Constrained CIP Security Best Practices

Jakub Korbel
Networks Architect
Rockwell Automation

Jack Visoky
Principal Engineer and Security Architect
Rockwell Automation

Presented at the ODVA
2023 Industry Conference & 22nd Annual Meeting
October 18, 2023
El Vendrell, Spain

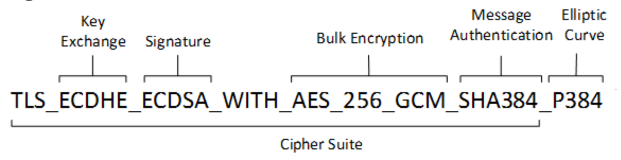
Abstract

Given the proliferation of cyberattacks it is of paramount importance to ensure that industrial equipment has sufficient cyber defensive capabilities. Even devices with low resources are not exempt from this, as attackers will target these devices knowing that it is not easy to support security capabilities on them. However, with the CIP Security Resource-Constrained Profile, a lightweight yet robust implementation of CIP Security is defined for this class of devices. Despite this, it may still not be straightforward to add these security capabilities. This paper will provide some best practices for applying the CIP Security Resource Constrained Profile to this class of devices.

Keywords

CIP Security, Cybersecurity, Industrial Security, Resource Constrained Devices, IIoT, TLS, DTLS

Glossary

Term	Description
AES	**Advanced Encryption Standard.** A specification for the encryption of electronic data established by the U.S. National Institute of Standards and Technology (NIST) in 2001. This block cipher is based on symmetric keys and it is used to encrypt the actual data flowing over a TLS or DTLS session.
Block cipher mode of operation	An algorithm that uses a block cipher to provide information security such as confidentiality or authenticity. A block cipher (such as AES) by itself is only suitable for the secure cryptographic transformation (encryption or decryption) of one fixed-length group of bits called a block, while the mode of operation describes how to repeatedly apply a cipher's single-block operation to securely transform amounts of data larger than a block.
CA	**Certificate Authority** [1]. An authority trusted by one or more entities to create and digitally sign public-key certificates. Optionally the certificate authority may create the subjects' keys.
CBC	**Cipher block chaining.** A block cipher mode of operation for symmetric-key cryptographic block ciphers (such as AES). Does not provide data authenticity, for that, HMAC can be used in addition.
Certificate	**X.509 Certificate** [1]. The public key of an entity, together with some other information, rendered unforgeable by digital signature with the private key of the certificate authority (CA) that issued it.
ChaCha20	A stream cipher. This stream cipher is based on symmetric keys and it is used to encrypt the actual data flowing over a TLS or DTLS session.
Cipher Suite	A cipher suite is a set of cryptographic algorithms used in establishment of and communication over a secure connection [2]. A cipher suite specifies one algorithm for each of the following tasks: • Key exchange • Signature • Bulk encryption • Message authentication e.g.:  Key Exchange Signature Bulk Encryption Message Authentication Elliptic Curve TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384_P384 Cipher Suite
DH	**Diffie-Hellman.** Key-agreement protocol, that allow two parties, each having a public-private key pair, to establish a shared secret over an insecure channel.
DTLS	**Datagram Transport Layer Security** [3]. This protocol allows client/server applications to communicate in a way that is designed to prevent eavesdropping, tampering, or message forgery. The DTLS protocol is based on the TLS protocol and provides equivalent security guarantees.
EC	**Elliptic-curve (cryptography)**. An approach to public-key cryptography based on the algebraic structure of elliptic curves over finite fields. ECC allows smaller keys compared to RSA cryptography while providing equivalent security.

ECC	**Elliptic-curve cryptography.** Same as EC.
ECDH	**Elliptic-curve Diffie-Hellman.** Same as DH but based on ECC.
ECDHE	**Elliptic-curve Diffie-Hellman with Ephemeral key.** Same as ECDH but using Ephemeral key.
ECDSA	**Elliptic-curve Digital Signature Algorithm.**
Ephemeral key	A key freshly generated for every key-agreement. This is the basis of the PFS feature. Cipher suites are marked with an extra E after the key-agreement protocol string, such as ECDH (Elliptic-Curve Diffie-Hellman) vs ECDHE (Elliptic-Curve Diffie-Hellman with Ephemeral key)
FS	**(Perfect) Forward Secrecy**. A security property where past negotiated session keys are protected even if the associated static private keys are eventually compromised. An example of this is: • A and B establish a secure session via key agreement with static public/private key pairs (usually using certificates) • This session encrypts and authenticates the data exchanged using the session key • The session ends • At some future date, A and B have their static private keys compromised, which were the keys used to create the secure session key • Even though these static keys are compromised, the data sent/received during the previous secure session cannot be compromised; this is a due to the property of perfect forward secrecy In practice there are different way to achieve this security property, but it usually has to do with generation of an intermediate ephemeral key pair by each communicating party, and then using that to generate the session key
GCM	**Galois/Counter Mode.** A block cipher mode of operation for symmetric-key cryptographic block ciphers (such as AES) which is widely adopted for its performance. The GCM algorithm provides both data authenticity (integrity) and confidentiality.
IIoT	**Industrial Internet of Things** [4]. Refers to interconnected sensors, instruments, and other devices networked together with computers' industrial applications, including manufacturing and energy management.
MAC	**Message authentication code.** A short piece of information used for authenticating a message. It gives an assurance that the message was not modified since the sender sent it.
PFS	**Perfect Forward Secrecy**. See FS.
Poly1305	A hash family. Can be used to produce a MAC.
PSK	**Pre-shared key**. A key that has been shared securely before any communication can happen. In (D)TLS-PSK, these are symmetric keys used for authentication, and can be combined with Diffie-Hellman key exchange.
RNG	**Random number generator.** Often found prefixed by T – True RNG, where the randomness is obtained from an entropy source, or P – Pseudo-RNG, where the randomness is calculated, usually by software, and needs to be seeded from an entropy source.
RSA	**Rivest–Shamir–Adleman.** An approach to public-key cryptography based on easy multiplication but difficult factorization of large prime numbers.
SHA	A hash family. Can be used to produce a MAC.

TLS	**Transport Layer Security protocol** [5]. TLS allows client/server applications to communicate in a way that is designed to prevent eavesdropping, tampering, and message forgery.
TRISIS	**Schneider Electric's Triconex safety instrumented system-targeting malware.**
X.509	**Recommendation ITU-T X.509 | ISO/IEC 9594-8** [1]**.** It defines frameworks for public-key infrastructure (PKI) and privilege management infrastructure (PMI). It introduces the basic concept of asymmetric cryptographic techniques. It specifies the following data types: public-key certificate, attribute certificate, certificate revocation list (CRL) and attribute certificate revocation list (ACRL). It also defines several certificates and CRL extensions, and it defines directory schema information allowing PKI and PMI related data to be stored in a directory. In addition, it defines entity types, such as certification authority (CA), attribute authority (AA), relying party, privilege verifier, trust broker and trust anchor. It specifies the principles for certificate validation, validation path, certificate policy, etc. It also includes a specification for authorization validation lists that allow for fast validation and restrictions on communications.

Introduction

It is well known that cyberattacks are increasing in both frequency and severity. Given the increased connectivity of IoT and other devices, attackers are no longer limited to “traditional” targets like servers and Internet websites. Examples of devices being attacked are readily available. One of the most famous is the Mirai Botnet, which used malware to control many IoT devices such as cameras to launch Distributed Denial of Service (DDoS) attacks [6]. Another infamous incident of device hacking occurred when security researchers demonstrated that they could take control of a Jeep and cause it to shut off the engine or disable braking [7]. Closer to the Industrial Controls space there was a hack of a Schneider Electric Triconix Safety Integrated System (SIS) done in 2017 [8].

Although much could be concluded from these and other examples of cyberattacks against devices, there are a few important takeaways to motivate the need for strong security protections at low levels:

1. Low-end devices will not be overlooked by hackers. Attacks targeting devices like IoT cameras shows that attackers are not going to simply overlook a device because it is resource constrained. The Mirai Botnet is especially interesting in the sense that the IoT devices were commandeered to launch other attacks; even if the device itself is not of great import it can still be hacked as a means to another end.
2. A connected device is a potentially vulnerable attack surface. If a device includes connectivity, then it should be considered a potential target for attackers. Although in the case of Mirai the devices were connected directly to the Internet, in the case of the Jeep exploit and TRISIS they were not. Therefore, just because a device is not directly on the Internet doesn’t mean it won’t be a target.
3. Given the above two, security protections need to be available for resource-constrained connected devices. It is important that these devices are able to implement protections, as they are clearly targets of attacks.

The rest of this paper will discuss different areas of consideration for applying security to a constrained device and highlight best practices.

Multitasking Discussion

When designing low-end communication device firmware, a decision needs to be made on how to handle concurrent tasks and events. In computing, there are two categories of computer multitasking – preemptive multitasking and non-preemptive (cooperative) multitasking.

Preemptive multitasking

In preemptive multitasking, used by most modern operating systems (OS), the task scheduling scheme includes preemption. Here preemption means interruption of a currently running task, usually by utilizing hardware interrupts and interrupt service routines (ISR) [9]. These routines are usually used by an operating system's scheduler, which schedules the highest-priority active task to run or/and employs more advanced methods like static and dynamic time-slicing [10].

Scheduling a different task means switching task context. Processor flags, and key registers such as Program Counter (PC) and Stack Pointer (SP) of the preempted task needs to be saved [9]. Then the new task's context needs to be restored, which is a performance-costly operation as context-switches can occur abundantly depending on the scheduler algorithm and its variables. In preemptive multitasking, task synchronization needs to be in place by implementing synchronization primitives, for example critical sections, mutexes, semaphores, etc. [11] and each task needs its own distinct stack, which increases the system's Random Access Memory (RAM) consumption and also impacts performance as tasks need to wait for other tasks using the same resource or entering the same critical section. Even though the context switches and synchronization are costly, any blocking operations and heavy calculations cannot halt the whole system's runtime as they will be preempted based on the scheduling algorithm.

Non-preemptive (cooperative) multitasking

In cooperative multitasking, the task scheduling is simpler. The task decides when it is ready to give up control of the processor [12]. Hence, there is no need to perform complex costly context switching in terms of RAM and CPU and the scheduler can be as simple as a loop periodically firing task routines which yield CPU by a simple return. Apart from synchronizing tasks and ISRs, which can preempt running tasks and run temporarily instead of them, no synchronization primitives are needed [11] and neither is OS providing them. On the other hand, tasks must behave nicely, i.e., not perform blocking or lengthy operations and even one misbehaving task can halt the whole system.

Implications for CIP Security Constrained Profile Devices

Modern operating systems are designed to run many programs at once and even allow users to run what they like. Thus, non-preemptive multitasking is unacceptable for them as any misbehaving program could bring the system to a halt. This is, however, not a concern for constrained communication device firmware, where programmers have complete control over what is running in the system. Due to the nature of stack allocation [13], which can reclaim used space for different operations, it is memory-effective to utilize stack allocations as much as possible and share one stack for all tasks. As it will be demonstrated in the "**Error! Reference source not found.**" section, this is even more pronounced considering transport layer security with RAM-intensive operations. It is hence a best practice to use cooperative multitasking in highly constrained devices, for less constrained, single-CPU devices with developers' full control over them, it should be considered as well based on the application and environment.

The Handshake and Connection Memory Footprint

Handshaking is an important part in network communication, where two counterparts establish a connection in between. For connectionless protocols, such as UDP, there can be no handshake involved and messages could be exchanged without the counterparts keeping any internal state. This, however, changes with DTLS, where encryption context containing shared secrets to encrypt and decrypt messages, is needed to be kept on both parts. In a way, this makes DTLS more complex and memory consuming than TLS, because TLS can exploit the underlying, already connected, TCP transport, as it will be demonstrated in Analysis section.

Here is how the handshake process looks like for DTLS v1.2 [3]:

```
 Client                                          Server
 ------                                          ------

 ClientHello             -------->                           Flight 1

                         <-------    HelloVerifyRequest      Flight 2

 ClientHello             -------->                           Flight 3

                                            ServerHello    \
                                           Certificate*     \
                                     ServerKeyExchange*      Flight 4
                                    CertificateRequest*     /
                         <--------      ServerHelloDone    /

 Certificate*                                              \
 ClientKeyExchange                                          \
 CertificateVerify*                                          Flight 5
 [ChangeCipherSpec]                                         /
 Finished                -------->                         /

                                     [ChangeCipherSpec]    \ Flight 6
                         <--------             Finished    /
```

Figure 1: Message flights for a full DTLS v1.2 handshake

This is how the DTLS v1.2 handshake looks like for IO over DTLS on port 2221/udp. For explicit message channel over DTLS on port 44818/udp, as defined in the CIP Security Constrained Profile, this is preceded by StartDTLS EtherNet/IP™ messages signifying that a DTLS session follows with the IP address and UDP port of the client.

1. **ClientHello.** The client sends a list of their TLS version, Cipher Suites alongside client random value.
2. **HelloVerifyRequest.** The server asks for verification using a cookie to prevent DoS attacks.
3. **ClientHello.** The client re-sends the Client Hello with a cookie.
4. **ServerHello.** The server picks the best version of TLS and the best Cipher Suite from client's list, if any is applicable and sends server random value.
5. If this is a ECDHE and PSK-based suite:
    a. **ServerKeyExchange.** The server sends its DH parameters, including a newly generated DH public key, if Ephemeral suites are used. Server sends an opaque, application dependent identity hint to the client. PSK is used for authentication and is embedded in the pre-master secret.
    b. **ServerHelloDone.** Indication of the server completing its ServerHello.
    c. **ClientKeyExchange.** The client sends its DH parameters, including a newly generated client public key, if Ephemeral suites are used. It also sends the identity opaque value based on the identity hint.
    d. The server and client establish a shared secret and send **ChangeCipherSpec**. After that, the communication is encrypted.
6. If this is an ECDHE and ECDSA-based suite:
    a. **Certificate.** The server sends its certificate or certificate chain.

   b. **ServerKeyExchange.** The server sends its DH parameters, including a newly generated server public key, if Ephemeral suites are used. It also sends a signature of previous messages authenticated by the private key that belongs to the server Certificate.
   c. **CertificateRequest.** If the server choses to verify the client, who then must send its **Certificate** and the **CertificateVerify** message.
   d. **ServerHelloDone.** Indication of the server completing its ServerHello.
   e. If the server requested client certificate authentication, **Certificate** is sent from the client.
   f. **ClientKeyExchange.** The client sends its DH parameters, including a newly generated public key, if Ephemeral suites are used. If certificate authentication was requested, it also sends a signature of previous messages authenticated by the private key that belongs to the client.
   g. The server and client establish a shared secret and send **ChangeCipherSpec**. After that, the communication is encrypted.
7. **Finish**. Signifies completion of the handshake and it is encrypted by the symmetric shared key.

Analysis

To provide basis for following claims, a testware application has been developed and published on github [14] and was used to check memory consumption (stack, heap) and performance profiling. This testware application used two common TLS libraries, wolfSSL [15] and mbedTLS [16] to make more general statements applicable to multiple practical applications. The test application tested the Server and Client side of the communication for TLSv1.2 and DTLSv1.2 using multiple cipher suites with various properties. Only one certificate has been set for a device and another for CA, which directly signed it. Thus, no certificate chain was used, which could potentially render ECC-based cipher suites more efficient.

The most memory intensive part of the (D)TLS communication is undoubtedly the handshake, as can be seen from the difference between “Heap” Column meaning handshake peak memory and “Heap 1 Conn” column in Table 1: Embedded TLS Libraries Handshake and Connection Heap and Stack Consumption in bytes, marking remaining allocated size for connection context extended of the memory needed to transmit or receive a simple 11-byte-long “Hello world” message.

The largest memory footprint can be seen when Elliptic-Curve Diffie-Hellman with Ephemeral EC keypair generation is used. Contrary to popular belief, Elliptic-Curve-based Cipher Suites can consume more memory during handshakes than RSA-based Cipher Suites even though their keys are smaller in size. This is especially true for the client side. On the server side, both TLS libraries consume similar amounts of memory and the difference is not so decisive.

DTLS is also more memory consuming than TLS in general for handshakes but also for keeping one connection alive, which is more pronounced on the server side. This is likely caused by the DTLS having to decrypt the packet as a whole and serve it to the user’s provided buffer at once as it is customary in the Berkley socket API. For example, WolfSSL is allocating the full size of user-specified receive buffer enhanced for padding and DTLS record headers for packet reception. This is offset by DTLS not needing a reliable stream protocol underneath, where the management of the session is the responsibility of the TCP/IP stack and the memory consumption can then be visible there.

Going for ECC and DTLS may thus not be the easy path for Constrained security, and it may require significant tweaks in the (D)TLS library compilation options.

Further memory savings by using PSK as opposed to ECC certificates are negligible (but can grow in importance when certificate chains are being sent), although for some reason mbedTLS had significant memory savings with PSK over ECC on the client side. Using PSK has significant memory benefits over RSA certificates with large keys.

ChaCha20-Poly1305 seems to be best suited for constrained purposes. With its post-handshake heap usage per 1 connection only NULL-SHA256 cipher suites is comparable (which provides no confidentiality).

Handshake operations are also stack intensive. If the client and server side run in two separate threads, in some configurations the RAM consumption can grow up to 20K just for parallel stacks of a multi-threaded design but staying at 10K for Single-Threaded. To lower stack consumption, it appears this mbedTLS configuration was better suitable, but the vast amounts of memory were spent on heap, rendering the actual RAM savings negative for one connection.

Method	Cipher Suite	wolfSSL			mbedTLS		
		Stack	Heap[1]	Heap 1 Conn	Stack	Heap[2]	Heap 1 Conn
TLS Client	ECDHE-ECDSA-AES128-CBC-SHA256	10488	8452	3280	9896	37838	5282
TLS Client	ECDHE-PSK-AES128-CBC-SHA256	10488	8452	3280	7928	33621	2994
TLS Client	DHE-RSA-AES256-SHA256	10136	5147	3280	8520	38432	6943
TLS Client	DHE-PSK-AES128-CBC-SHA256	10136	4015	3280	7256	34395	2994
TLS Client	ECDHE-ECDSA-AES128-GCM-SHA256	10488	8452	3330	9896	37846	5658
TLS Client	ECDHE-PSK-NULL-SHA256	10488	8452	1424	7928	33045	2418
TLS Client	ECDHE-PSK-CHACHA20-POLY1305	10488	8452	1816	7928	33053	2426
DTLS Client	ECDHE-ECDSA-AES128-CBC-SHA256	10488	9150	5434	9896	37786	5282
DTLS Client	ECDHE-PSK-AES128-CBC-SHA256	10488	8468	5434	7928	33560	2994
DTLS Client	DHE-RSA-AES256-SHA256	10136	5959	5947	8520	38344	6943
DTLS Client	DHE-PSK-AES128-CBC-SHA256	10136	4415	5947	7256	34159	2994
DTLS Client	ECDHE-ECDSA-AES128-GCM-SHA256	10488	9150	5484	9896	37794	5658
DTLS Client	ECDHE-PSK-NULL-SHA256	10488	8468	3738	7928	32984	2418
DTLS Client	ECDHE-PSK-CHACHA20-POLY1305	10488	8468	3970	7928	32992	2426
TLS Server	ECDHE-ECDSA-AES128-CBC-SHA256	9736	8066	3296	4968	33480	2994
TLS Server	ECDHE-PSK-AES128-CBC-SHA256	9560	8133	3296	3800	33375	2994
TLS Server	DHE-RSA-AES256-SHA256	9176	8524	3296	3688	37147	4810
TLS Server	DHE-PSK-AES128-CBC-SHA256	9176	3272	3296	3272	33875	2994
TLS Server	ECDHE-ECDSA-AES128-GCM-SHA256	9560	8067	3346	4968	33751	3370
TLS Server	ECDHE-PSK-NULL-SHA256	9560	8133	1440	3800	32799	2418
TLS Server	ECDHE-PSK-CHACHA20-POLY1305	9560	8133	1832	3800	32807	2426
DTLS Server	ECDHE-ECDSA-AES128-CBC-SHA256	9560	10608	8814	4968	34154	6113
DTLS Server	ECDHE-PSK-AES128-CBC-SHA256	9560	9888	8814	3800	33416	6113
DTLS Server	DHE-RSA-AES256-SHA256	9176	10711	6150	3688	38119	9743
DTLS Server	DHE-PSK-AES128-CBC-SHA256	9176	5004	6150	3272	34113	7927
DTLS Server	ECDHE-ECDSA-AES128-GCM-SHA256	9560	10605	8864	4968	34528	6489
DTLS Server	ECDHE-PSK-NULL-SHA256	9560	9884	7118	3800	32836	5537
DTLS Server	ECDHE-PSK-CHACHA20-POLY1305	9560	9884	7350	3800	32844	5545

Table 1: Embedded TLS Libraries Handshake and Connection Heap and Stack Consumption in bytes

[1] During receive operation, wolfSSL dynamically allocates a buffer for incoming encrypted UDP packet, thus these measured values contain additional 1500+ bytes on DTLS server side. For every receive call, the user will need to have two times the buffer, one for decrypted (user-allocated) and one for encrypted (wolfssl-allocated) data.
[2] Per mbedTLS configuration file documentation, mbedTLS allocates 32768 bytes (maximum 16384 bytes per [5] for both sides) for a handshake as a reasonable maximum. This is included in the measured values.

Performance Considerations for Handshakes

The handshake operation is also performance intensive. Here, handshake operation refers to the authentication and key agreement performed at the start of a DTLS session. This involves several asymmetric cryptography calculations as well as the generation of random data, which are both fairly processor intensive operations. In selected suites, where PFS needs to be assured, ephemeral versions of Diffie-Hellman key exchange and Elliptic-Curve Diffie-Hellman key exchange are used. Profiling on the memory analysis testware showed, that generating a new DH key-pair can be a very slow operation (e.g., on constrained, single-core devices up to 100MHz CPU clock rate without hardware acceleration, it can take seconds) and it takes similar time to sign and verify in the ECDSA authentication phase. Hashing is also a very time-consuming operation happening throughout the communication, but profiling did not draw it as a significant handshake performance bottleneck. If possible, it is beneficial to offload key generation RNG and Sign & Verify operations to the hardware or to use asynchronous operations. This allows for other communication channels, such as CIP I/O, to run without significant impact. In a wolfSSL analysis [17], various crypto operations, such as RNG, SHA and SHA256 but even ECC key generation, ECDHE and ECDSA [18] could be sped up by several (usually one or two) orders of magnitude. Time consuming handshakes are even more pronounced when the device uses a Single-Threaded approach, since during the calculation nothing else can run. Pre-calculating Ephemeral keys in advance and using them when a handshake is requested is also a potential mitigation to consider. These methods can be combined to provide even smoother operation, so pre-calculating ephemeral keys, using PSK instead of ECDSA, offloading SHA256 and AES operations to hardware and using asynchronous operations in Single-Threaded environment may be the correct path forward in Constrained environments.

Memory Savings

The CIP Security Resource Constrained Profile deliberately removed components of the other profiles that took up significant memory but are not strictly necessary for a secure protocol. The most significant component is the support for provisioned certificates. The EtherNet/IP Confidentiality Profile supports provisioning of a user-signed certificate, as well as the provisioning of multiple Certificate Authority (CA) or End-Entity (EE) certificates. This implies support for the following components:

- File Object
- Certificate Management Object
- X.509 Parsing Code
- Certificate-based Cipher Suites

However, removing support for these results in significant memory savings. Although exact values will vary greatly from one platform and implementation to another, it can be instructive to view some concrete numbers. As an example, one can compare two Rockwell Automation proprietary implementations, one with the EtherNet/IP Confidentiality Profile, the other with the CIP Security

Resource Constrained Profile. Of course, the former implemented the full EtherNet/IP Profile, and the latter the Constrained EtherNet/IP Profile. The constrained implementation reduced the build footprint by over 50%. Not all of this came from the differences in CIP Security profile, some was the base EtherNet/IP as well. However, this provides quite significant savings in terms of footprint. Note here the build footprint refers to code size, read-only memory (constants), and read-write memory (global/static variables).

File Object

CIP Security uses the CIP File Object to store any certificate, whether it is the device's identity certificate, an additional trusted CA certificate, or an additional trusted EE certificate. This is a reasonable decision as the File Object is a generic mechanism for storing files used in CIP, and certificates are files. However, with this generic mechanism comes the complexity of additional objects and services which can be removed if certificates are not supported. On an existing sample implementation the File Object resulted in about 2 Kilobytes of footprint, which can be a significant amount of savings within a resource constrained environment.

Certificate Management Object

The Certificate Management Object is a CIP object defined to allow for common configuration and management of a certificate. For CIP Security, it is used to manage a device's identity certificate, as well as trust for the Certificate Authority that has signed the device's identity certificate. However, to this end, multiple attributes and services are included. On an existing sample implementation the Certificate Management Object contributed around 300 bytes of footprint. Although not as significant as the File Object, this still is measurable savings in resource usage.

X.509 Parsing Code

Supporting certificates means supporting the ability to parse the X.509 data encoded within the certificate. X.509 certificates can include complex data types and custom extensions, which even if not used by a device would still need to be parsed. Then there are some extensions which are required for device usage like Subject Alternative Name. Much has been written about the complexity of X.509 certificates and the associated parsing needed [19] [20]. As such, including support for X.509 parsing adds significant complexity, as well as memory usage. Exact values for memory usage are not easy to determine as this code is often bound up in the (D)TLS library used. However, even a cursory review of open source (D)TLS libraries can show that X.509 support code is quite significant. Removing support for parsing certificates results in significant savings on memory.

Certificate-Based Cipher Suites

The Constrained CIP Security Profile removes support for any certificate-based cipher suites, except in the Factory Default state, in which case only one cipher suite is supported. Instead it makes use of PSK-based cipher suites. There are only 3 PSK suites required for the CIP Security Resource Constrained Profile:

- TLS_ECDHE_PSK_WITH_NULL_SHA256
- TLS_ECDHE_PSK_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_PSK_WITH_CHACHA20_POLY1305_SHA256

The choice of these ciphers was deliberate. ChaCha20-Poly1305 is particularly well-suited to constrained environments that do not have hardware accelerators. AES GCM is very efficient when a hardware accelerator is present [21]. The non-encrypted algorithm provides a lightweight option where the data is not encrypted and only uses SHA-256 to authenticate.

Note that one important decision regarding these cipher suites was to include cipher suites that provide perfect forward secrecy. All of these ciphers are Elliptic Curve Diffie Hellman ciphers, which include an ephemeral Elliptic Curve key generated during the handshake. This results in more memory usage during the handshake, as well as more code size. However, it was thought that this is a worthwhile tradeoff, as it provides good information assurance benefits against compromise of the PSK [22]. However, if code size becomes too much of an issue for constrained EtherNet/IP products, it remains an option to allow for even lighter-weight ciphers that do not provide perfect forward secrecy. However, note that TLS 1.3 and DTLS 1.3 is meant for ciphers which support perfect forward secrecy. That is, a PSK without an Ephemeral Diffie Hellman key can be used in TLS 1.3, although the expectation is that this is used for session resumption, to then be followed by a certificate exchange. To use TLS 1.3 without PFS would be to use a PSK without an Ephemeral Diffie Hellman key, which may be technically feasible but is not the documented/intended use. Although CIP Security is currently standardized on TLS 1.2 and DTLS 1.2, it will move to supporting TLS 1.3 and DTLS 1.3 in the near future.

Security Implications

Removing support for certificates does provide significant savings in terms of memory usage and complexity. However, there is of course a trade-off to these savings. From an information assurance standpoint, certificates provide a unique identity for each communicating party in a given group. Of course, this is only true if certificates are used for bi-directional authentication, support of which is required in CIP Security. However, PSKs do not provide this same assurance; rather a PSK is often shared amongst a group. If more than two parties share a given PSK then there is no assurance at the TLS layer that can distinguish one party from another, which has an impact on non-repudiation and spoofing security properties. However, for small-scale systems this is often a reasonable trade-off. Furthermore, some PSK uses may be vulnerable to reflection attacks of the type described in [23]. Regardless, Threat Modeling of a given system is necessary to understand the information assurance properties provided by the CIP Security Resource Constrained Profile. A given system may require further mitigations or compensating countermeasures to achieve a desired security posture.

Although user-generated certificates are not supported in the CIP Security Resource Constrained Profile, this profile does support an initial certificate. This has the benefit of allowing authentic and confidential deployment of security credentials like PSKs when a device is in the factory default state. Furthermore, assuming the initial certificate used is an IEEE 802.1AR IDevID, which is the recommended implementation, this can also provide identity and authenticity assurances of the device in question. Even though the device has a unique certificate, it still does not need to parse certificate data. That is, when the initial connection is made, the device simply “serves” the certificate in the (D)TLS handshake; it does not request a client certificate. Also, because this certificate is not dynamic there is no need to implement the File Object or Certificate Management Object. Therefore this certificate has very little memory impact, although maintaining it on a device provides significant benefit to the initial commissioning and device authenticity.

Conclusions

The CIP Security Resource Constrained Profile provides robust cyber security assurances scaled down to devices which cannot feasibly implement the other CIP Security profiles. However, this does not come without trade-offs. Using PSKs instead of certificates provides significant savings on code complexity and memory usage, although some of the information assurances change. PSKs do not provide the same level of non-repudiation, or of spoofing by already trusted endpoints as certificates. Depending on the threat model for a given product or system these trade-offs may be acceptable, and in some cases even negligible. However it is important to be aware of this and to spend the appropriate time and effort documenting risks and responding to them.

Although the CIP Security Resource Constrained Profile provides significant savings in terms of resources required to implement it, there are still important considerations for product vendors. One of the biggest areas is around the tasking architecture of the platform. Whether multi-tasking is used and what scheme is chosen can have significant impact on the performance of security within a constrained device. Vendors should consider non-preemptive multitasking as it is particularly well-suited to a constrained environment device supporting DTLS communication. Furthermore, it is recommended to preference stack allocations for memory usage in a constrained environment, as the stack memory may be easily reclaimed for other uses once it is no longer needed for security intensive operations like DTLS handshakes. It is very important to be mindful of the DTLS handshake and how that will be processed, especially if multiple handshakes can occur at a time, since this is likely to be the most resource intensive operation related to CIP Security in a constrained environment.

The CIP Security Resource-Constrained Profile currently mandates cipher suites which provide perfect forward secrecy, which is an important security attribute and is standardized in TLS 1.3. However, this comes at some non-trivial memory usage and performance cost. If in practice this is shown to be too burdensome to highly constrained devices then the SIG will need to consider providing support for cipher suites which do not provide Perfect Forward Secrecy (e.g. cipher suites with PSKs that do not use Ephemeral Diffie Hellman keys). This is an important area for the SIG to monitor and react to as product developers begin to implement the CIP Security Resource-Constrained Profile.

References

[1] International Telecommunication Union, "Recommendation ITU-T X.509," October 2019. [Online]. Available: https://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-X.509-201910-I!!PDF-E&type=items. [Accessed 10 September 2023].

[2] Microsoft, "Cipher Suites in TLS/SSL (Schannel SSP)," 14 July 2023. [Online]. Available: https://learn.microsoft.com/en-au/windows/win32/secauthn/cipher-suites-in-schannel. [Accessed 10 September 2023].

[3] E. Rescorla and N. Modadugu, "Datagram Transport Layer Security Version 1.2," January 2012. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6347. [Accessed 10 September 2023].

[4] H. Boyes, B. Hallaq, J. Cunningham and T. Watson, "The industrial internet of things (IIoT): An analysis framework," *Computers in Industry,* vol. 101, pp. 1-12, 2018.

[5] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol," August 2008. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc5246#section-6.2.1. [Accessed 10 July 2023].

[6] B. Krebs, "krebsonsecurity.com," 21 September 2016. [Online]. Available: https://krebsonsecurity.com/2016/09/krebsonsecurity-hit-with-record-ddos/. [Accessed 10 July 2023].

[7] C. Miller and C. Valasek, "Remote Exploitation of an Unaltered Passenger Vehicle," 10 August 2015. [Online]. Available: https://illmatics.com/Remote%20Car%20Hacking.pdf. [Accessed 10 July 2023].

[8] Dragos Inc., "TRISIS Malware - Analysis of Safety System Targeted Malware," Dragos Inc., 13 December 2017. [Online]. Available: https://www.dragos.com/wp-content/uploads/TRISIS-01.pdf. [Accessed 10 July 2023].

[9] M. Barr, "Introduction to Preemptive Multitasking.," 6 March 2003. [Online]. Available: https://www.embedded.com/introduction-to-preemptive-multitasking/. [Accessed 10 July 2023].

[10] M. Kalin, "CFS: Completely Fair Process Scheduling in Linux.," 5 February 2019. [Online]. Available: https://opensource.com/article/19/2/fair-scheduling-linux. [Accessed 10 July 2023].

[11] D. Sergio, L. Brandon and S. Kiran, "Computer Science 111 Lecture 9 - Synchronization Primitives and Deadlock," 16 February 2015. [Online]. Available: http://web.cs.ucla.edu/classes/winter15/cs111/scribe/9b/index.html. [Accessed 10 July 2023].

[12] P. K. Jr., "Heavyweight Tasking," *Embedded Systems Programming,* vol. 3, no. 4, pp. 43-52, April 1990.

[13] B. Kinariwala and T. Dobry, "Programming in C," University of Hawai`i, 3 September 1994. [Online]. Available: http://ee.hawaii.edu/~tep/EE160/Book/chap14/subsection2.1.1.8.html. [Accessed 10 July 2023].

[14] J. Korbel, "TLS Library Benchmark," 10 July 2023. [Online]. Available: https://github.com/JKorbelRA/tlsbm. [Accessed 10 July 2023].

[15] wolfSSL Inc., "wolfSSL - Embedded SSL/TLS Library," 10 July 2023. [Online]. Available: https://www.wolfssl.com/. [Accessed 10 July 2023].

[16] Linaro Limited, "Mbed TLS - Trusted Firmware," 11 September 2023. [Online]. Available: https://www.trustedfirmware.org/projects/mbed-tls/. [Accessed 10 July 2023].

[17] wolfSSL Inc., "Benchmarking wolfSSL and wolfCrypt," [Online]. Available: https://www.wolfssl.com/docs/benchmarks/. [Accessed 10 July 2023].

[18] Microchip Technology Inc., "ATECC508A Summary Data Sheet," 18 December 2017. [Online]. Available: https://ww1.microchip.com/downloads/aemDocuments/documents/OTH/ProductDocuments/DataSheets/20005928A.pdf. [Accessed 10 July 2023].

[19] N. M. G. P. Alessandro Barenghi, "SYSTEMATIC PARSING OF X.509: ERADICATING SECURITY ISSUES WITH A PARSE TREE," *Journal of Computer Security,* vol. 26, no. 6, p. 32, 2018.

[20] P. M. R. B. Arnaud Ebalard, "Journey to a RTE-free X.509 parser".

[21] Y. Nir and A. Langley, "RFC 8439: ChaCha20 and Poly1305 for IETF Protocols," [Online]. Available: https://www.rfc-editor.org/rfc/rfc8439.txt.

[22] A. Langley, "Protecting data for the long term with forward secrecy," Google, 22 November 2011. [Online]. Available: https://security.googleblog.com/2011/11/protecting-data-for-long-term-with.html.

[23] N. Drucker and S. Gueron, "Selfie: reflections on TLS 1.3 with PSK," *Cryptology ePrint Archive,* vol. 2019, no. 347, 2019.

[24] wolfSSL Inc., "wolfSSL Resource Use," wolfSSL Inc., 27 July 2016. [Online]. Available: https://www.wolfssl.com/files/flyers/wolfssl_resource_use.pdf. [Accessed 10 July 2023].