netty+证书认证

置顶 玄尺 2019-02-14 16:43:47 1130 收藏 2

分类专栏： 计算机安全

版权

netty客户端：

package com.lyf.csdn.netty;

import io.netty.bootstrap.Bootstrap;

import io.netty.channel.\*;

import io.netty.channel.nio.NioEventLoopGroup;

import io.netty.channel.socket.SocketChannel;

import io.netty.channel.socket.nio.NioSocketChannel;

import io.netty.handler.timeout.IdleState;

import io.netty.handler.timeout.IdleStateEvent;

import io.netty.handler.timeout.IdleStateHandler;

import io.netty.util.concurrent.DefaultEventExecutorGroup;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import java.util.Random;

import java.util.concurrent.TimeUnit;

/\*\*

\* 客户端：启动时发送1，服务器返回该整数的平方，添加一个心跳，每隔一段时间发送一个随机数

\*

\* @author xuanchi.lyf

\*/

public class NettyClient {

private static final Logger logger = LoggerFactory.getLogger(NettyClient.class);

public static void main(String[] args) {

EventLoopGroup workerGroup = new NioEventLoopGroup();

DefaultEventExecutorGroup bizGroup = new DefaultEventExecutorGroup(5);

try {

Bootstrap b = new Bootstrap();

b.group(workerGroup);

b.channel(NioSocketChannel.class);

b.option(ChannelOption.SO\_KEEPALIVE, Boolean.TRUE);

b.handler(new ChannelInitializer<SocketChannel>() {

@Override

public void initChannel(SocketChannel ch) {

IdleStateHandler idleStateHandler = new IdleStateHandler(20, 20, 20,

TimeUnit.SECONDS);

ch.pipeline().addLast(idleStateHandler);

ch.pipeline().addLast(new HeartBeatHandler());

ch.pipeline().addLast("decoder", new NettyDecoder());

ch.pipeline().addLast("encoder", new NettyEncoder());

ch.pipeline().addLast(bizGroup, new ClientHandler());

}

});

ChannelFuture f = b.connect("127.0.0.1", 9876).sync();

f.channel().closeFuture().sync();

} catch (Exception e) {

logger.info(e.getMessage(), e);

}

logger.info("netty tcp connection build success.");

}

}

/\*\*

\* 心跳发送[0000]

\*/

class HeartBeatHandler extends ChannelInboundHandlerAdapter {

@Override

public void userEventTriggered(ChannelHandlerContext ctx, Object evt) throws Exception {

if (evt instanceof IdleStateEvent) {

IdleState state = ((IdleStateEvent) evt).state();

if (state == IdleState.WRITER\_IDLE) {

ctx.channel().writeAndFlush(new Random().nextInt(100));

}

} else {

super.userEventTriggered(ctx, evt);

}

}

}

class ClientHandler extends ChannelInboundHandlerAdapter {

private static final Logger logger = LoggerFactory.getLogger(ClientHandler.class);

@Override

public void channelActive(ChannelHandlerContext ctx) {

logger.info("channel active");

ctx.channel().writeAndFlush(1);

}

@Override

public void channelInactive(ChannelHandlerContext ctx) {

logger.info("channel inActive");

}

@Override

public void channelRead(ChannelHandlerContext ctx, Object msg) {

logger.info("message = {}.", msg);

}

@Override

public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {

logger.info("throw exception", cause);

}

}

netty服务器端：

package com.lyf.csdn.netty;

import io.netty.bootstrap.ServerBootstrap;

import io.netty.buffer.ByteBuf;

import io.netty.channel.\*;

import io.netty.channel.nio.NioEventLoopGroup;

import io.netty.channel.socket.SocketChannel;

import io.netty.channel.socket.nio.NioServerSocketChannel;

import io.netty.handler.codec.LengthFieldBasedFrameDecoder;

import io.netty.handler.codec.MessageToByteEncoder;

import io.netty.util.concurrent.DefaultEventExecutorGroup;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import java.io.Serializable;

/\*\*

\* 服务器端

\*

\* @author xuanchi.lyf

\*/

public class NettyServer {

private static final Logger logger = LoggerFactory.getLogger(NettyServer.class);

public static void main(String[] args) {

EventLoopGroup bossGroup = new NioEventLoopGroup();

EventLoopGroup workerGroup = new NioEventLoopGroup();

DefaultEventExecutorGroup bizGroup = new DefaultEventExecutorGroup(50);

try {

ServerBootstrap b = new ServerBootstrap().group(bossGroup, workerGroup)

.channel(NioServerSocketChannel.class).option(ChannelOption.SO\_BACKLOG, 128)

.option(ChannelOption.SO\_REUSEADDR, Boolean.TRUE)

.childOption(ChannelOption.SO\_KEEPALIVE, Boolean.TRUE)

.childOption(ChannelOption.SO\_REUSEADDR, Boolean.TRUE)

.childOption(ChannelOption.TCP\_NODELAY, Boolean.TRUE)

.childOption(ChannelOption.ALLOW\_HALF\_CLOSURE, Boolean.FALSE)

.childHandler(new ChannelInitializer<SocketChannel>() {

@Override

public void initChannel(SocketChannel channel) {

channel.pipeline().addLast("decoder", new NettyDecoder());

channel.pipeline().addLast("encoder", new NettyEncoder());

channel.pipeline().addLast(bizGroup, new BizHandler());

}

});

ChannelFuture f = b.bind(9876).sync();

f.channel().closeFuture().sync();

} catch (Exception e) {

logger.info(e.getMessage(), e);

} finally {

workerGroup.shutdownGracefully();

bossGroup.shutdownGracefully();

}

}

}

class NettyDecoder extends LengthFieldBasedFrameDecoder {

NettyDecoder() {

super(1024, 0, 4);

}

@Override

protected Object decode(ChannelHandlerContext ctx, ByteBuf in) throws Exception {

ByteBuf frame = (ByteBuf) super.decode(ctx, in);

if (frame == null) {

return null;

}

int dataLength = frame.readInt();

if (dataLength == 0) {

return null;

}

return frame.readInt();

}

}

class NettyEncoder extends MessageToByteEncoder<Serializable> {

@Override

protected void encode(ChannelHandlerContext ctx, Serializable msg, ByteBuf out) {

out.writeInt(4);

out.writeInt((Integer) msg);

}

}

class BizHandler extends ChannelInboundHandlerAdapter {

private static final Logger logger = LoggerFactory.getLogger(BizHandler.class);

@Override

public void channelActive(ChannelHandlerContext ctx) {

logger.info("channel active.");

}

@Override

public void channelRead(ChannelHandlerContext ctx, Object msg) {

int data = (Integer) msg;

ctx.channel().writeAndFlush(data \* data);

logger.info("message = {}.", msg);

}

@Override

public void channelInactive(ChannelHandlerContext ctx) {

logger.info("channel inActive.");

}

@Override

public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {

logger.info("throw exception", cause);

}

}

以上我们可以服务器和客户端建立连接，正常的收发数据了。现在我们需要给netty加上SSL。其中证书可以是单向认证，也可以双向认证，两种方式各有特点。

单向认证是在服务器添加证书，客户端不添加证书，这样客户端需要校验服务器证书，但是服务器不用校验客户端证书，这种情况客户端是安全的，因为能够保证服务器是自己人，但是服务器本身不安全，因为服务器不知道客户端证书，没法校验。这种场景下主要是针对WEB场景。

双向认证是服务器端同时校验客户端身份信息，在点对点通信场景很重要。双向任务的问题是客户端证书过期了，如何更换客户端证书，这个问题很要命。

二、自签证书

1、生成服务器端和客户端keystore

keytool -genkey -alias server -keysize 2048 -validity 365 -keyalg RSA -dname "CN=localhost" -keypass mypassword -storepass mypassword -keystore server.jks

keytool -genkey -alias client -keysize 2048 -validity 365 -keyalg RSA -dname "CN=localhost" -keypass mypassword -storepass mypassword -keystore client.jks

其中：

    -alias：表示秘钥对的名称

  -keysize：秘钥长度，1024已经不安全了，至少2048起步

  -keypass、-storepass：密码，保持一样就好

  -keystore：生成keystore文件名

2、导出证书

keytool -export -alias server -keystore server.jks -storepass mypassword -file server.cer

keytool -export -alias client -keystore client.jks -storepass mypassword -file client.cer

3、将服务器端证书导入客户端的keystore（供客户端验证服务器证书）

keytool -import -trustcacerts -alias server -file server.cer -storepass mypassword -keystore client.jks

4、将客户端证书导入服务器端的keystore（供服务器端验证客户端证书）

keytool -import -trustcacerts -alias client -file client.cer -storepass mypassword -keystore server.jks

三、单向认证

服务器端代码：

package com.lyf.csdn.netty;

import io.netty.bootstrap.ServerBootstrap;

import io.netty.buffer.ByteBuf;

import io.netty.channel.\*;

import io.netty.channel.nio.NioEventLoopGroup;

import io.netty.channel.socket.SocketChannel;

import io.netty.channel.socket.nio.NioServerSocketChannel;

import io.netty.handler.codec.LengthFieldBasedFrameDecoder;

import io.netty.handler.codec.MessageToByteEncoder;

import io.netty.handler.ssl.SslHandler;

import io.netty.util.concurrent.DefaultEventExecutorGroup;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import javax.net.ssl.KeyManagerFactory;

import javax.net.ssl.SSLContext;

import javax.net.ssl.SSLEngine;

import java.io.FileInputStream;

import java.io.IOException;

import java.io.InputStream;

import java.io.Serializable;

import java.security.KeyStore;

/\*\*

\* 服务器端

\*

\* @author xuanchi.lyf

\*/

public class NettyServer {

private static final Logger logger = LoggerFactory.getLogger(NettyServer.class);

private static SSLEngine sslEngine;

static {

String sChatPath = "/Users/xuanchi.lyf/Desktop/testNetty/server.jks";

sslEngine = ServerSslContextFactory.getServerContext(sChatPath).createSSLEngine();

sslEngine.setUseClientMode(false);

}

public static void main(String[] args) {

EventLoopGroup bossGroup = new NioEventLoopGroup();

EventLoopGroup workerGroup = new NioEventLoopGroup();

DefaultEventExecutorGroup bizGroup = new DefaultEventExecutorGroup(50);

try {

ServerBootstrap b = new ServerBootstrap().group(bossGroup, workerGroup)

.channel(NioServerSocketChannel.class).option(ChannelOption.SO\_BACKLOG, 128)

.option(ChannelOption.SO\_REUSEADDR, Boolean.TRUE)

.childOption(ChannelOption.SO\_KEEPALIVE, Boolean.TRUE)

.childOption(ChannelOption.SO\_REUSEADDR, Boolean.TRUE)

.childOption(ChannelOption.TCP\_NODELAY, Boolean.TRUE)

.childOption(ChannelOption.ALLOW\_HALF\_CLOSURE, Boolean.FALSE)

.childHandler(new ChannelInitializer<SocketChannel>() {

@Override

public void initChannel(SocketChannel channel) {

channel.pipeline().addLast("ssl", new SslHandler(sslEngine));

channel.pipeline().addLast("decoder", new NettyDecoder());

channel.pipeline().addLast("encoder", new NettyEncoder());

channel.pipeline().addLast(bizGroup, new BizHandler());

}

});

ChannelFuture f = b.bind(9876).sync();

f.channel().closeFuture().sync();

} catch (Exception e) {

logger.info(e.getMessage(), e);

} finally {

workerGroup.shutdownGracefully();

bossGroup.shutdownGracefully();

}

}

}

class ServerSslContextFactory {

private static final String PROTOCOL = "TLS";

private static SSLContext SERVER\_CONTEXT;

static SSLContext getServerContext(String pkPath) {

if (SERVER\_CONTEXT != null) {

return SERVER\_CONTEXT;

}

InputStream inputStream = null;

try {

KeyManagerFactory keyManagerFactory;

if (pkPath != null) {

//密钥库KeyStore

KeyStore keyStore = KeyStore.getInstance("JKS");

//加载服务端证书

inputStream = new FileInputStream(pkPath);

//加载服务端的KeyStore ；sNetty是生成仓库时设置的密码，用于检查密钥库完整性的密码

keyStore.load(inputStream, "mypassword".toCharArray());

keyManagerFactory = KeyManagerFactory.getInstance("SunX509");

//初始化密钥管理器

keyManagerFactory.init(keyStore, "mypassword".toCharArray());

//获取安全套接字协议（TLS协议）的对象

SERVER\_CONTEXT = SSLContext.getInstance(PROTOCOL);

// 初始化此上下文

// 参数一：认证的密钥

// 参数二：对等信任认证

// 参数三：伪随机数生成器

// 由于单向认证，服务端不用验证客户端，所以第二个参数为null

SERVER\_CONTEXT.init(keyManagerFactory.getKeyManagers(), null, null);

}

} catch (Exception e) {

throw new Error("Failed to initialize the server-side SSLContext", e);

} finally {

if (inputStream != null) {

try {

inputStream.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

return SERVER\_CONTEXT;

}

}

class NettyDecoder extends LengthFieldBasedFrameDecoder {

NettyDecoder() {

super(1024, 0, 4);

}

@Override

protected Object decode(ChannelHandlerContext ctx, ByteBuf in) throws Exception {

ByteBuf frame = (ByteBuf) super.decode(ctx, in);

if (frame == null) {

return null;

}

int dataLength = frame.readInt();

if (dataLength == 0) {

return null;

}

return frame.readInt();

}

}

class NettyEncoder extends MessageToByteEncoder<Serializable> {

@Override

protected void encode(ChannelHandlerContext ctx, Serializable msg, ByteBuf out) {

out.writeInt(4);

out.writeInt((Integer) msg);

}

}

class BizHandler extends ChannelInboundHandlerAdapter {

private static final Logger logger = LoggerFactory.getLogger(BizHandler.class);

@Override

public void channelActive(ChannelHandlerContext ctx) {

logger.info("channel active.");

}

@Override

public void channelRead(ChannelHandlerContext ctx, Object msg) {

int data = (Integer) msg;

ctx.channel().writeAndFlush(data \* data);

logger.info("message = {}.", msg);

}

@Override

public void channelInactive(ChannelHandlerContext ctx) {

logger.info("channel inActive.");

}

@Override

public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {

logger.info("throw exception", cause);

}

}

客户端代码：

package com.lyf.csdn.netty;

import io.netty.bootstrap.Bootstrap;

import io.netty.channel.\*;

import io.netty.channel.nio.NioEventLoopGroup;

import io.netty.channel.socket.SocketChannel;

import io.netty.channel.socket.nio.NioSocketChannel;

import io.netty.handler.ssl.SslHandler;

import io.netty.handler.timeout.IdleState;

import io.netty.handler.timeout.IdleStateEvent;

import io.netty.handler.timeout.IdleStateHandler;

import io.netty.util.concurrent.DefaultEventExecutorGroup;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import javax.net.ssl.SSLContext;

import javax.net.ssl.SSLEngine;

import javax.net.ssl.TrustManager;

import javax.net.ssl.TrustManagerFactory;

import java.io.FileInputStream;

import java.io.IOException;

import java.io.InputStream;

import java.security.KeyStore;

import java.util.Random;

import java.util.concurrent.TimeUnit;

/\*\*

\* 客户端

\*

\* @author xuanchi.lyf

\*/

public class NettyClient {

private static final Logger logger = LoggerFactory.getLogger(NettyClient.class);

private static SSLEngine sslEngine;

static {

String clientKeystorePath = "/Users/xuanchi.lyf/Desktop/testNetty/client.jks";

sslEngine = ClientSslContextFactory.getClientContext(clientKeystorePath).createSSLEngine();

sslEngine.setUseClientMode(true);

}

public static void main(String[] args) {

EventLoopGroup workerGroup = new NioEventLoopGroup();

DefaultEventExecutorGroup bizGroup = new DefaultEventExecutorGroup(5);

try {

Bootstrap b = new Bootstrap();

b.group(workerGroup);

b.channel(NioSocketChannel.class);

b.option(ChannelOption.SO\_KEEPALIVE, Boolean.TRUE);

b.handler(new ChannelInitializer<SocketChannel>() {

@Override

public void initChannel(SocketChannel ch) {

IdleStateHandler idleStateHandler = new IdleStateHandler(1, 1, 1,

TimeUnit.SECONDS);

ch.pipeline().addLast("ssl", new SslHandler(sslEngine));

ch.pipeline().addLast("idleCheck", idleStateHandler);

ch.pipeline().addLast("heartbeat", new HeartBeatHandler());

ch.pipeline().addLast("decoder", new NettyDecoder());

ch.pipeline().addLast("encoder", new NettyEncoder());

ch.pipeline().addLast(bizGroup, new ClientHandler());

}

});

ChannelFuture f = b.connect("127.0.0.1", 9876).sync();

f.channel().closeFuture().sync();

} catch (Exception e) {

logger.info(e.getMessage(), e);

}

logger.info("netty tcp connection build success.");

}

}

class ClientSslContextFactory {

private static final String PROTOCOL = "TLS";

private static SSLContext CLIENT\_CONTEXT;

static SSLContext getClientContext(String caPath) {

if (CLIENT\_CONTEXT != null) {

return CLIENT\_CONTEXT;

}

InputStream inputStream = null;

try {

//信任库

TrustManagerFactory trustManagerFactory = null;

if (caPath != null) {

//密钥库KeyStore

KeyStore tks = KeyStore.getInstance("JKS");

//加载客户端证书

inputStream = new FileInputStream(caPath);

tks.load(inputStream, "mypassword".toCharArray());

trustManagerFactory = TrustManagerFactory.getInstance("SunX509");

// 初始化信任库

trustManagerFactory.init(tks);

}

CLIENT\_CONTEXT = SSLContext.getInstance(PROTOCOL);

//设置信任证书

TrustManager[] trustManagers = trustManagerFactory == null ? null

: trustManagerFactory.getTrustManagers();

CLIENT\_CONTEXT.init(null, trustManagers, null);

} catch (Exception e) {

throw new Error("Failed to initialize the client-side SSLContext");

} finally {

if (inputStream != null) {

try {

inputStream.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

return CLIENT\_CONTEXT;

}

}

/\*\*

\* 心跳发送[0000]

\*/

class HeartBeatHandler extends ChannelInboundHandlerAdapter {

@Override

public void userEventTriggered(ChannelHandlerContext ctx, Object evt) throws Exception {

if (evt instanceof IdleStateEvent) {

IdleState state = ((IdleStateEvent) evt).state();

if (state == IdleState.WRITER\_IDLE) {

ctx.channel().writeAndFlush(new Random().nextInt(100));

}

} else {

super.userEventTriggered(ctx, evt);

}

}

}

class ClientHandler extends ChannelInboundHandlerAdapter {

private static final Logger logger = LoggerFactory.getLogger(ClientHandler.class);

@Override

public void channelActive(ChannelHandlerContext ctx) {

logger.info("channel active");

ctx.channel().writeAndFlush(1);

}

@Override

public void channelInactive(ChannelHandlerContext ctx) {

logger.info("channel inActive");

}

@Override

public void channelRead(ChannelHandlerContext ctx, Object msg) {

logger.info("message = {}.", msg);

}

@Override

public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {

logger.info("throw exception", cause);

}

}

单向认证，客户端需要认证服务器身份，此时我们可以构造一个新服务器证书，换上试一下。

io.netty.handler.codec.DecoderException: javax.net.ssl.SSLHandshakeException: General SSLEngine problem

at io.netty.handler.codec.ByteToMessageDecoder.callDecode(ByteToMessageDecoder.java:459)

at io.netty.handler.codec.ByteToMessageDecoder.channelRead(ByteToMessageDecoder.java:265)

at io.netty.channel.AbstractChannelHandlerContext.invokeChannelRead(AbstractChannelHandlerContext.java:362)

at io.netty.channel.AbstractChannelHandlerContext.invokeChannelRead(AbstractChannelHandlerContext.java:348)

at io.netty.channel.AbstractChannelHandlerContext.fireChannelRead(AbstractChannelHandlerContext.java:340)

at io.netty.channel.DefaultChannelPipeline$HeadContext.channelRead(DefaultChannelPipeline.java:1359)

at io.netty.channel.AbstractChannelHandlerContext.invokeChannelRead(AbstractChannelHandlerContext.java:362)

at io.netty.channel.AbstractChannelHandlerContext.invokeChannelRead(AbstractChannelHandlerContext.java:348)

at io.netty.channel.DefaultChannelPipeline.fireChannelRead(DefaultChannelPipeline.java:935)

at io.netty.channel.nio.AbstractNioByteChannel$NioByteUnsafe.read(AbstractNioByteChannel.java:141)

at io.netty.channel.nio.NioEventLoop.processSelectedKey(NioEventLoop.java:645)

at io.netty.channel.nio.NioEventLoop.processSelectedKeysOptimized(NioEventLoop.java:580)

at io.netty.channel.nio.NioEventLoop.processSelectedKeys(NioEventLoop.java:497)

at io.netty.channel.nio.NioEventLoop.run(NioEventLoop.java:459)

at io.netty.util.concurrent.SingleThreadEventExecutor$5.run(SingleThreadEventExecutor.java:858)

at io.netty.util.concurrent.FastThreadLocalRunnable.run(FastThreadLocalRunnable.java:30)

at java.lang.Thread.run(Thread.java:748)

Caused by: javax.net.ssl.SSLHandshakeException: General SSLEngine problem

at sun.security.ssl.Handshaker.checkThrown(Handshaker.java:1478)

at sun.security.ssl.SSLEngineImpl.checkTaskThrown(SSLEngineImpl.java:535)

at sun.security.ssl.SSLEngineImpl.readNetRecord(SSLEngineImpl.java:813)

at sun.security.ssl.SSLEngineImpl.unwrap(SSLEngineImpl.java:781)

at javax.net.ssl.SSLEngine.unwrap(SSLEngine.java:624)

at io.netty.handler.ssl.SslHandler$SslEngineType$3.unwrap(SslHandler.java:292)

at io.netty.handler.ssl.SslHandler.unwrap(SslHandler.java:1248)

at io.netty.handler.ssl.SslHandler.decodeJdkCompatible(SslHandler.java:1159)

at io.netty.handler.ssl.SslHandler.decode(SslHandler.java:1194)

at io.netty.handler.codec.ByteToMessageDecoder.decodeRemovalReentryProtection(ByteToMessageDecoder.java:489)

at io.netty.handler.codec.ByteToMessageDecoder.callDecode(ByteToMessageDecoder.java:428)

... 16 more

Caused by: javax.net.ssl.SSLHandshakeException: General SSLEngine problem

at sun.security.ssl.Alerts.getSSLException(Alerts.java:192)

at sun.security.ssl.SSLEngineImpl.fatal(SSLEngineImpl.java:1728)

at sun.security.ssl.Handshaker.fatalSE(Handshaker.java:304)

at sun.security.ssl.Handshaker.fatalSE(Handshaker.java:296)

at sun.security.ssl.ClientHandshaker.serverCertificate(ClientHandshaker.java:1514)

at sun.security.ssl.ClientHandshaker.processMessage(ClientHandshaker.java:216)

at sun.security.ssl.Handshaker.processLoop(Handshaker.java:1026)

at sun.security.ssl.Handshaker$1.run(Handshaker.java:966)

at sun.security.ssl.Handshaker$1.run(Handshaker.java:963)

at java.security.AccessController.doPrivileged(Native Method)

at sun.security.ssl.Handshaker$DelegatedTask.run(Handshaker.java:1416)

at io.netty.handler.ssl.SslHandler.runDelegatedTasks(SslHandler.java:1408)

at io.netty.handler.ssl.SslHandler.unwrap(SslHandler.java:1316)

... 20 more

Caused by: sun.security.validator.ValidatorException: Certificate signature validation failed

at sun.security.validator.SimpleValidator.engineValidate(SimpleValidator.java:215)

at sun.security.validator.Validator.validate(Validator.java:260)

at sun.security.ssl.X509TrustManagerImpl.validate(X509TrustManagerImpl.java:324)

at sun.security.ssl.X509TrustManagerImpl.checkTrusted(X509TrustManagerImpl.java:281)

at sun.security.ssl.X509TrustManagerImpl.checkServerTrusted(X509TrustManagerImpl.java:136)

at sun.security.ssl.ClientHandshaker.serverCertificate(ClientHandshaker.java:1501)

... 28 more

Caused by: java.security.SignatureException: Signature does not match.

at sun.security.x509.X509CertImpl.verify(X509CertImpl.java:449)

at sun.security.x509.X509CertImpl.verify(X509CertImpl.java:392)

at sun.security.validator.SimpleValidator.engineValidate(SimpleValidator.java:213)

... 33 more

说明客户端验证了服务器的证书。

四、双向认证

服务器端：

package com.lyf.csdn.netty;

import io.netty.bootstrap.ServerBootstrap;

import io.netty.buffer.ByteBuf;

import io.netty.channel.\*;

import io.netty.channel.nio.NioEventLoopGroup;

import io.netty.channel.socket.SocketChannel;

import io.netty.channel.socket.nio.NioServerSocketChannel;

import io.netty.handler.codec.LengthFieldBasedFrameDecoder;

import io.netty.handler.codec.MessageToByteEncoder;

import io.netty.handler.ssl.SslHandler;

import io.netty.util.concurrent.DefaultEventExecutorGroup;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import javax.net.ssl.\*;

import java.io.FileInputStream;

import java.io.IOException;

import java.io.InputStream;

import java.io.Serializable;

import java.security.KeyStore;

import java.security.SecureRandom;

/\*\*

\* 服务器端

\*

\* @author xuanchi.lyf

\*/

public class NettyServer {

private static final Logger logger = LoggerFactory.getLogger(NettyServer.class);

private static SSLEngine sslEngine;

static {

String sChatPath = "/Users/xuanchi.lyf/Desktop/testNetty/server.jks";

sslEngine = ServerSslContextFactory.getServerContext(sChatPath, sChatPath)

.createSSLEngine();

sslEngine.setUseClientMode(false);

sslEngine.setNeedClientAuth(true);

}

public static void main(String[] args) {

EventLoopGroup bossGroup = new NioEventLoopGroup();

EventLoopGroup workerGroup = new NioEventLoopGroup();

DefaultEventExecutorGroup bizGroup = new DefaultEventExecutorGroup(50);

try {

ServerBootstrap b = new ServerBootstrap().group(bossGroup, workerGroup)

.channel(NioServerSocketChannel.class).option(ChannelOption.SO\_BACKLOG, 128)

.option(ChannelOption.SO\_REUSEADDR, Boolean.TRUE)

.childOption(ChannelOption.SO\_KEEPALIVE, Boolean.TRUE)

.childOption(ChannelOption.SO\_REUSEADDR, Boolean.TRUE)

.childOption(ChannelOption.TCP\_NODELAY, Boolean.TRUE)

.childOption(ChannelOption.ALLOW\_HALF\_CLOSURE, Boolean.FALSE)

.childHandler(new ChannelInitializer<SocketChannel>() {

@Override

public void initChannel(SocketChannel channel) {

channel.pipeline().addLast("ssl", new SslHandler(sslEngine));

channel.pipeline().addLast("decoder", new NettyDecoder());

channel.pipeline().addLast("encoder", new NettyEncoder());

channel.pipeline().addLast(bizGroup, new BizHandler());

}

});

ChannelFuture f = b.bind(9876).sync();

f.channel().closeFuture().sync();

} catch (Exception e) {

logger.info(e.getMessage(), e);

} finally {

workerGroup.shutdownGracefully();

bossGroup.shutdownGracefully();

}

}

}

class ServerSslContextFactory {

private static final String PROTOCOL = "TLS";

private static SSLContext SERVER\_CONTEXT;

static SSLContext getServerContext(String pkPath, String caPath) {

if (SERVER\_CONTEXT != null) {

return SERVER\_CONTEXT;

}

InputStream inputStream1 = null;

InputStream inputStream2 = null;

try {

KeyManagerFactory keyManagerFactory;

if (pkPath != null) {

//密钥库KeyStore

KeyStore keyStore = KeyStore.getInstance("JKS");

//加载服务端证书

inputStream1 = new FileInputStream(pkPath);

//加载服务端的KeyStore ；sNetty是生成仓库时设置的密码，用于检查密钥库完整性的密码

keyStore.load(inputStream1, "mypassword".toCharArray());

keyManagerFactory = KeyManagerFactory.getInstance("SunX509");

//初始化密钥管理器

keyManagerFactory.init(keyStore, "mypassword".toCharArray());

KeyManager[] keyManagers = keyManagerFactory.getKeyManagers();

//信任库

TrustManagerFactory trustManagerFactory = null;

if (caPath != null) {

KeyStore tks = KeyStore.getInstance("JKS");

inputStream2 = new FileInputStream(caPath);

tks.load(inputStream2, "mypassword".toCharArray());

trustManagerFactory = TrustManagerFactory.getInstance("SunX509");

trustManagerFactory.init(tks);

}

TrustManager[] trustManagers = trustManagerFactory == null ? null

: trustManagerFactory.getTrustManagers();

SERVER\_CONTEXT = SSLContext.getInstance(PROTOCOL);

SERVER\_CONTEXT.init(keyManagers, trustManagers, new SecureRandom());

}

} catch (Exception e) {

throw new Error("Failed to initialize the server-side SSLContext", e);

} finally {

if (inputStream1 != null) {

try {

inputStream1.close();

} catch (IOException e) {

e.printStackTrace();

}

}

if (inputStream2 != null) {

try {

inputStream2.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

return SERVER\_CONTEXT;

}

}

class NettyDecoder extends LengthFieldBasedFrameDecoder {

NettyDecoder() {

super(1024, 0, 4);

}

@Override

protected Object decode(ChannelHandlerContext ctx, ByteBuf in) throws Exception {

ByteBuf frame = (ByteBuf) super.decode(ctx, in);

if (frame == null) {

return null;

}

int dataLength = frame.readInt();

if (dataLength == 0) {

return null;

}

return frame.readInt();

}

}

class NettyEncoder extends MessageToByteEncoder<Serializable> {

@Override

protected void encode(ChannelHandlerContext ctx, Serializable msg, ByteBuf out) {

out.writeInt(4);

out.writeInt((Integer) msg);

}

}

class BizHandler extends ChannelInboundHandlerAdapter {

private static final Logger logger = LoggerFactory.getLogger(BizHandler.class);

@Override

public void channelActive(ChannelHandlerContext ctx) {

logger.info("channel active.");

}

@Override

public void channelRead(ChannelHandlerContext ctx, Object msg) {

int data = (Integer) msg;

ctx.channel().writeAndFlush(data \* data);

logger.info("message = {}.", msg);

}

@Override

public void channelInactive(ChannelHandlerContext ctx) {

logger.info("channel inActive.");

}

@Override

public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {

logger.info("throw exception", cause);

}

}

客户端：

package com.lyf.csdn.netty;

import io.netty.bootstrap.Bootstrap;

import io.netty.channel.\*;

import io.netty.channel.nio.NioEventLoopGroup;

import io.netty.channel.socket.SocketChannel;

import io.netty.channel.socket.nio.NioSocketChannel;

import io.netty.handler.ssl.SslHandler;

import io.netty.handler.timeout.IdleState;

import io.netty.handler.timeout.IdleStateEvent;

import io.netty.handler.timeout.IdleStateHandler;

import io.netty.util.concurrent.DefaultEventExecutorGroup;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import javax.net.ssl.\*;

import java.io.FileInputStream;

import java.io.IOException;

import java.io.InputStream;

import java.security.KeyStore;

import java.security.SecureRandom;

import java.util.Random;

import java.util.concurrent.TimeUnit;

/\*\*

\* 客户端

\*

\* @author xuanchi.lyf

\*/

public class NettyClient {

private static final Logger logger = LoggerFactory.getLogger(NettyClient.class);

private static SSLEngine sslEngine;

static {

String clientKeystorePath = "/Users/xuanchi.lyf/Desktop/testNetty/client.jks";

sslEngine = ClientSslContextFactory.getClientContext(clientKeystorePath, clientKeystorePath)

.createSSLEngine();

sslEngine.setUseClientMode(true);

}

public static void main(String[] args) {

EventLoopGroup workerGroup = new NioEventLoopGroup();

DefaultEventExecutorGroup bizGroup = new DefaultEventExecutorGroup(5);

try {

Bootstrap b = new Bootstrap();

b.group(workerGroup);

b.channel(NioSocketChannel.class);

b.option(ChannelOption.SO\_KEEPALIVE, Boolean.TRUE);

b.handler(new ChannelInitializer<SocketChannel>() {

@Override

public void initChannel(SocketChannel ch) {

IdleStateHandler idleStateHandler = new IdleStateHandler(1, 1, 1,

TimeUnit.SECONDS);

ch.pipeline().addLast("ssl", new SslHandler(sslEngine));

ch.pipeline().addLast("idleCheck", idleStateHandler);

ch.pipeline().addLast("heartbeat", new HeartBeatHandler());

ch.pipeline().addLast("decoder", new NettyDecoder());

ch.pipeline().addLast("encoder", new NettyEncoder());

ch.pipeline().addLast(bizGroup, new ClientHandler());

}

});

ChannelFuture f = b.connect("127.0.0.1", 9876).sync();

f.channel().closeFuture().sync();

} catch (Exception e) {

logger.info(e.getMessage(), e);

}

logger.info("netty tcp connection build success.");

}

}

class ClientSslContextFactory {

private static final String PROTOCOL = "TLS";

private static SSLContext CLIENT\_CONTEXT;

static SSLContext getClientContext(String pkPath, String caPath) {

if (CLIENT\_CONTEXT != null) {

return CLIENT\_CONTEXT;

}

InputStream inputStream1 = null;

InputStream inputStream2 = null;

try {

KeyManagerFactory keyManagerFactory;

if (pkPath != null) {

//密钥库KeyStore

KeyStore keyStore = KeyStore.getInstance("JKS");

//加载服务端证书

inputStream1 = new FileInputStream(pkPath);

//加载服务端的KeyStore ；sNetty是生成仓库时设置的密码，用于检查密钥库完整性的密码

keyStore.load(inputStream1, "mypassword".toCharArray());

keyManagerFactory = KeyManagerFactory.getInstance("SunX509");

//初始化密钥管理器

keyManagerFactory.init(keyStore, "mypassword".toCharArray());

KeyManager[] keyManagers = keyManagerFactory.getKeyManagers();

//信任库

TrustManagerFactory trustManagerFactory = null;

if (caPath != null) {

KeyStore tks = KeyStore.getInstance("JKS");

inputStream2 = new FileInputStream(caPath);

tks.load(inputStream2, "mypassword".toCharArray());

trustManagerFactory = TrustManagerFactory.getInstance("SunX509");

trustManagerFactory.init(tks);

}

TrustManager[] trustManagers = trustManagerFactory == null ? null

: trustManagerFactory.getTrustManagers();

CLIENT\_CONTEXT = SSLContext.getInstance(PROTOCOL);

CLIENT\_CONTEXT.init(keyManagers, trustManagers, new SecureRandom());

}

} catch (Exception e) {

throw new Error("Failed to initialize the server-side SSLContext", e);

} finally {

if (inputStream1 != null) {

try {

inputStream1.close();

} catch (IOException e) {

e.printStackTrace();

}

}

if (inputStream2 != null) {

try {

inputStream2.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

return CLIENT\_CONTEXT;

}

}

/\*\*

\* 心跳发送[0000]

\*/

class HeartBeatHandler extends ChannelInboundHandlerAdapter {

@Override

public void userEventTriggered(ChannelHandlerContext ctx, Object evt) throws Exception {

if (evt instanceof IdleStateEvent) {

IdleState state = ((IdleStateEvent) evt).state();

if (state == IdleState.WRITER\_IDLE) {

ctx.channel().writeAndFlush(new Random().nextInt(100));

}

} else {

super.userEventTriggered(ctx, evt);

}

}

}

class ClientHandler extends ChannelInboundHandlerAdapter {

private static final Logger logger = LoggerFactory.getLogger(ClientHandler.class);

@Override

public void channelActive(ChannelHandlerContext ctx) {

logger.info("channel active");

ctx.channel().writeAndFlush(1);

}

@Override

public void channelInactive(ChannelHandlerContext ctx) {

logger.info("channel inActive");

}

@Override

public void channelRead(ChannelHandlerContext ctx, Object msg) {

logger.info("message = {}.", msg);

}

@Override

public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {

logger.info("throw exception", cause);

}

}

重点理解：CLIENT\_CONTEXT.init(keyManagers, trustManagers, new SecureRandom());

第一个参数是是允许对方校验自己的证书，作用是加密

第二个参数是信任证书列表，作用是鉴权

第三个参数是随机种子

1、最严密的策略是双方互相校验对方证书，并且只信任对方证书

2、其次可以双方互相校验对方证书，并且信任所有证书

当信任所有证书：

new TrustManager[] {new X509TrustManager() {

@Override

public void checkClientTrusted(X509Certificate[] x509Certificates, String s) throws CertificateException {

}

@Override

public void checkServerTrusted(X509Certificate[] x509Certificates, String s) throws CertificateException {

}

@Override

public X509Certificate[] getAcceptedIssuers() {

return new X509Certificate[0];

}

}}