```
char* generetStringPublicKey(RSA* public_key, RSA* keypair)
{
    BIO* public_bio_key = BIO_new(BIO_s_mem());
    PEM_write_bio_RSAPublicKey(public_bio_key, keypair);
    size_t pub_len = BIO_pending(public_bio_key);
    char* public_key_char = (char*)malloc(pub_len + 1);
    BIO_read(public_bio_key, public_key_char, pub_len);
    public_key_char[pub_len] = '\0';
    return public_key_char;
}

EVP_PKEY* open_public_key(unsigned char* pub_key_file)
{
    EVP_PKEY* key = NULL;
    RSA* rsa = NULL;

    OpenSSL_add_all_algorithms();
    //BIO* bio_pub = BIO_new(BIO_s_file());
    BIO* bio_pub = BIO_new(BIO_s_mem());;
    BIO_read(bio_pub, pub_key_file, strlen((const char*)pub_key_file));
    if (NULL == bio_pub)
    {
        printf("open_public_key bio file new error!\n");
        return NULL;
    }

    rsa = PEM_read_bio_RSAPublicKey(bio_pub, NULL, NULL, NULL);
    if (rsa == NULL)
    {
        printf("open_public_key failed to PEM_read_bio_RSAPublicKey!\n");
        BIO_free(bio_pub);
        RSA_free(rsa);

        return NULL;
    }

    printf("open_public_key success to PEM_read_bio_RSAPublicKey!\n");
    key = EVP_PKEY_new();
    if (NULL == key)
    {
        printf("open_public_key EVP_PKEY_new failed\n");
        RSA_free(rsa);

        return NULL;
    }

    EVP_PKEY_assign_RSA(key, rsa);
    return key;
}

EVP_PKEY* open_private_key(const char* priv_key_file, const unsigned char* passwd)
{
    EVP_PKEY* key = NULL;
    RSA* rsa = RSA_new();
    OpenSSL_add_all_algorithms();
    BIO* bio_priv = NULL;
    bio_priv = BIO_new_file(priv_key_file, "rb");
    if (NULL == bio_priv)
    {
        printf("open_private_key bio file new error!\n");

        return NULL;
    }

    rsa = PEM_read_bio_RSAPrivateKey(bio_priv, &rsa, NULL, (void*)passwd);
```

```
    if (rsa == NULL)
    {
        printf("open_private_key failed to PEM_read_bio_RSAPrivateKey!\n");
        BIO_free(bio_priv);
        RSA_free(rsa);

        return NULL;
    }

    printf("open_private_key success to PEM_read_bio_RSAPrivateKey!\n");
    key = EVP_PKEY_new();
    if (NULL == key)
    {
        printf("open_private_key EVP_PKEY_new failed\n");
        RSA_free(rsa);

        return NULL;
    }

    EVP_PKEY_assign_RSA(key, rsa);
    return key;
}

int create_socket(int port)
{
    SOCKET s = 0;
    struct sockaddr_in addr;

    WSADATA wsaData;
    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)
    {
        printf("WSAStartup()fail:%d\n", GetLastError());
        return -1;
    }

    addr.sin_family = AF_INET;
    addr.sin_port = htons(port);
    addr.sin_addr.s_addr = htonl(INADDR_ANY);

    s = socket(AF_INET, SOCK_STREAM, 0);
    if (s < 0) {
        perror("Unable to create socket");
        exit(EXIT_FAILURE);
    }

    if (bind(s, (struct sockaddr*)&addr, sizeof(addr)) < 0) {
        perror("Unable to bind");
        exit(EXIT_FAILURE);
    }

    if (listen(s, 1) < 0) {
        perror("Unable to listen");
        exit(EXIT_FAILURE);
    }

    return s;
}

SSL_CTX* create_context()
{
    const SSL_METHOD* method;
    SSL_CTX* ctx;

    method = TLS_server_method();
```

```
    ctx = SSL_CTX_new(method);
    if (!ctx) {
        perror("Unable to create SSL context");
        ERR_print_errors_fp(stderr);
        exit(EXIT_FAILURE);
    }

    return ctx;
}

void configure_context(SSL_CTX* ctx)
{
    /* Set the key and cert */
    if (SSL_CTX_use_certificate_file(ctx, "cert_test.pem", SSL_FILETYPE_PEM) <= 0) {
        ERR_print_errors_fp(stderr);
        exit(EXIT_FAILURE);
    }

    if (SSL_CTX_use_PrivateKey_file(ctx, "key_test.pem", SSL_FILETYPE_PEM) <= 0) {
        ERR_print_errors_fp(stderr);
        exit(EXIT_FAILURE);
    }
}

std::string pem(X509* x509)
{
    BIO* bio_out = BIO_new(BIO_s_mem());
    PEM_write_bio_X509(bio_out, x509);
    BUF_MEM* bio_buf;
    BIO_get_mem_ptr(bio_out, &bio_buf);
    std::string pem = std::string(bio_buf->data, bio_buf->length);
    BIO_free(bio_out);
    return pem;
}

void createCertificate()
{
    EVP_PKEY* pkey;
    pkey = EVP_PKEY_new();

    RSA* rsa;
    rsa = RSA_generate_key(
        2048,   /* number of bits for the key - 2048 is a sensible value */
        RSA_F4, /* exponent - RSA_F4 is defined as 0x10001L */
        NULL,   /* callback - can be NULL if we aren't displaying progress */
        NULL    /* callback argument - not needed in this case */
    );

    EVP_PKEY_assign_RSA(pkey, rsa);

    X509* x509;
    x509 = X509_new();

    ASN1_INTEGER_set(X509_get_serialNumber(x509), 1);

    X509_gmtime_adj(X509_get_notBefore(x509), 0);
    X509_gmtime_adj(X509_get_notAfter(x509), 31536000L);

    X509_set_pubkey(x509, pkey);

    auto name = X509_get_subject_name(x509);

    int ret = X509_NAME_add_entry_by_txt(name, "C", MBSTRING_ASC,
        (unsigned char*)"CA", -1, -1, 0);
    std::cout << ret << std::endl;
```

```
    ret = X509_NAME_add_entry_by_txt(name, "O", MBSTRING_ASC,
        (unsigned char*)"MyCompany Inc.", -1, -1, 0);
    std::cout << ret << std::endl;
    ret = X509_NAME_add_entry_by_txt(name, "CN", MBSTRING_ASC,
        (unsigned char*)"localhost", -1, -1, 0);
    std::cout << ret << std::endl;

    ret = X509_set_issuer_name(x509, name);
    std::cout << ret << std::endl;

    ret = X509_sign(x509, pkey, EVP_sha1());
    std::cout << ret << std::endl;

    ret = X509_verify(x509, pkey);
    std::cout << ret << std::endl;

   /* BIO* f = BIO_new(BIO_s_mem());
    PEM_write_bio_X509(f, x509);
    size_t pri_len = BIO_pending(f);
    char* private_key_char = (char*)malloc(pri_len + 1);
    BIO_read(f, private_key_char, pri_len);
    private_key_char[pri_len] = '\0';*/

    //BIO* bio_file = NULL;

    //bio_file = BIO_new_file("AAAAAAA.pem", "w");
    //if (bio_file == NULL) {
    //    ret = -1;
    //}
    //ret = PEM_write_bio_X509(bio_file, x509);
    //if (ret != 1) {
    //    ret = -1;
    //}
    //BIO_free(bio_file);


    BIO* w = NULL;
    w = BIO_new_file("key_test.pem", "wb");
    PEM_write_bio_PrivateKey(
        w,                  /* write the key to the file we've opened */
        pkey,               /* our key from earlier */
        NULL, /* default cipher for encrypting the key on disk */
        NULL,       /* passphrase required for decrypting the key on disk */
        0,                 /* length of the passphrase string */
        NULL,               /* callback for requesting a password */
        NULL                /* data to pass to the callback */
    );
    BIO_free(w);

    BIO* f = NULL;
    f = BIO_new_file("cert_test.pem", "wb");
    PEM_write_bio_X509(
        f,   /* write the certificate to the file we've opened */
        x509 /* our certificate */
    );
    BIO_free(f);
}

int main(int argc, char** argv)
{
    /*RSA* rsa_pub_key = NULL;
    RSA* keypair = RSA_new();
    keypair = RSA_generate_key(2048, RSA_F4, NULL, NULL);
    char* pub_key = generetStringPublicKey(rsa_pub_key, keypair);
    EVP_PKEY* evp_pub_key = open_public_key((unsigned char*)pub_key);*/
```

```
    //createCertificate();

    int sock;
    SSL_CTX* ctx;

    ctx = create_context();

    configure_context(ctx);

    sock = create_socket(4433);

    /* Handle connections */
    while (1) {
        struct sockaddr_in addr;
        int len = sizeof(addr);
        SSL* ssl;
        const char reply[] = "test\n";

        int client = accept(sock, (struct sockaddr*)&addr, &len);
        if (client < 0) {
            perror("Unable to accept");
            exit(EXIT_FAILURE);
        }

        ssl = SSL_new(ctx);
        SSL_set_fd(ssl, client);

        if (SSL_accept(ssl) <= 0) {
            ERR_print_errors_fp(stderr);
        }
        else {
            char buf[1024];
            int ret = SSL_write(ssl, reply, strlen(reply));

            ret = SSL_read(ssl, buf, strlen(buf));
            buf[ret] = '\0';
            std::cout << buf << std::endl;

        }

        SSL_shutdown(ssl);
        SSL_free(ssl);
        closesocket(client);
    }

    closesocket(sock);
    SSL_CTX_free(ctx);
}
```