TLS-API

MULTOS TLS1.2-API

MAO-DOC-TEC-019 v0.7

Published by

MAOSCO Limited
1st Floor,
GPS House,
215 Great Portland Street,
W1W 5PN,
London,
United Kingdom.

General Enquiries

Email: dev.support@multos.com
Web: http://www.multos.com

Document References

All references to other available documentation is followed by the document acronym in square [ ] brackets. Details of the content of these documents can be found in the MULTOS Guide to Documentation, and the latest versions are always available from the MULTOS web site (http://www.multos.com).

Contents

1 Introduction

This document details the 'C' and Python 3 application programming interfaces (APIs) available with MULTOS for supporting the cryptographic functionality needed when using the Transport Layer Security (TLS) 1.2 and Datagram Transport Layer Security (DTLS) 1.2 protocols.

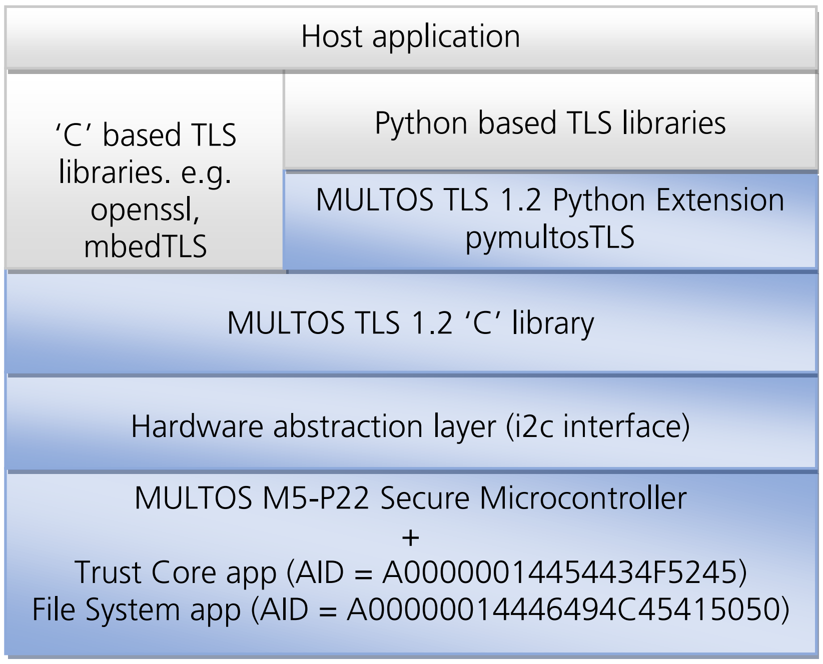
1.1 Security Considerations

The overriding design principle is to generate and store all keys, conduct all cryptographic operations and generate all random numbers within/using the high security MULTOS M5-P22 microcontroller. This is a highly certified tamper and attack resistant environment that can be used as a hardware root of trust for any IoT device.

1.2 Architecture

The MULTOS device is used as a secondary secure cryptographic processor connected to the main device microcontroller using i2c.

The software stack looks like this:-



Host application
'C' based TLS libraries. e.g. openssl, mbedTLS
Python based TLS libraries
MULTOS TLS 1.2 Python Extension pymultosTLS
MULTOS TLS 1.2 'C' library
Hardware abstraction layer (i2c interface)
MULTOS M5-P22 Secure Microcontroller + Trust Core app (AID = A00000014454434F5245) File System app (AID = A00000014446494C45415050)

Figure 1 - Software Stack

Those items in blue are currently provided as part of the MULTOS supplied package. Suggested integration points for openssl and mbedTLS are available on request.

1.1.1 Hardware Abstraction Layer

This is dependent on the host platform. Versions are available for Windows^{TM} , Raspberry Pi^{TM} (model 3b) and Arduino^{TM}. The source code is also available to allow porting to other host platforms. Wiring details are included with each.

The APIs described in this document have been built and tested on Windows^{TM} and Raspberry Pi^{TM} (running Raspbian O/S).

"Raspberry Pi" is a trademark of the Raspberry Pi Foundation.
This is not an official Arduino product and all support requests should be directed to MULTOS.

1.3 TLS 1.2 feature support

1.1.2 Certificate Formats

The API supports RSA and ECC certificates.

RSA

- The API currently operates with RSA – SHA256 formatted certificates.
- Device keys are generated as 2048 bit RSA keys with an exponent of 65537.
- CA keys can be up to 4096 bits long.

ECC

- Curves P-256, P-384 and P-521
- ECDSA with SHA-256, SHA-384 and SHA-512 (depending on curve)

1.1.3 Cipher Suites

The API currently supports the following cipher suites:

- TLS_RSA_WITH_AES_128_CBC_SHA
- TLS_RSA_WITH_AES_128_CBC_SHA256
- TLS_RSA_WITH_AES_256_CBC_SHA
- TLS_RSA_WITH_AES_256_CBC_SHA256
- TLS_RSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256

1.1.4 Other

- *Extended Master Secrets* **ARE** supported
- *Encrypt Then MAC* **IS** supported
- *Mutual Authentication* **IS** supported
- *CBC Digest Records* **ARE NOT** supported

2 Control Functions

2.1 Initialise

int **mtlsInit** (unsigned short wCipherSuite);

success = **pymultosTLS.init** (wCipherSuite)

The parameters are:
- wCipherSuite: The ciphersuite code as defined in RFC 5246, or 0.

This function initialises the hardware abstraction layer and selects the MULTOS TLS 1.2 application in the MULTOS device.

IMPORTANT: The user PIN for the Trust Core application must be provided in the file "user_pin.txt" located in the same directory as the application calling this function. By default, the user PIN is 1234.

Returns 1 if the function is successful, 0 if there is an error.

2.2 Finish

int **mtlsFinish** (void);

success = **pymultosTLS.finish** ()

This function erases the key data for the current TLS 1.2 session and deselects the ciphersuite.

Returns 1 if the function is successful, 0 otherwise.

2.3 Version

void **mtlsVersion** (unsigned char *bMajor, unsigned char *bMinor);

bMajor,bMinor = **pymultosTLS.version** ()

Returns the version of the library in bMajor and bMinor

3 RSA Functions

These functions are mainly concerned with generating and using the device private key for authentication purposes. When combined with other MULTOS applications (for example that for supporting the Device Authority IAM platform), the private key pair may come from another source.

3.1 Generate Key Pair

int **mtlsGenerateRsaKeyPair** (char *sFileName, char *sCountryName, char *sStateOrProvinceName, char *sLocalityName, char *sOrgName, char *sOrgUnit, char *sCommonName, char *sEmailAddress);

success = **pymultosTLS.generateRsaKeyPair** (sFileName, sCountryName, sStateOrProvinceName,sLocalityName, sOrgName, sOrgUnit, sCommonName, sEmailAddress)

The parameters are:

- sFileName: Name of the file to hold the certificate signing request (input)
- sCountryName: ‘C’ value in certificate request (input)
- sStateOrProvinceName: ‘ST’ value (input)
- sLocalityName: ‘L’ value (input)
- sOrgName: ‘O’ value (input)
- sOrgUnit: ‘OU’ value (input)
- sCommonName: ‘CN’ value (input)
- sEmailAddress: email address for certificate request (input)

NOTE: This function is not supported in this version of the API. The function returns 0 if called (failed). The function is left as a stub to ensure backwards compatibility. Key pairs and CSRs should instead by generated using the **p11keygen** application. The keyId parameter used in the command must be 0 (zero).

3.2 Verify Signature

int **mtlsRsaVerifySignature** (unsigned char *pModulus, unsigned short wModLen, unsigned char *pExponent, unsigned char bExpLen, unsigned char *pData, unsigned short wDataLen, unsigned char *pSignature);

The C parameters are:

- pModulus: Pointer to the public key modulus value to use to verify the signature (input)
- wModLen: Length of the public key modulus in bytes (input)
- pExponent: Pointer to the public key exponent (input)
- bExpLen: Length of the exponent value in bytes (input)
- pData: Pointer to the data whose signature is to be verified, typically an ASN.1 sequence (input)
- wDataLen: Length of the data (input)
- pSignature: Pointer to the raw signature to verify, having the same length as pModulus (input)

success = **pymultosTLS.rsaVerifySignature** (abModulus, abExponent, abData, abSignature);

The Python parameters are:

- abModulus: byte-like object containing the public key modulus to use for signature verification (input)
- abExponent: byte-like object containing the exponent of the key to use for verification (input)
- abData: byte-like object containing the unsigned data (input)
- abSignature: byte-like object containing the signature to be verified (input)

This function is for use in verifying the signatures on X.509 certificates signed with the *sha256WithRSAEncryption (PKCS#1)* method. The data supplied is hashed using SHA-256 and compared to the hash recovered from the signature.

Returns 1 if the signature is verified, 0 otherwise.

3.3 PKCS#1 Type 1 Signature

int **mtlsRsaSignPKCS1_type1** (unsigned char *pData, unsigned short wDataLen, unsigned char bDoHash, unsigned char **pOutPtr);

The C parameters are:

- pData: pointer to the data to sign (input)
- wDataLen: length of the data to sign (input)
- bDoHash: set (non zero) if the SHA256 is to be performed on pData. Otherwise it is assumed that pData is already the required hash value (input).
- pOutPtr: pointer to a pointer which will be set to point to the signature (output)
  Note: the caller is responsible for freeing the output buffer pointed to by *pOutPtr.

signature_length, abSignature = **pymultosTLS.rsaSignPKCS1_type1** (abData, bDoHash)

The Python parameters are:

- abData: a byte-like object containing the data to sign (input)
- bDoHash: set (non zero) to perform the SHA256 step (input)
- abSignature: a bytes object containing the signature (output)

The provided data is

- hashed using SHA256 (optional)
- the hash is inserted into an ASN.1 sequence denoting a SHA-256 hash
- the ASN.1 sequence is padded to 256 bytes (the length of the device public key) using PKCS#1 v1.15 type 1 padding,
- the padded sequence is signed using the device private key previously generated using 0

This function may be used to sign the "Certificate Verify" message during mutual authentication and returns the length of the signature if successful, 0 otherwise.

3.4 *PKCS#1 Signature with PSS padding*

int **mtlsRsaSignPKCS1_PSS** (unsigned char *pData, unsigned short wDataLen, unsigned char bDoHash, unsigned char **pOutPtr);

The C parameters are:

- pData: pointer to the data to sign (input)
- wDataLen: length of the data to sign (input)
- bDoHash: set (non zero) if the SHA256 is to be performed on pData. Otherwise it is assumed that pData is already the required hash value (input)
- pOutPtr: pointer to a pointer which will be set to point to the signature (output)

signature_length, abSignature = **pymultosTLS.rsaSignPKCS1_PSS** (abData, bDoHash)

The Python parameters are:

- abData: a byte-like object containing the data to sign (input)
- bDoHash: set (non zero) to perform the SHA256 step (input)
- abSignature: a bytes object containing the signature (output)

The provided data is

- hashed using SHA256 (optional),
- the hash is padded using the PSS method,
- the padded hash is signed using the device private key previously generated using 0.

This function may be used to sign the "Certificate Verify" message during mutual authentication and returns the length of the signature if successful, 0 otherwise.

Note: the caller is responsible for freeing the output buffer pointed to by *pOutPtr.

4 ECC Functions

ECC public keys used in these functions are big endian byte arrays, the X coordinate is followed by the Y coordinate. Each coordinate's length must equal the prime length of the curve (32, 48 or 66 bytes).

ECC signatures are similar to public keys, other than the array consists or R followed by S (again both must be the length of the prime).

Curves and identified by using their IANA named curve ID. P-256 = 0x17, P-384 = 0x18 and P-521 = 0x19

4.1 Generate Key Pair

Use the **p11keygen** application to generate key pairs and CSRs. The keyId parameter used in the command must be 0 (zero). Note that keyId = 3 is used for ephemeral ECC keys.

4.2 ECDSA Signature

int **mtlsECDSASign** (unsigned char *pHash, unsigned short wHashLen, unsigned char *pSignature);

The C parameters are:

- pHash: pointer to the data to sign (input)
- wHashLen: length of the data to sign (input)
- pSignature: pointer to a buffer to hold the signature. Is assumed to be big enough (output)

signature_length, abSignature = **pymultosTLS.ecdsaSign** (abHash)

The Python parameters are:

- abHash: a byte-like object containing the hash to sign (input)
- abSignature: a bytes object containing the signature (output)

This function may be used to sign the "Certificate Verify" message during mutual authentication and returns the length of the signature if successful, 0 otherwise.

4.3 ECDSA Verify

int **mtlsECDSAVerify** (unsigned char *pPubKey, unsigned short wPubKeyLen, unsigned char bNamedCurve, unsigned char *pData, unsigned short wDataLen, unsigned char *pSignature);

The C parameters are:

- pPubKey: Pointer to the public key to use to verify the signature (input)
- wPubKeyLen: Length of the public key in bytes (input)
- bNamedCurve: curve for pPubKey (input)
- pData: Pointer to the data whose signature is to be verified (input)
- wDataLen: Length of the data (input)

- pSignature: Pointer to the raw signature to verify (input). The length is assumed to equal wPubKeyLen (as it should).

success = **pymultosTLS.ecdsaVerify** (abPubKey, bNamedCurve, abData, abSignature);

The Python parameters are:

- abPubKey: byte-like object containing the public key to use for signature verification (input)
- bNamedCurve: curve of the PubKey (input)
- abData: byte-like object containing the data whose signature is to be verified (input)
- abSignature: byte-like object containing the signature to be verified (input)

This function is for use in verifying the signatures on X.509 certificates signed with the *shaXXXWithecdsa* methods (XXX is one of 256, 384 or 512). The hash method used is selected from the named curve (P-256 = SHA-256, P-384 = SHA-384, P-521 = SHA-512).

Returns 1 if the signature is verified, 0 otherwise.

4.4 Generate Ephemeral Key Pair

int **mtlsGenerateEphermeralECKey** (unsigned char bNamedCurve, unsigned char *pPubKey);

The C parameters are:

- bNamedCurve: curve to use (input)
- pPubKey: Pointer to the buffer to store the public key (output)

success, abPubKey = **pymultosTLS.generateEphemeralECKey** (bNamedCurve)

The Python parameters are:

- abHash: a byte-like object containing the hash to sign (input)
- abPubKey: a bytes-like object containing the public key (output)

Generates an EC key pair using the specified curve, internally storing the private key and returning the public key. If this function is called again, the existing private key will be overwritten.

Returns 1 if the key pair was successfully generated, 0 otherwise.

5 Handshake Functions

These functions are involved in the handshake process, ultimately resulting in the client and server having negotiated the key material that they are going to use for subsequent exchanges of messages.

5.1 Generate Client Random

int **mtlsGenerateClientRandom** (unsigned long dwServerTime, unsigned char bUseTime, unsigned char *pOut, unsigned short wOutSize);

The C parameters are:

- dwServerTime: the time on the server to include in the random value if used, otherwise ignored (input)
- bUseTime: if not zero, the dwServerTime value will be included in the output (input)
- pOut: Pointer to the buffer to hold the generated client random (output)
- wOutSize: Size of the buffer pointed to by pOut (input)

output_length, abOut = **pymultosTLS.generateClientRandom** (dwServerTime)

The Python parameters are:

- dwServerTime: the time on the server to include in the random value if not zero, otherwise ignored (input)
- abOut: a bytes object containing the client random value.

This function uses the MULTOS device's TRNG random number generator to generate the client random value using in the Client Hello message, optionally including the server time in the first eight bytes.

This function returns the length of the client random value placed into pOut / abOut

5.2 Generate Pre Master Secret

This function supports generation of the shared secret using RSA keys or ECDHE depending on the ciphersuite.

unsigned short **mtlsGeneratePreMasterSecret** (unsigned char bMajor, unsigned char bMinor, unsigned char *pPubKey, unsigned short wPubKeyLen, unsigned char *pExponent, unsigned char bExpLen, unsigned char *pOut, unsigned short wOutSize);

The C parameters are:

- bMajor*: Protocol major version (0x03 for TLS1.2, 0xFE for DTLS1.2) (input)
- bMinor*: Protocol minor version (0x03 for TLS1.2, 0xFD for DTLS1.2) (input)
- pPubKey: pointer to the server's public key / modulus (input)
- wPubKeyLen: length of the server's public key in bytes (input)
- pExponent*: pointer to the server's exponent value (input)
- bExpLen*: length of the server's exponent value (input)

- pOut*: pointer to a buffer to hold the encrypted pre-master secret (output)
- wOutSize*: size of the buffer pointed to by pOut. This must be a least the length of the server's public key modulus (input)

output_length, abOut = **pymultosTLS.generatePreMasterSecret**(abPubKey, abExponent, bMajor, bMinor)

The Python parameters are:

- abPubKey: bytes-like object containing the public key (input)
- abExponent*: bytes-like object containing the public key exponent (input)
- bMajor*: Protocol major version (0x03 for TLS1.2, 0xFE for DTLS1.2) (input)
- bMinor*: Protocol minor version (0x03 for TLS1.2, 0xFD for DTLS1.2) (input)
- abOut*: bytes object containing the encrypted pre master secret (output)

* These parameters are ignored when the ciphersuite employs ECDHE for shared secret creation so should be NULL for pointers and zero for lengths.

RSA: This function generates the premaster secret within the MULTOS device and directly encrypts it with the server's public key. The function returns the length of the data written to pOut.

ECDHE: The function creates a pre-master secret by using the ephemeral private key and provided public key. Returns 1 for success, 0 for failure.

5.3 Generate Master Secret

int **mtlsGenerateMasterSecret** (unsigned char *pServerRandom)

success = **pymultosTLS.generateMasterSecret** (abServerRandom)

The parameter is a pointer to the buffer / bytes-like object containing the 32 byte server random received from the server.

The MULTOS device internally creates and stores the master secret following the method defined in section 8.1 of RFC 5246 as follows:

```
/* From section 8.1 of RFC 5246
 * master_secret = PRF(pre_master_secret, "master secret",
 *                     ClientHello.random + ServerHello.random)
 *                     [0..47];
 */
```

This function returns 1 if successful, 0 otherwise.

5.4 Generate Extended Master Secret

int **mtlsGenerateMasterSecretExtended** (unsigned char *abHandshakeHash, unsigned short wHashLen, unsigned char *pServerRandom);

The C parameters are:

- abHandshakeHash: pointer to the hash of all the bytes exchanged so far during the handshake (input)
- wHashLen: length of the handshake hash, 32 for SHA256, 20 for SHA-1 (input)
- pServerRandom: pointer to the 32 byte server random value (input)

success = **pymultosTLS.generateMasterSecretExtended** (abHandshakeHash, abServerRandom)

The Python parameters are:

- abHandshakeHash: bytes-like object containing the hash of all the bytes exchanged so far during the handshake (input)
- abServerRandom: bytes-like object containing the 32 byte server random value (input)

The MULTOS device stores the server random for later use and computes the master secret using the extended method which uses the handshake hash as the PRF seed.

This function returns 1 if successful, 0 otherwise.

5.5 Generate Keys

int **mtlsGenerateKeys** (unsigned char **pClientIvPtr, unsigned char **pServerIvPtr);

The C parameters are:

- pClientIvPtr: pointer to a pointer to hold the location of the client iv value (output)
- pServerIvPtr: pointer to a pointer to hold the location of the client iv value (output)

success, abClientIv, abServerIv = **pymultosTLS.generateKeys**()

The Python parameters are:

- abClientIv: bytes-like object containing the generated client IV (output)
- abServerIv: bytes-like object containing the generated server IV (output)

The MULTOS device generates (and securely stores internally) the keys required for message encryption and MACing based on the master secret, ciphersuite and exchanged random values. The length of the returned IV's depends upon the cipher method (16 for AES-CBC and 4 for AES-GCM).

This function returns 1 if successful, 0 otherwise.

5.6 Generate Final Finish MAC

unsigned short **mtlsGenerateFinalFinishMAC** (char *sLabel, unsigned char *abHandShakeHash, unsigned short wHashLen, unsigned char *pOut, unsigned short wOutSize);

The C parameters are:

- sLabel: the string "client finished" or "server finished" (input)

- abHandShakeHash: pointer to the buffer containing the hash of all the messages exchanged during the handshake (input)
- wHashLen: length of the handshake hash, 32 for SHA256, 20 for SHA-1 (input)
- pOut: pointer to the buffer to hold the computed MAC value (output)
- wOutSize: length of the buffer pointed to by pOut, which must be at least 12 bytes (input)

output_len, abOut = **pymultosTLS.generateFinalFinishMAC** (sLabel, abHandShakeHash)

The Python parameters are:

- sLabel: utf-8 string "client finished" or "server finished" (input)
- abHandShakeHash: bytes-like object containing the hash of all the messages exchanged during the handshake (input)
- abOut: bytes object containing the MAC.

This function computes the MAC used when sending / verifying the "Client Finished" or "Server Finished" messages.

The function returns the length of the MAC written to pOut.

5.7 Initialise Handshake Hash

int **mtlsHandshakeHashInit** (void);

success = **pymultosTLS.handshakeHashInit** ()

This function must be called ONCE the first time a message is exchanged during the handshake to initiate the hash (SHA-256) which is going to be used at various points during the process (Generate Extended Master Secret, Client Finished, Server Finished messages).

This function returns 1 if successful, 0 otherwise.

5.8 Update Handshake Hash

int **mtlsHandshakeHashUpdate** (unsigned char *pData, unsigned short wLen);

The C parameters are:

- pData: pointer to a block of data to add to the hash (input)
- wLen: length of the block of data (input)

success = **pymultosTLS.handshakeHashUpdate** (abData)

For Python, abData is a bytes-like object containing the data to add to the handshake hash.

The MULTOS device internally updates the running handshake hash with the data provided.

This function returns 1 if successful, 0 otherwise.

5.9 Get Current Handshake Hash

unsigned short **mtlsHandshakeHashCurrent** (unsigned char *pOut);

output_len, abOut = **pymultosTLS.handshakeHashCurrent** ()

The C parameters are:

- pOut: pointer to buffer to hold the current value of the handshake hash. Must be 32 bytes in size. (output)

Python returns a bytes object containing the current hash value.

This function returns the length of the hash.

6 Message Encryption & MAC Functions

These functions are used once a successful handshake has been completed and all the keys agreed. By default TLS 1.2 operates a "MAC then Encrypt" scheme. This is supported by *Encrypt / Decrypt with HASH-MAC*. The other functions in this section are for supporting the "Encrypt then MAC" mode of operation.

6.1 Encrypt / Decrypt with HASH-MAC

unsigned short **mtlsEncryptDecrypt** (unsigned char *bSeqNum, unsigned char bContentType, unsigned short wProtocolVersion, unsigned short wDataLen, unsigned char *pData, int sending, unsigned char *pIV);

The C parameters are:

- bSeqNum: pointer to an 8 byte buffer containing the message sequence number to encode in the message (input)
- bContentType: a single byte value as defined in section 6.2.1 of RFC 5246 (input)
- wProtocolVersion: a two byte value signifying the protocol version; TLS1.2 = 0x0303 (input)
- wDataLen: length of data pointed to by pData (input)
- pData: pointer to buffer holding input data and where the result will be put (input/output)
- sending: 0 indicates receiving/decryption (uses server session keys), 1 indicates sending/encrypting (uses client session keys) (input)
- pIV: when sending, a pointer to a buffer to contain the random IV created for the encryption (output). When receiving, contains the IV to use for the decryption (input)

output_len, abOut, abIVOut = **pymultosTLS.encryptDecrypt** (abSeqNum, bContentType, wProtocolVersion, abData, sending, abIVIn)

The Python parameters are:

- abSeqNum: bytes-like object (8 bytes long) containing the message sequence number to encode in the message (input)
- bContentType: a single byte value as defined in section 6.2.1 of RFC 5246 (input)
- wProtocolVersion: an unsigned short value signifying the protocol version; TLS1.2 = 771 (input)
- abData: bytes-like object holding input data (input)
- sending: 0 indicates receiving/decryption (uses server session keys), 1 indicates sending/encrypting (uses client session keys) (input)
- abIVIn: a bytes-like object containing the IV to use for decryption (input)
- abOut: a bytes object containing the result of the encryption / decryption (output)
- abIVOut: a bytes object containing the random IV value generated for encryption (output)

This function constructs a fragment in the format specified by RFC 5246 in section 6.2.1 for encryption, decryption and MACing then uses the methods specified by the ciphersuite to perform those operations.

Note: Currently the maximum fragment size as specified by wDataLen is 2992 bytes.

This functions returns the length of the data returned in pOut / abOut.

6.2 *Encrypt / Decrypt Only*

int **mtlsEncryptDecryptOnly** (unsigned char *pData, unsigned short wDataLen, int sending, unsigned char *pIV);

The C parameters are:

- pData: pointer to buffer holding input data and where the result will be put (input/output)
- wDataLen: length of data pointed to by pData (input)
- sending: 0 indicates receiving/decryption (uses server session keys), 1 indicates sending/encrypting (uses client session keys) (input)
- pIV: pointer to a buffer holding the IV to use or NULL if the IV values generated during key generation are to be used (input)

success, abOut = **pymultosTLS.encryptDecryptOnly** (abData, sending, abIV)

The Python parameters are:

- abData: bytes-like object holding input data (input)
- sending: 0 indicates receiving/decryption (uses server session keys), 1 indicates sending/encrypting (uses client session keys) (input)
- abIV: bytes-like object holding the IV to use, empty if the IV values generated during key generation are to be used (input)
- abOut: a bytes object containing the result of the encryption / decryption (output)

This function takes data that has been formatted by the caller and simply encrypts or decrypts. Supports AES-CBC or AES-GCM depending upon the specified cipher suite.

This functions returns the length of the data returned in pData / abOut.

IMPORTANT NOTES – GCM.

1. The additional data must be set using the SetAdditionalData() function prior to calling this function.
2. wDataLen SHOULD NOT include the length of the tag value (16 bytes)
3. When encrypting, the Tag is appended to the end of output, therefore the buffer supplied (in the case of C) must be large enough to hold the ciphertext and the tag.
4. When decrypting, the Tag must be appended to the input data.

6.3 *Set Additional Data*

int **mtlsSetAdditionalData** (unsigned char *pData, unsigned short wDataLen);

The C parameters are:

- pData: pointer to the additional data to use for GCM encryption / decryption (input)
- wDataLen: length of data (input)

success = **pymultosTLS.setAdditionalData** (abData)

The Python parameters are:

- abData: bytes-like object holding the additional data to use for GCM encryption / decryption (input)

This function should be called before using Encrypt / Decrypt Only when the cipher suite specifies AES-GCM. It needs to be called each time the additional data is changed.

6.4 HASH-MAC

unsigned short **mtlsHMAC** (unsigned char *pIn, unsigned short wInLen, unsigned char client, unsigned char *pOut, unsigned short wOutSize);

The C parameters are:

- pIn: pointer to data to HASH-MAC (input)
- wInLen: length of data to HASH-MAC (input)
- client: 0 == use server_write_MAC_key, 1== use_client_write_MAC_key (input)
- pOut: pointer to buffer to hold the resulting MAC (output)
- wOutSize: size of the pOut buffer (input)

output_len, abOut = **pymultosTLS.hmac** (abIn, bClient)

The Python parameters are:

- abIn: a bytes-like object containing the data to HASH-MAC (input)
- bClient: 0 == use server_write_MAC_key, 1== use_client_write_MAC_key (input)
- abOut: bytes object containing the resulting MAC (output)

This function computes a HASH-MAC using the hash algorithm set by the ciphersuite and the keys generated during the handshake. It can be used to generate a HASH-MAC to send (in which case *client = 1*) or to verify a HASH-MAC received (in which case *client = 0*).
Note: Currently the maximum data size as specified by wInLen is 2992 bytes.
The function returns the length of the HASH-MAC written to pOut / abOut.

6.5 Generate Random

int **mtlsGenerateRandom** (unsigned short len, unsigned char client, unsigned char *pOut);

success, abOut = **pymultosTLS.generateRandom** (len)

This function uses the MULTOS TRNG to generate a stream of *len* random bytes written to the buffer pOut / bytes object abOut. It returns 1 for success, 0 otherwise. Its primary use is to generate random IV values for encryption.