Browser Clients and Resource-Constrained Embedded IoT Devices

Christian Schwinne * and Jan Pelzl

Hamm-Lippstadt University of Applied Sciences
* Correspondence: christian.schwinne@hshl.de

Abstract

This contribution outlines a completely new, fully local approach for secure web-based device control based on browser inter-window messaging. Modern smart home IoT (Internet of Things) devices are commonly controlled with proprietary mobile applications via remote servers, which can have significant adverse implications for the end user. Given that many IoT devices in use today are limited both in available memory and processing speed, standard approaches such as HTTPS-based transport security are not always feasible and a need for more suitable alternatives for such constrained devices arises. This local method for lightweight and secure web-based device control using inter-window messaging leverages existing standard web technologies to enable a maximum degree of privacy, choice and sustainability within the smart home ecosystem. The implemented proof-of-concept shows that it is feasible to meet essential security objectives in a local web IoT control context while utilizing less than a kilobyte of additional memory compared to an unsecured solution; thus promoting sustainability through hardening the control protocols used by existing devices with too little resources for implementing standard web cryptography. Therefore this work contributes to achieving the vision of a fully open and secure local smart home.

Keywords: browser; Inter-window messaging; JavaScript; network security; IoT

---

1. Introduction

In the last few years, Internet of Things (IoT) has been gaining significant traction and proliferation especially in domestic environments. Marketed as "Smart Home" systems, IoT platforms aim to automate the operation of common household appliances and make controlling them more convenient. IoT products targeted at the consumer market are typically connected via a remote cloud-hosted server, commonly run by the device manufacturer. These online services offer limited interfaces for the end user to control their devices - often no more than a manufacturer-provided proprietary smartphone app, sometimes accompanied by voice control services. While this implementation - at least initially - satisfies the use cases of some customers, a multitude of users wish to control their devices fully locally within their own home network, without the need to connect to an external server hosted on the public Internet. However, most commercially available IoT devices do not offer such an application programming interface (API) to facilitate local control, instead running a closed-source device firmware that makes control possible solely via the aforementioned manufacturer cloud service and associated apps.

There are a multitude of reasons why a user may prefer to control their devices locally, of which some of the most prominent are outlined below:

Firstly, all **security** considerations are constrained to the network to which the device is connected and its immediate physical vicinity. While refraining from connecting to any external server alone does not adequately protect a device, it significantly reduces the number of possible attack vectors. Even if a manufacturer takes care to implement state-of-the-art security measures, regular updates might not be provided down the line, or they may not be installed by users. Given that device firmware, the

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 December 2025 doi:10.20944/preprints202510.1808.v2

control software used on cloud servers and end-user apps are all commonly proprietary, it becomes impossible to independently ascertain that the application contains no critically exploitable bugs or even deliberate backdoors. In contrast, open source software allows the user to verify and - if desired - modify the codebase. This allows for independent review, thus it is no longer a necessity to put unconditional trust in the manufacturer - both in their ability to produce safe, bug-free code and in their benevolence - to allow their device to connect to one's internal network. However, even in the suboptimal case that device firmware is closed source, the existence of a feasible local control mechanism makes it possible to block the device from connecting to the Internet, which significantly limits the possible attack surface to exploit the device or even gain unauthorized access to the internal network itself.

Privacy is another common concern. Remotely controlled IoT devices usually require registering an account with the manufacturer, necessitating the disclosure of personally identifiable data, commonly at least an e-mail address. This puts the user at risk in the event the IoT control server is compromised. Particularly, their e-mail address might be exposed to third parties, which could lead to spam and phishing attacks. Ensuring an appropriate level of privacy protection is particularly essential with devices capable of recording and monitoring highly personal living spaces in a home, such as security cameras and voice assistants.

Furthermore, if storage in the server's database is not implemented according to best practices, the user's credentials - most commonly a password - might be trivial to obtain, a particular concern in case users re-use the same password on multiple services.

Another reason to prefer local control is **latency**. Sending a network packet within one's local network is typically accomplished within less than 50 ms, which is perceived as instant to the user. Contrast this with a remote manufacturer server that may be in another country, where sending the same packet may take about half a second. The user-perceived responsiveness of the device is severely impacted, as the minimum perceivable latency is only 200 ms at most (cmp. [1] p. 32).

Availability is another major concern. If the IoT control server is unreachable or the user's internet connection is disrupted, control of the devices is not possible.

Another essential factor is **longevity**. After an IoT product has reached the end of its commercial availability, the manufacturer might opt to disable servers or support for that device. This could also occur due to manufacturer bankruptcy. If the remote IoT control server ceases to function, the device is not controllable and depending on its design, severely limited in its functionality or rendered entirely inoperable. An example for this are home security cameras and systems by Nest Secure, which have been disabled in April 2024[2]. Such shutdowns are not only costly for the user, since they have to replace the device, but it is also unsustainable for the environment to dispose of a device in physically perfect working order merely because the means of controlling it no longer function.

Flexibility is another advantage. Proprietary solutions often require the user to install a mobile app for each vendor, which consumes limited mobile device resources and leads to a disjunct system composed of a multitude of different apps and accounts a user requires on their mobile phone just to control their devices from different vendors. With a sufficiently open local API, advanced automation via software like Home Assistant or other custom integrations becomes a straightforward possibility. Such options are typically severely limited with proprietary servers. This often results in vendor lock, where a user is unable to automate a process despite owning hardware that has the technical capability to do so. As a simple example, consider a motion detector and a light. If only one of these two devices uses a proprietary app and there is no way to interface with it over a common API or compatibility adapter, even a simple automation use case such as turning on the light if motion is detected can not be implemented using these components. This example highlights the importance of open - and preferably local - APIs, as they are a base prerequisite for effective home automation. Without such basic provisions for **interoperability**, deeming IoT devices "smart" may be considered an overstatement.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 December 2025
doi:10.20944/preprints202510.1808.v2

Remote access is one of the key reasons manufacturers often opt for preferring remote server control over local control. It is a selling point to be able to e.g., turn on the heat at home before one leaves work, or check that all the lights are off after leaving the house. However, remote and local control APIs are not mutually exclusive, so a decision to not offer a local API might be due to manufactures desiring a higher degree of control or reducing implementation complexity by foregoing the need for e.g., an embedded webserver on the device. Philips Hue is a positive example for a large scale IoT ecosystem that offers both remote and local APIs.

Ownership of hardware should empower users to use the device as they see fit. The possibility for local control is essential to a non-revocable freedom of use.

1.1. Methods to Proliferate Local Control

In 2024, the *Open Home Foundation* was created to facilitate the goals of **privacy**, **choice** and **sustainability** (cmp. [3]) with the vision of enabling a truly open and local smart home system. The WLED project, which will be used for the main example implementations in this work, is also a member of the Open Home Foundation.

Open source software is a key element in making IoT devices more accessible - including local control. When a device runs an open source firmware, users are free to customize its software behavior to their specific needs, which can include adding APIs to enable local control.

Consumer behavior is essential for a long-term improvement of implemented IoT solutions. If customers are sensitized to the issue and prefer purchasing devices that allow local control or are even fully open source, this is likely to influence manufacturer behavior to offer more flexible means to use their devices.

Web technology and in particular, JavaScript APIs present in modern browsers has evolved rapidly in the last few years, to the point where it is often feasible to replace proprietary native apps with open web-based apps that offer cross-platform compatibility with almost all Internet-connected end user devices that have a standard web browser installed. This could be expanded further in the future to offer built-in non-TLS means for browsers to communicate securely with low resource locally connected IoT devices.

The goal of this research is to identify and analyze methods for offering a secure local means of control for constrained IoT devices, with particular focus on browser-based web apps, i.e., without requiring the installation of custom software on client control devices. As an implementation example, the ESP8266 and ESP32 microcontrollers by Espressif are considered due to their widespread use both in home-made and commercial IoT devices. For demonstration, the open source lighting control project WLED is used - see 2.9.

In the following, firstly the technical background is given in 2. Furthermore, the novel approach method for secure and lightweight browser-based device control is outlined in 3.

2. Technical Background

Herein, fundamental concepts used for the subsequent implementation are outlined and the necessary prerequisites established. These include both the security objectives that the solution aims to achieve and possible approaches to implementing cryptographic protection measures in a web browser context.

2.1. System and Threat Model

It is useful to first establish an overview over the local IoT control system and the security objectives the implemented solution shall guarantee.

A direct user to device IoT control system by definition involves two core participants: The user and the IoT device. As the user uses a standard web browser, they assume the client role, while the device acts as a HTTP server. We define a third participant, a trusted server to download JavaScript code to be executed securely in the user browser. The reason this server is required for our approach is outlined in more detail in 2.3. There are two dedicated external communication channels: Firstly,

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 December 2025 doi:10.20944/preprints202510.1808.v2

an insecure HTTP and WebSocket connection between user and IoT device, and secondly, a trusted TLS connection between the user and trusted external server. Additionally, there is a third notable communication channel internal to the user browser enabling message exchange between the two browser windows - one with the page served by the trusted server and the second served insecurely by the IoT device server - see 2.5. This channel is also regarded as insecure, as it can be viewed as an extension to the external HTTP channel with the IoT device and is thus indirectly subject to similar security constraints. User and IoT device are in control of a Pre-Shared Key (PSK) that is established via a secure out-of-band method - e.g., a hardware serial bus. Lifecycle management of the PSK is considered out-of-scope.

We assume an active network attacker in the channel between the user and IoT device. The attacker has full knowledge of the system design and the ability to intercept, block, read, modify, replay, and send arbitrary messages, i.e., aligns with the adversary as specified in the Dolev-Yao model[4] (cmp. [5], sec. 1.3) and is thus able to mount Man-in-the-Middle (MitM) attacks. By extension, the attacker has the same capabilities for the internal browser inter-window message channel. The attacker is unable to break cryptographic primitives and has no knowledge of the user password or the derived PSK and is thus unable to forge valid message authentication codes for a chosen plaintext. Furthermore, the attacker is assumed to be incapable of interfering with the TLS connection between user and trusted external server in any way. While a denial-of-service attack would be possible, it is regarded out-of-scope in this threat model. On the user side, the attacker has no direct control over code downloaded to or executed in the user browser, with the crucial exception of JavaScript downloaded directly from the IoT device. In particular, the attacker is assumed to be incapable of modifying the user browser and device hardware or software, adding certificates to the user truststore, exploiting hardware access to the user device, or using social engineering to influence user behavior. On the device side, similarly, the attacker does not have hardware access to the IoT device; the attack surface is limited to the HTTP control network. The initial PSK agreement between user and device occurs securely out-of-band.

Regarding the security goals (cmp. [6] pp. 303-304) for low-risk smart devices that do not handle personal data, such as lighting equipment, keeping control commands secret may not be required; thus Confidentiality protection is not unconditionally required. However, the proposed approach must be able to protect against illegitimate or modified control commands being accepted by the IoT device; therefore Integrity and Authenticity protection is regarded essential. Furthermore, mitigation against replay attacks is crucial to prevent an attacker from re-sending previously captured legitimate control messages at a later time. Availability is also important, though there are only limited measures that can be taken against both attacks that intercept and discard messages as well as denial-of-service (DoS) attacks. While some mitigations, such as VPN tunnels, redundant channels, and firewalls, are compatible with our approach, they complement it and are not considered in detail in this approach. Non-repudiation is also not critical in a typical end-user IoT environment, omitting this requirement enables use of lightweight, purely symmetric mechanisms based on a pre-shared key. In conclusion, for this IoT application scenario, only Authenticity and Integrity with replay attack mitigation are considered as fundamental security goals.

A visual overview of the system model is given in Figure 1. Note that the functionality "Control UI" is listed in both tabs - in practice, user interface elements may be defined in either tab. In the PoC, the entire UI is part of the insecure tab hosted by the device. In order to fully mitigate injection of attacker-chosen control commands through the UI, the UI code must be hosted by the trusted external server and part of the secure crypto tab - see 5.3.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 December 2025 doi:10.20944/preprints202510.1808.v2

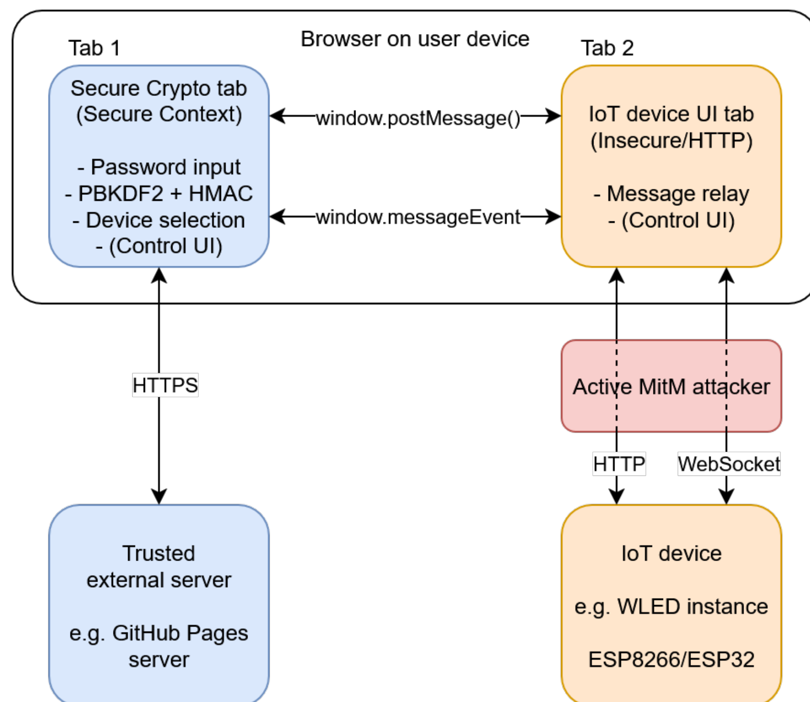

Figure 1. Architectural overview of the system model

2.2. Web-Based Apps as an Alternative to Native Apps and Browser-Based Cryptography

In order to successfully design a secure local IoT system controlled exclusively via a web app running in a standard browser, it is first necessary to gather an overview of the cryptographic functionalities present in modern browsers and their associated limitations.

The primary means of control for most commercially available IoT devices for home use are native smartphone applications. While major mobile operating system software development kits (SDKs) offer a high degree of flexibility for the application developer in the interfaces and device features they can utilize, they can have drawbacks for the end user. Since the source code is typically not accessible, users have to trust manufacturers and distributors not to abuse the power granted to them. Both modern operating systems and browsers implement a permission system to let the user decide which device functionality and data the app or website is allowed to access. A key problem with this system is that it is often not clear to the user why and in which scope particular permissions are required for the app to function; the choice given to them is often not granular enough. A specific example pertinent to IoT control apps is location access. Many Wi-Fi-based devices open an access point (AP) for initial setup. The app then scans for and connects to this AP to provision the device with the credentials for the user's home Wi-Fi network. In recent versions of Android, the location permission is required to connect to Wi-Fi networks within the app, as network SSIDs could theoretically be referenced against publicly available SSID maps such as WiGLE and thus a likely location of the user inferred. Therefore, granting the location permission is often required for initial device setup. However, this allows the app to store the location of the user during setup or even to continually track them, which constitutes data collection that for most categories of smart home devices (such as lights, air conditioning, and cameras) is not strictly necessary and thus not only fails to adequately protect the users privacy, but also risks non-compliance with laws enacted to protect individual privacy, such as the General Data Protection Regulation (GDPR) in the EU.

If control of an IoT device from user end devices is desired without requiring additional apps, web browsers can be a viable choice. Virtually all computers and smartphones and sometimes even more specialized devices such as e-book readers have modern browsers pre-installed. CSS offers a rich toolkit to create visually appealing user interfaces and JavaScript offers the majority of features needed to implement client-side logic - to the point that many natively installable apps are

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 December 2025 doi:10.20944/preprints202510.1808.v2

merely wrappers around a web page. This approach is used by the WLED project as well as other widely used applications such as the chat platform *Discord*. An even more lightweight alternative are progressive web apps (PWA), which appear like native apps to the user - for instance by having a dedicated app icon, running full-screen without visible browser UI, and being able to run offline using caching[7]. Additionally, the client-side code of a website or PWA is in most cases open source by definition, as JavaScript is an interpreted language that is executed directly from the corresponding source code. While code obfuscation and minification - that is, removing formatting and comments and shortening variable and function names - are possible and often advisable to reduce the size of the page when transferred over the network, such minified JavaScript can still be analyzed and customized more readily than an assembled and packaged binary application. Another feature built into modern browsers that is highly useful for secure local IoT control is the *Web Crypto subtle* API, which implements select cryptographic primitives for in-browser use. However, particularly with devices that do not support transport encryption, one first has to carefully consider whether cryptography code downloaded from the device's embedded web server and executed in the browser can be trusted.

2.3. Secure JavaScript Cryptography Code from Untrusted Origins

There is a theory of the so-called "Browser Cryptography Chicken and Egg Problem"[8], which states that "*if you can't trust the server [or message transport] with your secrets, then how can you trust the server [or message transport] to serve secure crypto code?*"[9]. Essentially, in a server and client architecture without TLS, the web page is sent to the client unencrypted. In that case, there is no easy way for one to implement client side cryptographic mechanisms without TLS communication, as all page JavaScript comes from the server and could thus be subject to a MitM attack. Therefore it is inherently untrustworthy and any trust in JavaScript code run on the client side would first need to be established using either built-in browser functionality, or external HTTPS servers.

Using external servers for download or verification of certain client-side code is less than ideal, as part of the availability and privacy concerns of full remote device control still apply. However it is still deemed preferable to remote control, as no user accounts are required and caching - such as possible when deployed as a PWA - may allow the solution to continue working in temporary offline conditions. Moreover, users could host such a server themselves on more performant hardware.

2.4. Secure Contexts in Browsers

An important consideration in browser-based cryptography are Secure Contexts. A multitude of powerful browser APIs are only available in Secure Contexts, i.e., when the page is loaded via TLS. This is meant to protect personal data, as access to potentially sensitive APIs, for example camera, location, or microphone, is denied. One notable API that is also only available in a Secure Context is the aforementioned Web Crypto subtle API, which provides cryptographic primitives and useful functions, for instance password-based key derivation (PBKDF2) and implementations of various hash functions. The Subtle Crypto API is only available in Secure Contexts likely due to the fact that, as outlined above, any client-side JavaScript transferred over HTTP is inherently insecure - at least without additional integrity checking of the downloaded JavaScript via trusted in-browser code. The *Subresource Integrity* feature of modern browsers supports checking the integrity of additionally downloaded JavaScript files against a hash provided in the `integrity` attribute of the `<script>` tag to be loaded. However, to the authors knowledge, such a built-in integrity check mechanism is not natively available for the root HTML page itself, therefore a trusted root page served over TLS is still required.

Furthermore, mixed content prohibition complicates interconnection between TLS and non-TLS servers: unsecured sites may fetch additional content from TLS servers, but the inverse is not true - browsers will in general block all unsecured HTTP requests from pages loaded via HTTPS. In the context of a non-TLS capable locally controlled IoT device, approaches such as loading the control interface HTML, stylesheets, and JavaScript from a TLS-enabled external server and only establishing

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 December 2025 doi:10.20944/preprints202510.1808.v2

an HTTP connection with the device for some API control commands are therefore not trivially possible.

2.5. Inter-Window Messaging

Even though a direct HTTP or unencrypted WebSocket connection to the device from a Secure Context is disallowed by modern browsers, indirect communication is possible under certain conditions. Two different browser windows that have a mutual reference can communicate with each other using the *window.postMessage()* API to send a message and the *window.messageEvent* API to listen for incoming messages.

Note that *window* refers to a independent page context here, it does not need to be a different browser window as seen by a user, but can also be e.g., a second tab. The reference to the other window can be made by e.g., loading the second window in an <*iframe*> element embedded in the first window (though unencrypted *iframe* content is also disallowed on secure pages) and also by opening the second window in a new tab programmatically, which makes a reference to the original window available via *window.opener*, thereby allowing the opened window to send an initial message. The window message API is even available if the two windows have different origins - which often means they are hosted on different servers - and, for the JavaScript cryptography use case crucially, when one window is a Secure Context and the other is not. This allows developers a controlled circumvention of enforced Secure Context constraints, as they can pass a message to the Secure Context window, let it process the message in a trusted environment using an API that is only available in a Secure Context, like Web Crypto, and then return the result to the untrusted window via messaging.

2.6. TLS-Less Client Authentication Mechanisms

For a server to be able to discern authorized from unauthorized users, i.e., clients, an authorized user is required to present proof of their authorization to the server. Using TLS, this proof can be a client certificate, however this is not an option without TLS. Most commonly, passwords are used for authentication, though they require both transport and storage protection measures against eavesdropping and subsequent misuse.

2.7. Deriving a Shared Key from the User Password

Most servers that employ transport layer security (TLS) rely on it to transmit the cleartext password and then carry out hashing operations within the server itself. This raises the question why password hashing is not more commonly done on the client side, so that the cleartext password is at no time exposed to the server. One likely explanation may trace back to the "Browser Cryptography Chicken and Egg Problem" in the stance that if the server can not be trusted to handle the password, it can also not be trusted to serve secure cryptographic code for client-side password hashing. Another reason may be to reduce complexity or the reliance on JavaScript on the client side, or to increase performance when combining a slow client device and an expensive password hashing algorithm.

In an IoT device context where TLS is not available, client-side password hashing becomes indispensable in order to avoid exposing the cleartext password during the unencrypted transmission between the browser and IoT device.

An option for password hashing built-in to browsers is *HTTP Digest access authentication*. It allows for client side password hashing and subsequently avoiding sending the password in the clear without any use of JavaScript, however unfortunately it is susceptible to MitM attacks ([10], sec. 5.8). Since the connection is unencrypted, the attacker can downgrade the authentication scheme to use *HTTP Basic access authentication*, which transmits the password in cleartext with a simple Base64 encoding. This manipulation is not readily detectable to the end user since the HTTP authentication dialog is rendered identically regardless whether Basic or Digest authentication is in use. Furthermore, the appearance is not customizable by the page, but is always rendered as a generic popup login dialog as shown in Figure 2.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 December 2025 doi:10.20944/preprints202510.1808.v2

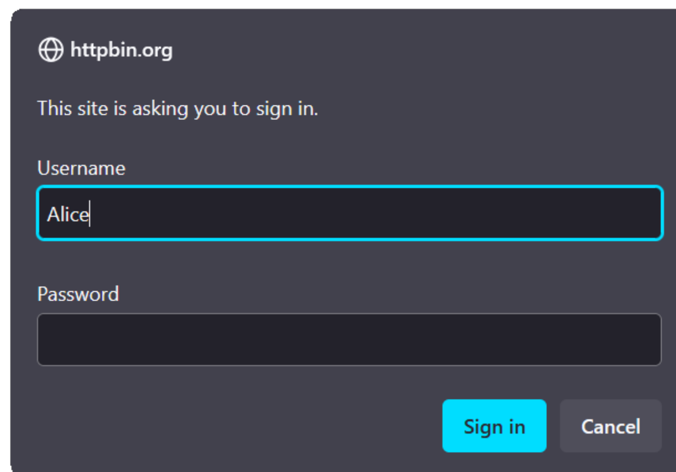

Figure 2. Generic HTTP authentication popup dialog in the Firefox browser

On a page sent via unencrypted HTTP, the JavaScript *Web Crypto* API is not available as the window is not in a secure context, therefore neither the built-in password based key derivation (PBKDF2) can be used directly, nor can one implement their own algorithm directly in JavaScript or download it from an external server in a secure manner due to the "Browser Cryptography Chicken and Egg Problem".

2.8. Non-TLS Data Transport Security Mechanisms

In order to implement confidentiality, integrity, and authenticity protected message transport, full TLS is not unconditionally required. In particular, the asymmetric key exchange steps and required buffers drive up implementation complexity and memory requirements. In cases where use of a pre-shared key (PSK) is acceptable, asymmetric cryptography is no longer required and in principle, symmetric ciphers can be used directly. Some symmetric cipher suites that may be suitable in the context of IoT devices and JavaScript client implementations include the Advanced Encryption Standard (AES) and ChaCha20, which "is considerably faster than AES in software-only implementations, making it around three times as fast on platforms that lack specialized AES hardware" ([11], p. 3), a property that is very useful on constrained devices with low processing power and without hardware support for AES. *Ascon* is a noteworthy family of symmetric ciphers designed to be particularly lightweight and standardized by the US National Institute of Standards and Technology (NIST) as "Lightweight Cryptography Standards for Constrained Devices"[12].

In applications where confidentiality is not required, data encryption can be omitted entirely, as for integrity and authenticity protection a cryptographically secure message authentication code (MAC) is sufficient. This increases performance compared to data encryption, as calculating a cryptographic digest (hash) of the data and authenticating it is typically significantly faster than encrypting the entire message.

A widely used MAC is HMAC. The integrity of a message can be asserted by the sender calculating $HMAC(msg, psk)$ and including the resulting hash in the transmission. The receiver, which also possesses the PSK, may now repeat this calculation, if the calculated hash matches the one included by the sender, it is valid and the receiver can be assured that the message has not been tampered with and originated from someone who knows the PSK.

2.9. Introduction to WLED

WLED (backronym: Wireless Lighting Effects Driver) is an open-source software project conceived and maintained by the author C. Schwinne since 2016, designed for the ESP8266 and ESP32 microcontroller series by Espressif Systems. It is engineered to drive a large quantity of digital individually addressable full-color LEDs; each LED may be set to a different color and brightness via a serial protocol and thus enables an abundance of different lighting effects. WLED has over a

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 December 2025
doi:10.20944/preprints202510.1808.v2

hundred built-in effect modes - from simple blinking, flickering candles and twinkling fairy lights up to complex visualizations intended to be displayed on a two-dimensional matrix of LEDs. WLED offers an easy-to-use and feature complete web based control user interface, additionally native mobile applications and smart home automation system integrations are available.

The primary interface for controlling a WLED instance from external hosts such as home automation software, but also from the included web-based user interface, is the WLED JSON API, which is publicly documented in detail (cmp. [13]). For the scope of this work, it is important to note that the JSON API is available both via an HTTP endpoint (`/json`) and via a WebSocket connection. WebSocket is a protocol for bi-directional messaging communication between an HTTP server and client based on a persistent TCP connection. It can thus be used to replace resource-heavy and high latency HTTP polling for receiving status updates from the server. The syntax of a WLED API command is a simple JSON object containing the keys to be set. For instance, the following command turns on the light and set it to approximately half brightness (`bri` is an 8-bit value denoting the current overall brightness, which has a range of 0-255):

```
{
    "on": true, "bri": 128
}
```

3. New Approach Methodology

This section describes how the components outlined in 2 are combined to facilitate secure and lightweight control of devices via standard web browsers.

3.1. Hosting of Browser-Based Secure Cryptography

Due to the "Browser Cryptography Chicken and Egg Problem" outlined in 2.3, doing most cryptographic operations or even just accepting input of a user password is fundamentally insecure on pages that have been loaded via HTTP only. Thus, to avoid this problem, facilitate secure message authentication, and ensure the user password is not transmitted over an insecure connection, a trusted execution environment is required. This could for instance be a page locally stored on the user device - although loading pages from downloaded HTML/JS source files is typically not implemented in browser apps for mobile devices and is also inconvenient for users. Alternatively, it could be hosted via HTTPS on a domain the user trusts (e.g., `https://rc.wled.me`, or alternatively the user could host their own server running this page for cryptographic functionality, denoted as the "secure crypto tab" (SCT) in the following.

A key goal of the system to be implemented is handling all cryptographic operations within the browser client, which has two advantages: Firstly, apart from initial WLED ESP provisioning, where the PBKDF2-derived pre-shared key needs to be stored on the ESP for HMAC verification, no secrets have to leave the user device. Secondly, a static web page is sufficient for hosting, which greatly simplifies deployment and allows hosting on e.g., GitHub Pages or an out-of-the-box Apache server without advanced configuration as would be necessary for a full server-side framework such as Django.

For the secure crypto tab to be able to connect to the WLED instance despite the browser's mixed content prohibition, inter-window messaging is used as explained in section 2.5. The user is unable to control the light by directly visiting the address of the WLED instance, as depicted in Figure 3. Instead, as the first step, they need to open the SCT and can subsequently open the instance user interface through it by first entering their password and clicking the "Connect" button, as shown in Figure 4. The abbreviated code for the initial window messaging handshake is given below:

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 December 2025 doi:10.20944/preprints202510.1808.v2

wled00/data/index.js (WLED UI, ll. 250+):

```
function onLoad() {
    //...
    if (window.opener) {
  window.opener.postMessage(
            '{"wled-ui":"onload"}', '*');
        );
}
    //...
}
```

WLED-SecureRemoteAccess/src/messages.ts (secure crypto tab):

```
export function handleMessageEvent(event : MessageEvent) {
    //... origin verification, error handling
    var json = JSON.parse(event.data)
    if (json['wled-ui'] === 'onload') {
        event.source!.postMessage(
            '{"wled-rc":"ready"}',
            {'targetOrigin':event.origin}
        );
    }
    // handling other message types
}
```

wled00/data/index.js (WLED UI, ll. 315+):

```
function handleWindowMessageEvent(event) {
    //... origin verification, JSON parsing
    if (json['wled-rc'] === 'ready') {
        useSRA = true;
        sraWindow = event.source;
        sraOrigin = event.origin;
    }
}
```

In the first UI code block, note the second parameter `'*'` in the call to `postMessage`. This sends the message to any opener window, regardless of its origin, which is necessary since browsers do not allow access to `window.opener.origin` for cross-origin privacy reasons. This is not ideal, since with no initial way of verifying the opener origin, a malicious site could hypothetically link to the control UI and eavesdrop on the control commands sent. However, since the HTTP-based transport does not offer confidentiality protection in the first place, this is a largely theoretical concern and could also be prevented by only allowing the *rc.wled.me* origin here, at the cost of preventing users from hosting the secure crypto tab themselves on a different origin. Once the reply to the handshake message by the SCT is received, `event.origin` is accessible though and can be used for custom filtering rules.

3.2. Cross-Origin Resources and Security Header Profile

For window messaging to be possible, at least one of the two involved windows needs to retain a reference to the other window. This is either the value returned from `window.open()` in the SCT or `window.opener` in the UI tab. As the UI tab and SCT are different origins, the Same-Origin policy already restricts the possible operations on the window reference. Not only is there no access to

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 December 2025 doi:10.20944/preprints202510.1808.v2

`window.opener.origin` as outlined above, but direct access to the DOM (Document Object Model) or JS is also inhibited - thereby ensuring the UI tab and potential MitM attacker can only post messages to the secure tab, but not modify it or access sensitive JS variables in any other way. Still, it would be desirable to make this relation unidirectional, so that only the SCT can send the UI tab messages by default. This is however not supported, as the measures to inhibit (cross-origin) opener references break the relation in both ways. This is achieved by e.g., setting the `noopener` feature on `window.open()` or setting the `Cross-Origin-Opener-Policy` (COOP) header on the SCT to `same-origin`, which causes the opened UI window to be opened in a new Browsing Context Group (BCG), which has no references to the opener. Therefore, to enable `postMessage`, the SCT must necessarily open the UI tab in the same BCG. Still, the COOP header on the SCT can be set to `same-origin-allow-popups`, which ensures that the SCT can open the UI window in the same BCG using `window.open()`, as the UI window is using a COOP of `unsafe-none`. At the same time, it safeguards the SCT itself from being opened by other - potentially malicious - origins in the same BCG[14]. For additional defense-in-depth, it is advisable to set additional security headers to take advantage of modern browser features to limit the attack surface for cross-site scripting (XSS) and other types of cross-origin attacks such as UI redress. It is important to note that headers on the UI page can be modified by an MitM adversary - therefore their value can not be relied on and should be regarded only as a supplemental measure. Headers sent by the server for the SCT are protected against manipulation through TLS, but can not always be set at will when using static hosters such as GitHub Pages. For the PoC, the Security headers in Table 1 are used.

Table 1. Security-releated HTTP headers on the SCT.

Header	Value	Effect
Content-Security-Policy[1]	`frame-ancestors 'none'`	No window can embed this SCT window; protects against some forms of UI redress.
Cross-Origin-Embedder-Policy	`require-corp`	Cross-origin requests are blocked by default and only allowed under certain conditions[15].
Cross-Origin-Opener-Policy	`same-origin-allow-popups`	Ensure other windows opening the SCT will do so in a new BCG. Still allows opening the UI tab in the same BCG.
Cross-Origin-Resource-Policy	`same-origin`	Ensure other origins cannot load resources from the SCT.
X-Content-Type-Options	`nosniff`	Blocks CSS and JS requests where the Content-Type header is set incorrectly.

[1] A more strict CSP is advisable in a production deployment to further restrict loading of all cross-origin resources unless explicitly permitted.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 December 2025 doi:10.20944/preprints202510.1808.v2

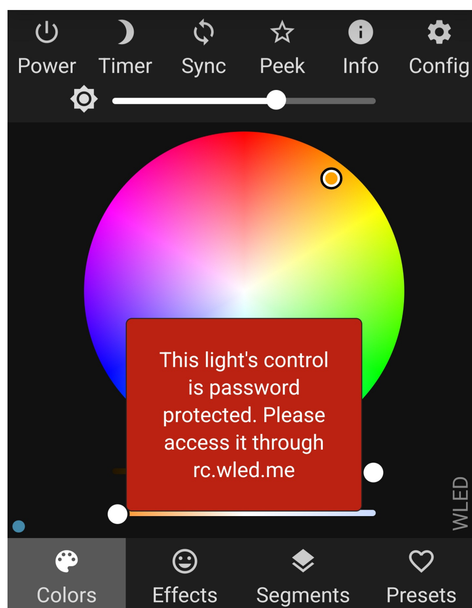

Figure 3. WLED control UI with error message asking the user to open the secure crypto tab first. It occurs when attempting to control lights directly in the UI without having opened it through the SCT.

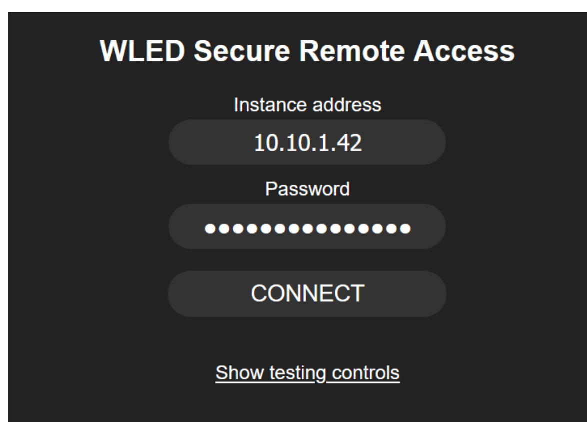

Figure 4. Secure crypto tab user interface.

3.3. Technology Stack

The ESP-side code, which expands the functionality of WLED by HMAC verification capabilities, is written in C++. The cryptographic primitive implementations for *SHA256* and *HMAC* are provided by the *ESP8266 Crypto* library, though the code is expected to be fully portable except for the ESP-specific random number generator implementation. It also provides an *AES* encryption implementation that may be useful in case the implementation is to be modified to offer confidentiality protection. Being contained in a single source file *Crypto.cpp* that is below 1.000 lines of code, it is easy to independently analyze.

The WLED web UI functionality is implemented in a single JavaScript file, *index.js*.

For the secure crypto tab, which implements all in-browser cryptographic operations, a TypeScript project is used that relies on modules to improve code structure. JavaScript modules allow for more efficient implementation of object-oriented programming concepts by for instance requiring all public functions of a module to be explicitly exported for use in other modules. Other functions are private by default. TypeScript is a superset of JavaScript that adds static typing and thus enables type checking. The *Vite* build tool is used to transpile the modules back to JavaScript that can be executed in standard browsers and merge them into a single file again. The user-facing appearance of the SCT is shown in Figure 4.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 December 2025 doi:10.20944/preprints202510.1808.v2

3.4. PBKDF2-Based PSK Derived from User Password

In order to avoid a potential unencrypted transmission of cleartext passwords between the browser and ESP, only a hash of the password is ever transmitted outside of the SCT. The *PBKDF2* algorithm is chosen for the PoC, as it is the only algorithm available in the Web Crypto API that is suitable for deriving keys from low-entropy passwords (cmp. [16]). Newer hashing algorithms designed to be used for passwords, for example *Argon2id*, would be preferable as it is more resistant to GPU (graphics processing unit) and ASIC (application specific integrated circuit) supported attacks by requiring the utilization of a large amount of system memory (cmp. [17], sec. 4). As the KDF only needs to run on the user browser side, there is no requirement to implement the KDF on the IoT device itself, since it can be provisioned directly with the derived key. Therefore, substitution of PBKDF2 with an algorithm such as *Argon2id*, that is hardened against GPU and ASIC attackers, is possible by employing a suitable and well-vetted implementation that can run within the browser, preferably using WebAssembly (WASM) for increased performance (cmp. [18] for an existing candidate implementation). For the PoC implementation of *PBKDF2* with the *SHA256* hash function, a work factor of 1.000.000 iterations is initially chosen. This offers some buffer over the 2023 OWASP recommendation[19] of 600.000 iterations. The Web Crypto implementation is subsequently tested for performance in 4.3, as the key derivation should be intentionally slow to mitigate attacks, but fast enough to not be a hindrance to the end user for a single derivation - ideally the key derivation time should stay within a period of time the user perceives as quick. The *Doherty Threshold*[20] of 400 *ms* could be considered a reasonable benchmark[21].

3.5. Generation of the MAC

The Web Crypto API is also used for generation of the HMAC message authentication code. The TypeScript function utilized for HMAC generation is given below - note that for brevity and readability, some non-critical lines are omitted:

WLED-SecureRemoteAccess/src/crypto.ts (secure crypto tab):

```
async function generateHMAC
    (message: string, key: CryptoKey) : Promise<string>
{
    const messageBuffer = new TextEncoder().encode(message);
    const sig = await crypto.subtle.sign('HMAC', key, messageBuffer);

    return Array.from(new Uint8Array(sig)).map(function(byte) {
        return ('0' + (byte & 0xFF).toString(16)).slice(-2);
    }).join('');
}
```

This function is a good example for how the Web Crypto API can be used. Calling the primitives is usually accomplished by a single line, though some setup code is needed, particularly to set parameters correctly and to convert formats, as the Web Crypto API only operates on byte buffers (*Uint8Array* in TypeScript). The last three lines of code in the function merely convert the returned HMAC to a hexadecimal string for sending it to the ESP over the network.

3.6. Transport of the HMAC

After authenticating a JSON message, the HMAC is to be sent along the message in order for the receiver to be able to verify that the sender is in possession of the pre-shared key by calculating the HMAC of the message itself and comparing it to that sent along with the message. There are multiple ways to implement such data transfer with an associated authentication code, most relavant for the application in WLED are JSON payloads either over HTTP or WebSocket. In a pure HTTP environment where JSON API commands are transmitted via a HTTP POST request, a straightforward

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 December 2025 doi:10.20944/preprints202510.1808.v2

implementation method would include the HMAC as an HTTP header, separate from the JSON payload sent in the POST request. However, this approach is not possible when using a WebSocket connection, as this is just a persistent TCP connection allowing for bi-directional messaging. Thus, the HMAC needs to be integrated into the message itself. For this purpose, the WLED JSON API calls are be wrapped, for this the following JSON syntax is chosen:

```
{
    "mac": "baddecafc0ffee[...]",
    "msg": {
        "on": true,
        "n": {"sid":"5e55101d[...]","c":41}
    }
}
```

Here, the `msg` object (`{"on":true}`) is the original API command, in this case to turn on the lights. An additional key `n` is added, this is a session ID and counter-based nonce used to prevent replay attacks. This is wrapped in another JSON object that also contains a string value `mac` containing the HMAC of the API command encoded as a hexadecimal string.

3.7. Verification of the HMAC

The verification of the HMAC itself - without nonce validation for replay attack mitigation - is simply implemented by re-calculating the HMAC of the message data and comparing it to the MAC sent along with the message.

3.8. Nonce Implementation

As mentioned above in 3.6, the nonce consists of a session ID and a counter that is incremented for every message sent. For the **session ID**, a 128 bit (16 bytes) length random value is chosen to balance the collision risk with the associated memory use and transmission length. The session ID is sent as a hexadecimal-encoded string, a single byte is represented by a set of two characters (e.g., `FF` for decimal 255), so the entire session ID is represented as a 32-character long hex string. A suitable source of entropy is required for secure random number generation, the ESP series of boards has a special register for obtaining hardware-seeded random numbers ([22], sec. 25). The **counter** is incremented by the client SCT on every message sent. The connection between the control UI tab and the WLED instance is always established either via WebSocket or HTTP. As both protocols utilize TCP for packet transport - which guarantees in-order delivery of packets, keeping track of a sliding window of allowed counter values is not required; it is sufficient for the ESP to reject any value less than or equal to the last received counter value for that session ID. For the counter, a 32-bit length is chosen, as this theoretically allows for over 4 billion messages to be authenticated before a new session ID must be used to prevent the unsigned counter from overflowing and wrapping back to 0. The ESP keeps track of allowable nonces by storing a configurable number of session ID / counter pairs. Each pair uses 20 bytes of memory (16 bytes for the session ID and 4 bytes for the counter); the array is sorted by which session IDs were most recently used in order to enable always replacing the oldest session ID with a new one if required and the array is already full. Generally, it is advisable to replace the oldest session ID. However, if multiple session IDs are requested without ever being successfully used in a MAC-authenticated message, it may be preferable to replace them first. This could mitigate against a potential "session ID exhaustion" denial of service attack, where new sessions are continually started, invalidating session IDs still in use by legitimate clients. It is crucial that the nonce is included *within* the HMAC-authenticated message; if it was sent alongside the message, an attacker could tamper with the nonce without invalidating the MAC.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 December 2025 doi:10.20944/preprints202510.1808.v2

3.9. Communication Flow

The implemented system consists of three codebase components running independently: the secure crypto tab, the user interface tab, and the WLED firmware running on the IoT device.

There are two core interactions between the components - the initial login authentication and subsequent authentication of control commands. Both core interactions are visualized below; Figure 5 shows the initial login process, while Figure 6 outlines the control commands authentication process. The "pipe" symbol in the UI tab denotes relay functionality - the UI tab does not process the message itself, but just forwards it from the SCT to the ESP or vice versa, changing the message transport protocol from window messaging to WebSocket or back from WebSocket to window messaging, respectively.

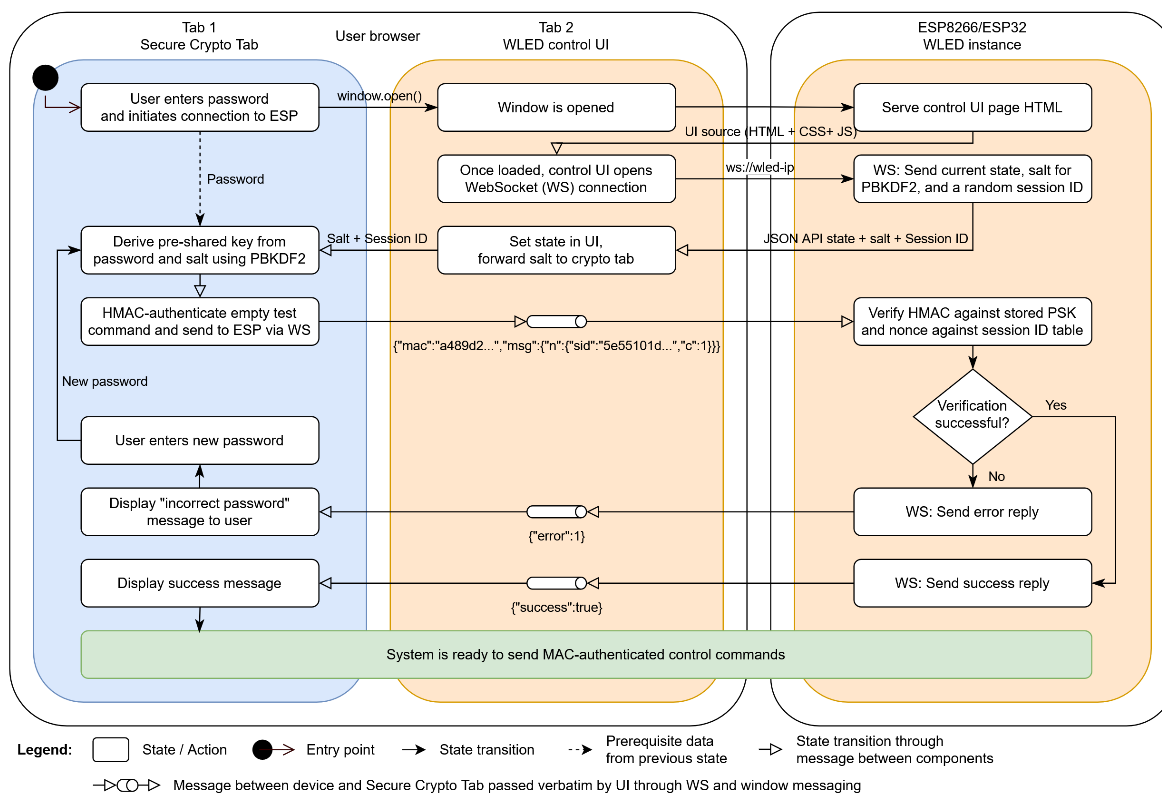

Figure 5. Initialization of the HMAC-based secure API access system

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 December 2025
doi:10.20944/preprints202510.1808.v2

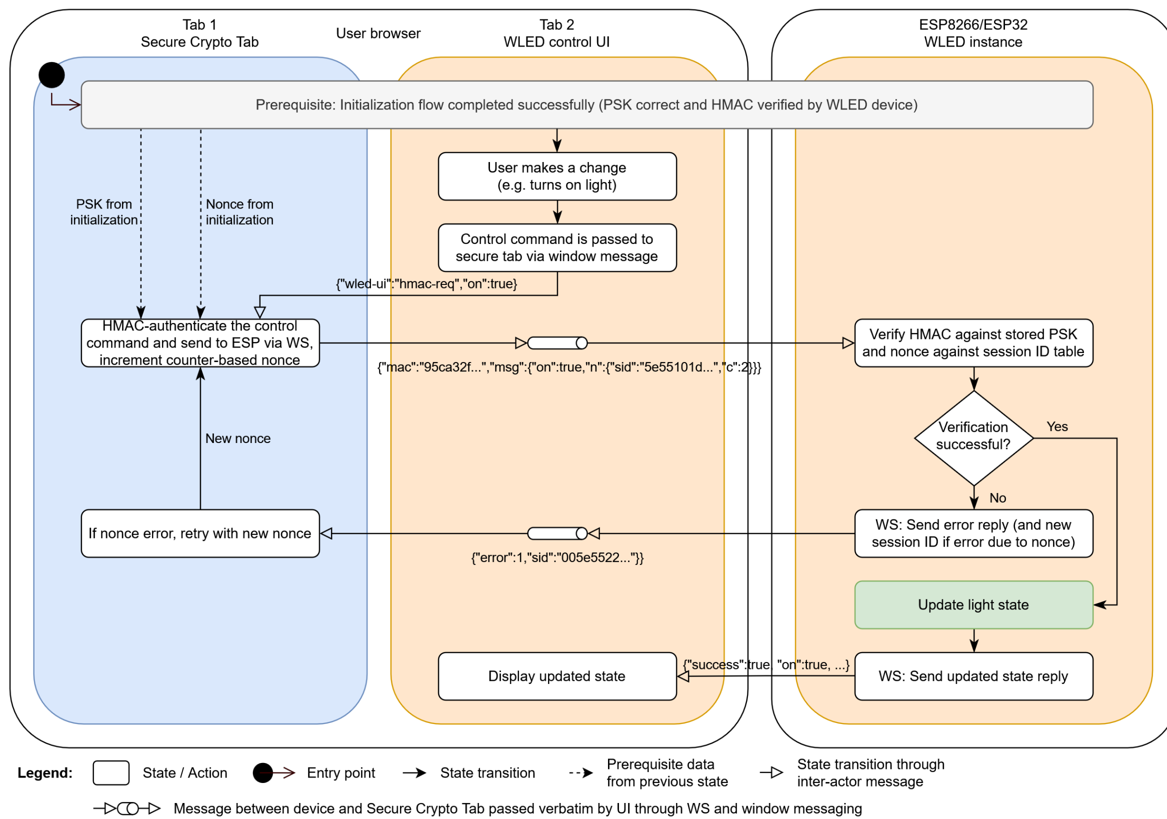

Figure 6. Use of the HMAC-based secure API access system to authenticate a control command

3.10. Design Choices Based on Threat Model

Table 2 justifies the design choices that have an impact on system security against an explicit threat that they aim to mitigate with regards to the threat model defined in 2.1. Concrete, testable security objectives are also provided.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 December 2025 doi:10.20944/preprints202510.1808.v2

Table 2. Mapping of identified explicit threats to implemented design choices and associated testable security objectives for verification.

Explicit Threat	Design Choice	Testable Objective
IoT device accepts forged control messages by an active attacker.	Authenticate messages with Message authentication codes using a symmetric PSK.	Assert IoT device rejects any forged message.
Attacker can replay captured legitimate control commands and thus repeat existing commands at will.	Each control message is unique and only accepted once by the device using a session ID and counter-based nonce.	Ensure IoT device rejects all replayed or duplicate messages - especially in multi-client stress test scenarios.
Attacker can observe response timing differences to obtain a valid hash for a chosen message over multiple requests.	Use constant-time comparison between expected and provided HMACs on IoT device.	Ensure that HMAC generation and comparison times run in constant time for given message and hash lengths.
Attacker can brute-force small key spaces.	PSK is of sufficient length.	Prove it would statistically take at least several years of computation to find the correct key.
Attacker can use dictionary attacks to selectively try commonly used values.	PSK is of high entropy - user passwords are run through a password-based KDF.	Ensure PSK length is greater or equal to current best practices.
Attacker can compute PSKs associated with common passwords at a rapid rate, enabling dictionary attacks.	PBKDF algorithm and difficulty are set appropriately.	Assert that KDF is sufficiently slow even using GPUs or ASICs, but performance for a single derivation is still adequate on typical hardware. (0.5 s to 1 s range)
Attacker can manipulate device UI tab to send arbitrary control commands that are accepted and authenticated by the secure tab.	The inter-window messaging channel is only used to relay data to and from the IoT device; it cannot be used by the attacker to inject control commands to be authenticated by the secure tab.	Ensure no control commands are accepted by the secure tab via inter-window messaging[1].

[1] Crucially, this is not yet considered in the PoC, see 5.3.

4. Results

The implemented PoC successfully uses browser inter-window messages to facilitate communication between a trusted client environment (Secure Context) and an IoT client device connected via insecure HTTP.

4.1. Resource Utilization

For a security solution for resource-constrained embedded devices to be viable in practice, it does not only need to be able to operate within the memory and computing resources available, but also leave enough resources available for the application logic to run. Especially in the case of more complex use cases - such as WLED hosting a full web server and lighting effects engine designed to drive hundreds of individual LEDs on the ESP8266 platform with just 80 kB of available user data memory - keeping the memory usage to a minimum is a crucial requirement. The additional memory usage of the HMAC functionality implemented on the ESP-side using the SHA256 hash based on the *ESP8266 Crypto* library is determined by comparing both the static and dynamic (heap) memory usage of a build B_{Demo} - of the revision with the commit hash `f3429a6c` that has the demo implementation

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 December 2025 doi:10.20944/preprints202510.1808.v2

integrated - with build B_{Base} of the base WLED software at the point of the commit with hash `6dc2c680`, on which the implemented demo was based. The ESP8266 is considered, as in contrast to the ESP32 platforms - due to their significantly higher amount of available memory - optimization is essential. Both the static RAM usage as well as the flash utilization, which is equal to the compiled binary size of the firmware, are considered. As the static RAM usage merely includes static variables allocated at compile time, but not dynamic data allocated at runtime (e.g., with `malloc` or the C++ `new` keyword), in addition the free heap memory after a reboot of the microcontoller is considered, as it denotes the available memory after persistent dynamic variables have been allocated, for example the buffer holding the color of each addressed LED.

It can be observed that static RAM use only increased by 280 bytes over build B_{Base} by implementing HMAC-SHA256 in build B_{Demo}. The free heap memory decreased by approximately 600 bytes, though this value is subject to fluctuation due to factors like network packet caching. The compiled binary increases in size by 14 kB.

4.2. Replay Attack Mitigation

The nonce-based replay attack mitigation outlined in 3.8 is stress-tested successfully under various scenarios, which are outlined in Table 3.

Table 3. Nonce tests and outcome.

Test Case	Outcome
A legitimate control message is replayed with identical session key and counter value.	Message is rejected by the device.
A legitimate control message is replayed with identical session key and counter value after IoT device reboot.	Message is rejected by the device. The counter is reset to 0, but the message is still not valid as the list of allowed session IDs is also fully cleared.
Two legitimate control messages with the same session ID and counter values of e.g., 6 and 7 respectively, are sent out of order - the message with counter 7 is sent first.	The first message with the higher counter value 7 is accepted, the second message with lower counter value is rejected.
Two separate clients connect to the device and both send legitimate control messages in arbitrary order.	All legitimate control messages are accepted, as the device sends each client a separate session ID, thus their counters are independent from each other.
Attacker captures a valid session ID sent to a client and sends an illegitimate control message to the device using an allowed session ID (but incorrect PSK).	Message is rejected by the device. The validity of the nonce for the legitimate client is not impacted, as device-side nonce validation and counter update occur only after the MAC itself is validated.

4.3. In-Browser Key Derivation Benchmark

The performance of the Web Crypto Subtle implementation of *PBKDF2* was tested to ensure it is adequately slow to increase the cost of brute-force guessing attacks, but still fast enough to run on low-end mobile devices. A minimum iteration count of 600.000 is desirable as outlined in 3.4. The required time for the *PBKDF2* algorithm on a spectrum of different devices can be compared in Table 4; ten runs were performed per configuration, the resulting mean runtime taken and standard deviation are given. All tests were done with identical parameters, a 23 byte key, 8 byte salt and 1.000.000 iterations. It can be seen that while the time taken for a single password attempt is significant, i.e., $>= 100\,ms$, modern devices stay within the 400 *ms Doherty Threshold* set as the target in 3.4 and thus should be perceived as quick by the user. Even though it can be argued that a more than decade-old smartphone from 2013 or an e-book reader is not a suitable device to run in-browser password key derivation on, they also successfully complete 1.000.000 iterations of *PBKDF2* within less than 1.5 *s*,

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 December 2025 doi:10.20944/preprints202510.1808.v2

an acceptable waiting time considering the overall user-perceived speed of these devices. Therefore, 1.000.000 iterations appear to be a suitable difficulty for the types of devices a user is likely to use to access their IoT devices.

Table 4. Time taken for in-browser PBKDF2 with 1.000.000 iterations.

Device Type and Browser	Mean PBKDF2 runtime in *ms*	Std. Deviation in *ms*
Laptop[1] - Firefox	136.4	3
Laptop[1] - Edge	100.2	1.7
Phone S23 Ultra - Firefox	154.3	7.6
Phone S21 FE - Chrome	148.6	8.4
Phone Galaxy Note 3 - Chrome	1291	44.2
E-Book Reader Kindle Scribe - Silk	1199.4	51

[1] System specifications: Ryzen 7 7840HS CPU, Windows 11, 32GB DDR5 memory at 6400 MT/s, no active background programs

4.4. HMAC Generation Benchmark

The speed of the HMAC generation is also essential for keeping server response time low and therefore lead to satisfactory user perceived performance (cmp. 1). Therefore, the performance of the `hmacSign` method is evaluated on the target platform. To reduce variability in runtime readings due to e.g., network and LED update interrupts, the measurements are taken prior to initialization of WiFi connectivity and all other startup code except the serial console. Moreover, the time taken for 100 consecutive iterations is measured to reduce measurement interval quantization errors. The measurements are visualized in Figure 7 for two exemplary message lengths; 16 *bytes*, resembling a typical short control message like `{"on":true}` for turning on the device, and 2048 *bytes*, which may be considered a reasonable upper estimate of the length of a control message. All measured times are within approximately 3 *ms* per iteration, which is fast enough to not have an impact on the perceceived network response performance.

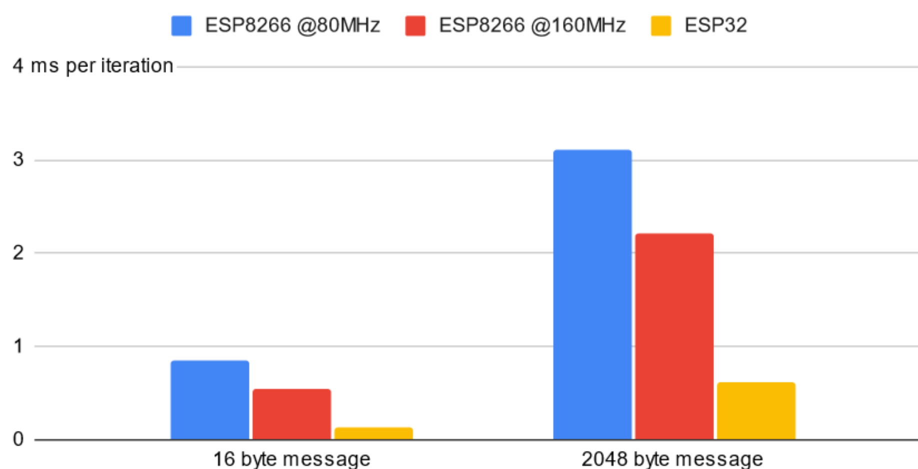

Figure 7. HMAC calculation times for 16- and 2048-byte long messages on different platforms.

4.5. Latency Benchmark

As already established in 3.4, keeping round-trip latency low is essential for user perceived responsiveness. Therefore, we have measured the round-trip time (RTT) from the control message being sent from by the UI to the resulting feedback state message from the device. The results of the latency tests are shown in Table 5. For each test, ten measurements are taken. Both the client device with the SCT and UI tab as well as the ESP IoT device were connected over wireless network. The first two ping tests give an insight on the latency that is solely due to the network - about 10 *ms*. The

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

third test does not measure device RTT, but instead the RTT of the initial control command being sent from the UI tab to the SCT and the SCT returning the HMAC-authenticated control message, which only adds 2 *ms* of delay. The next two tests 4 and 5 are a baseline measurement of the latency in the B_{Base} build without any transport security. The light is toggled via the UI power button and the approximate RTT is measured in the page JS using `performance.now()`. Tests 6 and 7 are for the same action on ESP8266 and ESP32 respectively, but including the message authentication in the B_{Demo} build. It can be seen that both platforms stay below 150 *ms* RTT, with the ESP32 being about 10 *ms* faster in the base and 20 *ms* faster in the demo scenario. The final test 8 does not involve the proposed approach or even the WLED software, instead it measures the RTT in the `Websocket-Chat` example of the `esp32_https_server` library, which hosts a full TLS server on the ESP32. The chat example is a trivial application of WebSockets - the same underlying protocol the WLED JSON API uses - and is therefore suitable for testing RTT. It can be seen that even with the complex application logic of WLED, JSON parsing, HMAC and nonce verification, and response building, the B_{Demo} latency, at about 118 *ms*, is lower than this minimal TLS example at 167 *ms*; thus, the latency improvement possible by using our approach over TLS (even where it is theoretically feasible) is significant.

Table 5. Experimental round-trip latency measurements.

Latency test	Mean RTT in *ms*	Std. Deviation in *ms*
1: ESP8266 Ping	7.4	3.7
2: ESP32 Ping	9.5	3.8
3: SCT HMAC gen. and window messages only	2.1	0.6
4: ESP8266 Toggle light (B_{Base})	38.4	6.6
5: ESP32 Toggle light (B_{Base})	27.8	5.6
6: ESP8266 Toggle light with auth. (B_{Demo}, device RTT only)	138.5	35
7: ESP32 Toggle light with auth. (B_{Demo}, device RTT only)	118.2	17.8
8: ESP32 TLS WS Chat example	166.7	27.6

5. Discussion

In the following, several aspects of the implementation are reflected upon and analyzed in terms of their advised usability in production systems and their long-term safety against quantum computing algorithms.

5.1. Secure Production Deployment of the Implemented Solution

While the architecture drafted and implemented in the above was continuously scrutinized on a best faith effort to be secure, it should strictly be considered a demo implementation for research and testing purposes only. The authors do not guarantee the security of the implemented solution and it should only be used in practical applications once thoroughly and independently verified , e.g., through penetration testing and carrying out formal analysis of the protocol message exchanges. Although only known and well-vetted cryptographic primitives are implemented, custom cryptographic system designs are easily prone to subtle errors that render the entire system ineffective - one example for an aspect that is not immediately obvious but essential for the security of the design is the nonce-based replay attack mitigation for HMAC-based message authentication schemes. Additional defense-in-depth measures, such as schema validation of incoming messages and stricter headers than used in the PoC, should also be implemented.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

5.2. Post-Quantum Cryptography Considerations

There are currently no indications that state-of-the-art symmetric ciphers, such as AES-256, or MAC-based approaches, such as implemented in this work, would be meaningfully reduced in their security value by quantum-based approaches; they are therefore deemed quantum-resistant (cmp. [23], p. 11). Thus, given the current state of research, unlike for instance current TLS implementations, the security of the implemented solution is likely to remain unaffected by the availability of powerful quantum computing resources, which is particularly important with embedded systems such as used in IoT devices, as they are typically in use for significantly longer periods of time than conventional IT hardware.

5.3. Potential for MitM Command Injections

While the implemented solution demonstrates that implementing hash-based security measures is highly feasible on constrained devices with limited resources such as the ESP8266, there are some use cases and limitations not addressed by the current implementation yet.

While the implemented system already significantly raises the transport security value and makes it more difficult for unauthorized parties to send valid unauthorized commands, there is still a significant risk for MitM attacks allowing for MAC verification of unauthorized commands - which again has its root in the "Browser Cryptography Chicken and Egg Problem". Although the implemented system delegates the relevant cryptographic operations to the SCT, all commands to be authenticated still originate from the untrusted control UI that is loaded from the ESP web server via insecure HTTP. This would allow the attacker to insert arbitrary control commands into the UI page source that could automatically be authenticated by the HMAC generation in the SCT and pass the ESP-side HMAC verification as regular MAC-authenticated commands.

A possible way to address this problem is hosting the entire control UI in the SCT, which would ensure no MitM command injection would be possible. The page loaded from the ESP web server over the insecure HTTP connection would be reduced to acting as a gateway that receives the window message from the SCT and passing it to the ESP via the WebSocket connection. This approach has one drawback, namely the SCT must have a version of the UI available that is API-compatible with the WLED version installed on the ESP. While this may be straightforward for standard builds, it would once again make it difficult for developers of custom features to have their version of the UI used by the SCT. This problem could be in part alleviated by allowing the ESP to specify a URL where a compatible version of the UI is hosted. While this approach requires retrieving the UI code via that URL and incurs the associated limitations regarding availability of online services, this URL could be part of the ESP-side firmware and also be protected against MitM URL replacement attacks using an HMAC generated on the ESP side.

5.4. Comparison to Alternative Approaches

It would be highly useful to effectively compare key characteristics of the proposed approach with other lightweight secure communication protocols for IoT, particularly with regards to resource utilization and asserted security goals to be able to select an architecture that is most suitable for a given application. Therefore, additional comparative benchmarks should be conducted in future work, not only against a baseline build with no security provisions as done in 4.1, but also against a full TLS server implementation - e.g., *mbedTLS* on an *ESP32*. We included an initial indicative latency comparison in 4.5. For the ESP8266, no practically usable TLS server implementation was found to be available, which is to be expected due to its severely limited memory. Furthermore, comparisons with more lightweight IoT protocols, based e.g., on lightweight symmetric protocols such as Ascon[12] would be highly interesting, even if they are not directly applicable to a direct browser-to-device control scenario.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 December 2025 doi:10.20944/preprints202510.1808.v2

5.5. A Case for TLS-PSK Browser Support

A standardized alternative to the implemented approach could be *TLS-PSK* or the functionally similar *TLS-SRP* (Secure Remote Password), which allow for a TLS connection without requiring asymmetric key agreement or an X.509 public key infrastructure by relying on a pre-shared key, similar to our approach. This makes *TLS-PSK* potentially more suitable for private networks and constrained IoT devices than standard *TLS*, as only symmetric primitives such as *AES* are required and limited IoT devices are a primary use case[24]. However, standard web browsers unfortunately do not support *TLS-PSK*; even if they did, its primary use case would likely be session resumption for connections that are initially established using public key cryptography[25], with no provision to use it as the exclusive mutual authentication scheme. Were browsers to support *TLS-PSK* with external PSKs established out-of-band, we would expect the feasibility of a very lightweight TLS server implementation to greatly increase on low-end IoT devices such as the ESP8266.

5.6. Browser Evolution and Possible Direction

Modern web browsers are continually evolving to improve the security of the web, protect user privacy, and mitigate newly discovered vulnerabilities[26]. In that process, legacy APIs or those deemed insecure can be restricted or removed. Notably, mixed content - i.e., accessing an HTTP resource from an HTTPS page - was allowed by default in all major browsers up until 2013 (Firefox), 2016 (Safari), and 2019 (Chromium)[27]. It is possible that the new approach presented herein could also be inhibited in future browser versions, for example by always opening mixed content windows in a new BCG, thereby removing the window reference needed to post messages to opened windows. The authors propose careful exemptions to the mixed-content prohibition with strict guardrails to enable a more seamless user experience without requiring our dual-tab approach. Such guardrails could include, but are not limited to: 1) Insecure content cannot be directly embedded into a HTTPS page; instead, the insecure response can only be read by JS in escaped text or binary form. 2) Insecure requests must be explicitly enabled for each request with a flag called e.g., `unsafe-mixed`. We believe this would be a welcome addition to facilitate interoperability between local and legacy devices and the wider, secured web.

5.7. Multi-Device and Multi-User Handling

A practical usability concern arises when implementing a system based on the proposed approach for actual daily use in a smart home environment: In a typical household, A) users are likely to own more than one device of a given category - especially in lighting - and B) multiple users may wish to control the device(s)[28]. Both these use cases should be supported in a frictionless manner in a practical IoT application. The implemented PoC does not yet adequately cover these use cases - the SCT only allows establishing a connection to a single device at a time; thus it is necessary to open two browser tabs for each device to be controlled. Furthermore, only a single password and associated PSK are considered in the PoC - this makes use in a multi-user environment infeasible without password sharing, which is a fundamentally flawed practice from a security perspective. However, the implemented PoC can be easily augmented to mitigate both of these multi-user and multi-device usability constraints while keeping the same underlying approach to lightweight security. For proper multi-user support, the IoT device could be configured to store a map of multiple user IDs and associated PSKs; users could then enter a username or ID in addition to the password in the SCT, which is sent within the authenticated message JSON and subsequently used by the device to select the corresponding PSK for HMAC verification. Multi-device support is easily enhanced by adding the capability for a single SCT to be shared by and interface with N devices, thereby reducing the amount of overall required tabs from $2 * N$ to $N + 1$. If the control UI is moved to the SCT to mitigate the possible command injection attack outlined in section 5.3, the user could dynamically select which device(s) to control via a list or dropdown interface. This approach reduces the tabs the user needs to

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

actively interact with from $2 * N$ (two tabs for each device, its SCT for authentication and the UI tab for control) to just a single secure crypto tab, allowing for a streamlined user experience.

5.8. Credential Lifecycle Management

Our approach is primarily intended for operational remote control of IoT devices that are part of a shared or public and thus untrustworthy network. It defines no methods for the initial key agreement or subsequent PSK rotations - aka. the user changing their password. It is assumed that the user occasionally has access to a secure out-of-band channel, for example a physical device bus. Implementing credential management as part of our approach would necessarily require transport confidentiality protection, thus encryption is required and a purely MAC-based approach becomes infeasible. One aspect of lifecycle management that can trivially be accommodated into the current approach is credential revocation - if the user suspects their PSK has been compromised, they can send a special authenticated control message that invalidates the PSK on the device so it is no longer accepted for authenticating additional messages.

6. Conclusions

In the course of this research, alternative methods for securing the control command communication between the user browser and IoT device were identified, aiming to be more **lightweight** than a TLS server implementation. It was found that approaches involving purely symmetric ciphers with a pre-shared key can be feasible depending on the requirements of the specific application. Furthermore, message authentication codes are a viable option in cases where message integrity and authenticity protection are required, but confidentiality protection is not needed, which greatly reduces the amount of necessary computation and memory resources that would be needed for encipherment and decryption; and thus is an excellent candidate for implementation on constrained devices that do not handle sensitive data. As the WLED software meets these criteria, an HMAC-based command message authentication scheme was successfully realized in a proof-of-concept demonstrator implementation that uses particularly few resources - less than a kilobyte of additional RAM usage was observed for the HMAC functionality. Moreover, novel methods to securely use cryptographic functionality in client browsers - even when the connection to the IoT device itself is untrusted - were designed. This results in an overall more user-friendly implementation compared to a direct TLS server.

Conclusively, despite undebatable hurdles faced by system designers when designing local control schemes that are to both interoperate fully with standard browsers and be functional on resource constrained devices, the work demonstrated that a practically usable and secure implementation of a lightweight web-based local IoT control system is definitely feasible. Once further refined and more widely deployed, a similar solution to that implemented herein has the potential to elevate the IoT ecosystem towards a vision that enables complete privacy, choice and sustainability for device owners.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: All source code created for the demo implementation is open source and freely available on GitHub. The code for the secure crypto tab is available at: https://github.com/Aircoookie/WLED-SecureRemoteAccess The secure crypto tab demo is also hosted at https://rc.wled.me for direct use. The WLED branch implementing the demo is available at: https://github.com/Aircoookie/WLED/tree/secure-api This branch will be maintained for a considerable length of time. If it is ever removed, the implementation may be found using the commit reference: https://github.com/Aircoookie/WLED/commit/f3429a6c9355 The *ESP8266 Crypto* library is available at: https://github.com/Aircoookie/arduino-crypto Note that this library is the work of Chris Ellis et al. and has merely been forked by the authors to integrate some minor bug fixes. The *esp32_https_server* library is available at: https://github.com/fhessel/esp32_https_server This library is

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 December 2025 doi:10.20944/preprints202510.1808.v2

the work of Frank Hessel. The example modified by the authors for latency measurement is available at: https://github.com/Aircoookie/esp32_https_server_test.

Acknowledgments: The authors would like to give credit to Michal Madziar of *Doyensec*, who in 2021 gave a security advisory on a buffer over-write vulnerability in the WLED API and sparked the idea to add a message authentication scheme to the WLED software. Furthermore, the authors would like to thank the peer-reviewers of the initial draft of this manuscript for their excellent review comments. The authors have not used any GenAI tools during the preparation of this manuscript.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

AES	Advanced Encryption Standard
AP	Access Point
ASIC	Application-Specific Integrated Circuit
API	Application Programming Interface
BCG	Browsing Context Group
COOP	Cross-Origin-Opener-Policy (header)
CSS	Cascading Style Sheets
ESP	Microcontroller series by Espressif Systems
GPU	Graphics Processing Unit
(H)MAC	(Hash-based) Message Authentication Code
HTML	HyperText Markup Language
HTTP(S)	HyperText Transfer Protocol (Secure)
IoT	Internet of Things
IT	Information Technology
JSON	JavaScript Object Notation
LED	Light-Emitting Diode
NIST	National Institute of Standards and Technology
MitM	Man-in-the-Middle (attack)
PBKDF2	Password-Based Key Derivation Function 2
PoC	Proof of Concept
PSK	Pre-Shared Key
PWA	Progressive Web App
RAM	Random Access Memory
RTT	Round Trip Time
SCT	Secure Crypto Tab
SDK	Software Development Kit
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UI	User Interface
URL	Uniform Resource Locator

References

1. Card, S.K.; Moran, T.P.; Newell, A. *The Psychology of Human-Computer Interaction*; CRC Press, 1983.
2. Google. Support for Nest Secure ended, 2024.
3. Open Home Foundation. About us, 2024.
4. Dolev, D.; Yao, A. On the security of public key protocols. *IEEE Transactions on Information Theory* **1983**, *29*, 198–208. https://doi.org/10.1109/TIT.1983.1056650.
5. Creese, S.; Goldsmith, M.; Roscoe, A.; Zakiuddin, I. The attacker in ubiquitous computing environments: Formalising the threat model, 2003.
6. Paar, C.; Pelzl, J.; Güneysu, T. *Understanding Cryptography*; Springer Berlin, Heidelberg, 2024.
7. MDN Web Docs. Progressive web apps.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 December 2025

doi:10.20944/preprints202510.1808.v2

8. Ptacek, T. Javascript Cryptography Considered Harmful, 2011.
9. Meixler Technologies Inc.. Browser Crypto, 2021.
10. Shekh-Yusef, R.; Ahrens, D.; Bremer, S. HTTP Digest Access Authentication. RFC 7616, 2015. https://doi.org/10.17487/RFC7616.
11. Nir, Y.; Langley, A. ChaCha20 and Poly1305 for IETF Protocols. RFC 8439, 2018. https://doi.org/10.17487/RFC8439.
12. Turan, M.S.; McKay, K.; Kang, J.; Kelsey, J.; Chang, D. Ascon-Based Lightweight Cryptography Standards for Constrained Devices: Authenticated Encryption, Hash, and Extendable Output Functions, 2025. https://doi.org/10.6028/NIST.SP.800-232.
13. Schwinne, C.; et al. JSON API, 2025.
14. MDN Web Docs. Cross-Origin-Opener-Policy (COOP) header.
15. MDN Web Docs. Cross-Origin-Embedder-Policy (COEP) header.
16. MDN Web Docs. SubtleCrypto: deriveKey() method.
17. Biryukov, A.; Dinu, D.; Khovratovich, D.; Josefsson, S. Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications. RFC 9106, 2021. https://doi.org/10.17487/RFC9106.
18. Scroggs, K. argon2-wasm, 2025.
19. OWASP. Password Storage Cheat Sheet, 2023.
20. Doherty, W.J.; Watson, T.J.; Thadani, A.J. The Economic Value of Rapid Response Time. *IBM Systems Journal* **1982**.
21. Yablonski, J. Doherty Threshold.
22. Espressif Systems. ESP32 Technical Reference Manual, 2024.
23. Preuß Mattsson, J.; Smeets, B.; Thormarker, E. Quantum-Resistant Cryptography, 2021.
24. Housley, R.; Hoyland, J.; Sethi, M.; Wood, C.A. Guidance for External Pre-Shared Key (PSK) Usage in TLS. RFC 9257, 2022. https://doi.org/10.17487/RFC9257.
25. Sy, E.; Burkert, C.; Federrath, H.; Fischer, M. Tracking Users across the Web via TLS Session Resumption. In Proceedings of the Proceedings of the 34th Annual Computer Security Applications Conference, New York, NY, USA, 2018; ACSAC '18, p. 289–299. https://doi.org/10.1145/3274694.3274708.
26. Kim, J.; van Schaik, S.; Genkin, D.; Yarom, Y. iLeakage: Browser-based Timerless Speculative Execution Attacks on Apple Devices. In Proceedings of the Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, New York, NY, USA, 2023; CCS '23, p. 2038–2052. https://doi.org/10.1145/3576915.3616611.
27. MDN Web Docs. Mixed content.
28. Zeng, E.; Roesner, F. Understanding and Improving Security and Privacy in Multi-User Smart Homes: A Design Exploration and In-Home User Study. In Proceedings of the 28th USENIX Security Symposium (USENIX Security 19), Santa Clara, CA, 08 2019; pp. 159–176.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.