Keywords: TLS protocol, secure microcontrollers, secure companion ICs, embedded security, Transport Layer Security protocol, secure communications

APPLICATION NOTE 6436

USING SECURE COMPANION ICS TO PROTECT A TLS IMPLEMENTATION

By: Stephane di Vito, Security Expert, Maxim Integrated

Abstract: This application note discusses the security challenges of TLS implementations and examines how secure companion ICs can protect such implementations from security breaches.

Introduction

It's more important than ever to protect the sensitive data that is collected and shared between our many smart, connected devices and the cloud. Industrial sensors, home appliances, wearable medical devices, and the like all utilize data that, if in the wrong hands, could lead to severe consequences.

The Transport Layer Security (TLS) protocol, formerly known as Secure Socket Layer (SSL), is the most used protocol for protecting data in transit. While it was initially created for bi-directional secure communications over the internet, between computers and web sites, the TLS protocol is now critical for securing the communication of IoT devices over the internet, preventing eavesdropping or tampering of data in transit. The protocol is considered to be robust since it has been vastly studied, attacked, and fixed and is widely adopted. To ensure that the TLS protocol can be trusted on the device side, secret keys, private keys, and certificates that must never be disclosed or modified must be stored in the devices and used in the course of the execution of the protocol. However, since IoT devices are deployed into the wild, these security assets can be exposed to attackers who might attempt invasive as well as non-invasive attacks. In an invasive attack, a cyber criminal opens the device's enclosure to manipulate its memory content, replace firmware, or probe PCB traces. The non-invasive variety, usually performed through communication ports, targets logical bugs in the device's firmware. Without a secure IC companion, protecting a TLS implementation becomes impossible.

This application note discusses a low-cost, low-complexity solution to secure a TLS implementation in a connected embedded system while also reducing the burden on the devices' application processor.

Overview: TLS Protocol

TLS occurs between a client and a server. In this application note, the "client" is an embedded system (e.g., an IoT device) and the "server" is a remote machine in the cloud, reachable using the internet.[I]

As such, let's imagine a "client" device with sensors and actuators that is connected to a web service provided by the "server." Based on this scenario, the user can use a smartphone to view sensor data and send commands to the device through the server. The client device uses TLS to report data or fetch commands to/from the server.

There are two main phases for the TLS protocol:

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. The phase includes server and client authentication, as well as computation of a shared secret. The four steps to this phase are as follows:

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. The phase includes server and client authentication, as well as computation of a shared secret. The four steps to this phase are as follows:

1. **An agreement on the cipher suite,** which is the collection of cryptographic algorithms to be used in subsequent phases below. The server and the client decide which set of cryptographic protocols is used for negotiation of the shared secret keys and security of the application communication.
2. **Authentication of the server by the client.** During this phase, the client sends a message comprised of a random sequence of bytes to the server. In exchange, the server sends back its own certificate to the client, along 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, authentication of the client by the server.** Useful in embedded systems, this step follows the same process as for Step 2. Standalone client devices can not "enter their password" to authenticate to the server, a common practice on web sites. The TLS handshake addresses this using 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 Step 2).
4. **Negotiation of shared communication secret keys** (e.g., AES keys) for the secure communication phase. In this step, secret keys can be exchanged using complex public key-based key exchange protocols (such as ECDHE). Those protocols allow computing of the same secret, or symmetric, key on both the client and the server side without actually transmitting it. The secret keys 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). Low memory usage and high-performance requirements drive the need to use symmetric cryptography. Compared to AES, RSA or EC public key-based algorithms would be extremely slow and/or devoid of resources. However, distributing secret keys for secure communication requires the complexity of the key negotiation phase using RSA- or EC-based algorithms.

   A pre-shared (secret) key exchange algorithm offers an alternative to the public key-based key exchange algorithms. In this scheme, the same secret key is loaded in both the server and the device before the device is released in the field. While 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. This approach is risky, however, because the disclosure of a single secret key compromises the whole network if all the connected objects share the same key.

Secure Communication Phase

Once the shared communication secret keys have been exchanged using 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. The layer does so 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 simultaneous encryption and signature. If incoming messages are corrupt, then the TLS protocol layer returns errors.

After the communication has ended, the shared communication keys, also called session keys, are discarded.

Disadvantages of TLS

From a software perspective, the TLS protocol can be easily integrated into any application using off-the-shelf software libraries (such as mbedTLS, https://tls.mbed.org). Software integration of such a library seems easy to do. After all, 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 does not need to contain all 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 add the TLS protection (encryption and signature) to both the incoming and outgoing application data transfers (e.g. HTTP request/responses).

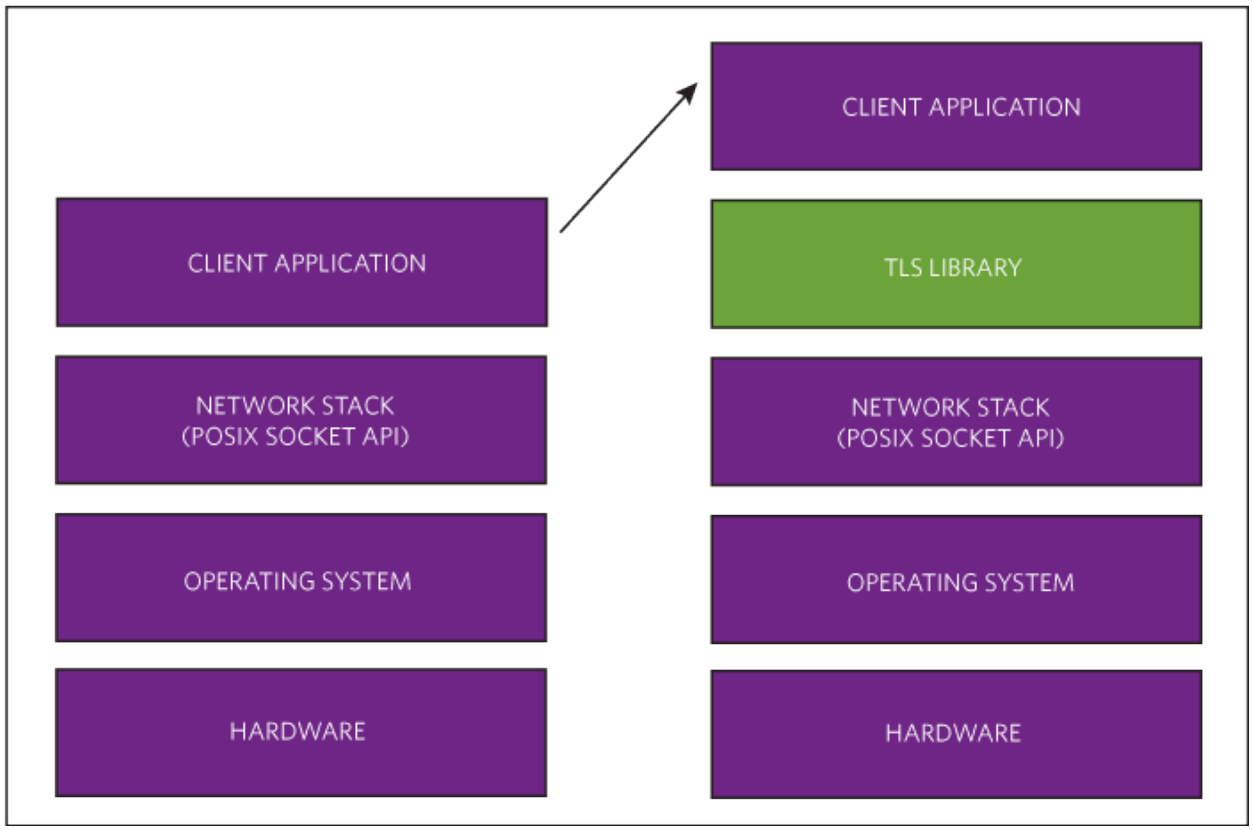

Figure 1. This diagram provides a rough picture of TLS implementation. More details can be found here: https://tls.mbed.org/kb/how-to/mbedtls-tutorial.

As straightforward as it might seem, integrating a TLS library into applications and their configurations, including configuration on the server, comes with some shortcomings. Even if using a bug-free TLS stack (see https://project-everest.github.io), the integration and use of the TLS library in the software can be flawed (see https://www.mitls.org/pages/attacks).

The following sections of this application note examine the common weaknesses in the TLS integration on the client side, i.e., our embedded device.

Skipped Certificate Verification

One of the first issues seen very often in TLS library usage by applications relates to certificate verification.

It's the application's role to enforce verification of the server certificates, so TLS libraries often don't do this automatically. They have to explicitly request the verification from the TLS library. But without this fundamental verification step, the communication can be exposed to a "man-in-the-middle" attack.

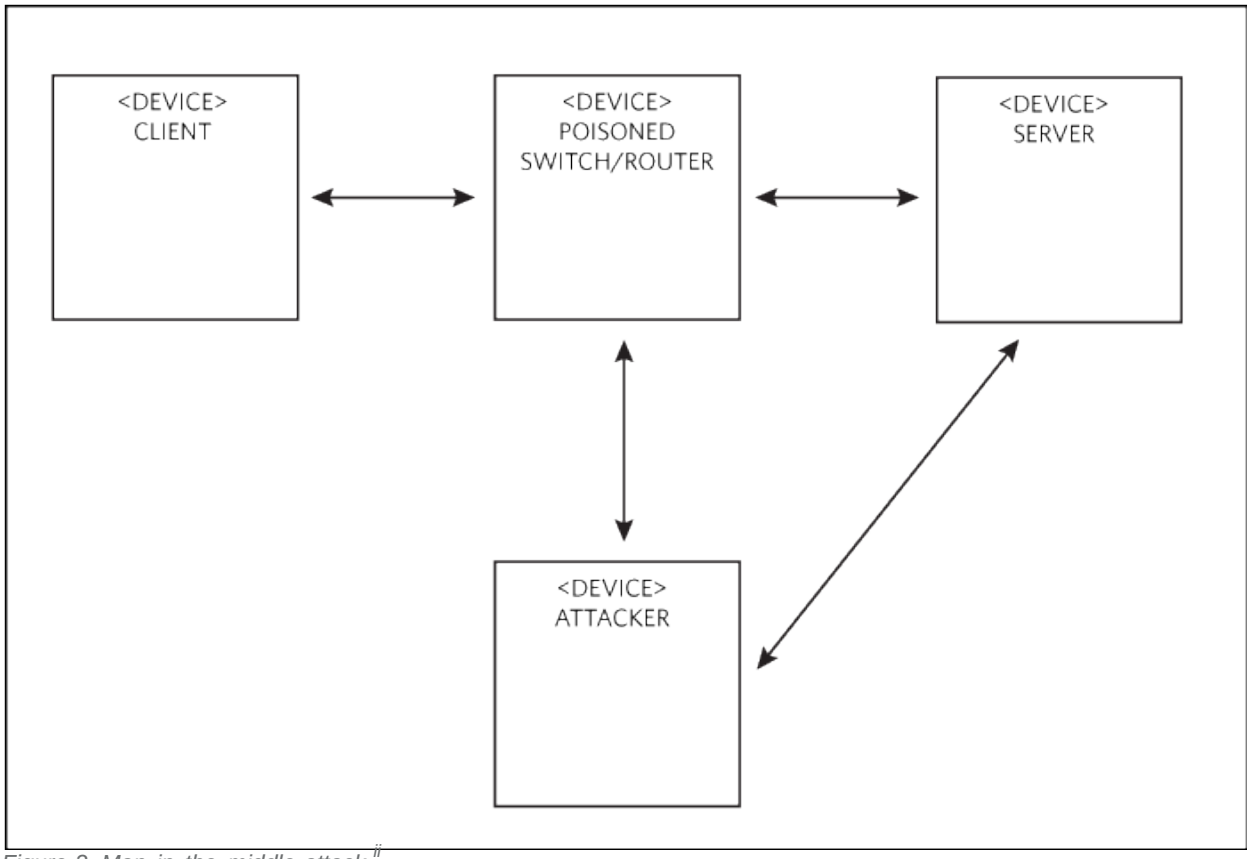

Figure 2. Man-in-the-middle attack.[ii]

Suppose the network traffic gets diverted to a computer that isn't the intended server. By not verifying the peer certificate, the client establishes a wrongly trustworthy TLS channel with the attacker. When this happens, the TLS protocol becomes useless and, worse, delivers a false impression of security.

Weak Cipher Suites

TLS failures can stem from the server's configuration. For example, the server can be configured to accept weak cipher suites. Weak cipher suites typically utilize deprecated or broken cryptographic algorithms (such as RC4) that are too weak to resist state-of-the-art attacks. Usually the TLS negotiation phase taps into the strongest protocol configuration, from a security standpoint, supported by both the client and the server. However, if the server supports obsolete cryptographic algorithms, an attacker can downgrade the security of the exchanges between the server and clients, and gain access to the exchanged data. Likewise, some algorithms use forward secrecy, which protects past sessions against future compromises of secret keys or passwords. The cipher suites that use ephemeral Diffie-Hellman (DHE) or the elliptic-curve variant (ECDHE) has perfect forward secrecy-these are the cipher suites servers should implement. There are options that do not have perfect forward secrecy. More details on cipher suites can be found here: https://github.com/ssllabs/research/wiki/SSL-and-TLS-Deployment-Best-Practices.

Vulnerable Certification Authority Certificates

Effective verification of a server's certificate is integral to resisting man-in-the-middle attacks. A root certificate stored in the embedded system verifies the server's certificate. Typically loaded during embedded device manufacturing, the root certificate belongs to a certification authority, which issued the device and the server certificates and can guarantee their integrity. If this root certificate gets replaced by a rogue one in the device's long-term memory, then

the certificate can fall victim to a security breach. When this occurs, an attacker's server can successfully authenticate as a valid server because the rogue server's certificate would be correctly verified by the rogue root certificate that was mischievously loaded into the client device. This scenario also requires some form of a man-in-the-middle attack; however, it isn't difficult to do.

Exposed Session Keys

Session keys are a short-term secret weapon in the security arsenal: their validity lasts as long as the TLS secure communication channel. One set of session keys cannot be used to guess another set of session keys. However, if these keys are leaked, this would still compromise a whole TLS session, whether it is a recorded past session or a currently running session.

Compromised Client Authentication Key

The TLS negotiation phase can 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.

Poor Cryptographic Implementations and Low-Quality Random Numbers

If the device's long-term memory was dumped, the exposure of this key can be "immediate." The exposure can also be indirect because the cryptographic algorithms in use, while functionally correct, aren't resistant to side-channel attacks. In a side-channel attack, 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.

Enabling a More Secure Client-Side TLS Implementation

There are a set of minimum rules to follow to maintain a truly secure TLS scheme and also to mitigate the pitfalls discussed in this application note:

1. The CA certificate must be truly immutable. Only the device manufacturer or operator should be able to replace root certificates. The challenge is then to protect access to the memory that stores the certificates, since any software injected in the application processor of an embedded system can modify the memory.
2. Server certificates must always be checked by the client to avoid MITM attacks.
3. Client's private authentication keys must be safely stored
4. Secure cryptographic algorithm implementations must be used (and be resistant to side-channel analysis)
5. The TLS library must be properly integrated and configured by the application in order to function safely

How Companion Secure ICs Safeguard TLS Implementations

The TLS handshake phase on the client side involves these long-term assets:

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)

Using a secure IC, such as the MAXQ1061, protects the TLS implementation by preventing the above vulnerabilities by design, without placing any additional burden on the application processor.

- The MAXQ1061 stores the CA root certificates inside its internal non-volatile memory. These certificates can only be replaced with the proper public key-based strong authentication. Typically, a remote administrator performs this strong authentication, and is the only entity that owns the private key for activating access into the MAXQ1061.
- The MAXQ1061 manages the TLS handshake phase, always guaranteeing that:

    - MITM attacks are prevented. Prior to use, server certificates are always verified. The MAXQ1061 refuses to use an arbitrary public key or certificate for further signature verification. Before usage, this public key must be signed using a valid certificate already present in the MAXQ1061.
    - Since they are computed using state-of-the-art cryptography (high-quality random numbers, no side-channel leakage), session keys do not leak
    - Since client authentication is performed using private keys stored within the MAXQ1061 long-term memory, clients cannot be impersonated. Those keys are always generated internally using a high-quality random number generator. They cannot, by design, be extracted from the IC.
    - Weak cipher suites cannot be used. Only ECDHE-ECDSA with AES-CCM or AES-GCM and SHA-256 or more cipher suites are available.
- The MAXQ1061 allows client authentication only when secure boot has succeeded. So long as the application processor's firmware and configuration have not been successfully verified, this secure IC can render the authentication private key unusable. With this approach, the client device can authenticate as a genuine one only if it has not been tampered with. Additionally, devices featuring a MAXQ1061 cannot be cloned: it's impossible to fake this secure IC since it contains a unique private key.

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

The above discussion does not preclude any weaknesses on the server side. Needless to say, it's imperative to follow security measures to protect the server private key and chain of certificate.

Conclusion

Securing an embedded system calls for robust software coding and locked system doors (physically and logically). It also pays to not underestimate the attackers.

Storing long-term secrets such as private keys for authentication, or vital data such as certificates, is much safer and easier within a secure IC. The alternative is storing these assets in a common application processor, but these processors typically provide, by design, many debug features that give access to their whole memory content. Another benefit of using a secure IC along with an application processor is that the physical isolation between the two ICs guarantees that a software vulnerability in the application processor won't endanger the assets stored in the secure IC.

---

[i] **A Primer on Public Key-Based Digital Signature**

Consider a scenario where a "sender" wants to transmit a message M to a "receiver," where M is a sequence of bytes. The receiver wants to ensure that the message comes unmodified from the genuine sender.

In the next sequence, consider message exchange between the "sender" and the "receiver." The names of the different items on the receiver's end are postfixed with "_r" (for "received") because they can 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 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 byte-long for SHA-256) with those important facts verified:
    - It's impossible to find or compute two different messages with the same hash value (no collisions)

Knowing a hash value, it is impossible to determine the matching message (one way), except by 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. If not, it is 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, this might not be the case.

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). Plus, "good" means that M_r is identical to the message signed using SprivK_r (integrity); it has not been modified since it has been signed.

As SprivK_t_ and SPubk_t are mathematically bound (but knowing SPubk_t does not 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.

However, the above signature verification is not entirely sufficient. To truly guarantee message authenticity, the public key used to verify the signature (SPubk_r) must be traced back to the sender. Indeed, 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 finds 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 not sent nor signed by the correct sender!

Certificate Verification, PKI

It's imperative for the receiver to verify that SPubk_r belongs to the expected sender (i.e., SPubk_r = SPubk_t). When there is 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 to verify incoming signed messages. However, in a networked environment, many senders might want to send messages to many receivers. This is led to the advent of public key infrastructures (PKI). In a PKI, public keys are transmitted by senders to receivers within certificates. Certificates are issued by certification authorities. 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 with the message M_t replaced by the certificate and the signing private key SPrivk_t being 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 for verifying the message's signature. To this end, the receiver uses the certificate of the issuer of Cert_SPUbk_r and proceeds as for verifying an incoming message (the message being Cert_SPUbk_r). A successful verification 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

When this occurs, it becomes impossible to forge a valid certificate with arbitrary identities and public keys inside, so an attacker cannot impersonate a sender. The certification authority's private key remains secret. But how do we validate the certification authority's certificate? There can be a few intermediate certificates 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.

[ii]In a "man-in-the-middle" attack, an attacker redirects network communications between a client and a server. Then it establishes regular TLS channels with, on the one hand, the client, and, on the other hand, the server. After receiving traffic from the client, it eavesdrops on it or tamper with it before transmitting to the server and vice versa. As a result, the communication channel is no longer secure.

Traffic diversion occurs 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 hikjacking (DNS is the protocol that resolves server names, e.g. www.mywebsite.com into its IP address).

Related Parts	
MAXQ1061	DeepCover Cryptographic Controller for Embedded Devices

More Information
For Technical Support: https://www.maximintegrated.com/en/support
For Samples: https://www.maximintegrated.com/en/samples
Other Questions and Comments: https://www.maximintegrated.com/en/contact

Application Note 6436: https://www.maximintegrated.com/en/an6436
APPLICATION NOTE 6436, AN6436, AN 6436, APP6436, Appnote6436, Appnote 6436