Electronic Design

STEPHANE DI VITO | Senior Security and Software Engineer, Maxim Integrated,
https://www.maximintegrated.com

Protect TLS in IoT Devices with Secure Companion ICs

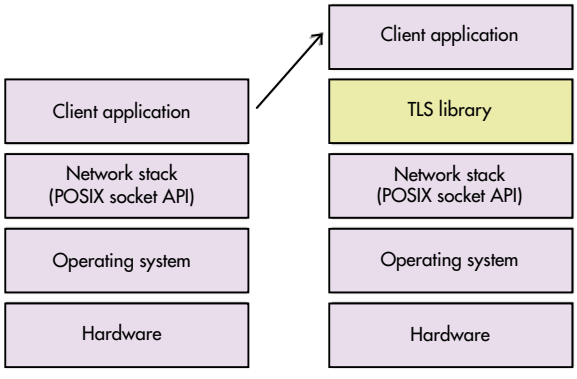
More connected devices means more exposure to personal, sensitive, or mission-critical data that can be disclosed and/or tampered with. Thus, it's essential to protect the data in transit and data at rest.

The proliferation of connected devices such as industrial sensors/actuators, utility meters, home appliances, wearable/medical devices, metering network gateways, connected toys for kids, etc. increases the exposure of personal, sensitive, or mission-critical data to disclosure and/or tampering. Therefore, it's essential to protect the data in transit and data at rest.

In terms of protecting data in transit, the most used protocol is TLS (Transport Layer Security, formerly known as SSL, Secure Socket Layer). Initially created for bidirectional secure communications over the internet, between computers and web sites, it's now a must-have for securing the communication of IoT devices over the internet. It prevents eavesdropping or tampering of data in transit.

Because the TLS protocol has been vastly studied, attacked, and fixed, let along being widely adopted, it's become quite robust. Full use of the TLS protocol relies on some assets, though: It requires storing of private keys that must never be disclosed and/or modified without strict control, and storing of certificates that must also never be modified unrestrainedly in the IoT devices.

However, since those IoT devices are deployed into the wild, the assets are exposed to attackers who may attempt invasive as well as non-invasive attacks. Invasive attacks consist of opening the enclosure of the device with the intent of manipulating memory content, replacing the firmware, or probing PCB traces. Non-invasive attacks (usually performed through communication ports on the device) target logical bugs in the device's firmware. Indeed, any software bug may be a security vulnerability, allowing a diversion of the system from its intended behavior. All of those attacks are aimed at discovering some keys, tampering with certificates, or tampering with the firmware itself to bypass some security features. Protecting the TLS implementation without a secure IC companion is an impossible challenge.



1. Integrating TLS in your software requires adding the TLS library. Note: This is a rough view of what has to be done. It's actually a little bit more complicated; all details can be found at https://tls.mbed.org/kb/how-to/mbedtls-tutorial.

After a quick description of the TLS protocol, this article will explain in more detail what assets are in danger. Then, the discussion will turn to how to secure this protocol's implementation in a connected embedded system, using a low-cost and low-complexity solution, while reducing the burden on the devices' application processor.

THE BASICS OF PUBLIC KEY-BASED DIGITAL SIGNATURE

Let's consider a "sender" wanting to transmit a message M to a "receiver" (M is a sequence of bytes). The receiver wants to make sure that the message comes from the genuine sender, unmodified.

In the following, we will consider exchanges of messages between the "sender" and the "receiver." The names of the different items on the receiver's end are postfixed with "_r" (for "received"). That's because they may not be the same as on the sender's end, due to error transmissions or attempts to

tamper with the values by an attacker. All of the items on the sender's end are postfixed with "_t" (for "transmitted").

SIGNATURE AND SIGNATURE VERIFICATION

On the sender's end, a message M_t is prepared for transmission:

1. The message M_t is first hashed, using a one-way secure hash function such as SHA-256 (mandated nowadays *). Note that a hash function "condenses" an arbitrary message into a fixed-size value (32 bytes long for SHA-256) with those important facts verified:

• It's impossible to find or compute two different messages giving the same hash value (no collisions).

• Knowing a hash value, it's impossible to find out the matching message (one-way), except via brute-force, which is practically impossible.

• A given message always produces the same hash (deterministic).

2. The resulting hash H_t then goes through a signature algorithm (e.g., ECDSA) involving the sender's pair of keys: a private key SprivK_t and a public key SPubk_t. The result of the execution of this signature algorithm over H_t is the signature of the message: S_t. Note: The private key SPrivK_t must be kept secret by the sender and inaccessible to the others. Otherwise, it becomes impossible to bind that key to the sender.

3. The sender then transmits the triplet (M_t, S_t, SPubk_t) to a receiver.

On the receiver's end, the triplet (M_r, S_r, SPubk_r) is received (message, signature, public key). In principle, it should be equal to (M_t, S_t, SPubk_t), but due to transmission errors, or voluntary modifications, it may not be.

Now the sender wants to verify the signature of the received message M_r.

1. To this end, M_r is hashed again with the same hash algorithm.

2. The resulting hash H_r goes through a signature verification algorithm involving the sender's public key SPubK_r.

3. The algorithm outputs the result of the verification: "good" or "not good." "Good" implies that M_r was signed using SprivK_r and no other private key. Consequently, only the owner of the private key SprivK_r can be the originator of the message (authenticity). Besides, "good" means that M_r is identical to the message signed using SprivK_r (integrity); it has undergone no modification since it has been signed.

As SprivK_t_ and SPubk_t are mathematically bound (but knowing SPubk_t doesn't allow one to find out SPrivK_t), if a signature is found to be correct using SPubk_t, then one knows that the signature was computed using SprivK_t and no other key.

But the above signature verification isn't sufficient per se. To really guarantee the authenticity of the message, the public key used to verify the signature (SPubk_r) must be traced back to the sender.

If an attacker catches the triplet of the genuine sender (M_t, S_t, SPUbk_t) during transit and silently replaces it with another correct, but evil, triplet (M'_t, S'_t, SPubk'_t) signed using SPrivK' (owned by the attacker itself), then the receiver will find the triplet (M_r, S_r, SPubk_r) = (M'_t, S'_t, SPubk'_t) as authentic and not corrupt because SPubk'_t matches S'_t . However, the message was neither sent nor signed by the correct sender!

CERTIFICATE VERIFICATION (PKI)

It's vital for the receiver to verify that SPubk_r belongs to the expected sender (i.e. SPubk_r = SPubk_t). When there's a single sender, this is quite trivial: It suffices to store SPubk_t on the receiver's end and use the stored version rather than the received version of the public key for the verification of incoming signed messages.

However, in a networked environment, many senders may want to send messages to many receivers. This is why public key infrastructures (PKIs) have been created. In a PKI, public keys are transmitted by senders to receivers within certificates. Certificates are issued by certification authorities.

Specifically, certificates are data sets that contain the full identity of the sender (usually the domain name of the server, e.g., "www.domainname.org," the name, address, country, and phone number of the owning organization), the identity of the certification authority who issued the certificate, and the public key of the sender. The whole certificate is digitally signed by the certification authority (following the same process as in our example above, but the message M_t is replaced by the certificate and the signing private key SPrivk_t is the one owned by the certification authority) and the resulting signature is appended to the certificate.

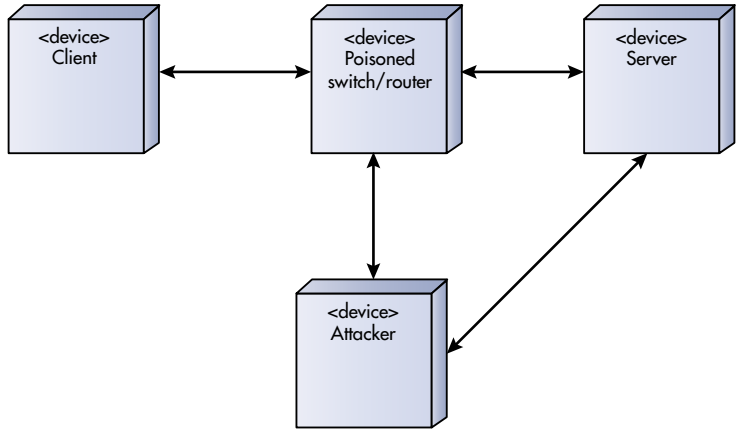
Now, when a receiver receives (M_r, S_r, Cert_SPUbk_r), he/she first needs to verify that the certificate Cert_SPUbk_r is valid before using SPUbk_r to verify the message's signature. To this end, the receiver uses the certificate of the issuer of Cert_SPUbk_r and proceeds to verify an incoming message (the message being Cert_SPUbk_r). If the verification is successful, it means that:

1. The sender's identity information found in Cert_SPUbk_r has not been tampered with.

2. The public key SPUbk_r contained in the certificate is the one of the certificate's owner, whose identity is stated in the certificate.

In this case, no attacker can impersonate a sender, because it's impossible to forge a valid certificate with arbitrary identities and public keys inside because the certification authority's private key is kept secret. But how do we validate the certification authority's certificate? A few intermediate certificates can be involved (a chain of certificates), but to

put an end to this chicken-and-egg issue, a root certification-authority certificate must ultimately be trusted. It means that the verifier must have the root certification-authority certificate stored in a safe place beforehand.



2. A man-in-the-middle attack consists of diverting the communication and interfering between the legitimate client and servers. The man in the middle authenticates with both parties, who think they're actually connected to each other. The man in the middle can eavesdrop messages and even modify them.

DESCRIPTION OF THE TLS PROTOCOL

The TLS protocol is a rich and complex protocol, but let's try to provide a 10,000-foot overview. The purpose of this section is to exhibit what assets are essential to the security of the TLS protocol.

TLS happens between two parties: a client and a server. For the sake of this article, the "client" will be an embedded system (e.g., an IoT device) and the "server" will be a remote machine in the cloud, reachable via the internet. Therefore, let's imagine a "client" device that has sensors and actuators, and it's connected to a web service provided by the "server." As a result of this, the user can, from his or her smartphone, look at sensor data and send commands to the device via the server. The client device uses TLS to report data or fetch commands to/from the server.

The TLS protocol has two main phases:

1. Establishment (handshake)
2. Secure communication (secure transport of application data with authentication and encryption)

ESTABLISHMENT (HANDSHAKE) PHASE

The handshake phase is a negotiation of the secure communication's configuration, followed by the authentication of the server and the client, and then the computation of a shared secret. Here are the four steps of this phase in details:

1. *An agreement on the cipher suite to use* (a cipher suite is the collection of cryptographic algorithms that will be used in subsequent phases below). The server and the client decide which set of cryptographic protocols will be used in the subsequent phases: negotiation of the shared secret keys and the security of the application communication.
2. *The authentication of the server by the client.* During this phase, the client sends a message made of a random sequence of bytes to the server. In return, the server sends back its own certificate to the client, together with the digital signature of this random message, calculated using the server's private key. The client then verifies the signature of the message using the server's certificate (after having validated this certificate).
3. *Optionally, the authentication of the client by the server.* This follows the same process as for the server authentication above, and proves useful in embedded systems. Indeed, standalone client devices may not "enter their password" to authenticate to the server, as is commonplace on web sites. To remedy this, the TLS handshake has an option to ask clients to prove their identity by asking them for a digital signature using their client private key (following exactly the same scheme as for the server authentication in phase 2).
4. *The negotiation of shared communication secret keys used for the secure communication phase* (e.g. AES keys). In this fourth phase, secret keys can be exchanged using complex public key-based key exchange protocols (e.g., ECDHE). Those protocols allow computing of the same secret (also called symmetric) key on both the client and the server side without actually transmitting it. The secret keys resulting from the key exchange are then used in the next phase to encrypt and then sign the messages exchanged between the server and the client. These secret keys are used in so-called symmetric cryptography (the same key is used for encryption and decryption, or signature and verification on both ends, as in AES-based algorithms).

The need to use symmetric cryptography comes from low memory usage and high-performance requirements. RSA or EC public-key-based algorithms would be extremely slow and/or void of resources compared to AES. But the distribution of secret keys for secure communication requires the complexity of the key negotiation phase using RSA- or EC-based algorithms.

As an alternative to the public-key-based key exchange algorithms, there's a pre-shared (secret) key exchange algorithm. It means that the same secret key is loaded in both the server and the device before the device is released in the field. If this option removes the burden of implementing RSA or EC cryptography, it requires the long-term storage of the same secret key both in the device(s) and on the server. Such an approach is risky—disclosing a single secret key compromises the whole network if all of the connected objects share the same key, which can happen since the embedded device lives in the wild. There are also risks because the key distribution is a sensitive operation.

SECURE COMMUNICATION PHASE

Once the shared communication secret keys have been exchanged following the above process, a secure communication channel is established; the two communicating parties can start exchanging application-level data (e.g., HTTP request/responses). The TLS protocol layer automatically encrypts and signs packets sent out, and decrypts and verifies received packets on the receiver's end using secret key-based algorithms such as AES-CBC for the encryption, AES-CMAC for signature, or more recent flavors such as AES-CCM for both encryption and signature at the same time. The TLS protocol layer returns errors if incoming messages are corrupt.

The shared communication keys, also called session keys, are discarded when the communication is ended.

PITFALLS IN TLS

From a software standpoint, the TLS protocol can be easily implemented (integrated) into any application by using off-the-shelf software libraries (such as mbedTLS, https://tls.mbed.org) *(Fig. 1)*. There are many claims that the software integration of such a library is easy, which is true at first sight. When communicating over the internet using the TCP/IP protocol, applications often use the POSIX socket API's "connect," "read," and "write" functions. To use TLS, the application only needs to redirect those calls to, for example, "mbedtls_ssl_connect," "mbedtls_ssl_read," and "mbedtls_ssl_write" when using the mbedTLS library.

The application source code doesn't need to contain all of the odds of implementing the main phases of TLS: "mbedtls_ssl_connect" automatically performs the authentication and key negotiation, and "mbedtls_ssl_read" and "mbedtls_ssl_write" automatically adds the TLS protection (encryption and signature) to both the incoming and outgoing application data transfers (e.g., HTTP request/responses).

Nevertheless, TLS library integration into applications and their configurations, including configuration on the server, have pitfalls and shortcomings. Even if you use a bug-free TLS stack (see https://project-everest.github.io), the integration and use of the TLS library in your software can be flawed (see https://www.mitls.org/pages/attacks).

The following sections discuss the common weaknesses in the TLS integration on the client side, i.e., our embedded device.

CERTIFICATE VERIFICATION SKIPPED

In a "man-in-the-middle" (MITM) attack, an attacker redirects network communications between a client and a server *(Fig. 2)*. Then it establishes regular TLS channels with, on the one hand, the client, and, on the other hand, the server. When it receives traffic from the client, it will eavesdrop it or tamper with it before transmitting to the server and vice versa. The communication channel is, therefore, no longer secure.

Diverting the traffic happens through switch or router poisoning (routers/switches transfer Ethernet packets to their legitimate recipient according to the Ethernet card MAC address and the IP address), or with DNS hijacking (DNS is the protocol that resolves server names, e.g., www.mywebsite.com, into its IP address).

One of the first issues seen very often in TLS library usage by applications relates to certificate verification. Often times, the TLS libraries don't enforce the verification of the server certificate; it's the application's duty. However, skipping this fundamental verification step exposes the communication to a "man-in-the-middle" attack.

If the network traffic is diverted to a computer that's not the intended server, by not verifying the peer certificate, the client will establish a wrongly trustworthy TLS channel with the attacker. In this case, the TLS protocol becomes useless and, worse, gives a false impression of being secure.

USE OF WEAK CIPHER SUITES

TLS failures may come from the server's configuration. The server may be configured to accept weak cipher suites. Weak cipher suites typically use deprecated or broken cryptographic algorithms (such as RC4) that are too weak to resist state-of-the-art attacks. Usually the TLS negotiation phase uses the strongest protocol configuration, security-wise, that both the client and the server support.

However, if the server supports obsolete cryptographic algorithms, an attacker may downgrade the security of the exchanges between the server and clients, and get access to the data exchanged. Likewise, forward secrecy is a property of some algorithms that protects past sessions against future compromises of secret keys or passwords.

The cipher suites that use ephemeral DHE or ECDHE will have perfect forward secrecy while the other options will not. Servers should implement only such cipher suites. This also eliminates the pre-shared key scheme that can't guarantee the forward secrecy. Recommendations can be found at https://github.com/ssllabs/research/wiki/SSL-and-TLS-Deployment-Best-Practices.

INSUFFICIENT PROTECTION OF CERTIFICATION-AUTHORITY CERTIFICATES

As we have seen, the resistance to man-in-the-middle attacks relies on the effective verification of the server's certificate. The server's certificate is verified using a root certificate that's stored in the embedded system. The root certificate is typically loaded during embedded-device manufacturing, when the device was commissioned.

This root certificate belongs to a certification authority, which issued the device and the server certificates, and can guarantee their integrity. A security breach will happen if this root certificate gets replaced by a rogue one in the device's long-

term memory. In such an event, an attacker's server may be successfully authenticated as a valid server because the rogue server's certificate would be correctly verified by the rogue root certificate mischievously loaded into the client device. This also requires some form of a man-in-the-middle attack, but isn't hard to achieve.

EXPOSURE OF SESSION KEYS

Session keys are a short-term secret asset—their validity lasts as long as the TLS secure communication channel. However, a set of session keys can't be used to guess another set of session keys. Still, if these keys are leaked, this would compromise a whole TLS session, whether it's a recorded past session or a currently running session.

COMPROMISED CLIENT AUTHENTICATION KEY

The TLS negotiation phase may include a step where the client authenticates to the server. If the client exposes its private key for authenticating to the server, then the client can be impersonated by an attacker who would reuse that key.

USE OF POOR CRYPTOGRAPHIC IMPLEMENTATIONS AND LOW-QUALITY RANDOM NUMBERS

The exposure of this key can be "immediate," e.g. because the device's long-term memory was dumped, or indirect because the cryptographic algorithms in use, although functionally correct, aren't resistant to side-channel attacks. A side-channel attack is a class of attack where attackers can guess private or secret keys by measuring timing and/or power consumption and/or electromagnetic emissions of the device when it runs a cryptographic algorithm involving such keys.

TOWARD A MORE SECURE TLS INTEGRATION ON THE CLIENT SIDE

To have a truly secure TLS scheme and mitigate the pitfalls listed above, one should follow these rules:

1. The CA certificate must be truly immutable. No one except the manufacturer of the device or the operator should be able to replace root certificates. The challenge is then to protect access to the memory that stores the certificates. Any software injected in the application processor of an embedded system can modify such a memory.
2. Server certificates must always be checked by the client to avoid MITM attacks.
3. Session keys must be protected while in use, and correctly removed from memory when not in use any more.
4. The client's private authentication keys must be safely stored.
5. Secure cryptographic algorithm implementations must be used (resistant to side-channel analysis).
6. The TLS library must be properly integrated and configured by the application in order to function safely.

BENEFITS OF A COMPANION SECURE IC FOR SECURE TLS IMPLEMENTATIONS

As seen above, the long-term assets involved in the TLS handshake phase on the client side are:

1. The server certificate
2. The certification authority certificate (one of the roots)
3. The client private key (if used)
4. The pre-shared key (if used)

A secure IC such as the MAXQ1061 makes the TLS implementation more secure because it prevents the above vulnerabilities by design, with no extra burden on the application processor. The IC stores the CA root certificates inside its internal non-volatile memory; they cannot be replaced without proper public key-based strong authentication. This strong authentication is usually performed by a remote administrator, the only entity owning the private key that can activate access into the device.

The MAXQ1061 handles the TLS handshake phase and makes the following possible:

- MITM attacks are prevented. Server certificates are always verified before use. The MAXQ1061 won't use an arbitrary public key or certificate for further signature verification. This public key has to be signed using a valid certificate already present in the IC before usage.
- Session keys don't leak, since they're computed using state-of-the-art cryptography (high-quality random numbers, no side-channel leakage)
- Clients cannot be impersonated because client authentication is performed using private keys stored within the MAXQ1061 long-term memory. Those keys are always generated internally using a high-quality random-number generator. By design they can't be extracted from the IC.
- Weak cipher suites can't be used. Only ECDHE-ECDSA with AES-CCM or AES-GCM and SHA-256 or more cipher suites are available.

In addition, the MAXQ1061 makes the client authentication possible only if secure boot has succeeded. It can keep the authentication private key unusable as long as the application processor's firmware and configuration haven't been successfully verified, thus allowing the client device to authenticate as a genuine one only if it hasn't been tampered with. As an aside, devices that integrate a MAXQ1061 can't be cloned. It's impossible to fake the IC since it contains a unique private key that never leaves the device.

A slightly modified TLS stack derived from mbedTLS, provided by Maxim Integrated, allows the application processor to leverage the above features of the MAXQ1061 and, thus, provide this increased level of security without requiring a major redesign. The modified mbedTLS stack leverages the 1061 for session key exchange, server certificate verification, and client authentication. The secure communication itself can be carried out by the 1061 itself or by the main application processor.

The above discussion doesn't preclude the weaknesses on the server side, where obvious security measures for the server private key protection and chain of certificate also apply! So, there are recommendations to follow for setting up a server, too.

CONCLUSION

Securing an embedded system requires implementation of a set of best practices:

- Robust software coding
- Lock system doors: physical and logical
- Don't underestimate attackers

The storage of long-term secrets such as private keys for authentication, or vital data like certificates, is much safer and easier to do within a secure IC rather than in a common application processor that offers, by design, many debug features that give access to its whole memory content. In addition, the physical isolation between the two ICs provides the guarantee that a software vulnerability in the application processor can't endanger the assets stored in the secure IC.

STEPHANE DI VITO is a senior security and software engineer at Maxim Integrated, leading software design and development for the company's secure microcontrollers. Prior to joining Maxim in 2011, he held software security roles at Newsteo, Atmel, and Gemalto. Stephane has been working in secure embedded software engineering since 1999.