TLS on Gemalto Cinterion BGS2-W

Version 0.1

Ed Goldstein

16/04/2018

Contents

[Assumptions: 1](#_Toc511635745)

[Introduction: 1](#_Toc511635746)

[Loading certificates into the modem: 1](#_Toc511635747)

[The command: 2](#_Toc511635748)

[Command signature: 2](#_Toc511635749)

[How this information was gathered: 3](#_Toc511635750)

[Other things to be aware of: 4](#_Toc511635751)

[Opening a secure socket: 5](#_Toc511635752)

# Assumptions:

User has already used the BGS2 to create a regular TCP socket.

User is familiar with TLS certificates and chain of trust model.

# Introduction:

The BGS2 modem has TLS support in its stack. To use it, certificates are loaded into the modem’s NVM, and a different set of AT commands are used to open a “secure socket” that will create a secure channel to the server.

Information in this document is drawn from wm01\_wm02\_an62\_tls\_v04 known as Application Note 62. Only available link: <https://ptelectronics.ru/wp-content/uploads/organizatsiya_bezopasnogo_ssl-soedineniya.pdf> so also attached here is a newer version (no real difference though):



# Loading certificates into the modem:

Certificates are loaded into the BGS2 NVM using a binary write AT command. This can be done at any time. The binary write command is signed so no other modem will accept the command.

The BGS2 has 10 slots (index) to hold certificates. Index 0 is always for holding a “local client certificate” this is a certificate that protects the loading of other certificates. The local client certificate must always be present before other certificates can be loaded. Unless the server requires client authentication, the local client certificate is not used in any TLS transaction with the server. In this (of server only authentication), the client certificate can be a generic/dummy, and does not need to be specific to the unit).

## The command:

The binary write command string has a number of fields known as parameter units:

* Certificate index – which slot to place the following certificate
* Certificate data – the public certificate in .der form (every byte from .der file)
* Client Private Key - in .der form (every byte from .der file), only needed if this is a certificate for index 0 (known as the local client certificate position).
* IMEI – the IMEI number of the modem. Makes the command unique to this modem.
* Command signature – This is a signature over all the previous parameter units. Means that no one can change any part of this command. e.g. the IMEI.

Each parameter unit has a length and parameter type number (Param ID).

Then at the beginning of the whole command, there is a total length, the SNBW command type which is WRITE and the number of following parameter units. All sizes and param IDs are 16 bits little endian.

All lengths include the two bytes of length themselves! (e.g. the IMEI is 15 ascii digits + 0x00 terminated + 16 bit param ID + 16 bits of the parameter unit length == 0x14.)

b9 05 01 00 05 00 06 00 01 00 00 00 69 02 02 00 30 82 02 … 58 69 b0 7f 02 05 00 30 82 02 … 53 d6 74 14 00 04 00 30 39 39… 34 35 36 31 00 b1 00 03 00 6b 4c 4c …63 3d 00

The above is an example of loading a client certificate, but the server certificate is the same except without the client private key parameter. (although the signature is still RSA’d with the client private key so the modem knows to accept it).

Here is a server certificate command loading into slot index 01.

1003 0100 0400 0600 0100 0100 3f02 0200 3082…fc9c 1400 0400 33353239383430383038383536373500 b100 0300 484966…733d00

### Command signature:

The command signature is a SHA1 hash of every byte of the command, from after the 16 bits of command length to the end of the IMEI parameter unit (underlined above). In the below function, this the parameter “buffer”. This SHA1 hash digest is then encrypted with RSA (default == RSA/None/PKCS1Padding) using the client private key. The output cipher from this RSA is then encoded to base 64 representation.

 private String getSignature(byte[] buffer)

 {

 Key pk = keystore.getKey(alias, keypass);

 MessageDigest sha1 = MessageDigest.getInstance("SHA1");

 byte[] digest = sha1.digest(buffer);

 Cipher cipher = Cipher.getInstance("RSA");

 cipher.init(1, (PrivateKey)pk);

 byte[] cipherText = cipher.doFinal(digest);

 return Base64.encode(cipherText);

 }

Key Store.ks PW < keypass

|  |  |  |
| --- | --- | --- |
| Alias | Pub Cert | Priv Key |

buffer: ~~b9 05~~ 01 00 05 00 06 00 01 00 … 36 31 00

pk = keystore.getKey

digest = sha1.digest

digest: 0C D4 7B AF A5 3A A2 C4 F6 9D 01 22 79 05 A6 70 A3 5A 43 2A

cipherText = cipher.doFinal

cipherText: ???

Base64.encode

Output bytes: 64795a4c…43745a6b3d00

In ASCII: dyZL+ … 3wp2SEkRCtZk=

SHA1

RSA + PKCS1Padding?

# How this information was gathered:

This is discussed on the Gemalto forum: <https://developer.gemalto.com/threads/bgs2-atsbnwiscert1-command-fails-cinterion-transport-layer-security-client-tcpip-services>

AN62 has a section with an overview of the format of the AT Binary Write command fields used to load certificate data into the modem. However, real examples are needed to show byte by byte the field size and data etc.

AN62 has attachments for a java tool that helps create this serial command to load into the modem. The document contains commands that:

1. Create a keystore file which is like a database to store sets of security data such as public and private keys and public certificates. Each set has an alias. The keystore itself has a password.
2. Create a chain of certificates. Note that these are example certificates, and you are not necessarily expected to create them in this way.
3. Call the java tool that takes various certificate data from the keystore, convert them into modem binary write commands and use a com port to load the certificates into the chosen slots in the modem.

To see the command in order to be able to decode it, the serial data output was recorded by using software to create a loop of two virtual com ports.

For this testing the serial program Termite was used. <https://www.compuphase.com/software_termite.htm>

<http://com0com.sourceforge.net/>

Creates two virtual com ports connected together. The java script outputs to one port, and a Termite serial terminal was opened on the other one to view the output of the tool without any hardware.

Note: Using this virtual com port system causes a mismatch in the flow control of the java tool. Therefore, the following considerations need to be made to get the full output:

1. The serial terminal needs RTS/CTS flow control, even though a real modem would not.
2. The first OK should be sent from termite before the java script is run.

The following are considerations when trying to use the Java tool:

1. The java version needs to be 32 bit. (because the RXTX dll for serial output is 32 bit).
2. Java 7 is the latest this has been tested with.
3. The JDK is needed, not just the JRE.
4. Depending on the paths used for the commands, if java doesn’t find the rxtx.dll, which is in wm02\_tls\_tools\cmd\_ipcertmgr\win-x86 you may need to copy it to a java bin directory.

# Other things to be aware of:

1. Older modem firmware may behave differently. For example:
	1. The algorithms for the signature in the certificates are limited. For example, if you load a certificate which has a SHA266 signature algorithm, the SNBR read shows the Signature Algorithm is “Unknown” this will therefore not work. If you load a certificate with a SHA1 signature, you will see this recognised in the SNBR:

^SBNR: 0, size: "595", issuer: "/C=Unknown/ST=Unknown/L=Unknown/O=Unknown/OU=Unknown/CN=Unknown", serial number: "5AC1D232", subject: "/C=Unknown/ST=Unknown/L=Unknown/O=Unknown/OU=Unknown/CN=Unknown", signature: "sha1RSA", thumbprint algorithm: "sha1", thumbprint: "332651DCD55FA746E9A87E3FC1A03FFB1C6CD932"

^SBNR: 1, size: "571", issuer: "/C=AU/ST=Some-State/O=Internet Widgits Pty Ltd/CN=74.traffilog.com", serial number: "00EB8D1AC060D27C6E", subject: "/C=AU/ST=Some-State/O=Internet Widgits Pty Ltd/CN=74.traffilog.com", signature: "Unknown", thumbprint algorithm: "sha1", thumbprint: "6AB627E872C75C46409D037FB48B326E7E472892"

# Opening a secure socket:

The commands and example in AN62 work.

If you open a secure socket with AT^SISO=0, you can call AT^SISE=0 and it will show you the connection error code. (look up value in the at command reference).

^SISE: 0,66 (=Peer certificate is not confirmed) was received for unknown signature above.

<https://developer.gemalto.com/threads/bgs2-atsbnwiscert1-command-fails-cinterion-transport-layer-security-client-tcpip-services>

<https://developer.gemalto.com/threads/bgs2-atsbnwiscert1-command-fails-cinterion-transport-layer-security-client-tcpip-services>

<https://stackoverflow.com/questions/38319086/setting-cinterion-bgs2-w-modem-certificates-in-code>

for running the tools in the documents:

for keytool, you need JDK (not just the JRE):

<https://stackoverflow.com/questions/4830253/where-is-the-keytool-application>

However, currently the cmd\_ipcertmgr only works with Java 6, 7, and not 10.

Note that the commands in the document use different passwords for keypass and storepass. Newer versions of java do not allow differing passwords, and ignore one. This could cause confusion later. It is better to use the same password, at least during initial testing.

Then As you said, it complains about the location of the rxtxserial. this thread says the  rxtxSerial.dll and the rxtxParallel.dllfiles need to be in **[...]/jre7/bin/**

Windows openssl (e.g to convert between der and pem)

<https://sourceforge.net/projects/openssl/?source=typ_redirect>

pem->der

C:\OpenSSL\bin\openssl x509 -outform der -in pem\_cert.cert.pem -out der\_cert.cert.der

Der->pem

C:\OpenSSL\bin\openssl x509 -inform der -in client01\_priv.der -out client01\_priv.pem

BGS2 needs firmware v4.0 to run TLS properly:

<https://developer.gemalto.com/threads/bgs2-where-can-i-find-firmware-v4>