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
    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);
    }

    // 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);
```

Revision #1
Created 8 August 2024 15:40:19 by lepus
Updated 8 August 2024 15:49:16 by lepus