

# WinAPI

- [Networking](#)
  - [Win32 Minimum Client Socket Transaction](#)
  - [Win32 SChannel TLS](#)
  - [Win32 SChannel TLS message send \(gross\)](#)

# Networking

# Win32 Minimum Client Socket Transaction

This is the minimal WinAPI using winsock2.h

References:

- <https://learn.microsoft.com/en-us/windows/win32/winsock/using-secure-socket-extensions>
- <https://learn.microsoft.com/en-us/windows/win32/api/winsock2/nf-winsock2-wssocketeta>
- [https://learn.microsoft.com/en-us/windows/win32/api/winsock2/ns-winsock2-wsaproTOCOL\\_infoa](https://learn.microsoft.com/en-us/windows/win32/api/winsock2/ns-winsock2-wsaproTOCOL_infoa)
- <https://learn.microsoft.com/en-us/windows/win32/api/ws2tcpip/nf-ws2tcpip-wsasetsocketsecurity>
- <https://learn.microsoft.com/en-us/windows/win32/api/winsock/nf-winsock-wsastartup>
- [https://learn.microsoft.com/en-us/windows/win32/api/mstcpip/ne-mstcpip-socket\\_security\\_protocol](https://learn.microsoft.com/en-us/windows/win32/api/mstcpip/ne-mstcpip-socket_security_protocol)

Requires linking against ws2\_32

```
#include <winsock2.h>
#include <mstcpip.h>

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR lpCmdLine, int nCmdShow){
    WORD wVersionRequired = MAKEWORD(2, 2);
    WSADATA wsaData;
    WSAStartup(wVersionRequired, &wsaData);

    // Create a socket
    SOCKET socket = WSASocketA(AF_INET, SOCK_STREAM, IPPROTO_TCP, NULL, 0, 0);
    if(socket == INVALID_SOCKET){
        cprintf("Socket was invalid with error %ld\n", WSAGetLastError());
    }
```

```

// Establish the peer
struct sockaddr_in peer;
peer.sin_family = AF_INET;
peer.sin_addr.s_addr = inet_addr("127.0.0.1");
peer.sin_port = htons(50007);

// Connect to a peer.
int result = WSAConnect(socket, (SOCKADDR*) &peer, sizeof(peer), NULL, NULL, NULL, NULL);
cprintf("Result: %ld\n", result);


// Send some data
// We use the WSABUF here.
WSABUF buffer = {0};
char* text = "This is some data. Hello.";
buffer.buf = text;
buffer.len = strlen(text) + 1;

// Send
DWORD dwBytesSent = 0;
result = WSASend(socket, &buffer, 1, &dwBytesSent, 0, NULL, NULL);
if (result == SOCKET_ERROR) {
    result = WSAGetLastError();
    cprintf("WSASend returned error %ld\n", result);
} else {
    cprintf("Sent %u bytes across the channel.\n", dwBytesSent);
}


// Receive some data
// Set the recipient buffer.
#define maximumRecvSize 4000
char* recvBuffer = (char*) valloc(maximumRecvSize);
buffer.len = maximumRecvSize;
buffer.buf = recvBuffer;

DWORD dataFlags = 0;
//Request the data.
result = WSAREcv(socket, &buffer, 1, &dwBytesSent, &dataFlags, NULL, NULL);
if (result == SOCKET_ERROR) {
    result = WSAGetLastError();

```

```
        cprintf("WSARecv returned error %ld\n", result);
    } else {
        cprintf("Received %u bytes across the channel.\n", dwBytesSent);
        cprintf("Data: %s\n", buffer.buf);
    }

    // Close the socket
    result = closesocket(socket);
    cprintf("Result: %ld\n", result);

    WSACleanup();

    cprintf("Socket: %u\n", socket);
    return 0;
}
```

# Win32 SChannel TLS

MVP here is Copilot. There's actually no solid guides on this at all. There's a gist but it missed a step in my case.

“ Here’s a basic outline and some example code to get you started:

## 1. Initialize the Security Package:

- Use `AcquireCredentialsHandle` to obtain a handle to the credentials.

## Create a Socket:

- Create a socket and connect it to the server.

## Perform the TLS Handshake:

- Use `InitializeSecurityContext` and `AcceptSecurityContext` to perform the handshake.

## Send and Receive Data:

- Encrypt data using `EncryptMessage` and decrypt data using `DecryptMessage`.

```
#include <winsock2.h>
#include <windows.h>
#include <sspi.h>
#include <secext.h>
#include <stdio.h>

#pragma comment(lib, "ws2_32.lib")
#pragma comment(lib, "secur32.lib")

int main() {
    WSADATA wsaData;
    SOCKET sock;
    struct sockaddr_in server;
    CredHandle hCred;
    CtxtHandle hCtxt;
```

```

SECURITY_STATUS secStatus;
TimeStamp tsExpiry;
SecBufferDesc OutBuffer;
SecBuffer OutBuffers[1];
ULONG ContextAttributes;

// Initialize Winsock
WSAStartup(MAKEWORD(2, 2), &wsaData);

// Create a socket
sock = socket(AF_INET, SOCK_STREAM, 0);
server.sin_addr.s_addr = inet_addr("127.0.0.1"); // Server IP
server.sin_family = AF_INET;
server.sin_port = htons(443); // HTTPS port

// Connect to the server
connect(sock, (struct sockaddr *)&server, sizeof(server));

// Initialize the security package
secStatus = AcquireCredentialsHandle(
    NULL, UNISP_NAME, SECPKG_CRED_OUTBOUND, NULL, NULL, NULL, NULL, &hCred, &tsExpiry);

if (secStatus != SEC_E_OK) {
    printf("AcquireCredentialsHandle failed: 0x%x\n", secStatus);
    return 1;
}

// Initialize the security context
OutBuffers[0].pvBuffer = NULL;
OutBuffers[0].BufferType = SECBUFFER_TOKEN;
OutBuffers[0].cbBuffer = 0;
OutBuffer.cBuffers = 1;
OutBuffer.pBuffers = OutBuffers;
OutBuffer.ulVersion = SECBUFFER_VERSION;

secStatus = InitializeSecurityContext(
    &hCred, NULL, NULL, ISC_REQ_SEQUENCE_DETECT | ISC_REQ_REPLAY_DETECT |
ISC_REQ_CONFIDENTIALITY,
    0, SECURITY_NATIVE_DREP, NULL, 0, &hCtxt, &OutBuffer, &ContextAttributes, &tsExpiry);

```

```

if (secStatus != SEC_I_CONTINUE_NEEDED) {
    printf("InitializeSecurityContext failed: 0x%x\n", secStatus);
    return 1;
}

// Send the initial token to the server
send(sock, (char *)OutBuffers[0].pvBuffer, OutBuffers[0].cbBuffer, 0);

// Continue the handshake...

// Clean up
closesocket(sock);
WSACleanup();
FreeCredentialsHandle(&hCred);
DeleteSecurityContext(&hCtxt);

return 0;
}

```

## “ TLS Handshake Process

### 1. Initialize the Security Package:

- Use AcquireCredentialsHandle to obtain a handle to the credentials.

### Create a Socket:

- Create a socket and connect it to the server.

### Perform the TLS Handshake:

- Use InitializeSecurityContext and AcceptSecurityContext to perform the handshake.

### Send and Receive Data:

- Encrypt data using EncryptMessage and decrypt data using DecryptMessage.

```

#include <winsock2.h>
#include <windows.h>
#include <sspi.h>
#include <secext.h>

```



```

#include <stdio.h>

#pragma comment(lib, "ws2_32.lib")
#pragma comment(lib, "secur32.lib")

int main() {
    WSADATA wsaData;
    SOCKET sock;
    struct sockaddr_in server;
    CredHandle hCred;
    CtxtHandle hCtxt;
    SECURITY_STATUS secStatus;
    TimeStamp tsExpiry;
    SecBufferDesc OutBuffer, InBuffer;
    SecBuffer OutBuffers[1], InBuffers[1];
    ULONG ContextAttributes;
    char recvBuffer[4096];
    int recvLen;

    // Initialize Winsock
    WSASStartup(MAKEWORD(2, 2), &wsaData);

    // Create a socket
    sock = socket(AF_INET, SOCK_STREAM, 0);
    server.sin_addr.s_addr = inet_addr("127.0.0.1"); // Server IP
    server.sin_family = AF_INET;
    server.sin_port = htons(443); // HTTPS port

    // Connect to the server
    connect(sock, (struct sockaddr *)&server, sizeof(server));

    // Initialize the security package
    secStatus = AcquireCredentialsHandle(
        NULL, UNISP_NAME, SECPKG_CRED_OUTBOUND, NULL, NULL, NULL, NULL, &hCred, &tsExpiry);

    if (secStatus != SEC_E_OK) {
        printf("AcquireCredentialsHandle failed: 0x%x\n", secStatus);
        return 1;
    }
}

```

```

// Initialize the security context
OutBuffers[0].pvBuffer = NULL;
OutBuffers[0].BufferType = SECBUFFER_TOKEN;
OutBuffers[0].cbBuffer = 0;
OutBuffer.cBuffers = 1;
OutBuffer.pBuffers = OutBuffers;
OutBuffer.ulVersion = SECBUFFER_VERSION;

secStatus = InitializeSecurityContext(
    &hCred, NULL, NULL, ISC_REQ_SEQUENCE_DETECT | ISC_REQ_REPLAY_DETECT |
ISC_REQ_CONFIDENTIALITY,
    0, SECURITY_NATIVE_DREP, NULL, 0, &hCtxt, &OutBuffer, &ContextAttributes, &tsExpiry);

if (secStatus != SEC_I_CONTINUE_NEEDED) {
    printf("InitializeSecurityContext failed: 0x%x\n", secStatus);
    return 1;
}

// Send the initial token to the server
send(sock, (char *)OutBuffers[0].pvBuffer, OutBuffers[0].cbBuffer, 0);

// Receive the server's response
recvLen = recv(sock, recvBuffer, sizeof(recvBuffer), 0);
if (recvLen <= 0) {
    printf("Failed to receive data from server\n");
    return 1;
}

// Process the server's response
InBuffers[0].pvBuffer = recvBuffer;
InBuffers[0].BufferType = SECBUFFER_TOKEN;
InBuffers[0].cbBuffer = recvLen;
InBuffer.cBuffers = 1;
InBuffer.pBuffers = InBuffers;
InBuffer.ulVersion = SECBUFFER_VERSION;

secStatus = InitializeSecurityContext(
    &hCred, &hCtxt, NULL, ISC_REQ_SEQUENCE_DETECT | ISC_REQ_REPLAY_DETECT |
ISC_REQ_CONFIDENTIALITY,
    0, SECURITY_NATIVE_DREP, &InBuffer, 0, &hCtxt, &OutBuffer, &ContextAttributes, &tsExpiry);

```

```

if (secStatus != SEC_E_OK && secStatus != SEC_I_CONTINUE_NEEDED) {
    printf("InitializeSecurityContext failed: 0x%x\n", secStatus);
    return 1;
}

// Send the next token to the server if needed
if (OutBuffers[0].cbBuffer > 0) {
    send(sock, (char *)OutBuffers[0].pvBuffer, OutBuffers[0].cbBuffer, 0);
}

// Continue the handshake until it completes...

// Clean up
closesocket(sock);
WSACleanup();
FreeCredentialsHandle(&hCred);
DeleteSecurityContext(&hCtxt);

return 0;
}

```

“ Once you’ve established the TLS context, you can send and receive messages securely by using the `EncryptMessage` and `DecryptMessage` functions provided by the SSPI (Security Support Provider Interface). Here’s how you can do it:

## Sending Messages Securely

### 1. **Encrypt the Message:**

- Use `EncryptMessage` to encrypt the data before sending it over the network.

### 2. **Send the Encrypted Message:**

- Use the standard `send` function to send the encrypted data.

## Receiving Messages Securely

### 1. **Receive the Encrypted Message:**

- Use the standard `recv` function to receive the encrypted data.

```

#include <winsock2.h>
#include <windows.h>
#include <sspi.h>
#include <secext.h>
#include <stdio.h>

#pragma comment(lib, "ws2_32.lib")
#pragma comment(lib, "secur32.lib")

void send_secure_message(SOCKET sock, CtxtHandle *hCtxt, const char *message) {
    SecBufferDesc Message;
    SecBuffer Buffers[4];
    SECURITY_STATUS secStatus;

    // Prepare the message to be encrypted
    Buffers[0].pvBuffer = (void *)message;
    Buffers[0].cbBuffer = (unsigned long)strlen(message);
    Buffers[0].BufferType = SECBUFFER_DATA;

    Buffers[1].BufferType = SECBUFFER_EMPTY;
    Buffers[2].BufferType = SECBUFFER_EMPTY;
    Buffers[3].BufferType = SECBUFFER_EMPTY;

    Message.ulVersion = SECBUFFER_VERSION;
    Message.cBuffers = 4;
    Message.pBuffers = Buffers;

    // Encrypt the message
    secStatus = EncryptMessage(hCtxt, 0, &Message, 0);
    if (secStatus != SEC_E_OK) {
        printf("EncryptMessage failed: 0x%x\n", secStatus);
        return;
    }

    // Send the encrypted message
    send(sock, (char *)Buffers[0].pvBuffer, Buffers[0].cbBuffer, 0);
}

void receive_secure_message(SOCKET sock, CtxtHandle *hCtxt) {
    char recvBuffer[4096];

```

```

int recvLen;
SecBufferDesc Message;
SecBuffer Buffers[4];
SECURITY_STATUS secStatus;

// Receive the encrypted message
recvLen = recv(sock, recvBuffer, sizeof(recvBuffer), 0);
if (recvLen <= 0) {
    printf("Failed to receive data from server\n");
    return;
}

// Prepare the buffer for decryption
Buffers[0].pvBuffer = recvBuffer;
Buffers[0].cbBuffer = recvLen;
Buffers[0].BufferType = SECBUFFER_DATA;

Buffers[1].BufferType = SECBUFFER_EMPTY;
Buffers[2].BufferType = SECBUFFER_EMPTY;
Buffers[3].BufferType = SECBUFFER_EMPTY;

Message.ulVersion = SECBUFFER_VERSION;
Message.cBuffers = 4;
Message.pBuffers = Buffers;

// Decrypt the message
secStatus = DecryptMessage(hCtxt, &Message, 0, NULL);
if (secStatus != SEC_E_OK) {
    printf("DecryptMessage failed: 0x%x\n", secStatus);
    return;
}

// Print the decrypted message
printf("Decrypted message: %s\n", (char *)Buffers[0].pvBuffer);
}

int main() {
    WSADATA wsaData;
    SOCKET sock;
    struct sockaddr_in server;

```

```

CredHandle hCred;
CtxtHandle hCtxt;
SECURITY_STATUS secStatus;
TimeStamp tsExpiry;
SecBufferDesc OutBuffer, InBuffer;
SecBuffer OutBuffers[1], InBuffers[1];
ULONG ContextAttributes;
char recvBuffer[4096];
int recvLen;

// Initialize Winsock
WSAStartup(MAKEWORD(2, 2), &wsaData);

// Create a socket
sock = socket(AF_INET, SOCK_STREAM, 0);
server.sin_addr.s_addr = inet_addr("127.0.0.1"); // Server IP
server.sin_family = AF_INET;
server.sin_port = htons(443); // HTTPS port

// Connect to the server
connect(sock, (struct sockaddr *)&server, sizeof(server));

// Initialize the security package
secStatus = AcquireCredentialsHandle(
    NULL, UNISP_NAME, SECPKG_CRED_OUTBOUND, NULL, NULL, NULL, NULL, &hCred, &tsExpiry);

if (secStatus != SEC_E_OK) {
    printf("AcquireCredentialsHandle failed: 0x%x\n", secStatus);
    return 1;
}

// Initialize the security context
OutBuffers[0].pvBuffer = NULL;
OutBuffers[0].BufferType = SECBUFFER_TOKEN;
OutBuffers[0].cbBuffer = 0;
OutBuffer.cBuffers = 1;
OutBuffer.pBuffers = OutBuffers;
OutBuffer.ulVersion = SECBUFFER_VERSION;

secStatus = InitializeSecurityContext(

```

```

        &hCred, NULL, NULL, ISC_REQ_SEQUENCE_DETECT | ISC_REQ_REPLAY_DETECT |
ISC_REQ_CONFIDENTIALITY,
        0, SECURITY_NATIVE_DREP, NULL, 0, &hCtxt, &OutBuffer, &ContextAttributes, &tsExpiry);

if (secStatus != SEC_I_CONTINUE_NEEDED) {
    printf("InitializeSecurityContext failed: 0x%x\n", secStatus);
    return 1;
}

// Send the initial token to the server
send(sock, (char *)OutBuffers[0].pvBuffer, OutBuffers[0].cbBuffer, 0);

// Receive the server's response
recvLen = recv(sock, recvBuffer, sizeof(recvBuffer), 0);
if (recvLen <= 0) {
    printf("Failed to receive data from server\n");
    return 1;
}

// Process the server's response
InBuffers[0].pvBuffer = recvBuffer;
InBuffers[0].BufferType = SECBUFFER_TOKEN;
InBuffers[0].cbBuffer = recvLen;
InBuffer.cBuffers = 1;
InBuffer.pBuffers = InBuffers;
InBuffer.ulVersion = SECBUFFER_VERSION;

secStatus = InitializeSecurityContext(
    &hCred, &hCtxt, NULL, ISC_REQ_SEQUENCE_DETECT | ISC_REQ_REPLAY_DETECT |
ISC_REQ_CONFIDENTIALITY,
    0, SECURITY_NATIVE_DREP, &InBuffer, 0, &hCtxt, &OutBuffer, &ContextAttributes, &tsExpiry);

if (secStatus != SEC_E_OK && secStatus != SEC_I_CONTINUE_NEEDED) {
    printf("InitializeSecurityContext failed: 0x%x\n", secStatus);
    return 1;
}

// Send the next token to the server if needed
if (OutBuffers[0].cbBuffer > 0) {
    send(sock, (char *)OutBuffers[0].pvBuffer, OutBuffers[0].cbBuffer, 0);

```

```

}

// Now you can send and receive secure messages
send_secure_message(sock, &hCtxt, "Hello, secure world!");
receive_secure_message(sock, &hCtxt);

// Clean up
closesocket(sock);
WSACleanup();
FreeCredentialsHandle(&hCred);
DeleteSecurityContext(&hCtxt);

return 0;
}

```

- “
1. **Encrypt the Message:** Use `EncryptMessage` to encrypt the data before sending it.
  2. **Send the Encrypted Message:** Use `send` to send the encrypted data.
  3. **Receive the Encrypted Message:** Use `recv` to receive the encrypted data.
  4. **Decrypt the Message:** Use `DecryptMessage` to decrypt the received data.

---

The use of four buffers in the `SecBuffer` array within the `SecBufferDesc` structure is a common practice when working with SSPI functions like `EncryptMessage` and `DecryptMessage`. Here's why:

1. **Data Buffer:**
  - This buffer contains the actual data to be encrypted or decrypted. It's the primary buffer that holds the plaintext (for encryption) or ciphertext (for decryption).
2. **Signature Buffer:**
  - This buffer is used to hold the signature or MAC (Message Authentication Code) generated during encryption. It ensures the integrity and authenticity of the message.
3. **Padding Buffer:**
  - This buffer is used to hold any padding bytes that might be required to align the data to a specific block size. Padding is often necessary in block cipher modes of encryption.



#### 4. **Extra Buffer:**

- This buffer can be used for any additional data that might be needed during the encryption or decryption process. It often remains empty but is included to handle any extra requirements that might arise.

## Example Breakdown:

When you call `EncryptMessage`, the function might use these buffers as follows:

- **Buffers[0]:** Contains the plaintext data to be encrypted.
- **Buffers[1]:** Will be filled with the signature or MAC.
- **Buffers[2]:** Will be filled with any necessary padding.
- **Buffers[3]:** Typically remains empty but is available for any additional data.

Similarly, when you call `DecryptMessage`, the function will use these buffers to process the received encrypted data and extract the plaintext, verify the signature, and handle any padding.

## Code Example:

Here's a snippet showing how these buffers are set up for encryption:

```
SecBuffer Buffers[4];
SecBufferDesc Message;

// Prepare the message to be encrypted
Buffers[0].pvBuffer = (void *)message; // Plaintext data
Buffers[0].cbBuffer = (unsigned long)strlen(message);
Buffers[0].BufferType = SECBUFFER_DATA;

Buffers[1].BufferType = SECBUFFER_EMPTY; // Signature
Buffers[2].BufferType = SECBUFFER_EMPTY; // Padding
Buffers[3].BufferType = SECBUFFER_EMPTY; // Extra

Message.ulVersion = SECBUFFER_VERSION;
Message.cBuffers = 4;
Message.pBuffers = Buffers;
```

```
// Encrypt the message
SECURITY_STATUS secStatus = EncryptMessage(hCtxt, 0, &Message, 0);
if (secStatus != SEC_E_OK) {
    printf("EncryptMessage failed: 0x%x\n", secStatus);
    return;
}

// Send the encrypted message
send(sock, (char *)Buffers[0].pvBuffer, Buffers[0].cbBuffer, 0);
```

# Win32 SChannel TLS message send (gross)

This works but it's super gross.

For TLS you have to Init like 3 times to finish the exchange and negotiation.

```
typedef struct {
    SOCKET socket;
    CredHandle hCredential;
    CtxtHandle hContext;
    char* ServerCertificate;
    unsigned int received;
    SecPkgContext_StreamSizes sizes;
} TLS_SOCKET;

// https://gist.github.com/mmozeiko/c0dfcc8fec527a90a02145d2cc0bfb6d
// Use SCHANNEL

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR lpCmdLine, int nCmdShow){
    WORD wVersionRequired = MAKEWORD(2, 2);
    WSADATA wsaData;
    WSAStartup(wVersionRequired, &wsaData);
    // WS error codes https://learn.microsoft.com/en-us/windows/win32/winsock/windows-sockets-error-codes-2
    int result;
    // TLS_SOCKET;
    TLS_SOCKET tSocket = {0};
    tSocket.ServerCertificate = valloc(20000);

    // Create a socket
    tSocket.socket = WSASocketA(AF_INET, SOCK_STREAM, IPPROTO_TCP, NULL, 0, 0);
    if(tSocket.socket == INVALID_SOCKET){
        cprintf("Socket was invalid with error %ld\n", WSAGetLastError());
    }
}
```

```

// Establish the peer
struct sockaddr_in peer;
peer.sin_family = AF_INET;
peer.sin_addr.s_addr = inet_addr("127.0.0.1");
peer.sin_port = htons(3333);

// Connect to a peer.
result = WSAConnect(tSocket.socket, (SOCKADDR*) &peer, sizeof(peer), NULL, NULL, NULL, NULL);
if (result == SOCKET_ERROR) {
    result = WSAGetLastError();
    cprintf("WSAConnect returned error %ld\n", result);
} else {
    cprintf("Connected.\n");
}

// Initialise SChannel
// https://learn.microsoft.com/en-us/windows/win32/api/schannel/ns-schannel-schannel_cred
// https://learn.microsoft.com/en-us/windows/win32/api/schannel/ns-schannel-sch_credentials
SCHANNEL_CRED schannelCredential = {
    .dwVersion = SCHANNEL_CRED_VERSION,
    .dwFlags = SCH_CRED_MANUAL_CRED_VALIDATION
    | SCH_CRED_NO_SERVERNAME_CHECK
    | SCH_CRED_NO_DEFAULT_CREDS,
    .grbitEnabledProtocols = SP_PROT_TLS1_2
};

// https://learn.microsoft.com/en-us/windows/win32/api/sspi/nf-sspi-acquirecredentialshandlea
if (AcquireCredentialsHandleA(NULL, UNISP_NAME_A, SECPKG_CRED_OUTBOUND, NULL, &schannelCredential,
NULL, NULL, &tSocket.hCredential, NULL) != SEC_E_OK){
    WSACleanup();
    return -1;
}

CtxtHandle* context = NULL;
int count = 0;

```

```

// Perform the TLS Handshake.
SecBuffer inbuffers[2] = { 0 };
inbuffers[0].BufferType = SECBUFFER_TOKEN;
inbuffers[0].pvBuffer = tSocket.ServerCertificate;
inbuffers[0].cbBuffer = 0;
inbuffers[1].BufferType = SECBUFFER_EMPTY;

SecBuffer outbuffers[1] = { 0 };
outbuffers[0].pvBuffer = malloc(40000);
outbuffers[0].cbBuffer = 40000;
outbuffers[0].BufferType = SECBUFFER_TOKEN;

SecBufferDesc indesc = { SECBUFFER_VERSION, ARRAYSIZE(inbuffers), inbuffers };
SecBufferDesc outdesc = { SECBUFFER_VERSION, ARRAYSIZE(outbuffers), outbuffers };

DWORD flags = ISC_REQ_USE_SUPPLIED_CREDS | ISC_REQ_ALLOCATE_MEMORY | ISC_REQ_CONFIDENTIALITY |
ISC_REQ_REPLAY_DETECT | ISC_REQ_SEQUENCE_DETECT | ISC_REQ_STREAM;
// https://learn.microsoft.com/en-us/windows/win32/api/sspi/nf-sspi-initializesecuritycontexta
SECURITY_STATUS sec = InitializeSecurityContextA(
    &tSocket.hCredential,
    NULL,
    NULL,
    flags,
    0,
    0,
    NULL,
    0,
    &tSocket.hContext,
    &outdesc,
    &flags,
    NULL);

//After the first round this will have a cert in it and the buffer type will become SECBUFFER_EXTRA
if (sec == SEC_E_OK)
{
    cprintf("OK!");
    char* buffer = "Hello world!";
    int size = 0;
    size= bstrlen(buffer);

```

```

    send(tSocket.socket, buffer, size, 0);
    // tls handshake completed
}
else if (sec == SEC_I_INCOMPLETE_CREDENTIALS)
{
    cprintf("INCOMPLETE");
    // server asked for client certificate, not supported here
    result = -1;
}
else if (sec == SEC_I_CONTINUE_NEEDED)
{
    // need to send data to server
    char* buffer = outbuffers[0].pvBuffer;
    int size = outbuffers[0].cbBuffer;
    cprintf("CONTINUE with data %s, %u\n", buffer, size);

    // DWORD dwBytes = 0;
    // char* buffer2 = valloc(50000);
    // dwBytes = recv(tSocket.socket, buffer2, 50000, 0);
    // cprintf("Recv %u bytes '%s'.\n", dwBytes, buffer2);

    while (size != 0)
    {
        int d = send(tSocket.socket, buffer, size, 0);
        if (d <= 0)
        {
            cprintf("Sent %u bytes.\n", d);
        }
        size -= d;
        buffer += d;
        cprintf("Sent %u bytes.\n", d);
    }
    // REceive the response
    int r = recv(tSocket.socket, tSocket.ServerCertificate, 4000, 0);
    cprintf("RECV %u bytes.\n", r);
    inbuffers[0].cbBuffer = r;
    inbuffers[0].BufferType = SECBUFFER_TOKEN;

    //Request the TLS Certificate to negotiate

```

```

sec = InitializeSecurityContextA(
    &tSocket.hCredential,
    &tSocket.hContext,
    NULL,
    flags,
    0,
    0,
    &indesc,
    0,
    &tSocket.hContext,
    &outdesc,
    &flags,
    NULL);

cprintf("SEC: %x\n", sec);
if (sec == SEC_E_BUFFER_TOO_SMALL){
    cprintf("Buffers too small\n");
    // outbuffers[0].pvBuffer = var

}

if (sec != SEC_E_OK && sec != SEC_I_CONTINUE_NEEDED) {
    cprintf("InitializeSecurityContext failed: 0x%x\n", sec);
    return 1;
}

if (outbuffers[0].cbBuffer > 0) {
    send(tSocket.socket, (char *)outbuffers[0].pvBuffer, outbuffers[0].cbBuffer, 0);
}

if (sec == SEC_I_CONTINUE_NEEDED){
    //We need to negotiate further.
    //Receive into the inbuffer.
    cprintf("Further negoatiation.");
    // REceive the response
    int r = recv(tSocket.socket, tSocket.ServerCertificate, 4000, 0);
    cprintf("RECV %u bytes.\n", r);
    inbuffers[0].cbBuffer = r;
    inbuffers[0].BufferType = SECBUFFER_TOKEN;
    sec = InitializeSecurityContextA(

```

```

        &tSocket.hCredential,
        &tSocket.hContext,
        NULL,
        flags,
        0,
        0,
        &indesc,
        0,
        &tSocket.hContext,
        &outdesc,
        &flags,
        NULL);

    cprintf("Sec= %x.\n", sec);
    if(sec == SEC_E_OK){
        cprintf("SEC_E_OK.\n");

    }

}

//FreeContextBuffer(outbuffers[0].pvBuffer);
if (size != 0)
{
    // failed to fully send data to server
    result = -1;
    cprintf("Oh");
}
}
else if (sec != SEC_E_INCOMPLETE_MESSAGE)
{
    cprintf("INCOMP_MSG");

    // SEC_E_CERT_EXPIRED - certificate expired or revoked
    // SEC_E_WRONG_PRINCIPAL - bad hostname
    // SEC_E_UNTRUSTED_ROOT - cannot verify CA chain
    // SEC_E_ILLEGAL_MESSAGE / SEC_E_ALGORITHM_MISMATCH - cannot negotiate crypto algorithms
    result = -1;
}

```



```

if(sec == SEC_E_CERT_EXPIRED){
    cprintf("EXPIRED");
}

if(sec == SEC_E_WRONG_PRINCIPAL){
    cprintf("PRINCIPAL");
}

if(sec == SEC_E_ILLEGAL_MESSAGE || sec == SEC_E_ALGORITHM_MISMATCH){
    cprintf("ALGORITHM");
}

// if (sec == SEC_E_OK)
// {
//     cprintf("OK!");
//     char* buffer = "Hello world!";
//     int size = 0;
//     size= bstrlen(buffer);
//     send(tSocket.socket, buffer, size, 0);
//     // tls handshake completed
// }

// int r = recv(tSocket.socket, tSocket.ServerCertificate + tSocket.received , 20000, 0);
// if (r == 0)
// {
//     // server disconnected socket
//     return 0;
// }
// else if (r < 0)
// {
//     // socket error
//     result = -1;
//     cprintf("SOCKET ERROR");
//     break;
// }
// tSocket.received = r;
// cprintf("RECEIVED %u", r);

```

```

// Get the sizes for the context

```

```

// SECURITY_STATUS secStatus;
SECURITY_STATUS secStatus = QueryContextAttributes(&tSocket.hContext, SECPKG_ATTR_STREAM_SIZES,
&tSocket.sizes);
if (secStatus != SEC_E_OK) {
    cprintf("QueryContextAttributes failed: 0x%x\n", secStatus);
    char* data = valloc(10000);
    int r = recv(tSocket.socket, data, 10000, 0);
    char* a = "asd123";
    cprintf("Received %d bytes: '%s'", r, data);
    int k = send(tSocket.socket, a, 6, 0);
    cprintf("sent %d bytes: '%s'", r, data);
    return -1;
} else {
    cprintf("SIZES MAX-MESSAGE: %d\n", tSocket.sizes.cbMaximumMessage);
    cprintf("SIZES HEAD-MESSAGE: %d\n", tSocket.sizes.cbHeader);
}

```

//ENCRYPT AND SEND 'ASD'

```

#define MAXIMUM_MESSAGE 20000
SecBufferDesc Message;
SecBuffer Buffers[4];

// Prepare the message to be encrypted

char* msg = valloc(MAXIMUM_MESSAGE);
// Allocate the header
Buffers[0].cbBuffer = tSocket.sizes.cbHeader;
Buffers[0].pvBuffer = msg;
Buffers[0].BufferType = SECBUFFER_STREAM_HEADER;

// Allocate the data
Buffers[1].pvBuffer = msg + tSocket.sizes.cbHeader;
bmemcpy(msg + tSocket.sizes.cbHeader, "asd123", 7);
Buffers[1].cbBuffer = 7;
Buffers[1].BufferType = SECBUFFER_DATA;

```

```

// Allocate a trailer?
Buffers[2].BufferType = SECBUFFER_STREAM_TRAILER;
Buffers[2].pvBuffer = msg + tSocket.sizes.cbHeader + 7;
Buffers[2].cbBuffer = tSocket.sizes.cbTrailer;

// NULL
Buffers[3].BufferType = SECBUFFER_EMPTY;

Message.ulVersion = SECBUFFER_VERSION;
Message.cBuffers = 3;
Message.pBuffers = Buffers;

// Encrypt the message
secStatus = EncryptMessage(&tSocket.hContext, 0, &Message, 0);
if (secStatus == SEC_E_OK) {
    cprintf("EncryptMessage OK");
} else if (secStatus == SEC_E_ENCRYPT_FAILURE){
    cprintf("EncryptMessage failed: 0x%x\n", secStatus);
    cprintf("EncryptMessage failed: SEC_E_ENCRYPT_FAILURE\n");
    if (secStatus == SEC_E_BUFFER_TOO_SMALL){
        cprintf("Small\n");
    }
    if (secStatus == SEC_E_INVALID_TOKEN){
        cprintf("TOKEN\n");
    }
    if (secStatus == SEC_E_CONTEXT_EXPIRED){
        cprintf("EXP\n");
    }
}

// Send the encrypted message
int total = Buffers[0].cbBuffer + Buffers[1].cbBuffer + Buffers[2].cbBuffer;
int sB = 0;
while (sB < total) {
    sB += send(tSocket.socket, msg + sB, total - sB, 0);
}
cprintf("\nSB: %d, T: %d\n", sB, Buffers[0].cbBuffer + Buffers[1].cbBuffer + Buffers[2].cbBuffer );

return 0;
}

```

