Глава 11 Secure Sockets

Један од сталних страхова потрошача који купују робу путем Интернета је да ће им неки хакер украсти број кредитне картице и потрошити неколико хиљада долара са рачуна. У стварности, вероватније је да ће радник у робној кући прочитати број њихове картице са исечка рачуна него да ће га неки хакер зграбити док путује Интернетом. Заправо, средином 2004. главне online крађе бројева кредитних картица извршене су крађом информација из слабо обезбеђених база података и фајл система након што су информације безбедно прошле кроз Интернет. Без обзира, како би се Интернет конекције учиниле фундаментално безбедним, сокети морају бити шифровани (енг. encrypted). То омогућује да трансакције буду поверљиве, аутентификоване и тачне.

Међутим, енкрипција је комплексна тема. Њена исправна примена захтева детаљно разумевање, не само математичких алгоритама коришћених за енкрипцију података, већ и протокола коришћених за размену кључева и шифрованих података. Чак и мала грешка може отворити огромну рупу у Вашем оклопу и открити ваше комуникације прислушкивачу. Према томе, писање софтвера за енкрипцију је посао који је боље препустити експертима. На срећу, особе са само лаичким познавањем протокола и алгоритама могу обезбедити своје комуникације софтвером који су дизајнирали експерти. Сваки пут када наручите нешто из online продавнице, трансакција је вероватно криптована и аутентификована коришћењем протокола и алгоритама о којима не морамо знати ништа. Као програмер који жели да пише мрежни софтвер клијента који комуницира са online продавницама, неопходно је знати мало више о протоколима и алгоритмима који су у игри, али не много више, јер се користи библиотека коју су написали експерти. Ако желите да пишете софтвер сервера који се извршава у online продавници, неопходно је знати још мало више, али и даље не толико као што би требало када би се све то писало од нуле, без реферисања на постојећи рад.

До скоро, такав софтвер био је предмет закона о контроли наоружања (енг. arms control laws) USA. Донекле још увек јесте. Закони о енкрипцији у другим државама крећу се од много стриктнијих него у USA до непостојећих. То је ограничило могућности Sun-у и другима који на интернационалном нивоу покушавају да обезбеде јак софтвер за енкрипцију. До Јаве 1.4 такве могућности нису биле уграђене у стандардне `java.net` класе. Пре тога, биле су доступне као стандардна екстензија Java Secure Sockets Extensions (JSSE). Иако је сада JSSE део стандардне дистрибуције JDK-а, још увек је спутан дизајном који подржава раније, мање либералне верзије, те је мање једноставан за употребу него што би то требало бити.

Ипак, JSSE може осигурати мрежну комуникацију коришћењем Secure Sockets Layer (SSL) Version 3 и Transport Layer Security (TLS) протокола и њима придружених алгоритама. **SSL је сигурносни протокол који омогућује web browser-има да комуницирају са web серверима користећи различите нивое поверљивости и аутентификације.**

Сигурне комуникације (secure communications)

Поверљива комуникација кроз отворени канал попут Интернета апсолутно захтева да подаци буду шифровани. Већина шема за шифровање које имају рачунарску имплементацију заснива се на нотацији *кључа*, незнатно општијој врсти шифре која није ограничена на текст. Чиста текстуална порука комбинује се са битовима кључа у складу са математичким алгоритмом како би произвео шифрат (encrypted cipher text). Коришћење кључева са већим бројем битова чини поруке експоненцијално тежим за дешифровање погађањем кључа методом грубе силе (brute-force).

У традиционалној енкрипцији *са тајним кључем* (*симетричној*), исти кључ се користи и за шифровање и за дешифровање података. И пошиљалац и прималац морају да поседују један кључ. Замислимо да Анђела жели да пошаље Гасу тајну поруку. Она му најпре шаље кључ који ће обоје користити. Али кључ не може бити шифрован јер Гас још увек нема кључ, па Анђела мора да га пошаље некриптованог. Сада претпоставимо да Едгар прислушкује конекцију између Анђеле и Гаса. Он ће добити кључ у исто време када и Гас. Од те тачке па надаље, он **може читати све** што Анђела и Гас кажу једно другоме користећи тај кључ.

У енкрипцији *са јавним кључем* (*асиметричној*), различити кључеви се користе за шифровање и дешифровање података. Јавни кључ се користи за шифровање података. Он се може дати било коме. Други кључ, тајни, користи се за дешифровање података. Он се мора чувати тајним и треба да га поседује само један од учесника. Ако Анђела жели да пошаље поруку Гасу, она тражи његов јавни кључ. Гас јој шаље свој јавни кључ преко некриптоване конекције. Анђела користи Гасов јавни кључ да криптује своју поруку коју му потом шаље. Ако Едгар прислушкује када Гас шаље Анђели свој кључ, он такође добије Гасов јавни кључ. Међутим, то му не омогућује да дешифрује поруку коју Анђела шаље Гасу, јер је за дешифровање потребан Гасов приватни кључ. **Порука је безбедна иако је јавни кључ откривен у путу.**

Асиметрична енкрипција се такође може користити за аутентификацију и проверу интегритета поруке. За ту сврху Анђела ће шифровати поруку својим тајним кључем пре слања. Када је Гас прими, он ће је дешифровати Анђелиним јавним кључем. Ако дешифровање успе, Гас **зна да је порука дошла од Анђеле**. Након свега, нико други не може произвести поруку која ће се правилно дешифровати њеним јавним кључем. Гас ће такође **знати да порука није промењена у путу**, било од стране малициозног Едгара или ненамерно софтвером који багује или шумом на мрежи, пошто би свака таква порука покварила дешифровање. Уз још мало труда, Анђела може извршити дупло шифровање, једном својим приватним кључем, а једном Гасовим јавним, добијајући тако све три предности: **приватности, аутентификације и интегритета**.

У пракси, енкрипција са јавним кључем (асиметрична) је процесорски много захтевнија и много спорија него традиционална енкрипција са тајним кључем (симетрична). Према томе, уместо шифровања целе поруке Гаусовим јавним кључем, Анђела шифрује **само традиционални тајни кључ** и шаље га Гаусу. Гаус га дешифрује својим тајним кључем. Сада Анђела и Гас **обоје знају тајни кључ, али Едгар га не зна**. Према томе, Гас и Анђела сада могу користити бржу енкрипцију са тајним кључем како би приватно комуницирали без да их Едгар прислушкује.

Међутим, Едгар **још увек има добар напад на овакав протокол**. (Врло важно: напад је **на протокол** који се користи за примање и слање порука, не на алгоритме шифровања који се користе. Овај напад не захтева да Едгар разбије Гасову и Анђелину енкрипцију и потпуно је независан од дужине кључа.) Едгар може, не само да чита Гасов јавни кључ када га он шаље Анђели, већ га може **заменити сопственим јавним кључем!** Када Анђела мисли да шифрује поруку Гасовим јавним кључем, она заправо користи Едгаров. Када она пошаље поруку Гасу, Едгар је пресретне, дешифрује користећи сопствени тајни кључ, шифрује користећи Едгаров јавни кључ и пошаље је Гасу. Ово се назива "man-in-the-middle attack". Радећи само на несигурном каналу, Гас и Анђела немају једноставан начин да се заштите од овога. Решење које се користи у пракси је да и Гас и Анђела сместе и верификују своје јавне кључеве користећи trusted third-party certification authority. Ни та шема још увек није идеална - Едгар може да се постави између Гаса и certification authority, Анђеле и certification authority, као и између Гаса и Анђеле - али то му компликује живот.

Многи занимљиви детаљи су изостављени из ове дискусије. Ако желите да знате више, http://www.faqs.org/faqs/cryptography-faq/ је добро место за почетак. За дубљу анализу протокола и алгоритама за поверљивост, аутентификацију и интегритет порука, "Applied Criptography" (аутор: Bruce Schneier, издавач: Wiley & Sons), је стандардни уводни текст. Коначно, "Java Cryptography" (аутор: Jonathan Knudsen, издавач: O'Reilly), и "Java Security" (аутор: Scott Oak, издавач: O'Reilly) покривају пакете за криптографију и аутентификацију на којима се заснива JSSE.

Као што претходни пример указује, теорија и пракса криптовања и аутентификације, како за алгоритме тако и за протоколе, је изазовно поље препуно замки које могу изненадити аматере. Много је једноставније дизајнирати лош алгоритам или проткол за шифровање, него добар. И није увек очигледно који алгоритми и протоколи су добри, а који нису. На срећу, не морамо да будемо експерт за криптографију како бисмо користили јаку криптографију у Јава мрежним програмима. JSSE нас штити од детаља ниског нивоа: како алгоритми преговарају, како се размењују кључеви, како се аутентификују учесници и како се подаци шифрују. JSSE нам омогућује да креирамо сокете и серверске сокете који транспарентно рукују преговорима и криптовањем неопходним за сигурну комуникацију. Све што треба да урадимо је да пошаљемо своје податке преко истих токова и сокета који су нам познати из претходних поглавља.

JSSE је подељен у 4 пакета:

`javax.net.ssl`

> Апстратне класе које дефинишу Јава API за сигурну мрежну комуникацију.

`javax.net`

> Апстрактне сокет factory класе које се користе уместо конструктора за креирање сигурних сокета.

`javax.security.net`

> Минимални скуп класа за руковање сертификатима јавних кључева неопходних за SSL у Java 1.1 (почев од Java 1.2 уместо њега је потребно користити пакет `java.security.cert`)

`com.sun.net.ssl`

> Конкретне класе које имплементирају алгоритме и протоколе за енкрипцију у Sun-овој референтној имплементацији JSSE. Технички, оне нису део JSSE стандарда. Друге имплементације могу заменити овај пакет сопственим; нпр. неким који користи native кôд да убрза процесорски захтевно генерисање кључева и процес енкрипције.

Ниједан од ових није био укључен као стандардни део JDK-а пре Java 1.4. Да би се користили са Java 1.3 било је неопходно преузети JSSE са адресе http://java.sun.com/products/jsse/ и инсталирати га.

Sun-ова референтна имплементација дистрибуирана је као .zip фајл, који се може распаковати и сместити било где на систему. У директоријуму lib овог .zip фајла налазе се 3 JAR архиве: jcert.jar, jnet.jar и jsse.jar. Њих је потребно сместити у свој class path или у jre/lib/ext директоријум.

Потом је потребно регистровати cryptography provider едитовањем свог jre/lib/security/java.security фајла.
Отворити овај фајл у текстуалном едитору и потражити линије облика:

```
security.provider.1=sun.security.provider.Sun
security.provider.2=com.sun.rsajca.Provider
```

Можете имати мање или више провајдера од овога. Међутим, колико год да их имате, додајте још линију:

```
security.provider.3=com.sun.net.ssl.internal.ssl.Provider
```

Можда треба да промените „3“ у 2, 4, 5 или који год је следећи број у секвенци security провајдера. Ако инсталирате third-party JSSE имплементацију, додаћете још једну линију попут ове са именом класе одређеном документацијом Ваше JSSE имплементације.

Ако користите већи број копија JRE, потребно је да поновите ову процедуру за сваку од њих.

Уколико ово одмах не „укопчате“, виђаћете изузетке попут „java.net.SocketException: SSL implementation not available“ када покушате да извршите програме који користе JSSE.
Алтернативно, уместо едитовања java.policy фајла, можете додати ову линију класама које користе Sun-ову имплементацију JSSE:

```
java.security.Security.addProvider(
      new com.sun.net.ssl.internal.ssl.Provider());
```

Ово може бити корисно ако пишете софтвер који ће се извршавати на неком другом систему, а не желите да замолите да се тамо модификује java.policy фајл.

Креирање сигурних клијентских сокета

Уколико не марите превише за детаље на којима су засновани, сâмо коришћење SSL сокета за комуникацију са постојећим сигурним сервером је заиста праволинијско.
Уместо конструисања `java.net.Socket` објекта конструктором, један се добије од `javax.net.ssl.SSLSocketFactory` коришћењем метода `createSocket()`.
Класа `SSLSocketFactory` је апстрактна:

```
public abstract class SSLSocketFactory extends SocketFactory
```

Како је сâма класа `SSLFactorySocket` апстрактна, њену инстанцу добијамо позивом статичког метода `SSLSocketFactory.getDefault()`:

```
public static SocketFactory getDefault() throws InstantiationException
```

Он или враћа инстанцу класе `SSLSocketFactory` или избацује `InstantiationException` ако не може пронаћи конкретну потклasу. Када имамо одговарајућу референцу, користимо један од следећих 5 преклопљених (overloaded) `createSocket()` метода да креирамо `SSLSocket`:

```
public abstract Socket createSocket(String host, int port)
   throws IOException, UnknownHostException
public abstract Socket createSocket(InetAddress host, int port)
   throws IOException
public abstract Socket createSocket(String host, int port,
   InetAddress interface, int localPort) throws IOException, UnknownHostException
public abstract Socket createSocket(InetAddres host, int port,
   InetAddress interface, int localPort) throws IOException
public abstract Socket createSocket(Socket proxy, String host, int port,
   boolean autoClose) throws IOException
```

Прва два метода креирају и враћају сокет који је конектован на задати хост и порт или избацују `IOException` ако не могу да се конектују. Трећи и четврти метод се конектују и враћају сокет који је конектован на задати хост и порт са задатог локалног мрежног интерфејса и порта. Последњи, пети, метод, међутим, је мало другачији. Он почиње са постојећим `Socket` објектом који је конектован на прокси сервер. Враћа `Socket` који „тунелује“ кроз овај прокси сервер до задатог хоста и порта. Аргумент `autoClose` одређује да ли `proxy` сокет треба да се затвори када се овај сокет затвори. Ако је `autoClose` `true`, прокси сокет ће бити затворен, а иначе неће.

`Socket` који враћају сви ови методи биће типа `javax.net.ssl.SSLSocket`, поткласе од `java.net.Socket`. Међутим, не морамо то да знамо. Након што је сигурни сокет креиран, користимо га као и сваки други сокет, кроз његове `getInputStream()`, `getOutputStream()` и друге методе. Нпр. претпоставимо да се сервер извршава на login.ibiblio.org на порту 7000 који прихвата поруџбине. Свака поруџбина се шаље као ASCII стринг користећи једну TCP конекцију. Сервер прихвата поруџбину и затвара конекцију. (Доста детаља који су неопходни у стварном систему је изостављено, попут тога да сервер шаље клијенту response code казујући клијенту да ли је поруџбина прихваћена). Поруџбине које клијенти шаљу изгледају овако:

Name: John Smit
Product-ID: 67X-89
Address: 1280 Deniston Blvd, NY NY 10003
Card number: 4000-1234-5670-9017
Expires: 08/05

Има довољно информација у овој поруци да омогући некоме ко забада нос у пакете да искористи број кредитне картице Џона Смита у нефер сврхе. Према томе, пре слања ове поруџбине, потребно је шифровати је; најједноставнији начин да се то уради не оптерећујући ни клијент ни сервер са много, компликованог и склоног грешкама, кôда за криптовање је користити сигурни сокет. Следећи кôд шаље поруџбину преко сигурног сокета:

```
try{
	// This statement is only needed if you didn't add
	// security.provider.3=com.sun.net.ssl.internal.ssl.Provider
	// to your java.security file
	Security.addProvider(new com.sun.net.ssl.internal.ssl.Provider());

	SSLSocketFactory factory =
				(SSLSocketFactory) SSLSocketFactory.getDefault();
	Socket socket = factory.createSocket("login.metalab.unc.edu", 7000);

	Writer out = new OutputStreamWriter(socket.getOutputStream(),"ASCII");
	out.write("Name: John Smith\r\n");
	out.write("Product-ID: 67X-89\r\n");
	out.write("Address: 1280 Deniston Blvd, NY NY 10003\r\n");
	out.write("Expires: 08/05\r\n");
	out.flush();
	out.close();
	socket.close();
}catch(IOException ex){
	ex.printStackTrace();
}
```

Само прве 3 наредбе су приметно другачије од оних које се користе када се ради са обичним сокетом. Остатак кôда просто користи обичне методе класа `Socket`, `OutputStream` и `Writer`.

Ни читање улаза није теже.

Пример 1 (HTTPSClient) је једноставан програм који се конектује на сигуран HTTP сервер, шаље једноставан GET захтев и штампа одговор.

Покретање:

```
>> java HTTPSClient www.usps.com
```

Када је овај пример тестиран приликом писања књиге, иницијално је одбио да се конектује на www.usps.com јер није могао да верификује идентитет удаљеног сервера. Проблем је био што су root сертификати допремљени уз JDK истекли и решен је тако што је урађен upgrade на последњу верзију JDK-а. Ако видите поруке типа “No trusted certificate found”, покушајте да урадите upgrade JDK-а.

Приликом покретања овог програма може се приметити да он спорије одговара него што би се очекивало. Приметно процесорско и мрежно време троши се на генерисање и размену јавних кључева. Чак и преко брзе конекције, лако може протећи 10 секунди или више док се конекција не успостави. Према томе, вероватно не желите да размењујете сав садржај преко HTTPS, већ само садржај који заиста треба да буде приватан.

Методи класе SSLSocket

Осим метода које смо већ дискутовали и који су наслеђени од класе `java.net.Socket`, класа `SSLSocket` поседује известан број метода за конфигурисање колико тачно и која врста аутентификације и шифровања да се врши. Нпр. Sun имплементација у склопу Java 1.4 подржава само 128-битну AES енкрипцију, док IAIK-ов iSaSiLk

(http://jce.iaik.tugraz.at/sic/Products/Communication-Messaging-Security/iSaSiLk)подржава 256-битну AES енкрипцију. Метод `getSupportedCipherSuites()` каже која комбинација алгоритама је доступна за дати сокет:

```
public abstract String[] getSupportedSuites()
```

Међутим, није нужно да сви комплети (cipher suites) који су препознати буду допуштени за дату конекцију. Неки могу бити преслаби и према томе онемогућени.
Метод `getEnabledCipherSuites()` каже које комплете је овај сокет вољан да користи:

```
public abstract String[] getEnabledCipherSuites()
```

О стварном комплету који ће бити коришћен преговарају клијент и сервер у време конектовања. Могуће је да се не сложе ни за један. Такође је могуће да иако је комплет омогућен и на клијенту и на серверу, један или обојица немају неопходне кључеве и сертификате да би га користили. У сваком случају, метод `createSocket()` ће избацити `SSLException`, поткласу од `IOException`. Могуће је променити комплете које клијент покушава да користи помоћу метода `setEnabledCipherSuites()`:

```
public abstract void setEnabledCipherSuites(String[] suites)
```

Аргумент овог метода треба да буде листа комплета које желимо да користимо. Свако име мора бити један од комплета које излиста `getSupportedSuites()`. Иначе бива избачен `IllegalArgumentException`. Sun-ов JDK 1.4 подржава ових 23 комплета:

- SSL_RSA_WITH_RC4_128_MD5
- SSL_RSA_WITH_RC4_128_SHA
- TLS_RSA_WITH_AES_128_CBC_SHA
- TLS_DHE_RSA_WITH_AES_128_CBC_SHA
- TLS_DHE_DSS_WITH_AES_128_CBC_SHA
- SSL_RSA_WITH_3DES_EDE_CBC_SHA
- SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA
- SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA
- SSL_RSA_WITH_DES_CBC_SHA
- SSL_DHE_RSA_WITH_DES_CBC_SHA
- SSL_DHE_DSS_WITH_DES_CBC_SHA
- SSL_RSA_EXPORT_WITH_RC4_40_MD5
- SSL_RSA_EXPORT_WITH_DES40_CBC_SHA
- SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
- SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
- SSL_RSA_WITH_NULL_MD5
- SSL_RSA_WITH_NULL_SHA
- SSL_DH_anon_WITH_RC4_128_MD5
- TLS_DH_anon_WITH_AES_128_CBC_SHA
- SSL_DH_anon_WITH_3DES_EDE_CBC_SHA
- SSL_DH_anon_WITH_DES_CBC_SHA
- SSL_DH_anon_EXPORT_WITH_RC4_40_MD5
- SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA

Свако име има алгоритам подељен у 4 дела: протокол, алгоритам размене кључева, алгоритам за шифровање и checksum. Нпр. SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA значи Secure Sockets Layer Version 3; Different Hellman метод за договор кључева; no authentication; Data Encryption Standard енкрипција са 40-битним кључевима; Cipher Block Chaining, и Secure Hash Algorithm checksum.

Подразумевано, JDK 1.4 имплементација омогућује све криптоване аутентификоване комплете (првих 15 на листи). Ако желимо неаутентификоване трансакције или аутентификоване, али некриптоване, морамо омогућити те комплете експлицитно методом `setEnabledCipherSuites()`.

Осим дужине кључева, постоји битна разлика између DES/AES и RC4-заснованих комплета. DES и AES су блоковске шифре, тј. они шифрују известан број битова одједном. DES увек шифрује 64 бита. Ако није доступно 64 бита, врши се допуна одговарајућим бројем битова. AES може шифровати блокове од 128, 192 или 256 бита, али такође мора да допуни улаз ако нема паран умножак величине блока. То није проблем за file transfer апликације попут сигурног HTTP и FTP, где су мање-више сви подаци доступни одједном. Међутим, јесте проблематично за user-centered протоколе попут chat-а и Telnet-а. RC4 је токовска шифра, може шифровати бајт по бајт и прикладнија је за протоколе који треба да шаљу бајт по бајт.

Нпр. претпоставимо да Едгар има прилично моћне паралелне рачунаре на располагању и може брзо да разбије било коју 64-битну или крађу енкрипцију и да Гас и Анђела то знају. Даље, они сумњају да Едгар може да изнуди од једне од њихових ISP или телефонских компанија да прислушкује линију, тако да желе да избегну анонимне конекције подложне “man-in-the-middle” нападима. Како би били сигурни, Гас и Анђела одлучују да користе бар 111-битну аутентификовану енкрипцију. Тиме постаје потребно да омогуће само најјаче доступне алгоритме. Следећи фрагмент кôда то чини:

```
String[] strongSuites = {"SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA",
      "SSL_RSA_WITH_RC4_128_MD5", "SSL_RSA_WITH_RC4_128_SHA",
      "SSL_RSA_WITH_3DES_EDE_CBC_SHA"};
socket.setEnabledCipherSuites(strongSuites);
```

Ако друга страна конекције не подржава строгу енкрипцију, сокет ће избацити изузетак када проба да се чита или пише у њега обезбеђујући да се поверљиве информације случајно не пошаљу преко слабог канала.

Event Handlers

Мрежне комуникације су споре у поређењу са брзином већине рачунара. Аутентификоване мрежне конекције су још спорије. Неопходно генерисање кључева и подешавање сигурне конекције лако могу потрајати неколико секунди. Према томе, можда желите да рукујете конекцијом асинхроно. JSSE користи стандардни модел догађаја уведен у Java 1.1 да обавести програме када се handshaking између клијента и сервера заврши. Шема је препознатљива. Како бисте добијали нотификације о handshake-completed догађајима, просто имплементирате интерфејс `HandshakeCompletedListener`:

```
public interface HandshakeCompletedListener
                                        extends java.util.EventListener
```

Интерфејс декларише метод `handshakeCompleted()`:

```
public void handshakeCompleted(HandshakeCompletedEvent event)
```

Метод као аргумент прима `HandshakeCompletedEvent`:

```
public class HandshakeCompletedEvent extends java.util.EventObject
```

Класа `HandshakeCompletedEvent` обезбеђује 4 метода за добијање информација о догађају:

```
public SSLSession getSession()
public String getCipherSuite()
public X509Certificate[] getPeerCertificateChain()
                                throws SSLPeerUnverifiedException
public SSLSocket getSocket()
```

Појединачни `HandshakeCompletedListener` објекти региструју своје интересовање за handshake-completed догађаје за одређени `SSLSocket` својим методима `addHandshakeCompletedListener()` и `removeHandshakeCompletedListener()`:

```
public abstract void addHandshakeCompletedListener(
                                HandshakeCompletedListener listener)
public abstract void removeHandshakeCompletedListener(
            HandshakeCompletedListener listener) throws IllegalArgumentException
```

Session Management

SSL се обично користи на web серверима и то с добрим разлогом. Web конекције теже да буду краткотрајне; свака страна захтева посебан сокет. Нпр. Amazon.com на свом сигурном серверу захтева 7 одвојених учитавања страна или више ако треба да едитујете адресу или изаберете паковање за поклон. Замислите да Вам свака од тих страница одузме 10 секунди или више за преговарање сигурне конекције. Због високог overhead-а укљученог у handshaking два хоста за сигурне комуникације, SSL омогућује да се успоставе сесије које обухватају већи број сокета. Различити сокети унутар исте сесије користе исти скуп јавних и тајних кључева. Ако сигурна конекција са Amazon.com има 7 сокета, свих 7 ће бити успостављени унутар исте сесије и користити исте кључеве. Само први сокет те сесије ће претрпети због генерисања и размене кључева.

Као програмер који користи JSSE не морате да радите ништа додатно како бисте користили предности сесија. Ако отворите већи број сигурних сокета ка једном хосту на једном порту у оквиру разумно кратког времена, JSSE ће аутоматски користити кључеве сесије. Међутим, у апликацијама високе сигурности, можда желите да онемогућите овакво дељење међу сокетима и форсирате реаутентификацију сесије. У JSSE сесије су представљене инстанцама `SSLSession` интерфејса; могу се користити методи овог интерфејса како би се проверило време када је сесија креирана и време последњег приступа сесији, урадила инвалидација сесије и добиле различите информације о њој:

```
public byte[] getId()
public SSLSessionContext getSessionContex()
public long getCreationTime()
public long getLastAccessedTime()
public void invalidate()
public void putValue(String name, Object value)
public Object getValue(String name)
public void removeValue(String name)
public String[] getValueNames()
public X509Certificate[] getPeerCertificateChain() throws SSLPeerUnverifiedException
public String getCipherSuite()
public String getPeerHost()
```

Метод `getSession()` класе `SSLSocket` враћа `Session` коме припада текући сокет:

```
public abstract SSLSession getSession()
```

Међутим, сесије представљају компромис између перформанси и безбедности. Безбедније је преговарати о кључу за сваку трансакцију. Ако имате заиста спектакуларан хардвер и покушавате да заштитите своје системе од подједнако богате, мотивисане и способне конкуренције, можда желите да избегнете сесије. Како би се спречило да сокет креира сесију, проследи се `false` методу `setEnableSessionCreation()`:

```
public abstract void setEnableSessionCreation(boolean allowSessions)
```

Метод `getEnableSessionCreation()` враћа `true` ако су сесије дозвољене, а `false` иначе:

```
public abstract boolean getEnableSessionCreation()
```

У ретким приликама можда чак желите да извршите поновну аутентификацију конекције, тј. да одбаците све сертификате и кључеве који су претходно договорени и изнова почнете са новом сесијом. Томе служи метод `startHandshake()`:

```
public abstract void startHandshake() throws IOException
```

Client Mode

Правило у већини сигурних комуникација је да се захтева да сервер аутентификује себе користећи одговарајући сертификат. Међутим, за клијенте се то не захтева. Када Харолд купује књигу од Амазона користећи њихов сигурни сервер, они морају да докажу његовом browser-у да су стварно Амазон, а не Џо Тамо Неки Хакер. Међутим, он не мора да докаже Амазону да је Елиот Расти Харолд. Углавном, то је онако како треба да буде, јер је куповина и инсталирање trusted сертификата неопходних за аутентификацију прилично непријатно искуство за корисника кроз које читаоци не би требало да пролазе само да би купили књигу. Међутим, ова асиметрија може довести до преваре са кредитном картицом. Како би се избегли проблеми попут овог, може се захтевати да се сокети аутентификују. Ова стратегија не би функционисала за јавно доступне сервисе. Међутим, може бити разумна за одређене интерне апликације високе сигурности.

Метод `setUseClientMode()` одређује да ли сокет треба да користи аутентификацију приликом свог првог handshake. Име метода може мало да доведе у заблуду. Може се користити и за сокете на страни клијента, али и за сокете на страни сервера. Међутим, када му се проследи `true`,

то значи да је сокет у client моду (без обзира да ли је на страни клијента или не) и неће понудити да се аутентификује. Када се проследи `false`, сокет ће покушати да се аутентификује:

```
public abstract void setUseClientMode(boolean mode)
                                      throws IllegalArgumentException
```

Ово својство се може поставити само једном за један сокет. Поновни покушај избацује `IllegalArgumentException`.

Метод `getUseClientMode()` просто каже да ли ће текући сокет користити аутентификацију приликом свог првог `handshake`:

```
public abstract boolean getUseClientMode()
```

Сигурни сокет на страни сервера (тј. сокет враћен методом `accept()` објекта `SSLServerSocket`) користи метод `setNeedClientAuth()` како би захтевао да се сви клијенти који се конектују аутентификују (или не):

```
public abstract void setNeedClientAuth(boolean needsAuthentication)
                                      throws IllegalArgumentException
```

Метод избацује `IllegalArgumentException` ако сокет није на страни сервера.

Метод `getNeedClientAuth()` враћа `true` ако сокет захтева аутентификацију од стране клијента, а `false` иначе:

```
public abstract boolean getNeedClientAuth()
```

Креирање сигурних серверских сокета

Сигурни клијентски сокети су само половина једначине. Друга половина су SSL серверски сокети. Они су инстанце класе `javax.net.SSLServerSocket`:

```
public abstract class SSLServerSocket extends ServerSocket
```

Као и код `SSLSocket`, сви конструктори ове класе су `protected`. Као и код `SSLSocket`, инстанце се креирају апстрактном factory класом, `javax.net.SSLServerSocketFactory`:

```
public abstract class SSLServerSocketFactory extends ServerSocketFactory
```

Такође као за `SSLSocketFactory`, инстанцу `SSLServerSocketFactory` враћа статички метод `SSLServerSocketFactory.getDefault()`:

```
public static ServerSocketFactory getDefault()
```

И као за `SSLSocketFactory`, `SSLServerSocketFactory` има три преклопљена `createServerSocket()` метода који враћају `SSLServerSocket` инстанце и једноставно их је разумети по аналогији са конструкторима класе `java.net.ServerSocket`:

```
public abstract ServerSocket createServerSocket(int port)
                                                 throws IOException
public abstract ServerSocket createServerSocket(int port,
                                int queueLength) throws IOException
public abstract ServerSocket createServerSocket(int port,
      int queueLength, InetAddress interface) throws IOException
```

Да је све само креирање сигурних серверских сокета, они би били прилично праволинијски и једноставни за употребу. Међутим, то није све. Factory који враћа `SSLServerSocketFactory.getDefault()` подржава само аутентификацију сервера. Не подржава енкрипцију. Да би се добила и енкрипција, server-side сигурни сокети захтевају још иницијализације и подешавања. Како тачно изгледа то подешавање **зависи од имплементације**. У Sun-овој референтној имплементацији, `com.sun.net.ssl.SSLContext` објекат је одговоран за креирање потпуно конфигурисаних и иницијализованих сигурних серверских сокета. Детаљи варирају од JSSE имплементације до JSSE имплементације, али за креирање сигурног серверског сокета у референтној имплементацији неопходно је:

- **генерисати јавне кључеве и сертификате** користећи **keytool**
- дати паре ☺ како би Ваши сертификати били аутентификовани од стране trusted third party попут Verisign
- **креирати `SSLContext` за алгоритам који ћете користити**
- **креирати `TrustManagerFactory`** одакле ће се узимати материјал сертификата који ћете користити
- **креирати `KeyManagerFactory`** за тип материјала кључа који ћете користити
- **креирати `KeyStore`** објекат за кључ и базу сертификата (Sun-ов подразумевани је **JKC**)
- **попунити `KeyStore` објекат кључевима и сертификатима**; нпр. њиховим учитавањем из фајлсистема користећи pass phrase којом су шифровани
- **иницијализовати `KeyManagerFactory` са `KeyStore` и његовом pass phrase.**
- **иницијализовати контекст** неопходним key manager-има **из `KeyManagerFactory`**, trust manager-има из `TrustManagerFactory` и извором случајности.
  (Последња два могу бити `null` ако желимо да прихватимо подразумеване вредности.)

Пример 2 (SecureOrderTaker) демонстрира ову процедуру комплетно. Поруџбине се прихватају и штампају на `System.out`. Наравно, у стварној апликацији, са поруџбинама би се радило нешто интересантније.

Пример учитава неопходне кључеве и сертификате из фајла *jnp3e.keys* у текућем директоријуму заштићеног шифром "*2andnotafnord*". Оно што наредни пример ради је да показује како је тај фајл креиран. Он је изграђен програмом *keytool* који је део JDK-а на следећи начин:

```
D:\JAVA> keytool -genkey -alias ourstore -keystore jnp3e.keys
Enter keystore password: 2andnotafnord
What is your first and last name?
   [Unknown]: Elliotte
What is the name of your organization unit?
   [Unknown]: Me, Myself, and I
What is the name of your organization?
   [Unknown]: Cafe au Lait
What is the name of your City or Locality?
   [Unknown]: Brooklyn
```

```
What is the name of your State or Province?
   [Unknown]: New York
What is the two-letter country code for this unit?
   [Unknown]: NY
Is <CN=Elliotte, OU="Me, Myself, and I", O=Caffe au Lait, L=Brooklyn,
ST=New York, C=NY> correct?
   [no]: y

Enter key password for <ourstore>
      (RETURN if same as keystore password):
```

Када се ово заврши, имате фајл са именом jnp3e.keys који садржи Ваше јавне кључеве. Међутим, нико неће веровати да су то Ваши јавни кључеви осим ако их не сертификујете код trusted third party као што је Verisign (https://www.verisign.com/). Сертификати нису бесплатни. Најјевтинија опција је $14.95 годишње за Class 1 Digital ID. Verisign крије пријавну форму за ову врсту ID дубоко унутар свог web сајта, очигледно како бисте пристали на скупље опције које су уочљиво истакнуте на њиховом home page. У време писања књиге жељена форма налазила се на адреси: https://www.verisign.com/client/. Verisign је мењао овај URL више пута у прошлости чинећи је тежом за проналажење од својих скупљих опција. У скупљим опцијама, Verisign иде на веће дужине како би гарантовао да сте ко кажете да јесте ☺ Пре него што се одлучите за било коју врсту дигиталног ID-а, треба да будете свесни да куповина једног има потенцијално озбиљне легалне последице. У појединим правосудним системима слабо промишљени закони чине власнике дигиталних ID-јева одговорним за све куповине и уговоре потписане њиховим дигиталним ID-ом, без обзира да ли је ID украден или фалсификован. Ако само желите да истражите JSSE пре но што одлучите да ли да прођете кроз муке, трошкове и одговорност куповине верификованог сертификата, Sun укључује верификовани keystore фајл „testkeys“, заштићен шифром „passphrase“, који садржи неке JSSE примере (стара адреса: http://java.sun.com/products/jsse/). Међутим, то није довољно за стварни рад.

За више информација о томе шта се тачно дешава и које су различите опције, као и други начини за креирање кључева и фајлова са сертификатима, консултујте online документацију за keytool који долази уз JDK, Java Cryptography Architecture guide на (стара адреса: http://java.sun.com/j2se/1.4.2/docs/guide/security/CryptoSpec.html, нова: http://docs.oracle.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html) или претходно поменуте књиге „Java Cryptography“ (аутор: Jonathan Knudsen) или „Java Security“ (аутор: Scott Oaks) (обе издао O'Reilly).

Други приступ је користити комплете који не захтевају аутентификацију.
Постоји 6 таквих у Java 1.4:

- SSL_DH_anon_WITH_RC4_128_MD5
- TLS_DH_anon_WITH_AES_128_CBC_SHA
- SSL_DH_anon_WITH_3DES_EDE_CBC_SHA
- SSL_DH_anon_WITH_DES_CBC_SHA
- SSL_DH_anon_EXPORT_WITH_RC4_40_MD5
- SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA

Они подразумевано нису омогућени јер су подложни man-in-the-middle нападима, али Вам бар допуштају да пишете једноставне програме без плаћања Verisign-у.

Методи класе SSLServerSocket

Након што смо успешно креирали и иницијализовали `SSLServerSocket`, постоји много апликација које можемо писати користећи ништа више осим метода наслеђених од `java.net.ServerSocket`. Међутим, постоје ситуације када је потребно мало прилагодити понашање. Као `SSLSocket`, и `SSLServerSocket` обезбеђује методе за избор комплета, руковање сесијама и утврђивање да ли се захтева да се клијенти аутентификују. Већина ових метода је веома слична са истоименим методима из `SSLSocket`. Разлика је што они раде на страни сервера и постављају подразумеване вредности за сокете које је прихватио `SSLServerSocket`. У неким случајевима, након што је `SSLSocket` прихваћен, још увек је могуће користити методе класе `SSLSocket` за конфигурисање тог сокета уместо свих сокета прихваћених текућим `SSLServerSocket`.

Избор комплета

Класа `SSLServerSocket` има иста три метода за утврђивање који комплети су подржани и омогућени као и `SSLSocket`:

```
public abstract String[] getSupportedCipherSuites()
public abstract String[] getEnabledCipherSuites()
public abstract void setEnabledCipherSuites(String[] suites)
```

Ови методи користе иста имена комплета као и истоимени методи класе `SSLSocket`. Разлика је у томе што се ови методи примењују на све сокете прихваћене са `SSLServerSocket` уместо на само један `SSLSocket`. Нпр. следећи фрагмент кôда има ефекат омогућавања анонимних неаутентификованих конекција на `SSLServerSocket server`. Заснива се на именима комплета која садрже стринг "_anon_". Ово је тачно за Sun-ову референтну имплементацију, а нема гаранције да и друге имплементације прате ову конвенцију:

```
String[] supported = server.getSupportedCipherSuites();
String[] anonCipherSuitesSupported = new String[supported.length];
int numAnonCipherSuitesSupported = 0;
for(int i = 0; i < supported.length; i++){
     if(supported[i].indexOf("_anon_") > 0){
            anonCipherSuitesSupported[numAnonCipherSuitesSupported++] = supported[i];
     }
}

String[] oldEnabled = server.getEnabledCipherSuites();
String[] newEnabled = new String[oldEnabled.length + numAnonCipherSuitesSupported];
System.arrayCopy(oldEnabled, 0, newEnabled, 0, oldEnabled.length);
System.arrayCopy(anonCipherSuitesSupported, 0, newEnabled, oldEnabled.length,
                                                    numAnonCipherSuitesSupported);

server.setEnabledCipherSuites(newEnabled);
```

Овај фрагмент дохвата листу подржаних и омогућених комплета коришћењем метода `getSupportedCipherSuites()` и `getEnabledCipherSuites()`. Прегледа име сваког подржаног комплета да види да ли садржи подстринг "_anon_". Ако то јесте случај, комплет се додаје у листу анонимних комплета. Након што је листа анонимних комплета изграђена, комбинује се у новом низу са претходном листом омогућених комплета. Нови низ се затим

прослеђује методу `setEnabledCipherSuites()` тако да се онда могу користити и претходно омогућени и анонимни комплети.

Управљање сесијом

И клијент и сервер се морају сложити око успостављања сесије. На страни сервера користи се метод `setEnableSessionCreation()` за задавање да ли ће то бити допуштено, а метод `getEnableSessionCreation()` за утврђивање да ли је тренутно допуштено:

```
public abstract void setEnableSessionCreation(boolean allowSessions)
public abstract boolean getEnableSessionCreation()
```

Креирање сесија је подразумевано омогућено. Ако сервер то онемогући, онда ће клијент који жели сесију и даље моћи да се конектује. Само неће добити сесију и мораће да ради handshake изнова за сваки сокет. Слично, ако клијент одбија сесије, али их сервер допушта, они ће још увек моћи да комуницирају, али без сесија.

Client Mode

Класа `SSLServerSocket` има два метода за утврђивање и задавање да ли се од клијентских сокета захтева да се аутентификују. Прослеђивањем `true` методу `setNeedClientAuth()` задаје се да ће бити прихваћене само конекције у којима је клијент способан да се аутентификује. Прослеђивањем `false` задаје се да се не захтева аутентификација клијената. Подразумевана вредност је `false`. Ако из неког разлога треба да знате тренутно стање овог својства, метод `getNeedClientAuth()` ће Вам га рећи:

```
public abstract void setNeedClientAuth(boolean flag)
public abstract boolean getNeedClientAuth()
```

Метод `setUseClientMode()` омогућује програму да укаже да чак и ако је креирао `SSLServerSocket`, он јесте и треба да буде третиран као клијент у комуникацији када се ради о аутентификацији и другим преговорима. Нпр. у FTP сесији клијентски програм отвара серверски сокет да прими податке од сервера, али то га не чини мање клијентом. Метод `getUseClientMode()` враћа `true` ако је `SSLServerSocket` у клијентском моду, а `false` иначе:

```
public abstract void setUseClientMode(boolean flag)
public abstract boolean getUseClientMode()
```