Rocket®

MultiValue University 2015

D3 Security – BASIC SSL Functions

Lab Guide

Developed by
D3 MVU Team

Lab Overview

Abstract

This lab demonstrates the use of the FlashBASIC SSL functions. These functions enable your server-side code to protect your data in transmission by using encryption. The API for these functions is briefly presented, followed by a pair of sample programs that uses all of the functions to communicate securely.

This lab used a controlled environment at MV University. You can do this exercise in your environment if you download a special account – SSL. This can be found at our github site:

https://github.com/RocketSoftware/multivalue-lab

About the Lab Environment

The lab environment uses the following:

Desirable Prerequisites:

- Some BASIC programming skills
- D3 10.1 Linux, D3 9.2 Windows
- SSL Account

Lab Overview

- Time estimate: 70 minutes
- There are four sections to this lab:
  - Section 1: Overview of the FlashBASIC SSL functions
  - Section 2: Walkthrough the server sample
  - Section 3: Walkthrough the client sample
  - Section 4: Running the sample programs

Exercise 1: Overview of the FlashBASIC SSL functions

Purpose of the Exercise

This exercise briefly goes over some of the core FlashBASIC communications functions for both non-secure and secure (SSL) communication.

After this exercise you will be able to:

List the FlashBASIC functions used in secure communication

Understand how they differ from their non-secure counterparts

Exercise Instructions

Perform the following steps:

__ 1. Functions that are common to both regular and secure communication

`%socket()` – creates a socket resource and returns a handle to it

`%bind()` – binds a socket resource to a port

`%listen()` – informs the OS how many concurrent connections to accept

__ 2. Functions that are used in regular (non-secure) communication

- **Unix and Windows**
  - **`%accept()`** – accepts connections from a client, and returns the client's IP address, port, and a new handle to communicate using this specific connection
  - **`%connect()`** – connects to the given server on the given port
- **Unix Only**
  - **`%close()`** – closes the connection and releases the socket resource. For the server, this must be called twice: once for the handle returned by **`%socket()`** and once for the handle returned by **`%accept()`**.
  - **`%read()`** – reads data from the connection
  - **`%write()`** – writes data to the connection

For **`%read()`** and **`%write()`** the client uses the handle returned by **`%socket()`**, while the server uses the handle returned by **`%accept()`**.

- **Windows Only**
  - **%closesocket()** – closes the connection and releases the socket resource. For the server, this must be called twice: once for the handle returned by **%socket()** and once for the handle returned by **%accept()**.
  - **%recv()** – reads data from the connection
  - **%send()** – writes data to the connection

  For **%recv()** and **%send()** the client uses the handle returned by **%socket()**, while the server uses the handle returned by **%accept()**.

__ 3. Functions that are used in secure (SSL) communication

Both Unix and Windows use the same functions

- **%accept_ssl()** – accepts connections from a client, and returns the client's IP address, port, and a new handle to communicate using this specific connection. Unlike its non-secure counterpart, it also requires the names of the certificate and private key files to load from the host OS.
- **%connect_ssl()** – connects to the given server on the given port. Unlike its non-secure counterpart, it returns a new handle to communicate using the specific connection.
- **%read_ssl()** - reads data from the connection
- **%write_ssl()** – writes data to the connection
- **%close_ssl()** – closes the connection. Unlike its non-secure counterpart, it takes two parameters: the handle returned by **%socket()** and either the handle returned by **%accept_ssl()** (server) or **%connect_ssl()** (client).

Exercise 1 summary: Familiarity with the FlashBASIC secure communications functions and how they differ from their non-secure counterparts.

End of Exercise 1

Exercise 2: Walkthrough the server sample

Purpose of the Exercise

This exercise walks through the FlashBASIC server sample program that uses the secure (SSL) communication functions.

After this exercise you will:

Become familiar with the server sample program

Understand how to use the FlashBASIC secure communication functions

Exercise Instructions

Perform the following steps:

__ 1. Log in to D3.

a. Do one of the following:

- On Linux, from the shell, enter **d3**.
- On Windows, Telnet to localhost.

b. Respond to the prompts as shown below:

```
user id: dm
master dictionary: SSL
```

__ 2. Review the SSL,bp, sslserver program.

There are multiple methods to review. For example:

- **ed SSL,bp, sslserver**, or
- **u SSL,bp, sslserver**, or
- **ct SSL,bp, sslserver**

Each section is described in its own step below.

In summary, the sslserver program asks the user if IPv4 or IPV6 is being used and the port number on which to communicate. It then listens for a connection from the client. Once a connection has been accepted, it behaves as a synchronous chat program, alternating between reading (listening), prompting the user for a message, and writing (talking) with the sslclient program.

The sslclient walkthrough exercise is very similar to this exercise. Therefore, some of the verbosity present in this walkthrough is eliminated from the sslclient

walkthrough. Translation: focus your attention on this exercise and refer back to it as needed.

__ 3. cfunction, includes, and program inputs.

Notice the **cfucntion** and **include** statements at the start of the program:

```
cfunction socket.builtin

include dm,bp,includes sysid.inc

include dm,bp,unix.h socket.h
```

The **cfunction** statement allows the program to recognize the communication related FlashBASIC %functions.

The **include** statements provide the necessary platform specific constants used when working with the FlashBASIC %functions.

The program then gathers the inputs required to listen for a client.

__ 4. Create a socket.

```
socketfd = %socket( addressFamily, SOCK$STREAM, 0 )
```

The **%socket()** function is used to create a socket structure. This allocates memory that is used to maintain both programmer provided configuration settings and internal settings.

The parameters instruct it to use the user specified IPv4 or IPv6 addressing, to communicate using streaming, and to let the system determine the best protocol. It returns a handle to the socket structure. Hang on to this. It is needed by other communication functions and also to close/free the memory when done.

__ 5. Bind the socket to a local port.

```
rtn = %bind( socketfd, addressFamily, INADDR$ANY, serverPort+0 )
```

The **%bind()** function is used to bind the socket resource to a port.

Among other things, the parameters include the handle to the socket returned by the **%socket()** call and the port.

Attention!

- Notice that the port parameter is passed to the function as follows: **serverPort+0**

- Internally, the %functions are implemented using C and C++, both of which are strongly typed languages. BASIC is not as strongly typed and typically needs to be coerced to pass the parameter as the correct type.
- In this case, **serverPort** was collected using the BASIC input statement, which stores the value as a string. The **+0** coerces BASIC into using the value as a number, which is what is required by **%bind()**.
- This is not specific to **%bind()** or even the communication functions. Be mindful of this when dealing with any FlashBASIC %functions.
- As a side exercise, try the following from TCL:
  search dm,bp,
- When prompted, type the following and press <Enter> twice.
  +0
- Notice that the **+0** in conjunction with %functions appears often.

__ 6. Wait for an incoming connection

```
rtn = %listen( socketfd, 1 )
```

The **%listen()** function tells the OS the maximum number of concurrent connections it can accept on this port. In this example, only one will be allowed.

__ 7. Accept a connection

```
char clientAddress[ 46 ] ;* IPv6 requires a char array.

rtn = %accept_ssl( socketfd, clientAddress, &clientPort, "mvu2015-
certificate.pem", "mvu2015-privatekey.pem", &sslfd )
```

The **%accept_ssl()** function accepts an incoming SSL connection from a client. The handle to the socket structure returned by the **%socket()** function is passed as a parameter.

Notice how the **clientAddress** parameter for IPv6 is prepared. A char array is allocated. This differs from IPv4, which allocates an integer and passes it with the **+0**. This is the only API difference between the IPv4 and IPv6 **%accept_ssl()** functions. The **clientAddress** is an output parameter. If a connection is successfully accepted, it will contain the IP address of the client.

The **clientPort** is also an integer output parameter, and upon a successful connection, will contain the port on which the client is listening to responses from the server.

Attention!

- Notice how the an integer output parameter is passed: **&clientPort**
- This tells BASIC to pass the address of the **clientPort** variable to the %function, not the value it contains. Having access to the address allows the %function to set the value of the variable. That is, it allows the variable to be an output parameter that is populated by the call.

The **%accept_ssl()** function also takes the names of two files containing the certificate and private key necessary for encrypted communication. There is a wealth of information about public key cryptography, certificates, keys, etc. A few very brief highlights are listed here in regards to how they were used in this example program.

- PEM files can contain one or more entries. For example, a single PEM file may contain both the certificate and private key. Then the same file name may be given for both certificate and private key file name parameters.
- The **openssl** utility was used to generate both the certificate and private key used in this lab. The following command was used.

```
openssl req -x509 -newkey rsa:2048 -keyout mvu2015-
privatekey.pem -out mvu2015-certificate.pem -days 60 -nodes
```

Finally, the **%accept_ssl()** function takes **sslfd**, which is an integer output parameter. On success, **sslfd** is a handle. Hang on to this. It is needed by other communication functions and also to close/free the memory when done.

__ 8. Use this new fd to communicate with the client.

This part of the program displays the client's IP address and port.
It then goes into a loop reading from and writing to the client.

__ 9. Read data from the socket.

```
char readBuffer[ 10000 ]

byteCount = %read_ssl( sslfd, readBuffer, 10000 )
```

This function reads data from the client. The programmer passes in the **sslfd** returned by **%accept_ssl()**, a char array output buffer that will be populated by the data read, and the maximum size of the buffer.
It returns the number of bytes read and displays the data.

```
print "From Client: " : readBuffer[ 1, byteCount ]
```

Attention! Notice how the text is extracted. This is necessary so that anything beyond the end of the actual string is eliminated.

__ 10. Ask for data to send.

This simply prompts the user for input to send to the client.

__ 11. Write data to socket.

```
byteCount = %write_ssl( sslfd, writeBuffer, len( writeBuffer )+0 )
```

This function writes data to the client. The programmer passes in the **sslfd** returned by **%accept_ssl()**, the buffer containing the user's input, and the length of the buffer. Notice the **+0**, which coerces the result of **len()** into an integer.

__ 12. Close socket.

```
rtn = %close_ssl( socketfd, sslfd )
```

This function wraps up the secure communication and releases the resources associated with the **socketfd** and **sslfd** handles.

Unlike its non-secure counterpart, which requires the handles returned by **%socket()** and **%accept()** to be closed separately, **%close_ssl()** closes both handles with a single call.

Exercise 2 summary: Familiarity with the sslserver sample program and how to use the secure communication functions.

End of Exercise 2

Exercise 3: Walkthrough the client sample

Purpose of the Exercise

This exercise walks through the FlashBASIC client sample program that uses the secure (SSL) communication functions.

After this exercise you will:

Become familiar with the client sample program

Understand how to use the FlashBASIC secure communication functions

Exercise Instructions

Perform the following steps:

__ 1. Log in to D3

a. Do one of the following:

- On Linux, from the shell, enter **d3**.
- On Windows, Telnet to localhost.

b. Respond to the prompts as shown below:

```
user id: dm
master dictionary: FLE
```

__ 2. Review the SSL,bp, sslclient program.

There are multiple methods to review. For example:

- **ed SSL,bp, sslclient**, or
- **u SSL,bp, sslclient**, or
- **ct SSL,bp, sslclient**

Each section is described in its own step below.

In summary, the sslclient program asks the user if IPv4 or IPV6 is being used, the host to which to connect, and the port number on which the server is listening. It then connects to the server. Once the connection has been accepted, it behaves as a synchronous chat program, alternating between prompting the user for a message, writing (talking), and reading (listening) with the sslserver program. Much of this is duplicated from the previous exercise discussing the sslserver program.

The core difference between the server and client is that the sslserver **%bind()**, **%listen()**, and **%accept_ssl()** calls are replaced with the **%connect_ssl()** call.

__ 3. cfunction, includes, and program inputs.

Notice the **cfucntion** and **include** statements at the start of the program:

```
cfunction socket.builtin

include dm,bp,includes sysid.inc

include dm,bp,unix.h socket.h
```

The **cfunction** statement allows the program to recognize the communication related FlashBASIC %functions.

The **include** statements provide the necessary platform specific constants used when working with the FlashBASIC %functions.

The program then gathers the inputs required to connect to a server.

__ 4. Create a socket

```
socketfd = %socket( addressFamily, SOCK$STREAM, 0 )
```

The **%socket()** function is used to create a socket structure. This allocates memory that is used to maintain both programmer provide configuration settings and internal settings.

The parameters instruct it to use the user specified IPv4 or IPv6 addressing, to communicate using streaming, and to let the system determine the best protocol.

It returns a handle to the socket structure. Hang on to this. It is needed by other communication functions and also to close/free the memory when done.

__ 5. Connect to the server

```
rtn = %connect_ssl( socketfd, addressFamily, hostname, port+0, &sslfd
)
```

The **%connect_ssl()** function opens a connection to the server.

Among other things, the parameters include the handle to the socket returned by the **%socket()** call, the hostname, and the port.

Notice the **+0** on the port parameter to coerce the variable into an integer. See the sslserver walkthrough exercise for a more detailed explanation on this.

The **%connect_ssl()** function takes **sslfd**, which is an integer output parameter. On success, sslfd is a handle. Hang on to this. It is needed by other communication functions and also to close/free the memory when done.

__ 6. Ask for data to send

This simply prompts the user for input to send to the server.

__ 7. Write data to socket

```
byteCount = %write_ssl( sslfd, writeBuffer, len( writeBuffer )+0 )
```

This function writes data to the server. The programmer passes in the sslfd returned by **%connect_ssl()**, the buffer containing the user's input, and the length of the buffer. Notice the **+0**, which coerces the result of **len()** into an integer.

__ 8. Read data from the socket

```
char readBuffer[ 10000 ]

byteCount = %read_ssl( sslfd, readBuffer, 10000 )
```

This function reads data from the client. The programmer passes in the **sslfd** returned by **%connect_ssl()**, a char array output buffer that will be populated by the data read, and the maximum size of the buffer.

It returns the number of bytes read and displays the data.

```
print "From Server: " : readBuffer[ 1, byteCount ]
```

Attention! Notice how the text is extracted. This is necessary so that anything beyond the end of the actual string is eliminated.

__ 9. Close socket

```
rtn = %close_ssl( socketfd, sslfd )
```

This function wraps up the secure communication and releases the resources associated with the **socketfd** and **sslfd** handles.

Unlike its non-secure counterpart, which requires the handles returned by **%socket()** and **%accept()** to be closed separately, **%close_ssl()** closes both handles with a single call.

Exercise 3 summary: Familiarity with the sslclient sample program and how to use the secure communication functions.

End of Exercise 3

Exercise 4: Running the sample programs

Purpose of the Exercise

This exercise demonstrates how to run the sslserver and sslclient example programs.

After this exercise you will:

- Become familiar with how to run the sample programs

Exercise Instructions

Perform the following steps:

__ 1. Notes:

- While running these programs, pressing <Enter> at any prompt without input will exit the program.
- This example demonstrates both the client and the server running on the same machine. You may run them on different machines, if you'd like. Simply make sure you can ping each machine from the other.

__ 2. Log in to D3.

a. Do one of the following:
   - On Linux, from the shell, enter **d3**.
   - On Windows, Telnet to localhost.

b. Respond to the prompts as shown below:

```
user id: dm
master dictionary: SSL
```

__ 3. Log in to D3, again.

a. Do one of the following:
   - On Linux, from the shell, enter **d3**.
   - On Windows, Telnet to localhost.

b. Respond to the prompts as shown below:

```
user id: dm
master dictionary: SSL
```

__ 4. Lookup your IP address

a. At TCL from the first line lookup the server's IP address as described below. Make note of it. It will be needed when starting the client.
   You may use either the IPv4 or IPv6 address.
   You may also use the hostname, but be sure to ping it to see if an IPv4 or IPv6 address is returned because the programs need to know which address format to use even when specifying a hostname.
   - On Linux:
     !ifconfig
   - On Windows:
     !ipconfig

__ 5. Start the server

a. At TCL, from the first line, enter:
   sslserver
b. Enter either 4 or 6 to select IPv4 or IPv6.
c. Enter the port number on which to listen. For example, 11122 (full-house, you are in Vegas).
   Several messages will be displayed indicating that the server is now waiting for a connection from a client.

__ 6. Start the client

a. At TCL, from the second line, enter:
   sslclient
b. Enter either 4 or 6 to select IPv4 or IPv6. This must match what was entered for the server.
c. Enter the hostname or IP address of the server as determined previously.
d. Enter the port number on which the server is listening. For example, 11122.

__ 7. Confirm that the client and server connected

The server will display that the connection was accepted, show the IP address and port from the client, and indicate that it is reading from the SSL connection. The client will display a message indicating that it is connecting with SSL and then prompt for input.

__ 8. Chat

The programs will alternate prompting for input.

```
Enter some data to send (enter to quit):
```

Enter some text. It will be read and displayed by the other program.

__ 9. Concluding the program

a. When done, press <Enter> at the prompt to quit the program.
   The secure connection will be closed and the resources released.

Exercise 4 summary: Running the sample secure communication programs.

End of Exercise 4