JWT Back to the future

On the (ab)use of JWTs in IoT transactions*

Alberto Battistello[0000−0001−8837−1356]1, Guido Bertoni[0000−0002−5122−1589]1, Filippo Melzani[0000−0002−0282−8894]1, and Maria Chiara Molteni[0000−0003−2901−2972]1

Security Pattern, Milan, Italy
{a.battistello,g.bertoni,f.melzani,m.molteni}@securitypattern.com

Abstract. The use of JSON Web Token (JWT)s has become ubiquitous in the Internet of Things (IoT) for the secure exchange of messages between the things and the Cloud. However, the standard that describes the JWT is fragmented, and interpretations may pave the way for abuses. In this work we show a supply chain attack that exploits a weakness in the JWT standard. In particular an attacker that takes control of one device during production, may be able to create a series of valid JWTs, that may be used further after the deployment, to impersonate the device when accessing the Cloud infrastructure. Among the advantages of the attack is that the network is completely unaware about the JWTs created in the supply chain by the attacker.

We show that the use of JWTs in the context of the Google Cloud IoT Core [15] infrastructure paves the way for a subtle attack, and that quite unexpectedly, the presence of an Secure Element (SE) on the connecting device does not allow to thwart the problem, but instead seems to make a solution harder to reach. We showcase our attack on the - now retired - Google Cloud IoT Core, in order to avoid malicious use of our findings, but our discovery can be applied to other services that provide token-based authentication. For example we further show that the same weaknesses applies to other tokens like the Concise Binary Object Representation Web Token (CWT) and the Entity Attestation Token (EAT) ([26,23]), and to platforms like HiveMQ and EMQX [13,17] providing a much wider attacker scope then merely a single token type or Cloud provider. Furthermore, our attack also applies to the Open Charge Metering Format (OCMF) standard, used for recording meter readings from charging station for Electric Vehicles (EV) [33].

In order to thwart the presented attack we provide a few countermeasures that can be applied, depending on the IoT infrastructure at hand.

* This work was partially funded by the European Union under grant agreement no. 101070008 (ORSHIN project). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them.

1 Introduction

The JWT standard described in RFC 7519 [21] specifies how to securely represent claims between two parties. It is one of the building blocks of the OAuth protocol [24], which allows authorization grants to be exchanged between services. For example, applications on computers, phones, TVs, printers etc. can use Google OAuth 2.0 to authorize access to Google's APIs [15]. This means for example that once you are logged in your phone, you can get access to your Google Drive documents without the need to input your credentials again.

JWT's have been developed in order to reduce both development complexity and friction in user's adoption. Developers can use a single API to create and validate JWTs for different services. At the same time the user experience is simplified by requiring a Single-Sign-On (SSO) to use different services. While the user experience has effectively been improved, the same is not completely true for developers. The main problem lies in the versatility of JWTs which can be declined in several different forms, use many different cryptographic algorithms and solve many different problems.

Lately, Google Cloud IoT Core adopted the use of JWTs to authenticate the requests coming from devices in the field. The idea is that the JWT token is much more lightweight to produce than a Transport Layer Security (TLS) authentication, thus improving the efficiency of the "things", as described in [14,13].

The present work stems from the fact that the JWT standard does not require the inclusion of a nonce in the token construction, thus the server receiving the token cannot validate its freshness. Despite a claim named "nonce" was defined and register with IANA for JWT (see [16]), none of the JWT claims are mandatory, thus in practice such nonces are never used. This remark is the starting point for replay attacks and also for a more subtle attack that we describe in this work as *JWT Back to the future*. We show an abuse of the JWTs in the supply chain, where a malicious attacker that can manipulate the time perceived by the IoT device under production can obtain a set of valid JWTs that she can use in the future to authenticate custom messages sent to the Cloud. After the malicious JWTs are produced, the device has no recollection or logs of the happening, and so the abuse cannot be detected. Naively, without an SE, an attacker may simply steal the signing key by accessing the microcontroller internal memory in debug mode, during production. One of our main contributions is to show that the presence of an SE does not necessarily thwart the attack, and we present the few options currently available to protect a system against our finding in section 6.

In the following, we showcase our work by using the Arduino MKR 1010 and the Google Cloud IoT Core. This example is illustrated by Google in [19]. We specifically opted to use a now retired platform in order to avoid the abuse of our attack in real world scenarios. However, this work straightforward applies to other uses of the JWT, like for example HiveMQ or the EMQX platforms [13,17], and also to other token standards, for example the CWT [23], described by the Object Security for Constrained RESTful Environments (OSCORE) IETF work-

ing group [6,25,31] in the context of the RESTful environments for Authentication and Authorization for Constrained Environments (ACE) project. CWTs were used for example in the EU Digital Covid certificate, or in EAT tokens from [26]. Finally, we remark that our work also applies to the emerging technology of EVs as the standard used to transfer charge information to the station is based on the JWT and CWT standard and is described in [33].

Recently, Shingala [34] published an interesting comparison between the use of JWTs versus the use of the mutual authentication from the TLS [32] protocol for authentications of the things on the Internet. Interested by such work and from the internet discussions on-going, we decided to contribute with our findings concerning the use of JWTs for IoT.

Interestingly, we remark that our work does not apply to the ARM PSA tokens ([5]), as such tokens include a *mandatory* nonce coming from the caller, to demonstrate freshness of the generated token (see [5] Section 4.1.1).

The rest of this work is organized as follows: we introduce the concepts of JSON Web Tokens and related structures and security in section 2. Then section 3 describe the general IoT architecture that we use in the rest of the paper, while section 4 the different phases of the life of an IoT device in order to illustrate our threat model. In section 5 we present our attack Back2TheFuture and possible countermeasures in section 6. Finally section 7 concludes this work.

2 JWTs, JWSs and JWEs

In this section we present in more details the different aspects of the JWT, JSON Web Signature (JWS) and JSON Web Encryption (JWE) standards, which collectively goes under the JOSE acronym (for JSON Object Signing and Encryption), described respectively in RFC 7519, RFC 7515, and RFC 7516 ([21,20,22]). Then we describe other token types, as the CWT, and EAT, defined by the ACE group [25], and the RATS (Remote ATtestation procedureS) working group [26], respectively.

2.1 JWT

A JWT, described in RFC 7519 [21], is a compact, URL-safe means of representing claims to be transferred between two parties. The claims in a JWT are encoded as a JSON object that is used as the payload of a JWS structure or as the plaintext of a JWE structure, enabling the claims to be digitally signed or integrity protected with a Message Authentication Code (MAC) and/or encrypted. A JWT is divided into three fields: header, payload, and signature. The header provides information about the content of the JWT, like the cryptographic algorithms used in the other blocks. The payload contains the actual claims: a set of statements about an entity. Claims can be public or private and standardized by the IANA Web Token Registry or not. Finally the latter field consists of the signature of the previous blocks, encoded in Base64. These three fields, encoded in Base64, and separated by dots, compose the JWT. An example

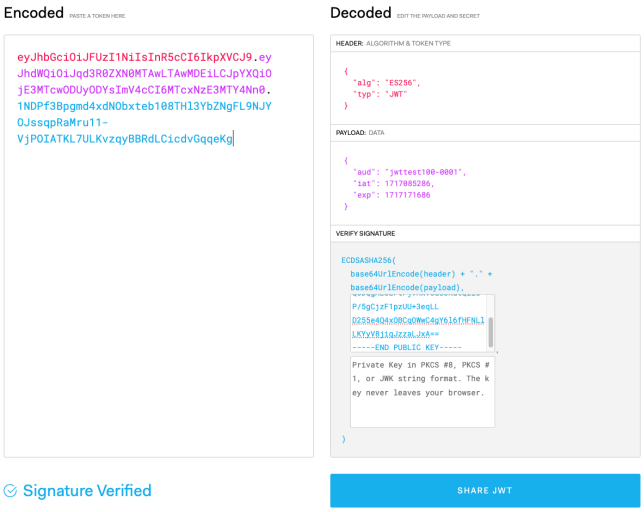


Fig. 1: Example of a legitimate JWT obtained by using an Arduino MKR WiFi 1010 and and decoded thanks to the https://jwt.io website. On the left is presented the base64 encoded value, while on the right are presented the decoded header and payload and the signature verification.

JWT from the site [30] is provided in Figure 1. The public key used to validate JWT in this work is the one indicated in subsection 5.1.

2.2 JWS

A JWS, described in RFC 7515 [20], allows the creation of a digital signature or Message Authentication Code (MAC) over a JSON-based data structure to ensure data integrity and authenticity. Cryptographic algorithms and identifiers for use with this specification are described in the separate JSON Web Algorithms (JWA) specification and an IANA registry defined by that specification.

The JWS does NOT protect the confidentiality of the data. Anyone who can get a hold of the JWS can decode and read the bytes that were signed. Related encryption capabilities are described in the separate JSON Web Encryption (JWE) specification.

2.3 JWE

A JWE, described in RFC 7516 [22], is a JSON-based data structure which protects the confidentiality of the payload by encrypting it and can sometimes ensure integrity of the payload data (depending on the algorithm chosen). A JWE can be used to store/transport sensitive data.

2.4 Sign then encrypt

JWTs can be signed then encrypted to provide confidentiality of the claims. While it is technically possible to perform the operations in any order to create a nested JWT, senders should first sign the JWT, then encrypt the resulting message. So, sign-then-encrypt is the preferred order for the following reasons: it prevents attacks in which the signature is stripped, leaving just an encrypted message; it provides privacy for the signer; finally, signatures over encrypted text are not considered valid in some jurisdictions. Certain papers advocate applying a second signature after the encryption [9]. This is not required with standard JWE algorithms due to their use of authenticated encryption [7].

In the following, we will use the term JWT to denote a JWT being it in plaintext, signed or encrypted (JWE, JWS). In particular, we will focus on the authenticity property, thus on JWSs, for which we demonstrate that the attacker can bypass the authenticity of some of the fields contained in the claims in section 5.

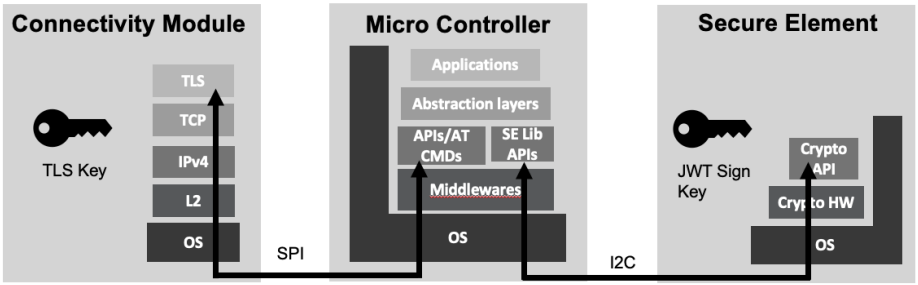
2.5 Other formats: CWTs and EATs

The ACE working group from IETF [25] is the responsible for the standardization of a solution framework to enable the protection of exchanges between a client and a server in a constrained environment. The method chosen by the working group to protect the exchanges uses tokens similar to the JWTs, but based on the CBOR format[8,11], called CWTs. A Go implementation for CWTs can be found for example in [1]. Furthermore, the RATS IETF working group developed a draft[26] in which they describe the EAT token ([26]) as a JWT or CWT token, with some attestation-oriented claims. Such tokens are used by a relying party, server or service to determine the type and degree of trust placed in the entity, like a smartphone, IoT device, network equipment and so on.

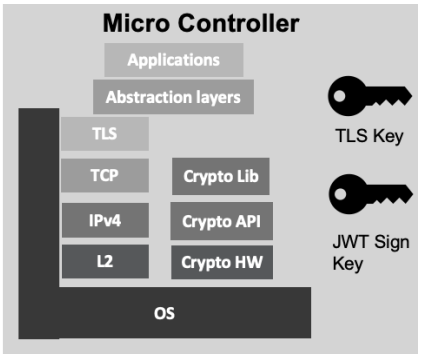
3 The IoT generic architecture

We consider a generic IoT environment, composed of a device which connects to the Cloud in order to securely send and receive data from the attached sensors. The device itself is composed of a micro-controller, which implements the application, and various modules in charge of different tasks. For example the micro-controller may implement the connectivity itself, or delegate it to an WiFi, BLE, or GSM module. Similarly, the micro-controller may use the cryptographic functionalities exposed by an SE.

In our scenario, we assume that the cryptographic primitives necessary for the security operations are provided by an SE module connected to the MCU, for example by the Inter-Integrated Circuit (I2C) channel [29], and not operated by the MCU itself. Such a situation is usually depicted as more secure, thanks to the presence of the SE and the assurance provided by it (see for example [14]). Then a WiFi module, for example connected to Serial Peripheral Interface (SPI)



(a) Modular architecture with different modules for the connectivity and security.



(b) Monolithic architecture with the connectivity and security on the MCU.

Fig. 2: Example of an IoT hardware and software stack architectures.

bus [27], is in charge of establishing the TLS connection to Cloud endpoint. This architecture is commonly realized by many IoT devices (e.g.: [35,12,28]).

As described in [14], the use of JWTs works as follows. The device will establish a secure connection to the global Cloud endpoint using TLS by using the WiFi module, but instead of triggering the mutual authentication it will generate a very simple JWT, sign it with its private key and pass it as authenticator. The JWT is received by the Cloud, the public key for the device is retrieved and used to verify the JWT signature. If valid, the mutual authentication is effectively established. As the SE offers the possibility to sign JWTs securely without ever exposing the private key, this scenario is generally considered more secure than in the absence of an SE.

We depict the two different architectures described above in Figure 2b and Figure 2a.

The latter architecture is realized for example by the Arduino MKR WiFi 1010, depicted in Figure 3. We use it in the following by connecting it to the Google IoT Core. Such a combination has been suggested in [14,2].

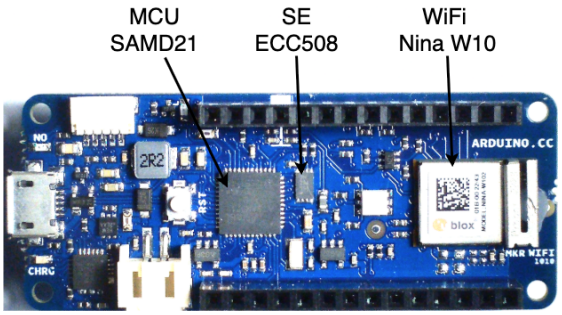


Fig. 3: Modules organization of the Arduino MKR WiFi 1010.

4 Device's lifecycle

This section presents an example life-cycle of an IoT device that we will use to illustrate the attack. The life-cycle of an IoT device is split into two main parts: manufacturing and deployment. Such splitting is useful to distinguish those steps which needs to be performed before and after the device is deployed in the field.

4.1 Manufacturing

During the first stages of life, the device is assembled, in particular all sub-modules are soldered together, the SE is connected to the micro-controller which is itself connected to the Wi-Fi module for communication. Afterwards, tests are performed on the complete device to verify all connections and functionalities. This step in general involves the generation of a cryptographic private key inside the SE on-boarded with the device, and the publication of the associated public key, together with an ID of the device, onto the Cloud Provider's fleet management system. Alternatively, it is possible that the SE comes pre-provisioned by the founder. For Google Cloud IoT Core, the instructions for device on-boarding can be found on different sources [18,28,10]. The final firmware is then loaded onto the micro-controller and thus the device is ready to go in the field. It is quite common that the manufacturing process is performed by a dedicated third-party company, where the use of secure manufacturing lines are not guaranteed. So the personnel having access to the devices during the manufacturing could be malicious and may try tampering, in particular, with the cryptographic assets. This aspect is commonly referred to as *supply chain attacks*, since it extends to all the steps of the supply chain, from component acquisition to manufacturing up to last mile shipping. During manufacturing, a supply chain attack can be problematic because the malicious attacker can act on the complete population of produced devices. Loading malicious firmware is the most naive supply chain attack. In such an environment, a *remote attestation* can be used to prove the device's software and hardware integrity to a remote party, providing cryptographic evidence that it is running as expected and has not been tampered

with. However there are attacks, such as the one we present here, that leave no evidence on the final device, and thus are hard to detect.

4.2 Deployment

After manufacturing is finished, the device is deployed in the field. From that moment on, every time the device needs to connect to the cloud to transfer data, the micro-controller connects to the Wi-Fi module trough the SPI channel, and requests the current time to a network time server. The current time thus retrieved is used to prepare a JWT. Claims in the JWT are generated by the micro-controller and signed by the SE. The signature generation is requested and returned through the I2C channel. The micro-controller then requires the opening of a TLS session with the cloud service to the Wi-Fi module, the interconnection between the micro-controller and the Wi-Fi module is ensured by the SPI channel. After the TLS session is established the micro-controller passes the signed JWT to the Wi-Fi module, to be sent to the Cloud service. Once the Cloud receives the JWT, it validates it and acts accordingly.

We want to stress that since the SPI and I2C channels are not protected, the micro-controller and the SE are not capable of understanding if the input data are coming from legitimate sources. This is true in particular for the signature request and data to be signed from the SE perspective, or time/date incoming from the SPI in the case of the micro-controller perspective.

4.3 Threat Model

This work considers a typical IoT device production scenario where a designer provides a factory with the hardware design and firmware. The factory's role is to manufacture a specified number of devices, each loaded with the designer's chosen firmware and secrets. It is assumed that the designer uses an SE to protect sensitive information and employs JWTs to ensure message authenticity between the IoT device and the Cloud after production.

For the purpose of this work, we can assume that the private key used for signing messages is the only secret, generated and accessible exclusively to the SE on the device. A key assumption is that factory employees are untrusted and could potentially tamper with devices during production to extract sensitive information like the signing key, by accessing the micro-controller internal memory in debug mode.

Specifically, attackers are assumed to be capable of eavesdropping and manipulating data on exposed communication channels (buses) before, during, and after firmware installation, to retrieve all secret material. However, the model excludes physical attacks on devices, such as side-channel or fault injection attacks.

In this context, the attacker's goal is to generate valid communication that the Cloud provider will accept, effectively impersonating legitimate IoT devices. The authors highlight that while JWTs are intended to provide authentication, they fail to protect against attackers who gain access to the private key. This

also implies that JWTs cannot protect against attackers who have access to the private key at any time, but also that even if the attacker cannot access the secret, the JWT construction fails to protect the communications that use the secret key.

In this perspective we want to remark two main aspects. First, if the SE is not present, then the attacker may simply steal the signing key by accessing the micro-controller internal memory in debug mode. Second, in order to thwart such a naive attack, an SE is usually mounted on the device. Although very surprisingly, using an SE in this context opens a new attack scenario (described in Section 5) which was not identified by the manufacturer nor the Cloud platform ([14]).

5 Back to the future

In this section we show that JWTs used in such context creates a security threat to the system. The JWT is created by the device based on its own time representation, while the Cloud side has no interaction in the generation of the JWT. This means that a JWT can be created at any time by the device or by anyone having access to the private key associated with the device. This represents a weakness in the authentication protocol, it can be used for example to mount replay attacks, by sending twice the same JWT.

We found a more subtle attack than a replay, and show in this section how to generate custom JWTs that will be valid in the future.

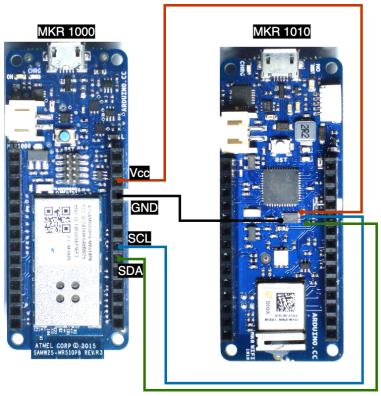
Our attack applies when the device is provided with an SE, being it pre-provisioned or not. An attacker having access to the device during manufacturing can interact with the SE and require the signature of a JWT for whatever moment in time. This is possible since the I2C between the micro-controller and the SE is not protected. Despite the possibilities for commercial SEs to establish a secure channel over the I2C with the micro-controller, we observe that this is not a sufficient countermeasure to thwart the attack presented in this work. As the key used to protect the I2C channel needs to be exchanged/rotated on the very same I2C channel on first boot, then an attacker that can eavesdrop such a communication would know the key and would thus be able to observe/modify all further communications.

Similarly, since the SPI connection between the micro-controller and the Wi-Fi module is not encrypted, an attacker can tamper with the time communicated by the Wi-Fi module as well. The attacker eavesdrops on the SPI channel and waits for the micro-controller to connect to a network time server. The attacker then alters the response, setting the time to some moment in the future. The micro-controller then receives the modified time and requests the signature on a JWT with a wrong time, to the SE. In this way, the micro-controller obtains a signed JWT with a custom time field ("iat"), which is controlled by the attacker, and can be used in the future.

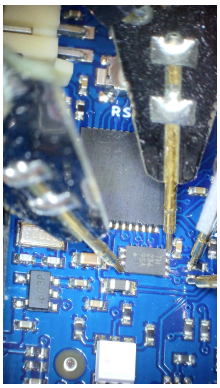
We observe that by targeting the Wi-Fi channel, only the JWT "iat" can be misused, and the attack is limited with respect to the one targeting the microcontroller-SE connection.

5.1 Attack

We present our attack by using the Arduino MKR WiFi 1010 [2] as victim's board. The attacker uses a second Arduino, the Arduino MKR 1000 WiFi, which is not provided of an SE, to abuse the SE of the victim's and make it generate a JWT for authenticating to the Cloud. As showed in Figure 3, the SE on the MKR WiFi 1010 is easily accessible. The experiment setup is depicted in Figure 4. It shows that the ground (GND), Serial DAta (SDA) and Serial CLock (SCL) signals are connected from the PINs of the attacker to the corresponding PINs on the victim' SE. In particular, from the SE's datasheet we can observe that the GDN, VCC, SCL, and SDA PINs are respectively PINs 4,8,6 and 7 on the SE. In order to ease the probes' positioning, we followed the SCL and SDA PINs to the nearest resistors, and connected the probes to them.



(a) Attack schematics.



(b) Probes positioning on the target.

Fig. 4: Attack schematics (a) and experimental setup (b).

We demonstrate our attack by using the sketch given by Arduino in [2] to connect the MKR to the Cloud Provider, in particular the version for the Google IoT Cloud. In order to emulate a pre-provisioned keypair, on the victim board we first load the Arduino ECCX08JWSPublicKey sketch [3] to generate the cryptographic material inside the SE. Further instructions on these procedures can be found in [4].

The obtained public key (necessary to validate the signature of the JWT) is:

```
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEoEPtFj
VHxvodsOKutQilCP/5gCjzF1pzUU+3eqLLD255e4Q4
xOBCqOWwC4gY6l6fHFNLlLKYyV8jiqJzzaLJxA==
-----END PUBLIC KEY-----
```

After registering the corresponding public key on the Cloud provider, we load the Arduino GCP_IoT_Core_WiFi [2] sketch to authenticate the node to the Cloud. This sketch executes the following steps: connects to the WiFi, retrieves the time information from the network, sign a JWT that contains the retrieved timing information in the "iat" claim, then use the signed JWT to authenticate to the MQTT server to publish and retrieve information. An example of a legitimate JWT generated by the victim is provided in Figure 1.

The attacker's board performs the same actions, but it cannot authenticate to the server as it has no SE connected. So it needs to "steal" it from the victim's board. In order to do so, she disconnects the power from the victim, and wire the two boards as depicted in Figure 4 (a). With the GND and VCC from the attacker to the victim' SE PINs, and the SDA and SCL signals to the corresponding signals of the victim's board.

Once this is done, the attacker can use the SE to sign any JWT of her choice, as if the SE was on its own board. So what she can do for example is creating a set of signed JWTs for the future, by manipulating the time, and store them for further use. We have performed such an attack, and edited the "iat" claim by setting it to a date far away in the future: `2077-12-02 17:26:22`. The corresponding obtained JWT is depicted in Figure 5.

We would like to stress that once the attacker obtains the JWTs, she doesn't need the devices anymore, and the JWTs can be used on any other device, i.e. a software tool executing an MQTT [36] client. Finally, as far as we know, if the attack is detected by the Cloud backend, the only possibility to mitigate it is to revoke all of the devices' keys. Alternatively, if the devices' private key can't be changed, dispose of all the devices.

5.2 A JWT weakness

JWTs were initially conceived to be used to authenticate users between interconnected servers, where all servers were controlled by the same entity and no (or little) clock skew was possible between them. The attack described above is the result of the use of the JWTs in a different context, where the token producer and the token consumer are different devices, under control of different entities. Thus, probably, the JWTs mechanism would require some adaptation in order to be securely applied to such use cases. Alternatively, one can argue that the problem is present due to a weakness in the JWT standards. Such weakness can be identified in the lack of communication between the Cloud and the device during the JWT generation. One of the building blocks of many security protocols is a random nonce generated by the party that verifies the claims and included in the signature by the party that wants to prove its identity. The lack

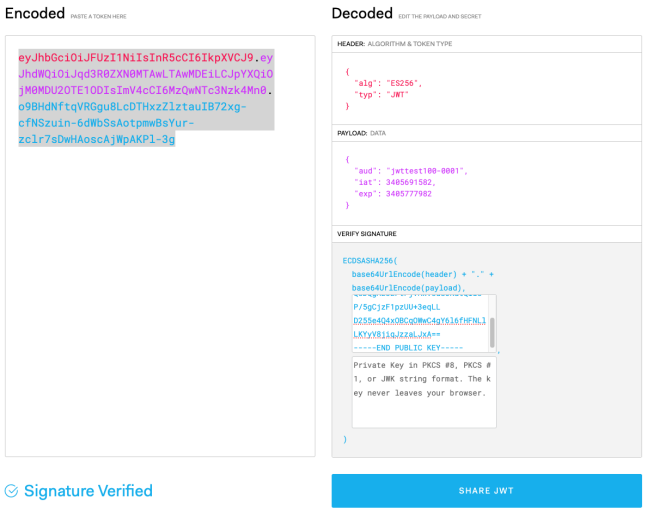


Fig. 5: Example of a JWT generated by the attacker with "iat" claim set to a date far in the future: 2077-12-02 17:26:22. Image taken from https://jwt.io website.

of such nonce in the JWT specification makes difficult to prove the protocol security, and opens the way to attacks as the one demonstrated in this work.

6 Countermeasures

In this section we present possible countermeasures that can be applied to thwart our attack. The applicability of the countermeasure obviously depends on the particular setup, device at hand, supply chain condition, resources, etc.

6.1 Use the TLS authentication

One simple solution is to use the TLS authentication mechanism instead of the JWT one. This has the obvious drawback that not all Cloud providers allows it, and of a heavier authentication mechanism. However it allows to thwart the attack as the nonces included in the TLS handshake prevents attackers to reuse previous commitments or generate JWTs valid in the future.

6.2 Pre-provisioned keys

A second solution would be to take advantage of the presence of pre-provisioned private-keys inside the SE.

For example, many commercial SEs contains a private key pre-provisioned by the founder (commonly referred to as *attestation key*). This countermeasure

assumes that such pre-provisioned keys are installed by a trusted party, in a trusted environment. The attestation key can only be used to sign data internal to the SE or data internal to the SE and some additional external bytes. The module also provides slots to store additional self generated private keys, which can be used to sign external data. We suggest the following protocol to thwart the attack described in this work. During provisioning, the public key corresponding to the attestation key is published to the Cloud. After deployment, the Cloud sends a random nonce to the SE. After receiving the nonce, the SE generates a private key in one of the free slots. Then, the SE uses the attestation key to sign the digest of the public key corresponding to the private self-generated key, and the nonce received from the Cloud. The signature is sent to the Cloud, which, upon verification, updates the public key to use during JWT verification with the one just received. The signature of subsequent JWTs can be signed by the self-generated private key. In such a way the Cloud can trust that the JWTs have been generated only after the nonce communication. This solution has the advantages of avoiding the need to implement a TLS stack on the micro-controller, and offering a robust mechanism to update the device's private keys in the field. The drawback of this solution is that a new step needs to be implemented in the Cloud provider's back-end, and the device must implement the new protocol. Another advantage of such countermeasure is that compromised keys can be revoked and compromised devices can be re-provisioned, simply by generating another signing key inside the SE and repeating the *nonce* procedure.

6.3 Use the nonce claim

A fourth solution to thwart the attack is similar to the previous one, but does not require the use of pre-provisioned keys in the SE module. It simply requires the Cloud to send the nonce to the device after deployment, prior to creating JWTs. Once received, the device includes the Cloud's nonce into the JOSE header of the JWTs. Afterwards, each JWT must contain the signed JOSE header containing the nonce, thus providing an assurance to the cloud concerning the generation time of the JWT. The main advantage of such solution is that it is the simplest solution to implement. The drawbacks involve a new step to be implemented in the Cloud's provider's back-end. Similarly to the solution with pre-provisioned keys, a further advantage is that compromised keys can be revoked and compromised devices can be re-provisioned by generating new sign keys and refreshing the *nonce* step.

6.4 Key use limit

A fifth solution takes advantage of the presence of monotonic counters in the SE, paired with a private key slot. Each time the key in the paired slot is used, the counter is incremented. By using an additional JWT field, the micro-controller can include the value of the counter in the JWT. The Cloud then verifies that the value in the counter is monotonic increasing, in order to mitigate the attack described in this work. The advantage of this solution is that it is simple and does

not require a TLS stack on the micro-controller. On the other hand, a new step is to be implemented in the Cloud provider's back-end, and a new field needs to be implemented in the JWT. Furthermore, differently from the solutions proposed in subsection 6.2 and subsection 6.3, this solution does not allow to revoke the used key if it is compromised.

6.5 Trusted supply chain

One final mention have to go to the simplest solution of all, at least from the technological point of view, which is of having a secure supply chain, or at least partially secure supply chain. Definition of a trusted supply chain is out of scope of this work, however we consider a supply chain to be trusted if it consists of a comprehensive and verifiable system encompassing all stages of an IoT device's lifecycle – from design and component sourcing to manufacturing, distribution, deployment, operation, maintenance, and eventual disposal – that ensures the security, integrity, and resilience of the device and its associated data throughout its entire existence. This would allow the first steps of the personalization of the SE to be performed in a trusty environment. The following steps, where the attack can not be mounted anymore, can be delegated to an untrusty supply chain. This solution, however have an intrinsic drawback of dividing the production into two different sites, the secure and insecure one, with obvious costs and troubles.

7 Conclusions

This work presents *JWT Back to the future*, the possibility of (ab)using an IoT device during production for preparing credential claims (JWTs) that will be later used by the malicious actor to connect to the Cloud service. A malicious user in the supply chain might have the possibility to collect a large number of JWTs for mounting a massive attack in the future.

The flaw is related to the lack of a mandatory nonce in the JWT standard. We show that it is possible to take advantage of such a flaw to abuse the authentication of IoT devices in the field, when JWT are used. In particular we show that an attacker in the supply chain may be able to make the IoT device generate some JWTs which are valid in the future and use them afterwards to impersonate the device with respect to the Cloud Provider.

Interestingly, we describe our attack against devices with different architectures. Showing that an SE attached to the micro-controller cannot protect from such an attack. We present a practical attack against an off-the-shelf IoT device by Arduino [2] and the mechanism used by the Google Cloud IoT Core to authenticate the devices. We demonstrate that an attacker on the production line can request a number of signatures on self generated JWTs, from the SE connected to the micro-controller, to be used in the future to connect to the Cloud.

We showcase our attack on the - now retired - Google Cloud IoT Core, in order to avoid malicious use of our findings, but our discovery can be applied to other services that provide token-based authentication. For example we further show that the same weaknesses applies to other tokens like the CWT and the EAT, and to platforms like HiveMQ and EMQX [13,17,26,23] providing a much wider attacker scope then merely a single token type or Cloud provider. Furthermore, our attack also applies to the OCMF standard, used for recording meter readings from charging station for EV [33].

In order to thwart the presented attack we provide a few countermeasures that can be applied, depending on the IoT infrastructure at hand. The simplest countermeasure is to use TLS authentication, while other countermeasures involve implementing a nonce-like mechanism in the JWTs or the authentication itself.

References

1. Keys, Algorithms, COSE and CWT in Go. `https://github.com/ldclabs/cose`.
2. Arduino. Arduino cloud provider examples. `https://github.com/arduino/ArduinoCloudProviderExamples`, 2019.
3. Arduino. Arduino eccx08jwspublickey. `https://github.com/arduino-libraries/ArduinoECCX08/tree/master/examples/Tools/ECCX08JWSPublicKey`, 2019.
4. Arduino. Securely connecting a mkr gsm 1400 to google cloud iot core. `https://docs.arduino.cc/tutorials/mkr-gsm-1400/securely-connecting-a-mkr-gsm-1400-to-google-cloud-iot-core/`, 2024.
5. ARM. ARM PSA. `https://datatracker.ietf.org/doc/html/draft-tschofenig-rats-psa-token`.
6. Victoria Beltran and Antonio F. Skarmeta. An overview on delegated authorization for coap: Authentication and authorization for constrained environments (ace). In *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, pages 706–710, 2016.
7. John Black. Authenticated encryption., 2005.
8. Carsten Bormann and Paul E. Hoffman. Concise Binary Object Representation (CBOR). RFC 8949, December 2020.
9. Don Davis. Defective sign & encrypt in s/mime, pkcs# 7, moss, pem, pgp, and xml. In *USENIX Annual Technical Conference, General Track*, pages 65–78, 2001.
10. Analog Devices. How to create a secure google to iot core connection with maxq1065. `https://www.analog.com/en/resources/app-notes/how-to-create-a-secure-google-iot-core-connection-with-maxq1065.html`.
11. Ericsson. ACE-OAuth – A new standard for lightweight authorization and access control. `https://www.ericsson.com/en/blog/2023/7/ace-oauth-standard-for-lightweight-authorization`.
12. Espressif. Esp32-wroom-32se. `https://docs.espressif.com/projects/esp-idf/en/release-v4.3/esp32/api-reference/peripherals/secure_element.html`.
13. HiveMQ GmbH. Step Up Your MQTT Security with JWT Authentication on HiveMQ Cloud Starter. `https://www.hivemq.com/blog/step-up-mqtt-security-jwt-authentication/`. Posted: 2024-03-18.

14. Google. Securing cloud-connected devices with cloud iot and microchip. https://cloud.google.com/blog/products/gcp/securing-cloud-connected-devices-with-cloud-iot-and-microchip, 2018.
15. Google. gcp-iot-core-examples. https://cloud.google.com/iot/docs/how-tos/credentials/jwts, 2023.
16. IANA. IANA JSON Web Token registered Claims. https://www.iana.org/assignments/jwt/jwt.xhtml.
17. EMQ Technologies Inc. Migrate Your Business from GCP IoT Core 03Use JSON Web Token (JWT) to Verify Device Credentials. https://www.emqx.com/en/blog/migrate-your-business-from-gcp-iot-core-03. Posted: 2022-11-28.
18. Google Inc. Best practices for running an iot backend on google cloud. https://cloud.google.com/architecture/connected-devices/bps-running-iot-backend-securely.
19. Google Inc. Google cloud platform iot arduino examples. https://github.com/GoogleCloudPlatform/google-cloud-iot-arduino, 2020.
20. M. Jones, J. Bradley, and N. Sakimura. JSON Web Signature (JWS). RFC 7515, RFC Editor, 5 2015.
21. M. Jones, J. Bradley, and N. Sakimura. JSON Web Token (JWT). RFC 7519, RFC Editor, 5 2015.
22. M. Jones and J. Hildebrand. JSON Web Encryption (JWE). RFC 7516, RFC Editor, 5 2015.
23. Michael B. Jones, Erik Wahlstroem, Samuel Erdtman, and Hannes Tschofenig. CBOR Web Token (CWT). RFC 8392, May 2018.
24. T. Lodderstedt, J. Richer, and B. Campbell. OAuth 2.0 Rich Authorization Requests. RFC 9396, RFC Editor, 5 2023.
25. Tim Hollebeek Loganaden Velvindron. Authentication and Authorization for Constrained Environments. Internet-Draft draft-ietf-ace-about, Internet Engineering Task Force, 2018. Work in Progress.
26. Laurence Lundblade, Giridhar Mandyam, Jeremy O'Donoghue, and Carl Wallace. The Entity Attestation Token (EAT). Internet-Draft draft-ietf-rats-eat-26, Internet Engineering Task Force, May 2024. Work in Progress.
27. Motorola. Single-Chip Microcomputer Data. https://archive.org/details/bitsavers_motoroladaSingleChipMicrocomputerData_68061538, 1984.
28. NXP. A71ch for secure connection to google cloud iot core. https://www.nxp.com/docs/en/application-note/AN12199.pdf.
29. NXP. I2C-bus specification and user manual. https://www.nxp.com/docs/en/user-guide/UM10204.pdf, 1 October 2021.
30. Inc. Okta. JWT.io. https://jwt.io.
31. Francesca Palombini and Marco Tiloca. Key Provisioning for Group Communication using ACE. Internet-Draft draft-ietf-ace-key-groupcomm-19, Internet Engineering Task Force, April 2024. Work in Progress.
32. E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, RFC Editor, 8 2018.
33. SAFE-eV. ECMF. https://github.com/SAFE-eV/OCMF-Open-Charge-Metering-Format/blob/master/OCMF-en.md.
34. Krishna Shingala. JSON web token (JWT) based client authentication in message queuing telemetry transport (MQTT). *CoRR*, abs/1903.02895, 2019.
35. Arduino S.r.l. Arduino security primer. https://blog.arduino.cc/2020/07/02/arduino-security-primer/, 2020.
36. OASIS Standard. Mqtt version 5.0. *Retrieved June*, 22:2020, 2019.