Отчёт о выполненой лабораторной работе

Тема: «Защищенное сетевое взаимодействие»

Выполнил:

Студент группы 15-ИТ-1
Стеняев Андрей Дмитриевич

Новополоцк 2018

Цель: Изучения функций openssl для создания защищенных соединений.

Задача: Организовать взаимодействие типа клиент - сервер. Клиент при входе в связь с сервером должен ввести пароль. Разрешено сделать три попытки. Если пароль не верен, сервер должен блокировать IP-адрес клиента на 5 минут. Протокол TCP.
Поддержка нескольких клиентов одновременно.

Ход работы

Предположим, что мы знаем достаточно о том, как запрограммировать простые клиентскую и серверную программы, поэтому пояснения по этой части будут опущены.

Первым делом я написал разного рода классы для выполнения поставленной задачи, но обо всё по-порядку.

Первые два класса, используемые для выполнения работы — **TlsServer** и **TlsClient**. Оба класса являются абстрактными и предназначены для выполнения основных процедур получения и настройки сокетов, а так же их последующая эксплуатация. Вся первоначальная настройка сокета выполняется в конструкторе. В случае ошибки отправляется исключение с объектом nullptr (мне было лень писать отдельный класс-исключение). Для начала работы у клиента и сервера вызывается метод start. В этом методе выполняются какие-то архиважные задачи по установке соединения, после чего управление передаётся абстрактному методу onConnect. Листинги интерфейсов клиента(листинг 1) и сервера (листинг 2) представлены ниже.

Листинг 1: Интерфейс класса **TlsClient**

```
class TlsClient
{
private:

    Socket _socket;
    unsigned short _port;
    sockaddr_in _addres;

    SSLContext _sslContext;

public:
    TlsClient(std::string serverAddres, unsigned short port);
    ~TlsClient();

    void start();

protected:
    virtual void onConnect(TLSConnect& connect) = 0;
};
```

Листинг 2: Интерфейс класса **TlsServer**

```
class TlsServer
{
private:
    std::list<std::thread> _listOfThreads;

    Socket _socket;
    unsigned short _port;
    sockaddr_in _addres;

    SSLContext _sslContext;

public:
    TlsServer(unsigned short port);
    ~TlsServer();

    void start();

protected:
    virtual void onConnect(TLSConnect& connect) = 0;

    void onConnectThread(std::shared_ptr<TLSConnect> connect);
};
```

Рассмотрим конструктор класса **TlsClient**(Листинг 3).

Листинг3: Конструктор класса **TlsClient**

```
TlsClient::TlsClient(std::string serverAddres, unsigned short port) :
_port(port),
_sslContext(SSLContext::SSL_Client)
{

    _socket = createSocketTCP();
    if (!_socket)
    {
        throw nullptr;
    }

    _addres.sin_family = AF_INET;
    _addres.sin_port = htons(port);
    if (inet_pton(AF_INET, serverAddres.c_str(), &(_addres.sin_addr)) <= 0)
    {
        throw nullptr;
    }
}
```

В теле конструктора **TlsClient**(Листинг 3) происходит получение сокета и заполнение структуры **sockaddr_in** (В конструкторе **TlsServer** так же происходит «bind» сокета).

В интерфейсах **TlsClient**(Листинг 1) и **TlsServer**(Листинг 2) присутствует поле _sslContext типа **SSLContext**. Это поле — объект класса **SSLContext**, который отвечает за создание и настройку **SSL_CTX**. Интерфейс класса представлен в листинге 4.

Листинг 4: Интерфейс класса **SSLContext**

```
class SSLContext
{
public:
    enum SSLContextType
    {
        SSL_Client,
        SSL_Server
    };

private:
    std::unique_ptr<SSL_CTX, decltype(&::SSL_CTX_free)> _sslContext;

public:
    SSLContext(SSLContextType type);

    SSLContext(SSLContextType type, std::string sertificateFilePath,
                                    std::string privateKeyFilePath);

    inline SSL_CTX* getSSLCTX();

    inline operator SSL_CTX* () { return _sslContext.get(); }

private:
    void m_generatePrivateKeyAndCertificate();
};
```

Класс **SSLContext**(Листинг 4) имеет два конструктора. Их отличие заключается в том, что первый конструктор вызывает метод генерации сертификата и ключа, что бы получить сертификат и ключ, а второй конструктор получает эти данные из файлов сертификата и ключа. Так же нужно отметить, что все классы, работающие с **OpenSSL**, в случае ошибки бросают исключение **SSLException** с сообщение о том, что пошло не так. Весь исходный код класса **SSLContext**(Листинг 4) можно посмотреть в приложении к отчёту.

После создания объектов сервера и клиента можно выполнить их запуск, вызвав метод start. У клиента этот метод выполняет соединение с сервером и создаёт объект защищённого соединения: **SSLConnect**. У меня не было желания пихать всё в один класс, поэтому я написал класс **TLSConnect**, который расширяет класс **SSLConnect**. После создания защищённого соединения вызывается метод onConnect, в который передаётся адрес объекта **TLSConnect**,

через который и будет происходить обмен сообщениями. Код метода start приведён в листинге 5.

*Л***истинг 5**: Метод start класса **TlsClient**

```
void TlsClient::start()
{
    typedef sockaddr* pSockaddr; // Для более читаемого приведения
    if(connect(_socket, pSockaddr(&_addres), sizeof(_addres)) != 0)
    {
        throw nullptr;
    }

    TLSConnect connect(_sslContext, _socket, _addres);
    connect.connect();
    onConnect(connect);
}
```

Метод start класса **TlsServer**(листинг 2) устроен иначе. В нём создаётся очередь для принятия соединений, после чего начинается приём входящих подключений. Каждое новое соединение сопровождается созданием безопасного соединения и потока для обработки этого соединения. Стоит отметить, что поточной функцией является метод onConnectThread. Это было сделано для удобства, не более. Код метода start класса **TlsServer**(листинг 2) представлен в листинге 6.

Структура **Socket**, которую в последствии можно будет использовать при написании программ в разных ОС (экспериментальное решение) хранит в себе адрес сокета, а так же определяет методы управления этим сокетом (в том числе перегрузки операторов) для удобства использования. Так же я написал несколько процедур для получения сокетов разных типов (Но это не важно).

Так же в начале программы необходимо выполнить инициализацию **OpenSSL**. Для этого я написал класс **SSLInitializer** с одним методом initOpenSSL, который выполняет всю инициализацию. Исходный код посмотрите в ресурсах, он пояснений не требует.

Класс **SSLConnect**, который является базовым классом для класса **TLSConnect**, служит в основном для организации более удобной пересылки и приёма данных, реализуя методы приёма и передачи данных. Так же для этого класса были реализованы перегрузки оператора **„<<“** для отправки данных и оператора **„>>“** для приёма данных. Считанная строка записывается в объект типа **std::string**, а её размер возвращается в качестве возвращаемого значения метода. Исходный код в приложении.

Листинг 6: Метод start класса **TlsServer**

```
void TlsServer::start()
{
    listen(_socket, 5);

    while (true)
    {
        try
        {
            sockaddr_in clientAddres;
            socklen_t sockLen = sizeof(clientAddres);
            typedef sockaddr* pSockaddr; // Для более читаемого приведения
            Socket clientSocket = accept(_socket, pSockaddr(&clientAddres),
                                                                &sockLen);
            if (!clientSocket)
            {
                continue;
            }

            auto connect = std::make_shared<TLSConnect>(   _sslContext,
                                                           clientSocket,
                                                           clientAddres);
            connect->accept();

            _listOfThreads.push_back(std::thread(&TlsServer::onConnectThread,
                                                           this, connect));
        }
        catch(SSLException ex)
        {
            std::cerr << ex.what() << std::endl;
        }
    }
}
```

Для выполнения задания я написал два класса: **MyTlsServer** и **MyTlsClient**, которые расширяют классы **TlsServer** и **TlsClient** соответственно и реализуют метод onConnect.

Клиент почти ничего не делает, поэтому с его исходным кодом ознакомитесь в ресурсах, а вот сервер — другое дело. Интерфейс сервера представлен в листинге 7. Поле _mapOfLockedAddresses — ассоциативный массив, в котором ключом является ip-адрес, а значением — пара из времени начала блокировки(на всякий случай) и времени конца блокировки. Для работы с этими данными реализованы три метода: ipIsLocked, lockAddres, unlockAddres и getBlockingTime. Метод ipIsLocked возвращает true, если адрес в данный момент заблокирован и false в противном случае. Так же метод может разблокировать адрес в случае, если время его блокировки истекло. Методы lockAddres и unlockAddres вносят ip-адрес в список и выносят его оттуда соответственно.

Листинг 7: Интерфейс класса **MyTlsServer**

```
class MyTlsServer : public TlsServer
{
private:
    unsigned int _ipLockTime = 300;

    std::mutex _mutexForMap;
    std::map<unsigned int, std::pair<long long int, long long int>>
                                                    _mapOfLockedAddresses;

    bool ipIsLocked(unsigned int ip);
    void lockAddres(unsigned int ip);
    void unlockAddres(unsigned int ip);
    int getBlockingTime(unsigned int ip);

    std::string _password;

public:
    MyTlsServer(unsigned short port, std::string password) : TlsServer(port),
_password(password) {}

    void setIpLockTime(unsigned int seconds);
    unsigned int getIpLockTime();

protected:
    virtual void onConnect(TLSConnect& connect) override;
};
```

Рассмотрим реализацию метода onConnect(листинг 8). В начале происходят всякие формальности, типа получения адреса клиента в виде строки и вывод сообщения о том, что подключился клиент с таким то адресом, в консоль сервера. После этого проверяется, заблокирован ли адрес этого клиента лна сервере или нет. Если адрес заблокирован, то клиенту высылается символ принудительного завершения соединения «!» и, после ответа клиента на ситуацию (Мне было лень парсер делать), высылается сообщение о причине закрытия соединения. В случае с блокировкой сообщение будет содержать информацию об остаточном времени блокировки. Если клиент не заблокирован на сервере, то сервер начинает ожидать от клиента ввода пароля, который передаётся серверу в конструкторе. Если клиент введёт пароль неверно 3 раза, то сервер его заблокирует и сообщит о намерении разорвать соединение. На этом всё.

Все скриншоты с работой программы пойдут в приложении к отчёту.

Весь код программы пойдёт в приложении к данному отчёту.

Листинг 8: реализация метода **MyTlsServer**::onConnect

```
void MyTlsServer::onConnect(TLSConnect& connect)
{
    std::string clientIP = connect.getIP_str();

    Log::Message("MyTlsServer", "New connect: " + clientIP);
    std::string clientMessage;
    if (ipIsLocked(connect.getIP()))
    {
        Log::Warning("MyTlsServer", "This IP(" + clientIP + ") is locked!");

        connect << "!";
        connect >> clientMessage;
        connect << "Your address has been blocked. Wait "
                  + std::to_string(getBlockingTime(connect.getIP())) + "s.";
        return;
    }
    else
    {
        connect << "+";
    }

    Log::Message("MyTlsServer", clientIP + " - logs in");
    for (int i = 0; i < 3; i++)
    {
        std::string userPassword;
        int bytes = connect >> userPassword;
        if (bytes < 0) return;

        if (userPassword == _password)
        {
            Log::Warning("MyTlsServer", clientIP + " - Correct password!");
            connect << "+";

            break;
        }

        if (i < 2)
        {
            Log::Warning("MyTlsServer", clientIP + " - Wrong password!");
            connect << "-";
        }
        else
        {
            lockAddres(connect.getIP());
            Log::Warning("MyTlsServer", clientIP + " - Locked!");
            connect << "!";
            connect >> clientMessage;
            connect << "Your address has been blocked. Wait "
                      + std::to_string(_ipLockTime) + "s.";
            return;
        }
    }
}
```

Вывод

Документация к OpenSSL очень странная.

Новополоцк 2018