



Neustar Trusted Device Identity Technical White Paper

13 March 2017

Primary Contact:

info.iot@neustar.biz

Table of Contents

Table of Contents.....	1
Executive Summary.....	2
Traditional PKI versus TDI	2
Message Authentication at Scale.....	2
Recoverability	3
TDI Core Design Principles	3
Confidentiality.....	3
Integrity.....	3
Availability.....	3
Non-Repudiation (Integrity + Authenticity)	4
Auditability.....	4
Asymmetric Keys.....	4
The TDI Solution.....	5
TDI System Elements.....	5
Keys on the TDI System.....	7
Basic Message Flow with TDI	8
Updating Firmware with TDI.....	9
Revocation	10
Recommended Key Storage Systems.....	11
Message Formats	11
Header Parameters should include	12
Standard Claims (from Section 4 of RFC 7519)	12
Conclusions	13
References	14

Executive Summary

The Neustar Trusted Device Identity (TDI) system is an open-source platform backed by enterprise-class security and management. It ensures that every element in an IoT deployment has strong identity, securely passes messages to other elements, and takes action based on flexible, robust, and easily managed policies.

Modern IoT deployments require devices and services to communicate securely at scale. This requires decentralized, manageable solutions. Tiny devices with constrained memory, processing, and connectivity may still produce sensitive data or control critical equipment. These devices need to be able to participate in a secure ecosystem.

Unfortunately, many deployments have been implemented with weak or non-existent security, resulting in well-known attacks.

The cornerstone of secure communication is a secure identity. The Neustar Trusted Device Identity system starts with a secure, revocable Identity for every element in the system and builds on that, allowing developers to build secure and performant ecosystems across a wide range of device types.

Traditional PKI versus TDI

Traditional Public Key Infrastructure (PKI) tries to tackle the problem of securing IoT devices, but there are significant challenges with this approach due to scalability and recoverability in the face of eventual breaches. TDI provides a new approach to PKI that offers a scalable and fault-tolerant implementation for identity as well as complimentary security solutions such as anomaly detection and route validation when a device breach occurs.

In typical PKI implementations, the burden of validating certificates falls on message recipients. That means when a message comes from a server, the receiving device reaches out to check the certificate against a Certificate Revocation List (CRL) or via the Online Certificate Status Protocol (OCSP). This requires another round trip for the device, or distribution and storage of CRLs on devices.

With TDI, deployment recipients are given two public keys. Through identity abstraction, recipients can authenticate ANY sender simply by verifying a message's signatures against those two keys. It is the validation, not the identity, which is forwarded to the recipient. Removing the need to do an additional CRL/OCSP lookup optimizes the footprint for lower-powered devices while maintaining robust multi-factor authentication.

Message Authentication at Scale

TDI provides a way to securely and at scale authenticate messages between devices. The key piece here is scale. Traditional PKI has been stretched to secure IoT, but scalability and recoverability are hugely problematic. Traditional SSL-flavored PKI was designed for client-server (one-to-many) authentication, in particular the assertion of identity by resources to end users. There are recognized

security issues with this architecture. Applying the same paradigm to many-to-many authentication, such as in IoT deployments, presents significant challenges. Many-to-many authentication requires broad public key distribution, so key and CRL management becomes extremely complicated at scale.

Recoverability

Revoking single entities in traditional PKI deployments with shared private keys is problematic. PKI remediation typically requires rekeying everything in the deployment and updating firmware. Instead of sharing private keys across servers or adding complex key management, TDI centrally manages all of the public keys in a deployment. Identities validate with the TDI service and only the validation is forwarded to recipients, a process we call Identity Abstraction. This greatly simplifies the structure of PKI while leveraging the strength of public key cryptography.

TDI's centralized management and identity abstraction enables real time provisioning and revocation for all entities in the deployment including devices, users, services and applications. This architecture provides tremendous flexibility and enables rapid breach recovery if, for example, every active server in a deployment is compromised. In such a case, Admins can utilize the TDI management capabilities to revoke any compromised server and add new servers to the pool, all without any rekeying or distribution of public keys. The very next request will be able to utilize the newly added servers.

TDI Core Design Principles

The TDI system addresses the following [core principles](#) of secure system design:

Confidentiality

Messages sent using TDI are encrypted using Elliptic Curve Diffie–Hellman [ECDH](#), which allows any sender to encrypt a message for one or more specific identities. Only the intended recipient can decrypt that message. Furthermore, a single message can be encrypted for a list of recipients, if needed, without having to craft unique messages.

Integrity

Every message, whether encrypted or not, is signed using the Elliptic Curve Digital Signature Algorithm (ECDSA) with the message creator's private key. In addition, intermediates can add their signatures and metadata to the message. These signatures ensure the recipient that the message has not been altered along the way. In addition, messages can be co-signed by TDI, an independent authority that attests to the integrity of the message and its authenticity.

Availability

The cloud-based TDI servers employ redundant HSMs as well as layers of security, disaster recovery plans, and other industry-standard techniques to ensure high availability. [Neustar's SiteProtect](#) DDoS mitigation service and [UltraRecursive](#) DNS also ensure high availability.

Non-Repudiation (Integrity + Authenticity)

Because every element has a unique Identity, recipients are assured that messages truly originated from the sender and could only have come from the sender. The recipient is informed if the message is not authentic when a sender's private key has been compromised. In this case, the system detects a compromise with machine learning or when it is informed by a User or other trusted Identity that an Identity has been compromised (see Revocation below).

Auditability

Each element in the system can maintain its own record of received messages and whether they were verified or rejected. At a minimum, the TDI co-signing service will maintain a log of every message that it has been asked to co-sign for. This data is used in forensic analysis and Anomaly Detection to detect compromised elements.

Asymmetric Keys

Before diving deeper into the TDI solution, it is important to understand how the keys underlying the TDI system generally work. If you are familiar with [public key \(or "asymmetric"\) cryptography](#), you may skip ahead to see our application of these primitives.

Key Legend

 FLEET PRIVATE Key

 FLEET PUBLIC Key

 SERVER PRIVATE Key

 SERVER PUBLIC Key

 DEVICE PRIVATE Key

 DEVICE PUBLIC Key

 TDI PRIVATE Key

 TDI PUBLIC Key

Each entity has a secret "private" key used for signing and a widely shared "public" key used to verify signatures. This key pair is mathematically related. You could not, for example, just make a key pair from two random numbers. It is nearly impossible to determine the private key given the public key because it would take trillions of years of time and energy to crack one key using known methods.

A piece of data gets a “signature,” generated using the data owner’s private key. Anyone with the owner’s public key can verify that the data could only have been signed using the corresponding private key. The recipient therefore knows that the data came from the owner and only could have come from the owner. This holds true as long as the owner has exclusive access to its private key.

Another important feature of a signature is that any change to the data results in a very different signature. This is important because it ensures that even with marginal data modifications, the signature will not match and will not be verified against the unmodified data. Therefore, someone trying to manipulate messages along the way (change a vote, for example) will not be able to do so successfully.

The TDI system signs data in the form of messages that contain various key-value pairs (see [Message Format](#) below). Most of these are the relevant information for the message, but others are there to help beef up the integrity of the message. For example, there are expiration dates and identifiers, but most importantly, they each contain a unique “[nonce](#)”. This specially-formed token is unique, and once a receiver sees it, they will never accept a message with that same one again. This means that someone can’t capture a signed, otherwise well-formed message, and send it again later, say to add votes or turn on a light.

Finally, there is another very useful property of these keys. That is the ability to derive a shared [symmetric encryption](#) key from my private key and your public key. Fortuitously, the math that let’s us do this works both ways, so when you do the same using your private key and my public key, the resulting value is identical! This is perfect for encrypting messages, because no one but the two parties involved will be able to generate the same secret key, and it can be done without ever having to share secret information (say a password) that someone else could steal.

The TDI Solution

Each entity in the TDI system is associated with a strong Elliptic Curve Cryptography ([ECC](#)) key, securing the Identity of that element. Ideally, these keys are maintained in a secure enclave that will not expose the private key. In practice, this means that the keys are kept in one or more redundant Hardware Security Modules ([HSMs](#)), and Devices will utilize tamper-proof memory, co-processors, Trusted Platform Modules, and/or apply other secure methods.

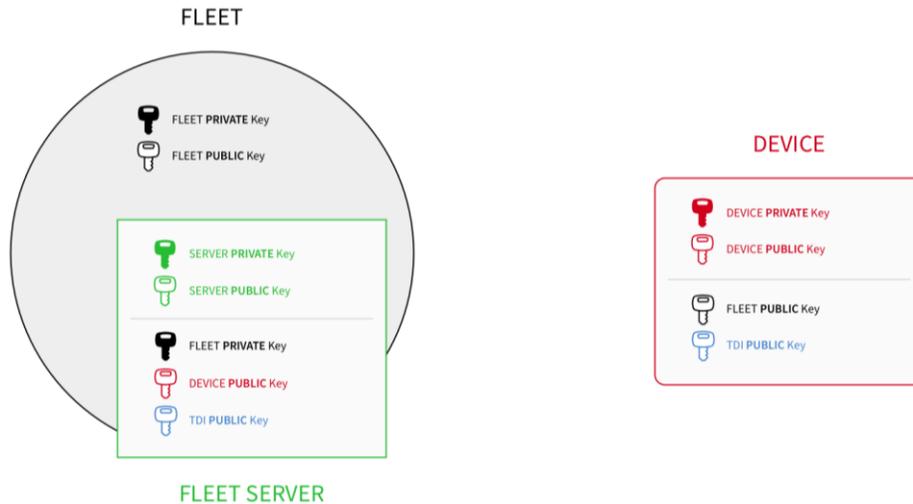
TDI supplies an embedded development toolkit that deploys trusted messaging between devices, servers, and gateways. Each entity has a key pair comprised of a public and private key.

TDI System Elements

At the core of a TDI deployment are devices and associated services, each with their own private signing key and public verification key. These are broken out into four key elements:

- A **Fleet** defines the scope of the deployment. Typically, a Fleet is comprised of devices with a certain SKU and those devices’ supporting services. The defining characteristic of a Fleet is the *Fleet Signing Key*, whose public key is recognized by all elements of the system and represents a Fleet’s base authority and identity.

- One or more **Fleet Servers** are services that sign messages on behalf of the Fleet. Typically they will be in data centers or cloud servers and have access to an HSM that holds the private *Fleet Signing Key* over a secure, private channel. Each Fleet Server has its own identity and ECC key pair as well. The separate roles of the *Fleet Server Key Pair* and the *Fleet Signing Key Pair* are described in the next section.
- Messages receive a second signature from the **TDI Fleet Co-Signing Server** to strengthen the integrity of the message and its authenticity. The TDI Fleet Co-Signing Server retains its key pair as well as the public keys for the Fleet Server and Devices.
- **Devices** are entities within a Fleet. A Fleet can have a few Devices or a few million. Each Device has its own unique private key and easy access to the public *Fleet Signing Key* and the *TDI Co-Signing Key*.
- Optional **Application Services**: Most applications will include services that don't require direct access to the public *Fleet Verification Key*. These applications may provide Fleet-level services, or they may be distributed (on premises or remote) for performance or other reasons. In many cases, the Device owners will want to deploy or implement systems that require secure interaction with Devices and the Fleet. The Application Services may have their own keys, or they may act on behalf of Users or Devices.



Neustar Trusted Device Identity (TDI)



Keys on the TDI System

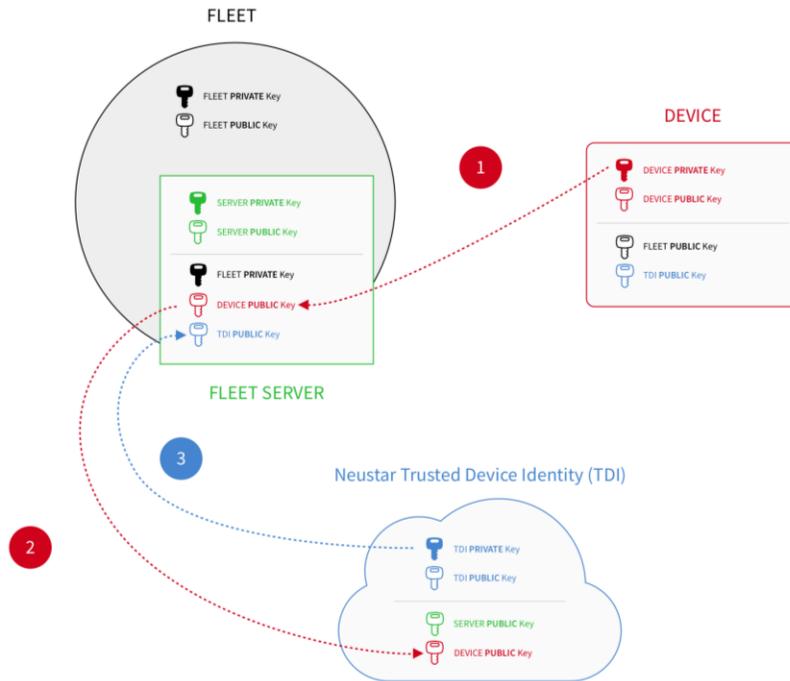
As mentioned above, a unique ECC (P-256) key pair embodies every Identity in the system. Each entity securely holds its own private key. The public key is communicated as needed, though is generally well-known throughout the system. Of course, not every Device keeps a record of every Identity in the system, so it relies on the Fleet and TDI's Fleet Co-Signing Server to verify identities.

In addition to the individual element keys, there is the *Fleet Signing Key* that the Fleet Servers use to sign messages on behalf of the Fleet and the *TDI Signing Key* that TDI uses to co-sign messages also on behalf of the Fleet.

Note that the Fleet Servers are unique because they have access to two private keys: the *Fleet Signing Key* and their own *Fleet Server Signing Key*. This enables a unique feature of a TDI-enabled deployment: Devices and other elements verify messages signed by any Fleet Server without having to know about every Fleet Server's individual Identity. This is important because Fleet Servers may come and go without having to update millions of Devices, Users, Application Services, etc. As load and application demands change, new Fleet Servers can be added to a deployment. If a Fleet Server is compromised, it can be revoked (see [Revocation](#), below) without having to re-key every Device, User, and Application Service in the system.

Basic Message Flow with TDI

To illustrate how messages are signed, co-signed and communicated, consider the following simplified diagram:



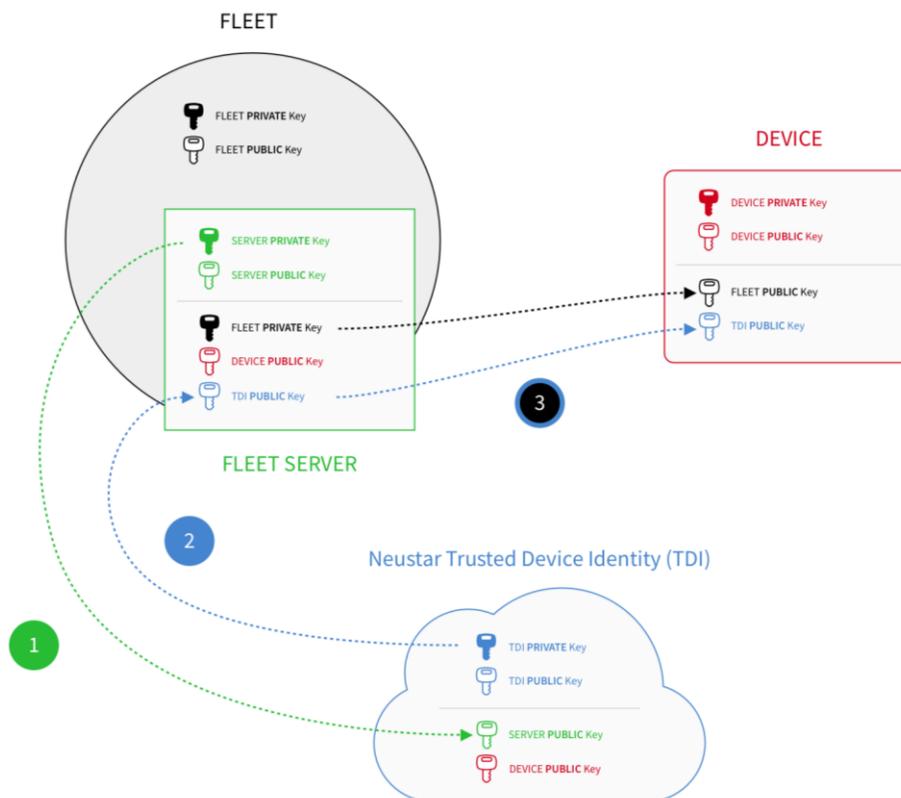
A Device sends a message to a Fleet Server, perhaps some sensitive telemetry data. The Fleet ensures that the reporting Device is authentic and that the message has not been forged or modified. The simplified flow is:

1. The Device generates a message and signs the message with its private *Device Signing Key*. The message is sent to one of the Fleet Servers. The Fleet Server:
 - i. Verifies the signature against the public *Device Verification Key*.
 - ii. Unpacks the message and verifies certain standard claims (see [Message Format](#) below).
2. The Fleet Server sends the message TDI's Fleet Co-Signing Server to get an additional signature. The TDI Fleet Co-Signing Server:
 - i. Checks that neither the Fleet nor the Device have been revoked
 - ii. Verifies the signature against the public *Device Verification Key*
 - iii. Unpacks the message and verifies the standard claims
 - iv. Signs the message using the TDI's *Fleet Co-Signing Key*
3. TDI's Fleet Co-Signing Server sends the co-signed message back to the Fleet Server. The Fleet Server:
 - i. Verifies the TDI co-signature against the TDI's public *Fleet Co-Signing Verification Key*
 - ii. Stores the now-validated data in the server

Of course, in any of these steps a failure might occur: a signature not validating, an entity found to be revoked, or a standard claim found invalid. In any of these cases, processing stops at that point and the error is logged.

Updating Firmware with TDI

In this second example, the Fleet wants to send a firmware update to one or more Devices. The Fleet sends a message that includes metadata about the upgrade, a hash or checksum of the image, and the location of the image:



1. A Fleet Server produces the firmware update message and signs it with its private *Fleet Server Signing Key*. The message is sent to TDI's Fleet Co-Signing Server. The Co-Signing Server:
 - i. Checks that neither the Fleet nor the Fleet Server have been revoked
 - ii. Verifies the signature against the public *Fleet Server Verification Key*
 - iii. Unpacks the message and verifies the standard claims
 - iv. Adds a signature to the message using the *TDI Fleet Co-Signing Key*
2. The Fleet Co-Signing Server returns the message to the Fleet Server. The Fleet Server:
 - i. Verifies TDI's co-signature against the public *Fleet Co-Signing Verification Key*
 - ii. (optionally) removes its own signature from the message and re-signs with the private *Fleet Signing Key*
3. The message is sent to the Devices. A Device receiving the message
 - i. Verifies the Fleet signature against the public *Fleet Verification Key*, and the TDI co-signature against the *Fleet Co-Signing Verification Key*
 - ii. Unpacks the message and verifies certain standard claims
 - iii. Acts on the message. In this case, the device downloads the firmware update, checks the package against the hash, and installs it.

Note that in this case, the Device never had to communicate with TDI, yet it had TDI's signature as assurance that the message was a valid one from the Fleet. It also didn't need to know anything about the specific Fleet Server that sent the message. The next update could come from another Fleet Server. The Device does not have to retain the Identity of any firmware update server in the Fleet.

Also, note that the Device never needed to access the internet to process this message, assuming internet was not required to download the firmware update.

Expanding these two examples presents more complex application use cases, such as:

- Local Gateway Device connected to non-internet-connected Devices
- User and Application Services authority
- Complex multi-Fleet applications

Revocation

We have referred to revocation or revoked elements in a few places in this document. Here we will describe what this means in more detail.

TDI's Fleet Co-Signing Server maintains a database of Fleet elements: the Fleet, Fleet Servers and Devices. It records the revocation status of each element, in addition to other metadata. Thus, TDI knows whether a given element is revoked by looking at the element's revocation status and the Fleet's revocation status. Neither can be revoked for co-signing to succeed. Furthermore, the entire Fleet can be revoked, which subsequently revokes each element in the Fleet. Limited to select Fleet Admins, this "big red button" is a scram safety measure yet is easily reversed.

Revocation is part of the TDI co-signing service and is built into the Fleet Servers. Because revocation is an important aspect of the system and fairly easy to implement, it should be included in any Fleet Server or Application Services in a deployment. Maintaining a similar flag for its elements and pushing upstream establishes redundant entry points for revocation.

For example, a Gateway Device detects a compromised Device and notifies a controlling Application Service, which revokes the Device in its tables and calls a Fleet API to revoke the device from the Fleet. The Fleet then calls TDI's Fleet Co-Signing Server API to revoke the Device. If for some reason only one of those points were accessible, revocation at any point would ensure that messages from the revoked Device would not be fully validated.

Reversing a revocation (or "unrevoking") has a different set of permissions in the TDI co-signing system. Different Fleet Admins could have unrevoking authority while a larger group (and automated agents) may only have permission to revoke. This allows quick response to certain attacks, while allowing measured *post hoc* diagnosis, analysis and restoration of specific elements or the Fleet as a whole by designated, trusted individuals.

Recommended Key Storage Systems

As discussed earlier, private keys must be kept as secure as possible to ensure that only the represented entity can sign with those keys. For computer systems in a secure data center, the best way to implement private key storage is in a high-end HSM, which generates and keep private keys internally with no way to access them directly. The HSM signs messages and encrypts messages as described above. Any attempt to physically break into the HSM will disable it, potentially destroying the keys. Using HSMs requires some expertise to ensure that only authorized entities can sign with keys they are allowed to use. It doesn't help to have a strong fortress with a screen door. HSMs should be connected through physically secure channels with redundancy and backups in place. Cloud and virtual HSMs may also be a reasonable solution, especially for cloud-based Fleets.

At the other end of the spectrum, small devices need similarly secure ways to store keys. Chips like Atmel's [ECC508A](#), or Maxim's [DS2476](#) are relatively cheap and provide many of the same functions as an HSM but at the board level. Some include tamper-protection but require care to ensure that access to the signing machinery is only from the intended software, both at the system level, through the use of [Trusted Execution Environments](#), at the board level, using a physical build technique that will render the chip useless if an attempt is made to probe its control inputs, or pry it from the board.

Note that these caveats present somewhat of a dilemma for system designers. While it is important to keep keys secure, the cost needs to be appropriate given the risk of compromise. The physical protection of a chip on a board is ideal, yet the compromise of a single device through physical means also requires physical access to one device at a time. This kind of attack cannot be weaponized. For many cases, the focus on detection of breach may be more helpful than the prevention of it.

Similarly, the choice to destroy keys rather than let them fall into the wrong hands is another system design choice, and it may be decided differently in different parts of the system. In some cases it may make more sense to rely on detection and revocation, the "live to play another day" approach. This is generally a reasonable approach for low-end distributed devices where attack at scale would be required. Alternatively, the "death before dishonor" approach may be more reasonable for things like Fleet Keys, where the release of this key could result in the ability to control millions of devices. Although even in this scenario, revocation and recovery can still be used to thwart attacks and prevent the system from falling under the control of an adversary.

Message Formats

The TDI system assumes that all messages are [JSON Web Signatures](#) (JWSs), with a format of their payloads that follows the [JSON Web Token](#) (JWT) specification. It requires that the signature algorithm used is ECC with a NIST P-256 key (alg: ES256).

A JWS consists of a payload, which is a sequence of arbitrary bytes, along with an array of signatures, which each include a header, that includes metadata for the signature, and the signature itself, which is a sequence of bytes encoded into base64url format without padding (trailing "=" characters removed).

For TDI messages, the payload is a set of "claims", per the JWT specification. These claims are arbitrary name-value pairs, although certain names are reserved. TDI expects and, in some cases, enforces some of these, especially around expiration (nbf and exp) and uniqueness (e.g. jti as a nonce).

Messages can be in either JWS "Compact" or "JSON" serialization format. Those serialized into Compact format with only one signature will be valid JWTs. This may be useful for using them in HTTP headers, etc., where JSON would not be allowed or practical. They are also designed to be URL safe.

The TDI SDKs are designed to allow for expansion of Compact JWSs into JSON JWSs when signatures are added. Since a compact JWT can only contain a single signature, the inverse is not possible.

As messages pass along a route, depending on the design decisions made, entities will add signatures to the message. The payload will not be altered, but different entities may include different information in the [JOSE](#) header associated with its signature. If the payload does need to be altered, a Nesting strategy may want to be employed (see below).

Because the order of signatures in a JWS is not defined, or directly cryptographically verifiable, we need another way to report the order that can be trusted and verified. One way this can be done is to include a unique and increasing "index" value in each signature header. Any entity verifying the message would verify the uniqueness and proper format of this index value, and any entity doing route validation would use this as the indication of order. Note that it may make sense to allow the originating Device to not add this index, making it equivalent to adding a value of zero (0). But if any other entity failed to include it, the message would fail to validate.

Messages may also be encrypted, and encoded in the [JSON Web Encryption \(JWE\)](#) format.

Header Parameters should include

- typ
 - Required
 - "JWT" for JWTs (or header extracted when extending a JWT into a JWS)
 - "JOSE+JSON" for JWSs
- alg
 - Required
 - Always "ES256"
- kid
 - Optional for JWT, required for JWS (except for header from JWT extended into a JWS)
- iss, sub, aud
 - Optional
 - May be required if claims are encrypted, but MUST be equal to equivalent claims per [Section 5.3](#) of [RFC 7519](#)

Standard Claims (from [Section 4](#) of [RFC 7519](#))

- iss
 - Required

- The UUID of the entity making the claims
- Should match one of the header kid values, but may not if signing was done by a proxy, Gateway Device, etc.
- sub
 - Optional
 - The identifier of the "subject" of the message, if different than the issuer or recipient
 - Although [RFC 7519](#) says that this is the entity to which the claims apply, we may more generally use this as a "target" of a request from the issuer, and the claims may be about either, both or something related to the relationship. The interpretations of the claims will be made in context
- aud
 - Optional
 - The identifier of the recipient(s) of the message, if different than or a superset of the issuer or audience
 - May be an array of identifiers
- jti
 - Required
 - A nonce unique to this message
- nbf
 - Required
- exp
 - Required

Conclusions

A secure identity is crucial to secure communication. TDI builds upon this core principle and offers a scalable solution for secure, revocable identities for every entity in the system. Traditional PKI has been employed to tackle the problem of securing IoT devices, but there are significant challenges with this approach due to scalability and recoverability in the face of eventual breaches. TDI provides a new approach to PKI that offers a scalable and fault tolerant implementation for identity as well as complimentary security solutions such as anomaly detection and route validation when a device breach occurs.

There are multiple problems with managing and securing devices that can act autonomously or are headless and must be controlled remotely. Establishing a level of security for these devices is critical. A small seemingly non-critical piece of equipment can be used as an entry point to a larger system. TDI resolves security breaches quickly and economically. Moreover, making TDI solution available open source supports faster and wider adaptation to support more deployments.

References

1. "Information Security." *Wikipedia*, en.wikipedia.org/wiki/Information_security#Key_concepts. Accessed 13 Mar. 2017.
2. "Elliptic curve Diffie–Hellman." *Wikipedia*, en.wikipedia.org/wiki/Elliptic_curve_Diffie%E2%80%93Hellman. Accessed 13 Mar. 2017.
3. "DDoS Protection Solutions." *Neustar*, neustar.biz/security/ddos-protection. Accessed 13 Mar. 2017.
4. "Recursive DNS." *Neustar*, neustar.biz/security/dns-services/recursive-dns. Accessed 13 Mar. 2017.
5. "Public-key Cryptography." *Wikipedia*, en.wikipedia.org/wiki/Public-key_cryptography. Accessed 13 Mar. 2017.
6. "Cryptographic nonce." *Wikipedia*, en.wikipedia.org/wiki/Cryptographic_nonce. Accessed 13 Mar. 2017.
7. "Symmetric-key algorithm." *Wikipedia*, en.wikipedia.org/wiki/Symmetric-key_algorithm. Accessed 13 Mar. 2017.
8. "Elliptic curve cryptography." *Wikipedia*, en.wikipedia.org/wiki/Elliptic_curve_cryptography. Accessed 13 Mar. 2017.
9. "Hardware security module." *Wikipedia*, en.wikipedia.org/wiki/Hardware_security_module. Accessed 13 Mar. 2017.
10. "ATECC508A In Production." *Microchip*, microchip.com/wwwproducts/en/ATECC508A. Accessed 13 Mar. 2017.
11. "DS2476 DeepCover Secure Coprocessor." *Maxim Integrated*, maximintegrated.com/en/products/digital/memory-products/DS2476.html. Accessed 13 Mar. 2017.
12. "Trusted execution environment." *Wikipedia*, en.wikipedia.org/wiki/Trusted_execution_environment. Accessed 13 Mar. 2017.
13. Jones, et al. "JSON Web Signature (JWS)." *Internet Engineering Task Force*, May 2015, tools.ietf.org/html/rfc7515. Accessed 13 Mar. 2017.
14. Jones, et al. "JSON Web Token (JWT)." *Internet Engineering Task Force*, May 2015, tools.ietf.org/html/rfc7519. Accessed 13 Mar. 2017.
15. Jones & Hildebrand. "JSON Web Encryption (JWE)." *Internet Engineering Task Force*, May 2015, tools.ietf.org/html/rfc7516. Accessed 13 Mar. 2017.