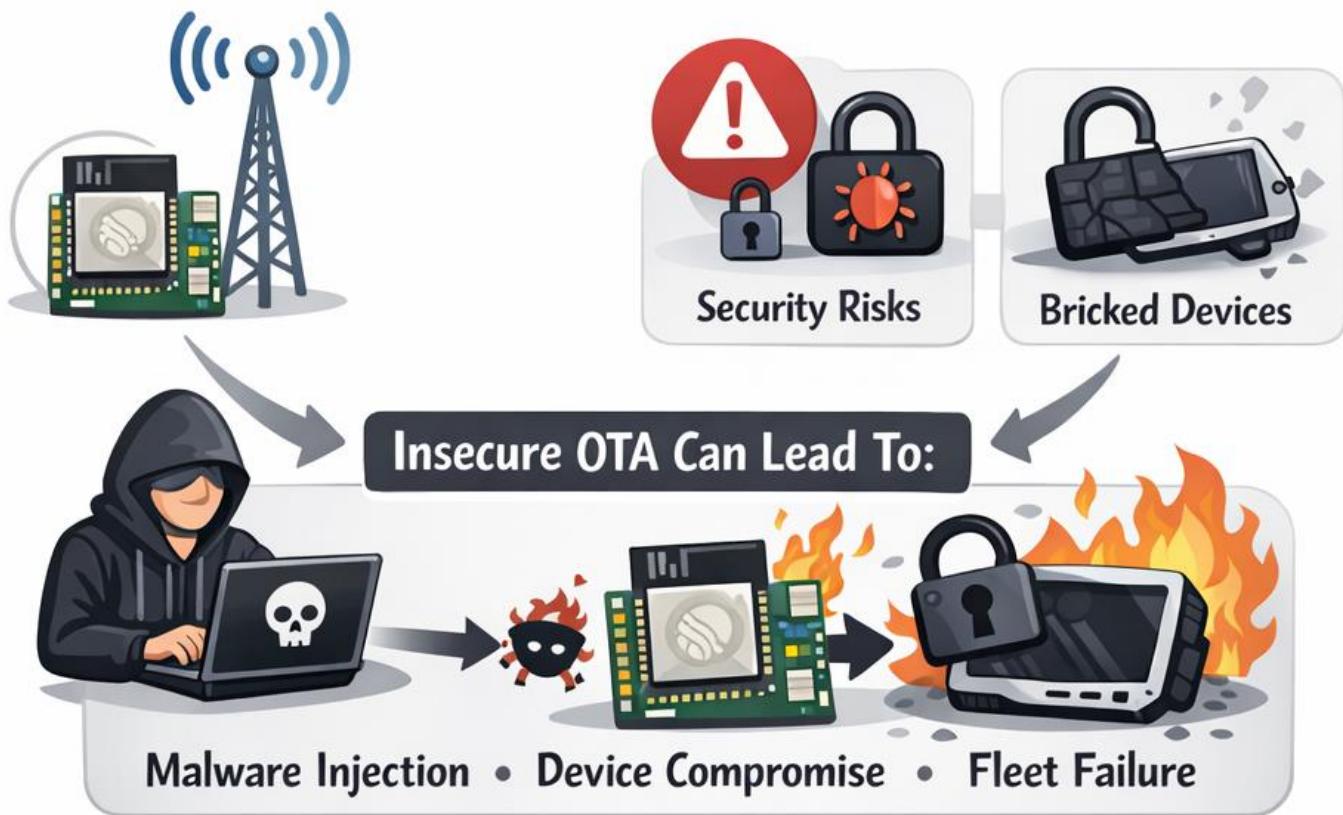


OTA Updates: Essential but Risky for ESP32 Devices



Implementing Secure OTA for ESP32



A Practical ESP-IDF Guide for Robust and Secure OTA Updates ➤➤➤

Secure OTA Firmware Updates on ESP32 Using ESP-IDF

Table of Contents

Table of Contents

1. Introduction
2. What Makes OTA "Secure" on ESP32
3. ESP32 Boot Process and OTA Chain of Trust
4. Flash Layout and OTA Partition Strategy
5. Secure Boot and Signed Firmware Images
6. HTTPS OTA and Transport Security
7. OTA Workflow in ESP-IDF
8. Implementing OTA with ESP-IDF APIs
9. Firmware Versioning, Validation, and Rollback
10. OTA Trigger Mechanisms in Real Products
11. Common OTA Failure Scenarios and Recovery

Table of Contents

12. Development vs Production OTA Workflows

13. Best Practices for Secure OTA in ESP32 Products

14. Conclusion

1. Introduction

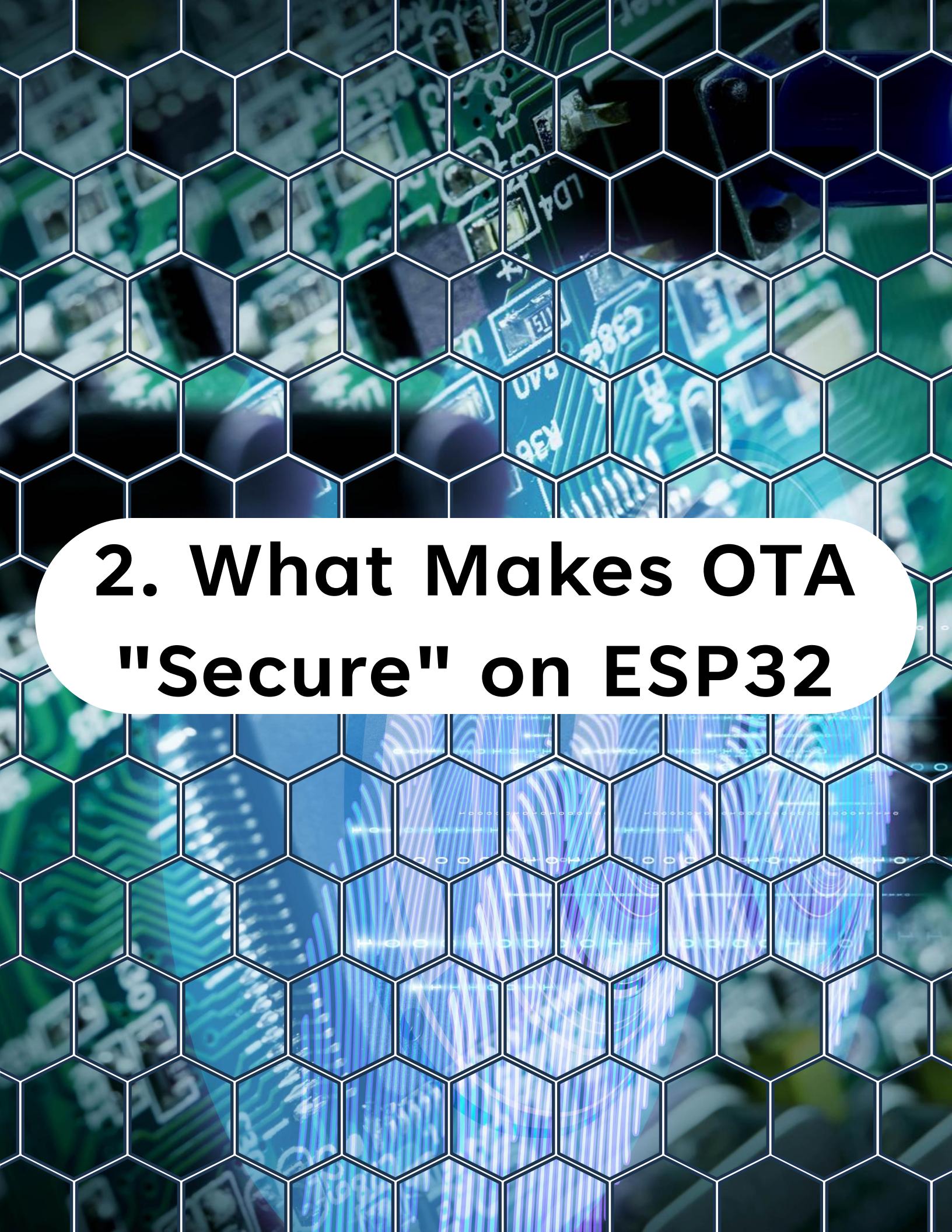
1. Introduction

Over-the-air firmware updates are no longer a convenience feature in embedded systems. For ESP32-based products deployed in the field, OTA is a fundamental system capability. Devices that cannot be updated remotely quickly become operational liabilities, security risks, or both.

However, OTA is also one of the most common attack vectors in IoT products. An insecure update mechanism can allow attackers to inject malicious firmware, permanently compromise devices, or brick entire fleets. For this reason, OTA must be designed as a security-critical subsystem rather than a simple download-and-flash feature.

1. Introduction

This article presents a practical, ESP-IDF-focused guide to implementing **secure OTA** on ESP32 devices. It explains how HTTPS, signed firmware images, and rollback mechanisms work together, and how ESP-IDF supports these features at the system level. The focus is on real-world engineering decisions, not minimal demos or shortcuts.



2. What Makes OTA "Secure" on ESP32

2. What Makes OTA "Secure" on ESP32

A basic OTA implementation answers only one question: how do I replace firmware remotely?

A secure OTA implementation answers several additional questions:

- How do I ensure firmware comes from a trusted source?
- How do I prevent unauthorized or modified images from running?
- How do I recover if an update fails?
- How do I avoid bricking devices during power or network failures?

2. What Makes OTA "Secure" on ESP32

On ESP32, secure OTA rests on three pillars:

1. Transport security

Firmware is downloaded over HTTPS, ensuring confidentiality and integrity during transfer.

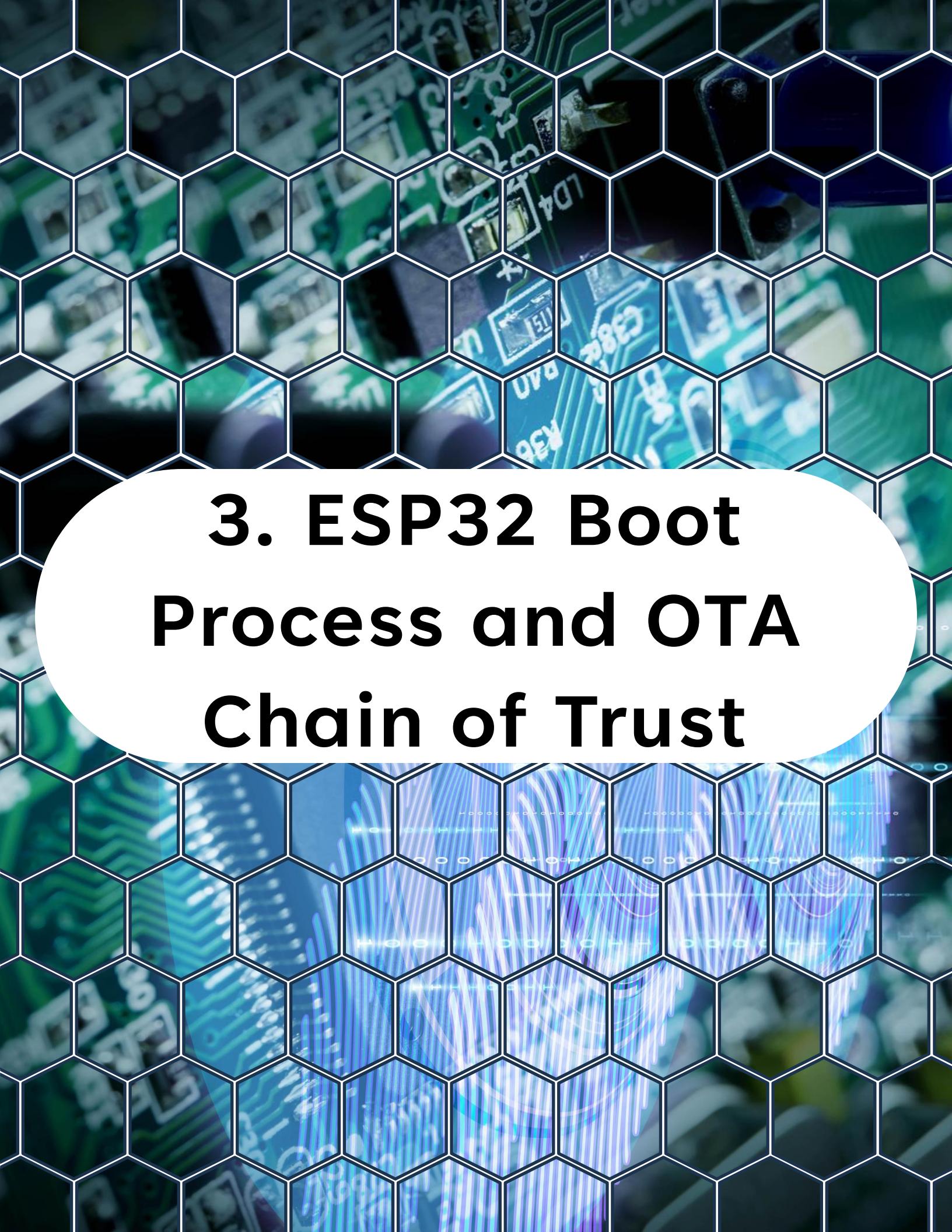
2. Firmware authenticity

Firmware images are cryptographically signed and verified before execution.

3. Runtime recovery

The system can roll back to a known-good firmware if a newly installed image fails.

ESP-IDF provides strong building blocks for all three, but it is the firmware architect's responsibility to connect them into a coherent system.



3. ESP32 Boot Process and OTA Chain of Trust

3. ESP32 Boot Process and OTA Chain of Trust

Understanding OTA requires understanding how the ESP32 boots.

At a high level, the boot process is:

1. ROM bootloader executes (immutable, stored in silicon).
2. Second-stage bootloader executes from flash.
3. The bootloader selects and launches an application image.

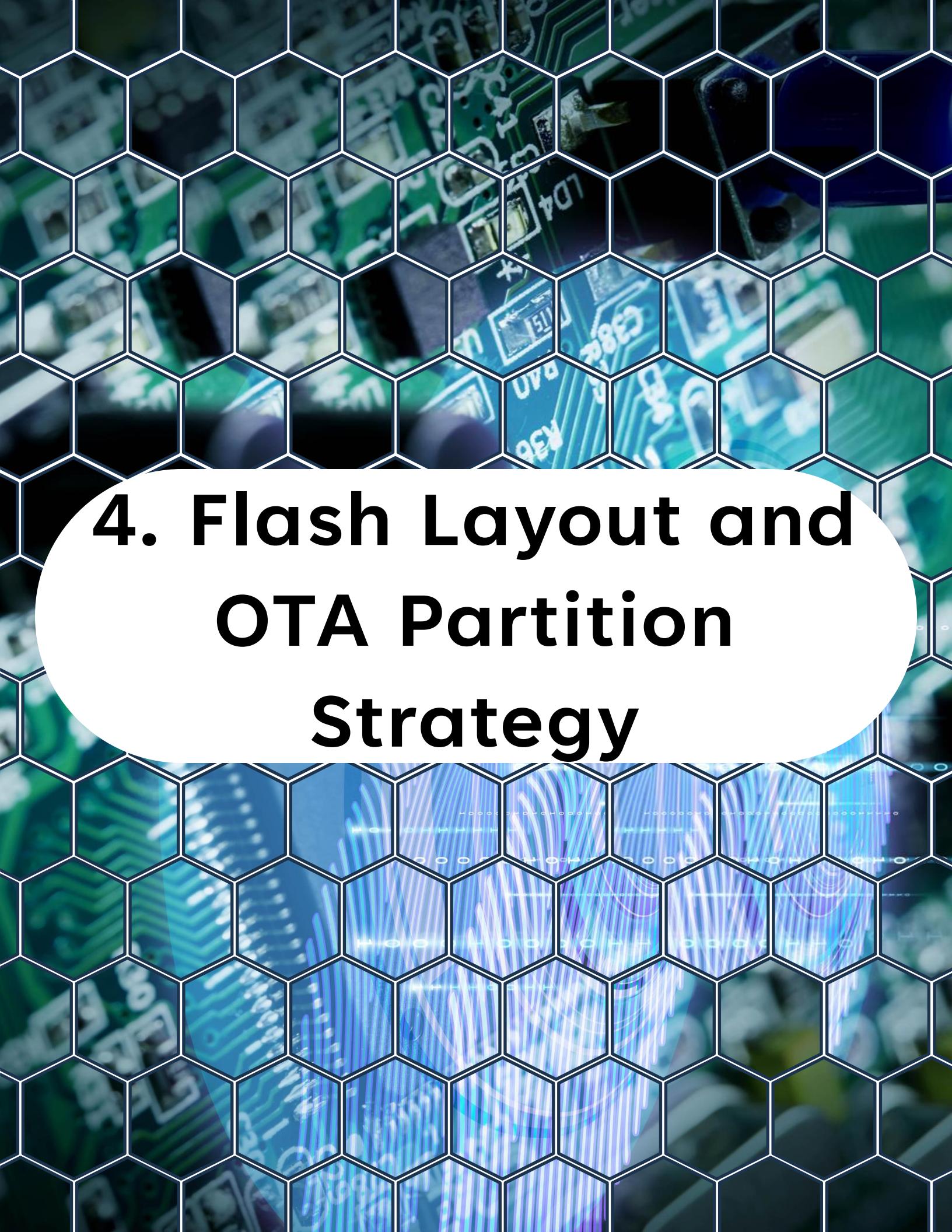
OTA integrates into step 3. Instead of always booting a fixed application partition, the bootloader can choose between multiple firmware slots. This selection is driven by metadata stored in flash, not by application code.

3. ESP32 Boot Process and OTA Chain of Trust

The concept of a **chain of trust** starts in ROM and extends forward:

- ROM code trusts only verified bootloader images.
- The bootloader trusts only valid application images.
- Applications inherit trust only if they pass verification.

Secure OTA ensures that this chain is never broken, even during updates.



4. Flash Layout and OTA Partition Strategy

4. Flash Layout and OTA Partition Strategy

OTA requires at least two application slots. One slot runs the current firmware, while the other receives the new image.

A typical OTA-capable partition table looks like this:

```
# Name,      Type, SubType, Offset,  Size
nvs,        data, nvs,     0x9000,   0x5000
otadata,    data, ota,     0xE000,   0x2000
ota_0,      app,  ota_0,   0x10000,  1M
ota_1,      app,  ota_1,   ,          1M
```

Key elements:

- **otadata** stores which application slot is active or pending.
- **ota_0** and **ota_1** are alternating firmware slots.

4. Flash Layout and OTA Partition Strategy

- The bootloader reads otadata to decide which image to boot.

During OTA, the running firmware writes the new image into the inactive slot, never overwriting itself. This alone eliminates many failure modes.



5. Secure Boot and Signed Firmware Images

5. Secure Boot and Signed Firmware Images

Signed firmware images ensure authenticity. Even if an attacker gains access to flash or the update channel, they cannot execute unsigned or modified firmware.

With Secure Boot enabled:

- Each firmware image includes a cryptographic signature.
- The bootloader verifies the signature before execution.
- Verification happens before any application code runs.

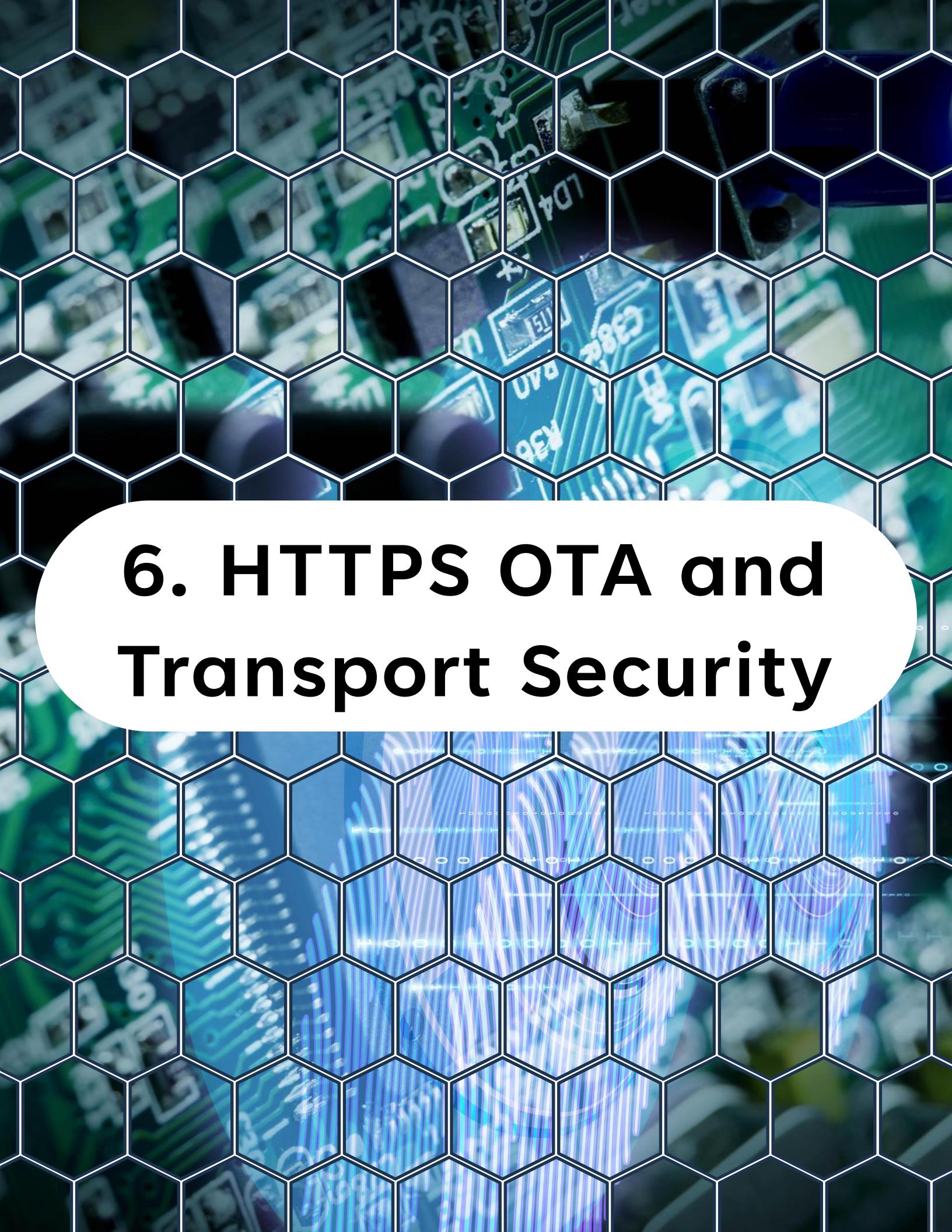
From an OTA perspective, this means:

- OTA can download any data it wants.

5. Secure Boot and Signed Firmware Images

- Only valid, signed images will ever execute.
- Invalid images fail safely and never boot.

It is important to understand that OTA does not replace secure boot. OTA relies on secure boot to enforce authenticity.



6. HTTPS OTA and Transport Security

6. HTTPS OTA and Transport Security

Transport security protects firmware while it is in transit.

ESP-IDF supports HTTPS-based OTA using standard TLS mechanisms. During OTA:

The device establishes a TLS session with the firmware server.

The server certificate is validated.

Firmware data is transferred securely.

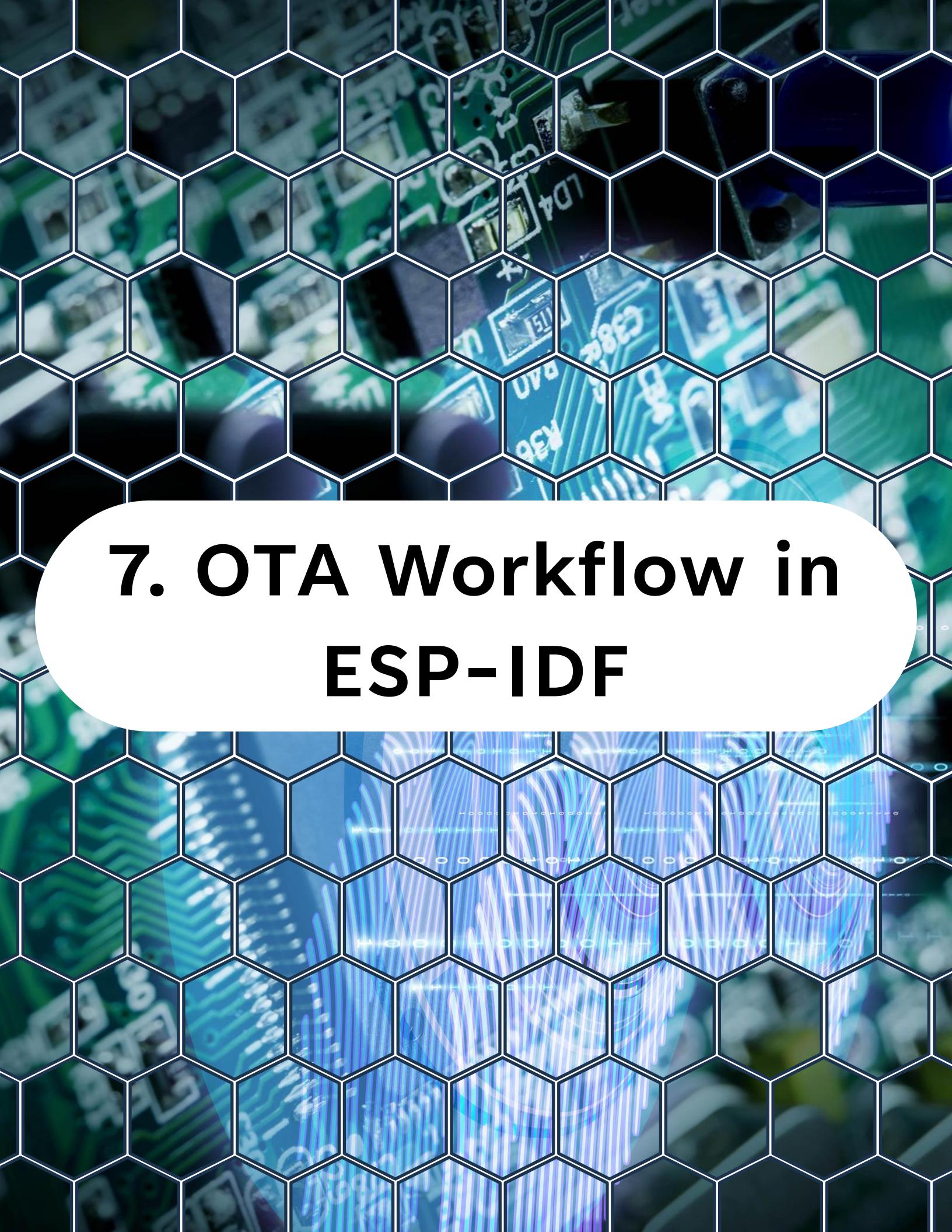
A typical HTTPS OTA configuration uses certificate pinning:

```
esp_http_client_config_t http_cfg = {  
    .url = firmware_url,  
    .cert_pem = server_cert_pem,  
};
```

6. HTTPS OTA and Transport Security

This ensures that even if a public CA is compromised, only the pinned certificate is accepted.

HTTPS protects against network-based attacks, but it does not guarantee firmware authenticity. That responsibility belongs to image signing and secure boot.



7. OTA Workflow in ESP-IDF

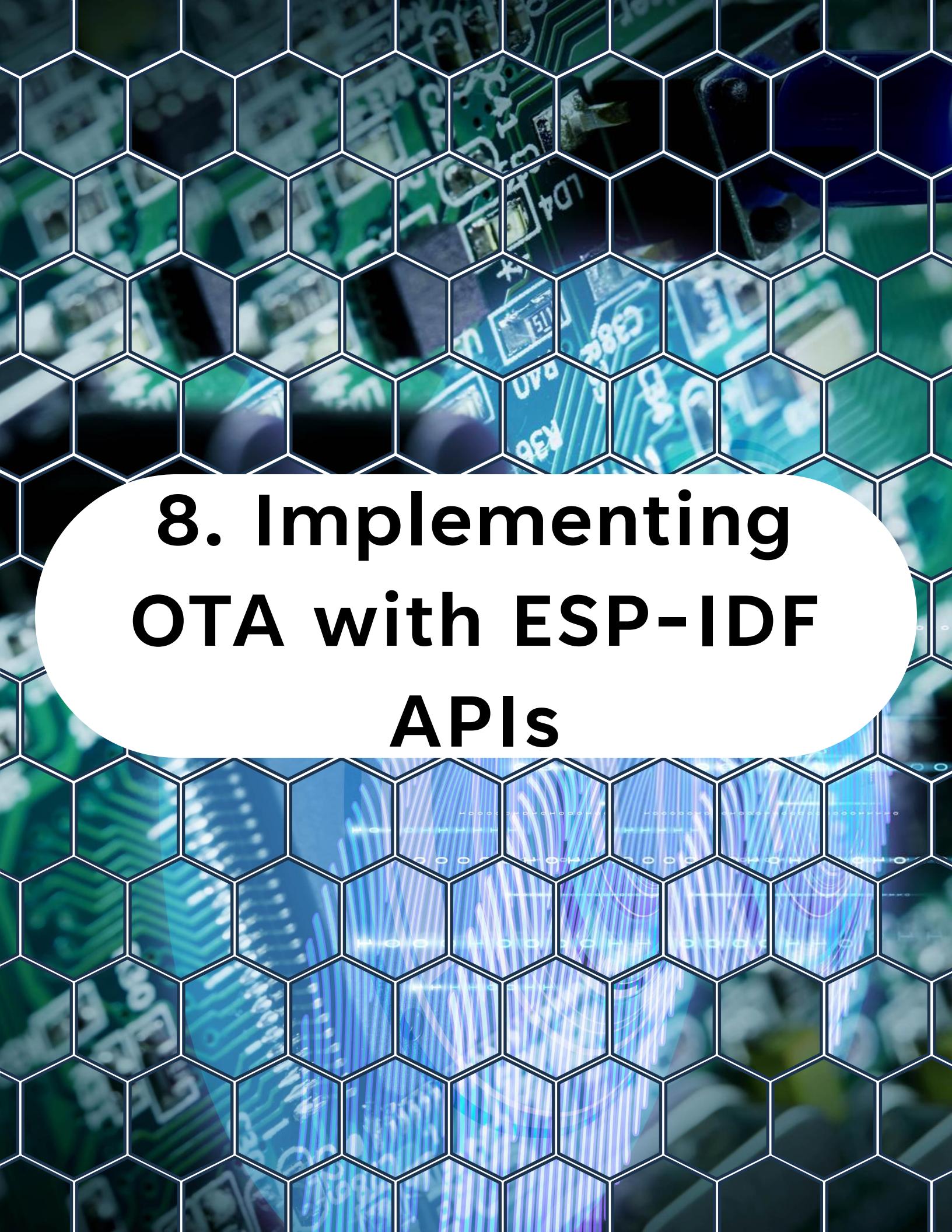
7. OTA Workflow in ESP-IDF

At a system level, OTA follows a predictable sequence:

1. Decide whether an update is allowed.
2. Download the firmware image.
3. Write the image to the inactive partition.
4. Validate the image.
5. Mark the new partition as bootable.
6. Reboot.

ESP-IDF enforces this flow internally. A failed step does not corrupt the running firmware. Reboot only occurs once a complete image is written and verified.

This design makes OTA robust even in unstable environments.



8. Implementing OTA with ESP-IDF APIs

8. Implementing OTA with ESP-IDF APIs

ESP-IDF provides both high-level and low-level OTA APIs.

The simplest secure approach uses

`esp_https_ota:`

```
esp_https_ota_config_t ota_cfg = {  
    .http_config = &http_cfg,  
};  
  
esp_err_t err = esp_https_ota(&ota_cfg);  
if (err == ESP_OK) {  
    esp_restart();  
}
```

This function handles:

- HTTPS connection
- Firmware download

8. Implementing OTA with ESP-IDF APIs

- Flash writes
- Image verification
- Partition switching

For advanced use cases, lower-level APIs such as `esp_ota_begin` and `esp_ota_write` allow full control over the update process. These are useful when integrating custom protocols or storage backends.

Regardless of API choice, OTA should always run in its own task and never block critical system functionality.



9. Firmware Versioning, Validation, and Rollback

9. Firmware Versioning, Validation, and Rollback

OTA is incomplete without rollback.

After booting into a newly updated firmware, the system must confirm that the image is healthy. If validation fails, the bootloader automatically reverts to the previous firmware.

Typical validation steps include:

- Successful boot completion
- Basic hardware initialization
- Optional self-tests
- Explicit call to mark the image as valid

If the application crashes, resets, or fails validation, rollback is triggered automatically.

Rollback transforms OTA from a risky operation into a controlled, recoverable process.



10. OTA Trigger Mechanisms in Real Products

10. OTA Trigger Mechanisms in Real Products

Real products do not update blindly.

Common OTA triggers include:

- User-initiated updates (button or UI).
- Cloud-controlled updates (command or policy).
- Scheduled maintenance windows.

Blind OTA on every boot is dangerous, especially in battery-powered devices. A well-designed system checks conditions such as power level, connectivity quality, and timing before updating.

OTA should be deliberate, not automatic.



11. Common OTA Failure Scenarios and Recovery

11. Common OTA Failure Scenarios and Recovery

Secure OTA systems must assume failure.

Common scenarios include:

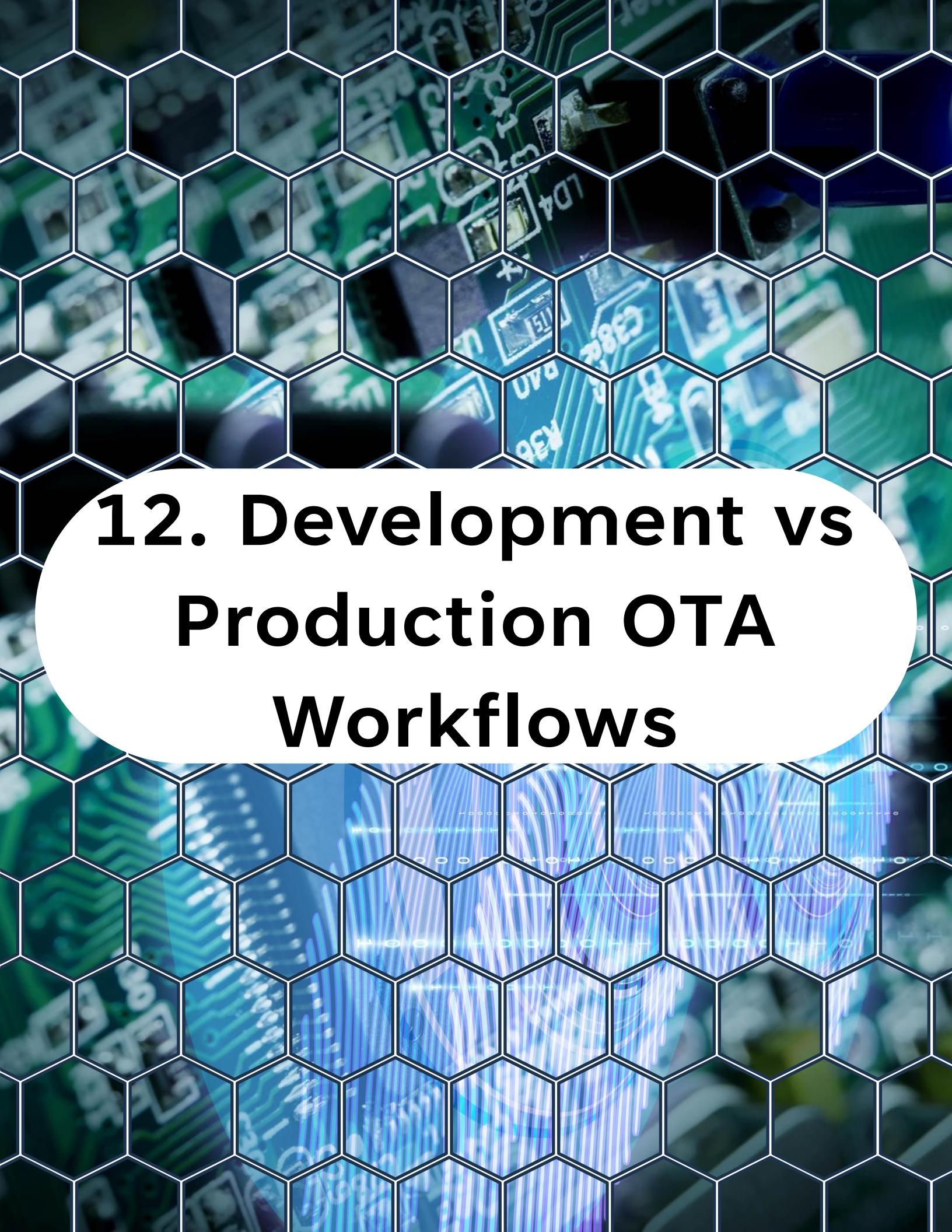
- Power loss during download.
- Network drop mid-transfer.
- Invalid or corrupted firmware.
- Incorrect partition layout.

ESP32 and ESP-IDF handle these cases

gracefully when OTA is implemented correctly.

The running firmware is never overwritten, and rollback ensures recovery.

Most catastrophic OTA failures are caused by incorrect partition tables or bypassing validation steps.



12. Development vs Production OTA Workflows

12. Development vs Production OTA Workflows

Development OTA and production OTA are not the same.

In development:

- Convenience may trump security.
- Keys may be stored locally.
- Manual recovery is acceptable.

In production:

- Keys must be protected.
- OTA must be auditable.
- Recovery must be automatic.

Secure OTA design should always target production, even during development.



13. Best Practices for Secure OTA in ESP32 Products

13. Best Practices for Secure OTA in ESP32 Products

- Treat OTA as part of system architecture.
- Never overwrite the running firmware.
- Always plan for rollback.
- Use HTTPS and signed images together.
- Log OTA decisions and failures.
- Test OTA under worst-case conditions.
- Document OTA behavior clearly.

Secure OTA is not difficult, but it requires discipline.

14. Conclusion

14. Conclusion

Secure OTA on ESP32 is not a single API call. It is a system composed of flash layout, bootloader logic, cryptography, network security, and firmware design.

ESP-IDF provides robust tools to implement HTTPS-based OTA with signed images and rollback support. When used correctly, these tools allow ESP32 products to update safely, recover automatically, and remain secure throughout their lifetime.

Design OTA early, design it deliberately, and treat it as a core system feature.

"Thanks for watching!

*If you enjoyed this video, make sure
to hit the like button and subscribe to
stay updated with my latest content.*

*Don't forget to check out my other
videos for more tips and tutorials on
Embedded C, Python, hardware
designs, etc. Keep exploring, keep
learning, and I'll see you in the next
video!"*

<https://www.linkedin.com/in/yamil-garcia>

<https://www.youtube.com/@LearningByTutorials>

<https://github.com/god233012yamil>