6.857 Final Project: Evaluating and Improving Over-the-Air Update Security for Arduino

Ryan Hennessey, Pedro Sales, Amarbold Byambajargal, and Uriel Guajardo

May 21, 2021

Contents

1 Introduction

The Internet of Things have grown rapidly in commercial use in recent years. Small, lightweight IoT devices need frequent firmware updates for reasons such as bug fixes and new features. In order to push the latest firmware version onto an IoT device, many use Over-the-Air (OTA) method, which consists of updating the device over a wireless connection. This way is beneficial in many cases, since most of IoT devices do not see human interaction regularly, making it tedious to install such updates with wired network.

In this paper, we will look at ArduinoOTA, which is one specific implementation of OTA for IoT devices. First, we give a context of our project and a security policy of ArduinoOTA. Then we discuss details on its implementation and security analysis, followed by our proposal for a better system. Finally, we go over discussion of our implementation, together with a summary of changes that overcome the shortcomings of the current ArduinoOTA library.

2 Context

The Internet of Things (IoT) ecosystem is everywhere, and most devices now have some level of internet connectivity. They do so for good reason: IoT provides an added level of connectivity between devices, allowing user-device and device-device interactions. As a result, many devices are connected to the internet that you might not expect: wearable watches, smoke alarms, doorbells, and even cars themselves.

However, all of these internet-connected devices tend to have a feature that may pose a security risk: they receive over OTA updates to update their software. In an ideal setting, the OTA process would be standardized and secure across devices. Unfortunately, that's not the reality: manufacturers use completely different standards and practices, each of which may have security holes. If a secure device updates its firmware by receiving an OTA update through a vulnerable framework, then the security of the device is completely broken.

A simple -and perhaps compelling- solution is to get rid of OTA update altogether for most devices. We could still live in an IoT world, but users would have to manually perform updates by connecting a physical interface to the device. This would avoid OTA security issue we presented, but we don't believe this is the best approach. Not only does getting rid of OTA updates prevent developers from adding extra functionality, but it also prevents important security updates. Therefore, getting rid of OTA updates is a security risk in itself: if you release a product with a

vulnerability, it would be near impossible for you to patch that vulnerability on products in the wild.

Specifications for secure update frameworks do exist [1], but to the best of our knowledge, none are specifically tailored to IoT devices. Instead, we wish to analyze existing frameworks to come up with an ideal specification that other frameworks can adopt to become more robust at handling different kinds of attacks. Specifically, we will be analyzing the Arduino's implementation of the OTA process: the ArduinoOTA library. We will be working with a wifi-enabled microcontroller ESP-8266 to evalute the library, and we will attempt to fix its flaws.

3 Intro to ArduinoOTA, Basic Usage, and Potential Attacks

3.1 Intro to ArduinoOTA

ArduinoOTA is the library will be analyzing. We care about ArduinoOTA specifically both because it showcases common problems in OTA security and because it is used in the real world. A GitHub search for the statement "`#include <ArduinoOTA.h>`" yields more than 27,000 hits. Looking at these, we see that ArduinoOTA is used in many hobbyist projects as well as some actual low volume products such as 3D printers and smart light switches.

ArduinoOTA admits in its documentation that no real security promises are made. Almost every measure of security is optional, but for the purposes of this analysis, we assume that all of those options are used by any reasonable developer. (However, this is not a safe assumption in the real world, and a wiser implementation of ArduinoOTA would implement security automatically.) At its best, ArduinoOTA provides data integrity and authenticity from the update server to the IoT device. A security policy for ArduinoOTA looks like the following.

3.2 Basic Usage

At a very high level, we will describe how ArduinoOTA works. First, the update server pings an IoT device, notifying it that there is a new update. The device accepts the update, and the server sends it over the air. There are optional layers of security that can be added to this process including password protection and binary signing. The optional layers of security aim to provide authenticity and integrity through a signing scheme.

3.3 Potential Attacks

Arbitrary installation: When the optional security measures are not implemented, an adversary can upload any binary to the device and cause unintended behavior in the device.

Rollback attack: The adversary stores valid, signed updates by sniffing the communication between the update server and the device. Whenever they want, the adversary can update the device to that version and use that information to do nefarious actions.

Old version attack: The adversary does the same attack as the rollback attack, but this time he updates the version to one that is newer than the version of the device, but older than the most up-to-date version. These attacks exploit schemes that only verify if a device is updating to a newer version than it has.

4 Implementation details and diagram

In order to get updated with ArduinoOTA, first an IoT device must be connected on wireless connection, which exposes the device to potential adversaries. Thus, ArduinoOTA offers some security to protect the device using the library from getting hacked during the OTA update process.

4.1 Update Protocol with MD5 hashing

The library uses Digest-MD5 to authenticate updates from a remote server. The update process

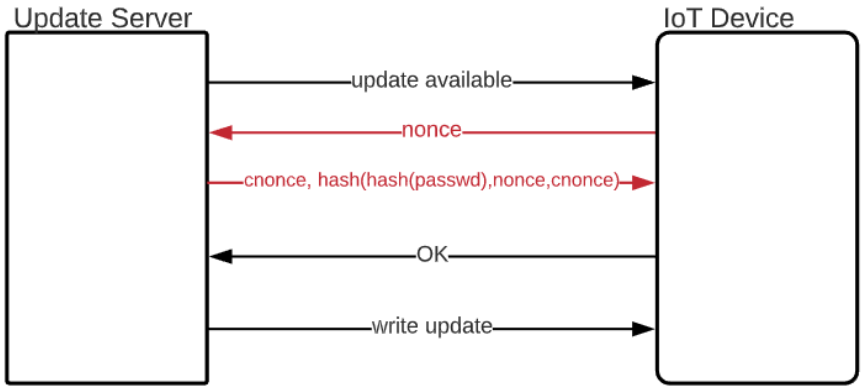


Figure 1: Communication between server and device during update process. Optional authentication scheme is highlighted in red.

commences with a packet sent from the server to the device informing a new update is available. If authentication is enabled, the device responds with a nonce, which is then answered by the

server together with a password. The password is obfuscated using Digest-MD5. The device then performs a check against an internally stored hash of the password, and accepts authentication if there is a match. Once authenticated, the device goes fully into update mode and starts receiving the binary.

4.2 Binary Download and Signing

There is not enough RAM in the device to hold the update contents. It is instead written to an unused section of flash memory. Once the update is downloaded, the device computes a checksum and reboots with bootloader flags if successful. On reboot, the bootloader copies the new sketch to the executable section which concludes the update process. In case the checksum fails, the device can always fall back to the original sketch.

ArduinoOTA also offers optional "advanced security" which consists of binary signing using public key cryptography. The server hashes the binary contents using SHA-256 and then signs it with their secret key SK using RSA-2048. This allows the device to check for authenticity after the full binary is downloaded into the free section of flash. It performs the same SHA-256 hash and then checks the signature using the corresponding PK which is stored in the current_sketch. The device instructs the bootloader to copy the update only if this check passes.

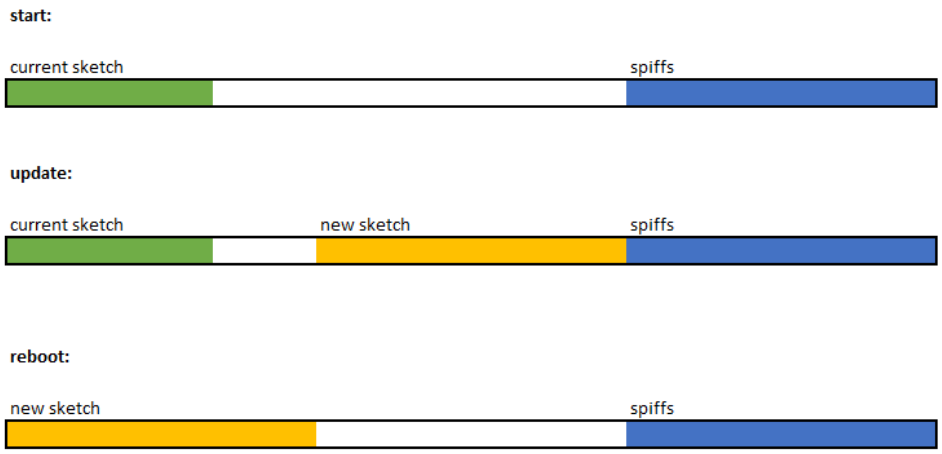


Figure 2: ArduinoOTA writes the new firmware into memory without executing it and checks for signatures before rewriting its current firmware

5 ArduinoOTA Security Analysis

ArduinoOTA's security policy is fairly simple; however, it may present some security vulnerabilities for more sophisticated attacks. In the following sections, we will outline the strengths and weaknesses of several security aspects associated with ArduinoOTA's policy and its implementation.

5.1 Strengths

5.1.1 Fault Tolerance

ArduinoOTA does a good job at having a reasonably robust updating process:

1. As shown in Fig. 2, the framework write the sketch on non-executable memory in order to check that everything was downloaded correctly. This is great not only for security, but it helps prevent cases where the sketch is incomplete or is corrupted

2. At each step of the update process, ArduinoOTA checks for errors and calls a callback function appropriately, which can notify the user of any errors that may have occurred.

As a result, the device can be reasonably confident that, if an update said it finished, it did so correctly, and it satisfied the security measures that the developer enabled.

5.1.2 Authenticity

Perhaps the strongest security feature that ArduinoOTA has is its RSA public-key encryption. It is not enabled by default, but if it is enabled, then it allows the sender of the update to cryptographically sign the binary using a private key. After receiving the binary + signature, the client can use the public key of the entity that issued the upload to make sure that the signature is correct.

This gives authenticity to the updates; in an ideal world, only the manufacturer issuing the update has the private key to write the signature. Without a correct signature, ArduinoOTA will reject the update, thus ensuring you that only the manufacturer (private key holder) created this binary.

5.2 Vulnerabilities/Attacks

5.2.1 Obscure Security Features

One flaw that we've noticed with the ArduinoOTA library is that it's often difficult as a developer to figure out how to adequately secure their application. In fact, the most glaring vulnerability in this area is that security features are all disabled by default. This runs contrary to most applications: usually, libraries enable many security measures by default, and it's up to the client to disable those features with full knowledge that their application may be more vulnerable. However, ArduinoOTA never prompts the client with this knowledge, which may give clients the incorrect assumption that there is more security than there really is.

5.2.2 Lack of Confidentiality

Although ArduinoOTA does do signatures to provide authenticity to uploads, there is no encryption done with uploads. This means that, if an adversary could observe the network traffic, the adversary could see potentially sensitive data that could live inside of the upload. Even if there was no explicit sensitive data inside the code, confidentiality is still a very important security measure for other reasons. For example, an adversary would be able to figure out what security features are enabled just by listening into the updates that are being sent to the device.

5.2.3 Arbitrary Installation Attacks

Without an RSA public key encryption, there are ways that an attacker can install virtually any firmware onto the Arduino. As mentioned in previous sections, ArduinoOTA's basic security includes setting a password; this would prevent random connections from giving updates, since they should not know the password. However, the password needs to live somewhere in the device's memory. So whenever an update is sent to a device (which includes the whole firmware to be replaced), the adversary can copy the binary and look for candidate password strings. Once they find the correct password, they can now upload binaries to the device itself using the password, thus breaking the basic password security that ArduinoOTA provides. To make things worse, the library gives the user the option to set the password MD5 hash instead of the plain text password, which gives a higher illusion of security. But what happens in reality is that the MD5 hash now becomes the password. Note the following two possibilities:

1. Device receives plaintext password from server, hashes it, and compares against local hash.

2. Device receives hash of password from server, and directly compares against local hash.

The first one is vulnerable to network eavesdroppers, while the second one is vulnerable to attacks that sniff the binary. There is really no proper way of doing authentication with a hash when both of these channels are unencrypted. The device cannot hold secret keys, and the exchange between device and server cannot send sensitive information.

5.2.4 Versioning Attacks

ArduinoOTA also provides no checking of versions in updates. This means that, even though it provides integrity, an adversary can still present the device with an older, vulnerable update that was created by the manufacturer. We present two attack strategies that an adversary can use:

Rollback Attack: The adversary knows a specific version of a software (version A) is vulnerable to an exploit. What the attacker can then do is ping the device and update its latest firmware on that device to version A, which would create security risks within the device itself. To mitigate this, ArduinoOTA would need to add versioning to its updates, and it would need to only allow versions that are higher than the version currently installed (so as to constantly improve the software, rather than downgrade it by sending older versions).

Old Version Attack: Even if ArduinoOTA provided basic versioning to prevent most cases of Rollback Attacks, there is still a subset of the Rollback Attacks which would still work. For example, if the device's firmware is version 1 (which is vulnerable to exploits), and if it's asking for an update, then an adversary can send them version 2 (which is also vulnerable to the same exploits) even though there is a newer version 3 out there that is not vulnerable like previous versions were. This is disastrous, since the user will receive an update and falsely believe that his system is secure.

6 Proposal for better system

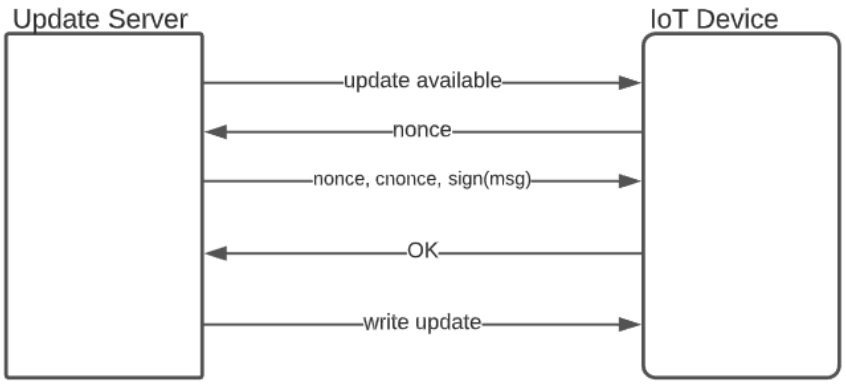
Here, we propose an improved version of the system. This revision adds support for secure authentication during the update process, as well as signed version checking before the update is committed to the executable section of memory. As an optional security feature, the scheme also supports one-time signing of the binaries, mimicking the secure update approach used by Apple [3].

6.1 Key Management

Before detailing the changes to the updates, it is important to understand how to handle key management and storage. Even though updates are authenticated, the binary is still being sent in plain text so no secret keys can be safely stored or transmitted to the device. We propose storing two public keys: PK_{main} and PK_{dev} on the on-board filesystem. These two keys act as a type of certificate chain, where *main* key signs the *dev* key, together with a key version number. The *dev* key can then be used during the binary signing and update authentication process. This allows the *master* secret key to be stored securely in a vault, while the *dev* key is in a computer connected to the internet. Then, if a *dev* secret key is leaked, the holder of the *main* secret key can sign a new *dev* key with a higher version number and update the devices accordingly. We note that the *main* key holder must do this quickly, before an attacker severely compromises the device. The key update follows the same update process authentication as a regular firmware update, except it only updates the *dev* key field in the internal filesystem.

6.2 Update Process Authentication

We propose replacing the insecure md5 password authentication with Public Key signature checks. As such, the update process flow remains largely the same, with the exception of the second server response and subsequent check by the IoT device. After receiving an update request, the IoT device



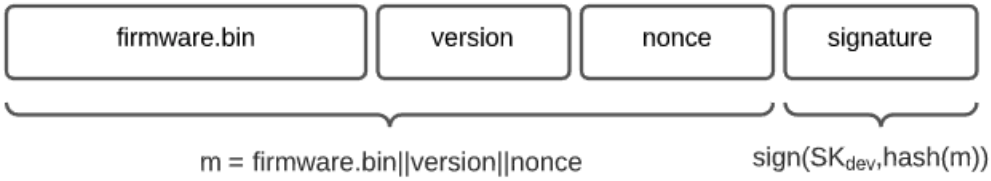
generates a random nonce that gets sent to the server. The server then generates a random nonce of its own, cnonce, concatenates both into message $m = \text{nonce}||\text{cnonce}$, and then signs the message using the current SK_{dev}. Once the device confirms the signature using PK_{dev}, it is authenticated

with the server and can safely [1] execute update commands. These commands can be one of three: i) updating the firmware binary, ii) updating the filesystem, or iii) revoking the existing *dev* key by presenting another signed key with a newer version number.

The use of nonce protects the authentication scheme from replay attacks. Since the device generates a random nonce every time, an attacker cannot use a previously signed command to trick the device into update mode. This effectively acts as a "proof" that the device is interacting with a SK_{dev} holder that is able to sign messages on-demand. Crucially, we also include a cnonce, because otherwise this handshake would expose a way to request arbitrary signatures from the server, which would completely break authentication! The cnonce is random and controlled by the server, therefore a PPT attacker cannot request arbitrary signatures with more than negligible probability respect to the cnonce length.

6.3 Binary Versioning

One of the vulnerabilities we uncovered were that despite binaries being properly signed, these are not versioned. We solve this with a simple modification, which consists of appending a version number to the binary before signing it. That way, the version number is correctly authenticated together with the binary and cannot be modified by an attacker. We can also protect against the



old version attack by enforcing expiring signatures on the binaries. Since trusting a time server can be a possible attack vector, we propose optional one-time signing of binaries. This consists of appending the same nonce sent by the IoT device during the initial authentication to the binary, and signing it. The device can then check that the received binary was in fact signed specifically for them, in the current session, by looking for the nonce they initially sent to the server.

As with the original ArduinoOTA implementation, the device writes the proposed binary to a special region in flash. After the download completes, then it can perform the signature verification and set a bootloader bit that copies the code to the executable region on reboot.

[1] this authentication is in place so attacker can not willfully put the IoT device in update mode. However, by itself does not protect against man in the middle attacks. All further commands (binary update, filesystem update, key update) must also be individually signed and contain the original nonce

6.4 Usability

One aspect that can not be ignored is making the security not be a hassle to the developer. No matter how good the system is in theory, if the developer chooses not to use them then the security falls apart. Substituting the md5 password for a signature is in fact easier for the developer. When the project is first created, the developer can be prompted to generate the public and private keys. Once this is done, the OTA updater is able to serve updates without user intervention, and importantly without having to come up with a secure password. The key management scheme outlined in **6.1** should also be easy to use. However, if this turns out to be a complication for the developer a single key can be safely used as long as the developer secures the dev key correctly.

6.5 Note on Random Number Generation

Random number generation on small microcontrollers can be a major challenge. The initial implementation of ArduinoOTA simply used the system uptime in microseconds as a random number. This is a major vulnerability since an attacker controlling the device's power or one that can remotely crash the program would be able to have control over this value. Any implementation of an authentication scheme that relies on device-side random numbers should ensure that these are random enough. Since these devices are usually connected to multiple sensors, these could be jointly used as a random noise signal to prime a pseudo random number generator. One must still be aware of the possibility of an attacker controlling a subset of these inputs. It is important that they are diverse and that enough readings are performed over time to ensure enough seed randomness.

7 Discussion of implementation

We implemented a subset of the proposed improvements. Specifically, we replaced the md5 authentication with the signature scheme. We also added versioning to the signed binaries to prevent rollback attacks. Here is the summary of changed files:

- `ArduinoOTA.cpp`: main library file. Removed md5 authentication, rewrote handshake to take PK from constant defined in the code
- `espota.py`: script executed by the IDE upon OTA upload. Modified to support PK handshake in place of md5.

- `Updater.cpp`: added versioning checks. While this could have been done on the ArduinoOTA library itself, the Updater class is part of esp-8266 Arduino "cores", and is also used by other update methods. We considered it made sense to improve this, as it benefits other OTA implementations as well.

- `signing.py`: script executed by the IDE before and after compilation. It splices a header containing PK_{dev} during pre-compilation and signs the binary after its compiled. Changes consist of now also including a version number during pre-compilation, and appending the version number to the compiled binary before signing.

The overall memory footprint is larger when compared to the unsecured or md5 protected version. This is because signature verification requires including BearSSL which handles all the cryptographic operations: sha-256 hashing and signature verification. Note that the memory footprint is practically the same as the "signed binary" original implementation, since we reuse the signature checking primitives for the improved handshake process.

We were able to get the versioned signature working. However, there are still some outstanding issues with the proposed handshake, primarily due to problems with proper handling of the network interface on the device. We have not yet made an attempt to implement the dual *main* and *dev* key scheme.

8 Future Work

The implementation is still a work in progress. Once we get it to work properly, we would like to properly document it and either open a pull request to the original OTA library or publish our own improved version.

9 Conclusion

In this paper, we analyzed the security vulnerabilities of the esp8266 implementation of the ArduinoOTA library. We used the uncovered vulnerabilities to design a new, secure, update scheme that can be implemented in any device with similar characteristics. The proposed scheme guarantees authentication during the full update process, even when it is carried out over an unecrypted network channel.

10 Acknowledgments

We would like to thank the course instructors, Ron Rivest and Yael Kalai for teaching us the material needed to perform an analysis such as this. We also want to thank the TAs and LAs, specifically Mike Specter, for feedback during the making of this project.

11 References

References

[1] Various Authors. *The Update Framework*
`https://theupdateframework.io/overview/`

[2] Open source. *ArduinoOTA documentation.*
`https://arduino-esp8266.readthedocs.io/en/latest/ota_updates/readme.html`

[3] Apple Inc. *Apple Security White Paper.* [*Secure software updates*]. Feb. 18, 2021
`https://support.apple.com/guide/security`

[4] Terry Dunlap. *The 5 Worst Examples of IoT Hacking and Vulnerabilities in Recorded History.* June 20, 2020
`https://www.iotforall.com/5-worst-iot-hacking-vulnerabilities`

[5] Nick G. *How Many IoT Devices Are There in 2021?.* March 29, 2021
`https://techjury.net/blog/how-many-iot-devices-are-there`