Цифрово подписване на документи в Java-базирани Web-приложения

Светлин Наков
http://www.nakov.com

Софийски Университет
“Св. Климент Охридски”

Резюме

Настоящата разработка има за цел да запознае читателя с проблемите, свързани с електронното подписване на документи в Java-базирани Web-приложения и да предложи конкретни подходи за тяхното решаване. Направен е кратък преглед на основните понятия, свързани с електронния подпис и инфраструктурата на публичния ключ – публичен ключ, личен ключ, цифров сертификат, сертифицираща организация, сертификационна верига, защитено хранилище за ключове и сертификати и др. Описват се процедурите и алгоритмите за цифрово подписване на документи и верификация на цифрови подписи. Разглеждат се библиотеките от класове за работа с цифрови подписи и сертификати, които Java 2 платформата предоставя. Дава се описание на класовете и интерфейсите от Java Cryptography Architecture (JCA) и Java Certification Path API, които имат отношение към работата с цифрови подписи и сертификати. Анализират се основните проблеми при подписването на документи в Web-базирани системи. Обосновава се необходимостта от използване на подписан Java-аплет, който се интегрира с Web-приложението и подписва файловете преди изпращането им от клиента към сървъра. Разглеждат се проблемите, свързани с подписването на Java-аплети и комуникацията между аплет и Web-браузър. Анализират се механизмите за верификация на цифрови подписи, сертификати и сертификационни вериги и възможностите за конкретното им прилагане. Представя се системата NakovDocumentSigner, която дава напълно функционален framework за цифрово подписване на документи в Java-базирани Web-приложения. Системата демонстрира как със средствата на Java 2 платформата могат да се подписват документи и верифицират цифрови подписи, сертификати и сертификационни вериги.

Основни понятия, свързани с цифровия подпис

При предаването на важни документи по електронен път често се налага да се удостовери по надежден начин кой в действителност е изпращач (автор) на даден документ. Един от подходите за удостоверяване на произхода на документи и файлове е чрез използването на т. нар. цифров подпис (електронен подпис).

Цифровото подписване на документи използва като математическа основа криптографията, базирана на публични ключове.

Криптографията, базирана на публични ключове (public key cryptography) е математическа наука, която се използва за осигуряване на конфиденциалност и автентичност при обмяната на информация чрез използването на криптографски алгоритми, работещи с публични и лични ключове. Тези криптографски алгоритми се използват за цифрово подписване на документи, проверка на цифров подпис, шифриране и дешифриране на документи.

Публичните и личните ключове представляват математически свързана **двойка криптографски ключове** (public/private key pair). На всеки публичен ключ съответства точно един личен ключ и обратното – на всеки личен ключ съответства точно един публичен ключ. За да използва криптография с публични ключове едно лице трябва да притежава двойка публичен ключ и съответен на него личен ключ.

Публичният ключ (public key) представлява число (последователност от битове), което обикновено е свързано с дадено лице. Един публичен ключ може да се използва за проверка на цифрови подписи, създадени със съответния му личен ключ, както и за шифриране на документи, които след

това могат да бъдат дешифрирани само от притежателя на съответния личен ключ. Публичните ключове не представляват тайна за никого и обикновено са публично достъпни. Публичният ключ на дадено лице трябва да е известен на всички, с които това лице си комуникира чрез криптография с публични ключове.

Личният ключ е число (последователност от битове), известно само на притежателя му. С личния си ключ едно лице може да подписва документи и да дешифрира документи, шифрирани със съответния на него публичен ключ. В известна степен личните ключове приличат на добре известните пароли за достъп, които са широко-разпространено средство за автентикация по Интернет. Приликата е в това, че както чрез парола, така и чрез личен ключ едно лице може да удостоверява самоличността си, т.е. да се автентикира. Освен това, както паролите, така и личните ключове по идея представляват тайна за всички останали освен за притежателя им. За разлика от паролите за достъп, личните ключове не са толкова кратки, че да могат да бъдат запомнени на ум и затова за съхранението им трябва да се полагат специални грижи. Ако един личен ключ попадне в лице, което не е негов собственик (бъде откраднат), цялата комуникация, базирана на криптографията с публични ключове, разчитаща на този личен ключ, губи своя смисъл. В такива случаи откраднатият личен ключ трябва да бъде обявен за невалиден и да бъде подменен, за да стане отново възможна сигурната комуникация с лицето, което е било негов собственик.

За целите на криптографията с публични ключове се използват такива криптографски алгоритми, че практически не е по силите на съвременната математика и съвременната изчислителна техника да се открие личният ключ на лице, за което е известен публичния ключ. Всъщност откриването на личен ключ, който съответства на даден публичен ключ е теоретически възможно, но времето и изчислителната мощ, необходими за целта, правят такива действия безсмислени. От математическа гледна точка не е възможно един документ да бъде подписан без да е известен личния ключ на лицето, което го подписва. Също не е възможно един документ, шифриран с публичния ключ на дадено лице, да бъде дешифриран без да е известен съответния му личен ключ. Науката, която се занимава с разбиването на криптографски ключове и кодове се нарича **криптоанализ**.

Цифровото подписване представлява механизъм за удостоверяване на произхода и целостта на информация, предавана по електронен път. При процеса на цифрово подписване на даден документ към него се добавя допълнителна информация, наречена цифров подпис, която се изчислява на базата на съдържанието на този документ и някакъв личен ключ. На по-късен етап тази допълнителна информация може да се използва за да се провери произхода на подписания документ.

Цифровият подпис (цифрова сигнатура) представлява число (последователност от битове), което се изчислява математически при подписването на даден документ (съобщение). Това число зависи от съдържанието на съобщението, от използвания алгоритъм за подписване и от личния ключ, с който е извършено подписването. Цифровият подпис позволява на получателя да провери истинския произход на информацията и нейната цялостност.

Инфраструктурата на публичния ключ (public key infrastructure – PKI) предоставя архитектурата, организацията, техниките, практиките и процедурите, които подпомагат чрез цифрови сертификати приложението на криптографията, базирана на публични ключове (public key cryptography) за целите на сигурната обмяна на информация по несигурни мрежи и преносни среди. За издаването и управлението на такива цифрови сертификати инфраструктурата на публичния ключ разчита на т. нар. сертифициращи организации, които позволяват да се изгради доверие между непознати страни, участнички в защитена комуникация, базирана на публични и лични ключове.

Цифровите сертификати свързват определен публичен ключ с определено лице. Те се издават от специални организации, на които се има доверие (сертифициращи организации) при строги мерки за сигурност, които гарантират тяхната достоверност. Цифровите сертификати можем да възприемаме като електронни документи, удостоверяващи, че даден публичен ключ е собственост на дадено лице. В практиката за целите на електронния подпис най-масово се използват X.509 сертификати.

X.509 е широко-възприет стандарт за цифрови сертификати. Един X.509 цифров сертификат съдържа публичен ключ на дадено лице, информация за това лице (име, организация и т.н.), информация за сертифициращата организация, която е издала този сертификат, информация за срока му на валидност, информация за използваните криптографски алгоритми и различни други детайли.

Сертифицираща организация (certification authority – CA) е институция, която е упълномощена да издава цифрови сертификати и да ги подписва със своя личен ключ. Целта на сертификатите е да потвърдят, че даден публичен ключ е притежание на дадено лице, а целта на сертифициращите организации е да потвърдят, че даден сертификат е валиден и може да му се вярва. В този смисъл сертифициращите организации се явяват безпристрастна доверена трета страна, която осигурява висока степен на сигурност при компютърно-базирания обмен на информация. Ако една сертифицираща организация е издала цифров сертификат на дадено лице и се е подписала, че този сертификат е наистина на това лице, можем да вярваме, че публичният ключ, който е записан в сертификата, е наистина на това лице, при условие, че имаме доверие на тази сертифицираща организация.

В зависимост от степента на сигурност, която е необходима, се използват сертификати с различно **ниво на доверие**. За издаването на някои видове сертификати е необходим само e-mail адрес на собственика им, а за издаването на други е необходимо лично присъствие на лицето-собственик, което полага подпис върху документи на хартия в някой от офисите на сертифициращата организация.

Не на всички сертифициращи организации може да се има доверие, защото е възможно злонамерени лица да се представят за сертифицираща организация, която реално не съществува или е фалшива. За да се вярва на една сертифицираща организация, тя трябва да е световно призната и утвърдена. В света на цифровата сигурност утвърдените световни сертифициращи организации разчитат на много строги политики и процедури за издаване на сертификати и благодарение на тях поддържат доверието на своите клиенти. За по-голяма сигурност тези организации задължително използват специален хардуер, който гарантира невъзможността за изтичане на важна информация, като например лични ключове. Сред най-известните утвърдени световни сертифициращи организации са компаниите: VeriSign Inc., Thawte Consulting, GlobalSign NV/SA, Baltimore Technologies, TC TrustCenter AG, Entrust Inc. и др.

Всяка сертифицираща организация има свой сертификат и съответстващ на него личен ключ, с който подписва сертификатите, които издава на своите клиенти. Една сертифицираща организация може да бъде от първо ниво (top-level certification authority; root CA) или да бъде от някое следващо ниво. **Сертифициращите организации от първо ниво** при започването на своята дейност издават сами на себе си сертификат, който подписват с него самия. Тези сертификати се наричат **Root-сертификати.** Root-сертификатите на утвърдените световни сертифициращи организации са публично достъпни от техните сайтове и могат да се използват за верификация на други сертификати. Сертифициращите организации, които не са на първо ниво, разчитат на някоя организация от по-горно ниво да им издаде сертификат, с който имат право да издават и подписват сертификати на свои клиенти.

Технически е възможно всеки сертификат да бъде използван за да се подпише с него всеки друг сертификат, но на практика възможността за подписване на сертификати е силно ограничена. Всеки сертификат съдържа в себе си неизменима информация за това дали може да бъде използван за подписване на други сертификати. Сертифициращите организации издават на своите клиенти сертификати, които не могат да бъдат използвани за подписване на други сертификати. Сертификати, с които могат да бъдат подписвани други сертификати, се издават само на сертифициращи организации при изключително строги мерки за сигурност. Ако един потребител си купи сертификат от някоя сертифицираща организация и подпише с него друг сертификат, новоподписаният сертификат няма да е валиден, защото ще е подписан от сертификат, в който е указано, че не може да се използва за подписване на други сертификати.

Един сертификат може да бъде подписан от друг сертификат (най-често собственост на някоя сертифицираща организация) или да е подписан от самия себе си. Сертификатите, които не са подписани от друг сертификат, а са подписани от самите себе си, се наричат **собственоръчно-подписани сертификати** (self-signed certificates). В частност Root-сертификатите на сертифициращите организации на първо ниво се явяват собственоръчно-подписани сертификати. В общия случай един self-signed сертификат не може да удостоверява връзка между публичен ключ и дадено лице, защото с подходящ софтуер всеки може да генерира такъв сертификат на името на избрано от него лице или фирма.

Въпреки, че не могат да се възприемат с доверие, собственоръчно-подписаните сертификати все пак имат свое приложение. Например в рамките на една вътрешно-фирмена инфраструктура, където е възможно сертификатите да се разпространят физически по сигурен начин между отделните служители

и вътрешните фирмени системи, self-signed сертификатите могат успешно да заместят сертификатите, издавани от сертифициращите организации. В такива вътрешно-фирмени среди не е необходимо някоя сертифицираща организация да потвърждава, че даден публичен ключ е на дадено лице, защото това може да се гарантира от начина издаване и пренасяне на сертификатите. Например, при постъпване на нов служител в дадена фирма, е възможно системният администратор да му издава self-signed сертификат и да му го дава на дискета или по друг сигурен път. След това администраторът може по сигурен път да пренася този сертификат до всички вътрешно-фирмени системи и по този начин да е гарантирано, че всички вътрешни системи имат истинските сертификати на всички служители.

Описаната схема за сигурност, базирана на self-signed сертификати може да се подобри, като фирмата изгради своя собствена **локална сертифицираща организация** за служителите си. За целта фирмата трябва първоначално да си издаде собственоръчно-подписан сертификат, а на всички свои служители да издава сертификати, подписани с него. Така първоначалният сертификат на фирмата се явява доверен Root-сертификат, а самата фирма се явява локална сертифицираща организация от първо ниво.

И в двете описани схеми за сигурност не е изключена възможността за евентуална злоупотреба на системния администратор, който има правата да издава сертификати. Този проблем би могъл да се реши чрез строги вътрешно-фирмени процедури по издаването и управлението на сертификатите, но пълна сигурност не е гарантирана.

При комуникация по Интернет, където няма сигурен начин да се установи дали даден сертификат, изпратен по мрежата не е подменен някъде по пътя, не се използват self-signed сертификати, а само сертификати, издадени от утвърдени сертифициращи организации. Например, ако един Web-сървър в Интернет трябва да комуникира с Web-браузъри по защитен комуникационен канал (SSL), той непременно трябва да притежава сертификат, издаден от някоя известна сертифицираща организация. В противен случай е възможно шифрираните връзки между клиентите и този Web-сървър да се подслушват от трети лица.

Сертификатите, издадени от утвърдените сертифициращи организации дават по-голяма степен на сигурност на комуникацията, независимо дали се използват в частна корпоративна мрежа или в Интернет. Въпреки това self-signed сертификати често се използват, защото сертификатите, издадени от сертифициращите организации струват пари и изискват усилия от страна на собственика им за първоначалното им издаване, периодичното им подновяване и надеждното съхраняване на свързания с тях личен ключ.

Когато една сертифицираща организация от първо ниво издава сертификат на свой клиент, тя го подписва със своя Root-сертификат. По този начин се създава верига от сертификати, състояща се от два сертификата – този на сертифициращата организация, предхождан от този на нейния клиент. **Вериги от сертификати** (сертификационни вериги, certificate chains) се наричат последователности от сертификати, за които всеки сертификат е подписан от следващия след него. В началото на веригата обикновено стои някакъв сертификат, издаден на краен клиент, а в края на веригата стои Root-сертификата на някоя сертифицираща организация. В средата на веригата стоят сертификатите на някакви междинни сертифициращи организации. Общовъзприетата практика е сертифициращите организации от първо ниво да издават сертификати на междинните сертифициращи организации, като отбелязват в тези сертификати, че могат да бъдат използвани за издаване на други сертификати. Междинните сертифициращи организации издават сертификати на свои клиенти или на други междинни сертифициращи организации. На сертификатите издавани на крайни клиенти се задават права, които не позволяват те да бъдат използвани за издаване на други сертификати, а на сертификатите, издавани на междинни сертифициращи организации не се налага такова ограничение.

На един сертификат, който се намира в началото на дадена сертификационна верига може да се вярва само ако тази сертификационна верига бъде успешно верифицирана. В такъв случай се казва, че този сертификат е **проверен сертификат** (verified certificate). Верификацията на една сертификационна верига включва проверка дали всеки от сертификатите, които я съставят, е подписан от следващия сертификат във веригата, като за последния сертификат се проверява дали е в списъка на Root-сертификатите, на които безусловно се вярва. Всеки софтуер за верификация на сертификати поддържа списък от **доверени Root-сертификати** (trusted root CA certificates), на които вярва безусловно. Това са Root-сертификатите на световно-утвърдените сертифициращи организации. Например Web-браузърът

Internet Explorer идва стандартно със списък от около 150 доверени Root-сертификата, а браузърът Mozilla при първоначална инсталация съдържа около 70 доверени сертификата. Проверката на една сертификационна верига включва не само проверка на това дали всеки сертификат е подписан от следващия и дали сертификатът в края на тази верига е в списъка на доверените Root-сертификати. Необходимо е още да се провери дали всеки от сертификатите във веригата не е с изтекъл срок на валидност, а също дали всеки от сертификатите без първия има право да бъде използван за подписване на други сертификати. Ако проверката на последното условие бъде пропусната, ще стане възможно краен клиент да издава сертификат на името на когото си поиска и верификацията на издадения сертификат да преминава успешно. При проверката на дадена сертификационна верига се проверява и дали някой от сертификатите, които я съставят, не е анулиран (revoked). Съвкупността от всички описани проверки има за цел да установи дали на един сертификат може да се вярва. Ако проверката на една сертификационна верига не е успешна, това не означава непременно, че има опит за фалшификация. Възможно е списъкът на доверените Root-сертификати, използван при верификацията, да не съдържа Root-сертификата, с който завършва веригата, въпреки че той е истински. В общия случай един сертификат не може да бъде верифициран, ако не е налична цялата му сертификационна верига или ако Root-сертификата, с който започва веригата му, не е в списъка на доверените сертификати. Сертификационната верига на един сертификат може да се построи и програмно, ако не е налична, но за целта трябва да са налични всички сертификати, които влизат в нея.

В системите за електронно подписване на документи се използват **защитени хранилища за ключове и сертификати** (keystores). Едно такова хранилище може да съдържа три типа елементи – сертификати, сертификационни вериги и лични ключове. Понеже съхраняваната в защитените хранилища информация е поверителна, от съображения за сигурност достъпът до нея се осъществява с пароли на две нива – парола за хранилището и отделни пароли за личните ключове, записани в него. Благодарение на тези пароли при евентуално откраднване на едно защитено хранилище за ключове и сертификати, поверителната информация, записана в него, не може да бъде прочетена лесно. В практиката личните ключове, като особено важна и поверителна информация, никога не се държат извън хранилища за ключове и сертификати и винаги са защитени с пароли за достъп.

Има няколко разработени стандарта за защитени хранилищата за ключове и сертификати. Най-разпространен е стандартът **PKCS#12**, при който хранилището се съхранява във вид на файл със стандартното разширение .PFX (или по-рядко използваното разширение .P12). Един PFX файл обикновено съдържа един сертификат, съответен на него личен ключ и сертификационна верига, удостоверяваща автентичността на сертификата. Наличието на сертификационна верига не е задължително и понякога в PFX файловете има само сертификат и личен ключ. В повечето случаи за улеснение на потребителя паролата за достъп до един PFX файл съвпада с паролата за достъп до личния ключ, записан в него. Поради тази причина при използване на PFX файлове най-често се изисква само една парола за достъп.

Когато една сертифицираща организация издаде цифров сертификат на свой клиент, клиентът в крайна сметка получава едно защитено хранилище за ключове и сертификати, което съдържа издадения му сертификат, свързания с него личен ключ и цялата сертификационна верига, която удостоверява автентичността на сертификата. Защитеното хранилище се предоставя на клиента или във вид на PFX файл или във формата на смарт-карта или се инсталира директно в неговия Web-браузър.

Обикновено когато един сертификат се издава през Интернет, независимо по какъв начин потребителят потвърждава самоличността си, в процедурата по издаването важно участие взима потребителският Web-браузър. При заявка за издаване на сертификат, отправена към дадена сертифицираща организация от нейния Web-сайт, Web-браузърът на потребителя генерира двойка публичен и личен ключ и изпраща публичния ключ към сървъра на организацията. Сертифициращата организация, след като установи верността на данните за самоличността на своя клиент, му издава сертификат, в който записва получения от неговия Web-браузър публичен ключ и потвърдените му лични данни. За някои видове сертификати личните данни могат се състоят единствено от един проверен e-mail адрес, докато за други те могат да включват пълна информация за лицето – имена, адрес, номер на документ за самоличност и т.н. Проверката на личните данни става по процедура определена от съответната сертифицираща организация. След като сертифициращата организация издаде сертификата на своя клиент, тя го препраща към Web-страница, от която този сертификат може да бъде инсталиран в неговия Web-браузър. Реално потребителят получава по някакъв начин от

сертифициращата организация новоиздаденият му сертификат заедно с пълната му сертификационна верига. Web-браузърът междувременно е съхранил съответния на сертификата личен ключ и така в крайна сметка потребителят се сдобива със сертификат и съответстващи му личен ключ, заедно със сертификационната верига на сертификата, инсталирани в неговия Web-браузър. Начинът на съхранение на личните ключове е различен при различните браузъри, но при всички случаи такава конфиденциална информация се защитава най-малко с една парола. При описания механизъм за издаване на сертификати личният ключ на потребителя остава неизвестен за сертифициращата организация и така този потребител може да бъде сигурен, че никой друг няма достъп до неговия личен ключ.

Повечето Web-браузъри могат да използват съхранените в тях сертификати и лични ключове за автентикация пред защитени SSL сървъри. Много E-Mail клиенти също могат да използват сертификатите, съхранявани в Web-браузърите при подписване, шифриране и дешифриране на електронна поща. Някои приложения, обаче, не могат да използват директно сертификатите от потребителските Web-браузъри, но могат да работят с PFX хранилища за ключове и сертификати. В такива случаи потребителите могат да експортират от своите Web-браузъри сертификатите си заедно със съответните им лични ключове в PFX файлове и да ги използват от всякакви други програми. В Internet Explorer експортирането на сертификат и личен ключ става от главното меню чрез командите Tools | Internet Options | Contents | Certificates | Export, а в Netscape и Mozilla – чрез командите Edit | Preferences | Privacy & Security | Certificates | Manage Certificates | Backup. По подразбиране при експортирането на сертификат и личен ключ в PFX файл Internet Explorer не включва пълната сертификационна верига в изходния файл, но потребителят може да укаже това от допълнителна опция.

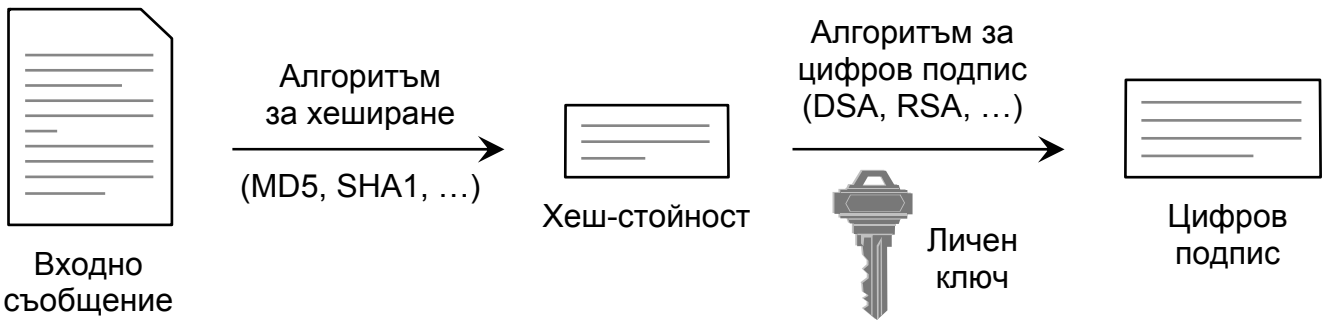
Има няколко стандарта за съхранение на X.509 цифрови сертификати. Най-често се използва кодирането ASN.1 DER, при което сертификатите се записват във файлове с разширение .CER (или по-рядко разширения .CRT или .CA). Един **CER файл** съдържа публичен ключ, информация за лицето, което е негов собственик и цифров подпис на някоя сертифицираща организация, удостоверяващ че този публичен ключ е наистина на това лице. Сертифициращите организации разпространяват от своите сайтове Root-сертификатите си във вид на CER файлове. Един CER файл може да бъде в двоичен вид или в текстов вид, кодиран с Base64.

Понякога се случва някое лице или фирма да загуби контрол над собствените си сертификати и съответстващите им лични ключове и те да попаднат в трети лица, които могат евентуално да злоупотребяват с тях. В такива случаи е необходимо тези сертификати да бъдат обявени за невалидни (revoked certificates).

Сертифициращите организации периодично (или по неотложност) издават списъци на определени сертификати, които са с временно преустановено действие или са анулирани преди изтичане на срока им на валидност. Тези списъци се подписват цифрово от сертифициращата организация, която ги издава и се наричат **списъци на анулираните сертификати** (certificate revocation lists – CRL). В един такъв списък се посочват името на сертифициращата организация, която го е издала, датата на издаване, датата на следващото издаване на такъв списък, серийните номера на анулираните сертификати и специфичните времена и причини за това анулиране.

Как работи цифровият подпис

Криптографията, базирана на публични ключове осигурява надежден метод за цифрово подписване, при който се използват двойки публични и лични ключове. Едно лице полага цифров подпис под дадено електронно съобщение (файл, документ, e-mail и др.) чрез личния си ключ. Разгледано технически цифровото подписване на едно съобщение се извършва на две стъпки:

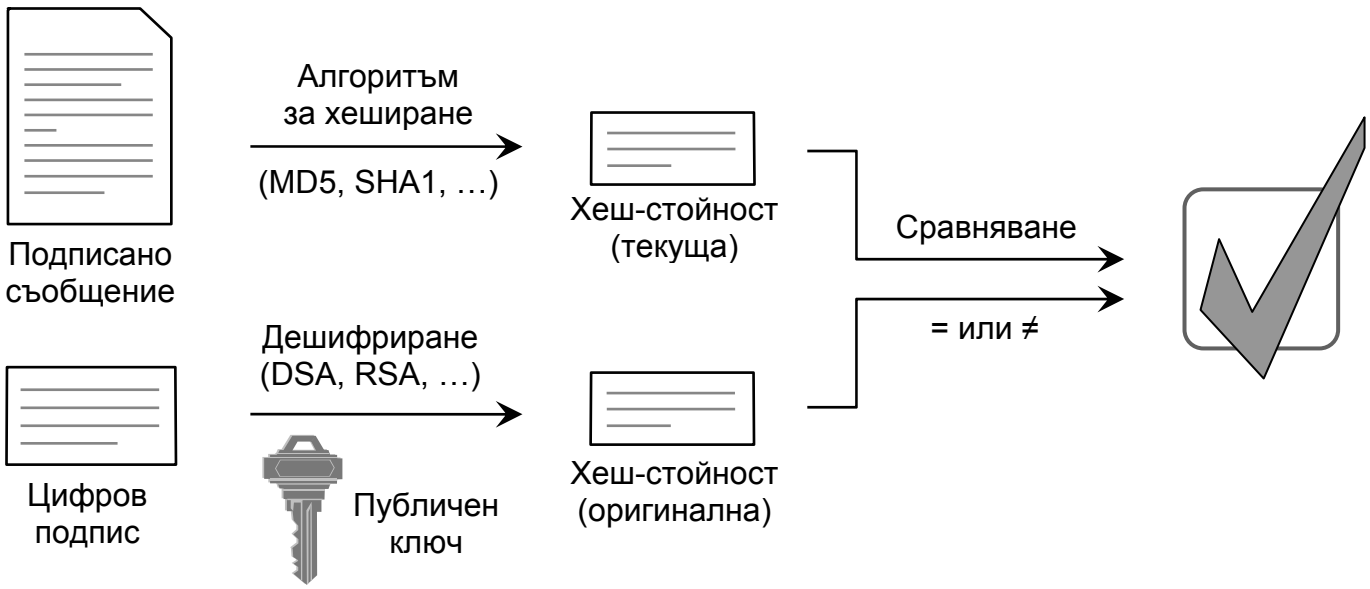


На първата стъпка се изчислява хеш-стойност на съобщението (message digest) по някакъв криптографски алгоритъм за хеширане (например MD4, MD5, SHA1 или друг). Хеш-стойността на едно съобщение представлява последователност от битове, обикновено с фиксирана дължина, извлечено по някакъв начин от съобщението. Алгоритмите, които изчисляват хеш-стойност, са такива, че при промяна дори само на 1 бит от входното съобщение се получава тотално различна хеш-стойност. Това поведение на тези алгоритми ги прави изключително устойчиви на криптоаналитични атаки, т.е. почти е невъзможно от дадена хеш-стойност на едно съобщение да се намери самото съобщение. Тази невъзможност за възстановяване на входното съобщение е съвсем логична, като се има предвид, че хеш-стойността на едно съобщение може да има стотици пъти по-малък размер от него самото. Възможно е два напълно различни документа да имат една и съща хеш-стойност, но вероятността това да се случи е толкова малка, че на практика тази възможност се пренебрегва.

На втората стъпка от цифровото подписване получената в първата стъпка хеш-стойност на съобщението се шифрира с личния ключ, с който се извършва подписването. За целта се използва някакъв математически алгоритъм за цифров подпис (digital signature algorithm), който преобразува хеш-стойността в шифрирана хеш-стойност, наричана още цифров подпис. Най-често се използват алгоритмите RSA (който се основава на теорията на числата), DSA (който се основава на теорията на дискретните логаритми) и ECDSA (който се основава на теорията на елиптичните криви). Полученият цифров подпис обикновено се прикрепя към съобщението в специален формат, за да може да бъде верифициран на по-късен етап, когато това е необходимо.

Цифровият подпис позволява на получателя на дадено подписано съобщение да провери истинския му произход и неговата цялостност (интегритет). Процесът на **проверка (верификация) на цифров подпис** има за цел да установи дали дадено съобщение е било подписано с личния ключ, който съответства на даден публичен ключ. Проверката на цифров подпис не може да установи дали едно съобщение е подписано от дадено лице. За да проверим дали едно лице е подписало дадено съобщение, е необходимо да се сдобием с истинския публичен ключ на това лице. Това е възможно или чрез получаване на публичния ключ по сигурен път (например на дискета или CD) или с помощта на инфраструктурата на публичния ключ чрез използване на цифрови сертификати. Без да имаме сигурен начин за намиране на истинския публичен ключ на дадено лице не можем да имаме гаранция, че даден документ е подписан наистина от него.

Технически погледнато проверката на цифров подпис се извършва на три стъпки:



На първата стъпка се изчислява хеш-стойността на подписаното съобщение. За изчислението на тази хеш-стойност се използва същия криптографски алгоритъм, който е бил използван при подписването му. Тази хеш-стойност наричаме текуща, защото е получена от текущия вид на съобщението.

На втората стъпка от проверката на един цифров подпис се дешифрира цифровия подпис, който трябва да бъде проверен, със същия алгоритъм, който е използван при шифрирането. Дешифрирането става с публичния ключ, съответстващ на личния ключ, който е използван при подписването на

съобщението. В резултат се получава оригиналната хеш-стойност, която е била изчислена при хеширането на оригиналното съобщение в първата стъпка от процеса на подписването му.

На третата стъпка се сравняват текущата хеш-стойност, получена от първата стъпка и оригиналната хеш-стойност, получена от втората стъпка. Ако двете стойности съвпадат, верифицирането е успешно и доказва, че документът е бил подписан с личния ключ, съответстващ на публичния ключ, с който се извършва верификацията. Ако двете стойности се различават, това означава, че цифровият подпис е невалиден, т.е. верификацията е неуспешна. Има три възможности за получаване на невалиден цифров подпис:

- Ако цифровият подпис е подправен (не е истински), при дешифрирането му с публичния ключ няма да се получи оригиналната хеш-стойност на съобщението, а някакво друго число.
- Ако документът е бил променян (подправян) след подписването му, текущата хеш-стойност, изчислена от подправения документ, ще бъде различна от оригиналната хеш-стойност, защото на различни документи съответстват различни хеш-стойности.
- Ако публичният ключ не съответства на личния ключ, който е използван за подписването, получената от цифровия подпис при дешифрирането с неправилен ключ оригинална хеш-стойност няма да е вярната.

Ако верификацията се провали, независимо по коя от трите причини, това доказва само едно: подписът, който се верифицира не е получен при подписването на документа, който се верифицира с личния ключ, съответстващ на публичния ключ, който се използва за верификацията. Неуспешната верификация не винаги означава опит за фалшификация на цифров подпис. Понякога верификацията може да се провали защото е използван неправилен публичен ключ. Такава ситуация се получава, когато съобщението не е изпратено от лицето, от което се очаква да пристигне или системата за проверка на подписи разполага с неправилен публичен ключ за това лице. Възможно е дори едно и също лице да има няколко валидни публични ключа с валидни сертификати за всеки от тях и системата да прави опит за верифициране на подписани от това лице документи с някой от неговите публични ключове, но не точно с този, който съответства на използвания за подписването личен ключ. За да се избегнат такива проблеми, често пъти, когато се изпраща подписан документ, освен документа и цифровия му подпис, се изпраща и сертификата на лицето, което е положило този подпис. При верификацията се използва публичния ключ, съдържащ се в сертификата и ако верификацията е успешна, се счита, че документът е изпратен от лицето, описано в сертификата. Разбира се, както винаги, когато се използват сертификати, трябва да се вярва на един сертификат, само ако е валидността му е проверена или ако е собственоръчно-подписан, но получен от изпращача по сигурен път.

Използване на цифрови подписи и сертификати в Java

За целите на подписването на документи, проверка на цифровия подпис и управление на цифрови сертификати в Java платформата се използва Java Cryptography Architecture (JCA). JCA представлява спецификация, която предоставя на програмистите достъп до криптографски услуги, работа с цифрови подписи и сертификати. От архитектурна гледна точка JCA е така проектирана, че да позволява различни имплементации от различни софтуерни доставчици. При работата с JCA програмистът избира имената на доставчиците на криптографски услуги (providers) и имената на алгоритмите, които иска да използва. Различните доставчици на криптографски услуги реализират различно множество от криптографски алгоритми, които са достъпни по име. Спецификацията JCA установява стандарти за различните типове криптографски услуги и специфицира начина на използване на криптографските алгоритми. Реализацията на самите алгоритми е оставена на софтуерните доставчици. Заедно с JDK 1.4 и всяка по-нова версия се разпространява и стандартна имплементация на JCA, която се използва по подразбиране, ако програмистът не посочи някоя друга.

Архитектурата JCA предоставя класове и интерфейси за работа с публични и лични ключове, цифрови сертификати, подписване на съобщения, проверка на цифрови подписи и достъп до защитени хранилища за ключове и сертификати. Тези класове и интерфейси се намират в пакетите `java.security` и `java.security.cert`. Ще дадем кратко описание на най-важните от тях:

`java.security.KeyStore` – използва се за достъп до защитени хранилища за ключове и сертификати. Предоставя методи за зареждане на хранилище от поток, записване на хранилище в поток,

преглед на записаната в хранилището информация, извличане на ключове, сертификати и сертификационни вериги, промяна на записаната в хранилището информация и др. Поддържа два формата за съхранение на хранилища – PFX (съгласно PKCS #12 стандарта) и JKS (Java Key Store). При създаване на обект от класа `KeyStore` форматът на хранилището се задава от параметър. Възможни са стойности "JKS" и "PKCS12". Обектите, записани в едно хранилище, са достъпни по име (alias), а ключовете са достъпни по име и парола.

`java.security.PublicKey` – представлява публичен ключ.

`java.security.PrivateKey` – представлява личен ключ.

`java.security.cert.Certificate` – представлява абстрактен базов клас за всички класове, които представляват цифрови сертификати. Съдържа в себе си публичен ключ и информация за лицето, което е негов собственик. За представяне на всеки конкретен тип сертификати (например `X.509`, `PGP` и т.н.) се използва съответен наследник на този клас.

`java.security.cert.X509Certificate` – представлява X.509 v.3 сертификат. Предлага методи за достъп до отделните му атрибути – собственик (Subject), издател (Issuer), публичен ключ на собственика, срок на валидност, версия, сериен номер, алгоритъм за цифровата сигнатура, цифрова сигнатура, допълнителни разширения и др. Данните за един X509Certificate са достъпни само за четене.

`java.security.Signature` – предоставя функционалност за подписване на документи и проверка на цифрови сигнатури. При създаването на инстанция на класа `Signature` се задава име на алгоритъм за цифрови подписи, който да бъде използван. Поддържат се различни алгоритми като `SHA1withRSA`, `SHA1withDSA`, `MD5withRSA` и др. При подписване на съобщения се използват методите `initSign()`, на който се подава личен ключ, `update()`, на който се подава съобщението, което трябва да се подпише и `sign()`, който подписва съобщението и връща изчислената сигнатура. При верификация на цифров подпис се използват методите `initVerify()`, на който се подава публичния ключ, който да бъде използван, `update()`, на който се подава подписаното съобщение и `verify()`, на който се подава сигнатурата, която ще се проверява. В резултат `verify()` връща булева стойност – дали верификацията на сигнатурата е успешна.

`java.security.cert.CertificateFactory` – предоставя функционалност за зареждане на сертификати, сертификационни вериги и CRL списъци от поток. При прочитането на сертификат от поток с метода `generateCertificate()` сертификатът трябва да е DER-кодиран (съгласно стандарта PKCS#7) и може да бъде представен или в бинарен или в текстов вид (с кодиране Base64). При прочитането на сертификационни вериги с метода `generateCertPath()` може да се зададе кодирането, което да бъде използвано. Допустими са кодиранията `PkiPath`, което отговаря на ASN.1 DER последователност от сертификати и `PKCS7`, което представлява PKCS#7 SignedData обект (обикновено такива обекти се записват във файлове със стандартно разширение .P7B). При използване на PKCS7 кодиране трябва да се има предвид, че при него подредбата на сертификатите не се запазва.

`java.security.GeneralSecurityException` и `java.security.cert.CertificateException` са класове за изключения, които се пораждат при работа с цифрови подписи и сертификати.

Класовете за проверка и построяване на сертификационни вериги се намират в Java Certification Path API. Тази спецификация дефинира класовете `java.security.cert.CertPathValidator`, който служи за верификация на дадена сертификационна верига и `java.security.cert.CertPathBuilder`, който служи за построяване на сертификационна верига по зададено множество от доверени Root-сертификати и зададено множество от сертификати, които могат да се използват за междинни звена в сертификационната верига. Ще дадем и кратко описание на по-важните класове:

`java.security.cert.CertPath` – представлява сертификационна верига (наредена последователност от сертификати). По конвенция една сертификационна верига започва от някакъв клиентски сертификат, следван нула или повече сертификата на междинни сертифициращи организации и завършва с Root-сертификат на някоя сертифицираща организация от първо ниво, като при това всеки от сертификатите във веригата е издаден и подписан от следващия след него. Класът `CertPath` е предназначен да съхранява такива правилно построени сертификационни вериги, но може да съхранява и просто съвкупности от сертификати без те да изграждат някаква сертификационна верига.

java.security.cert.TrustAnchor – представлява крайна точка на доверие при верификацията на сертификационни вериги. Състои се от публичен ключ, име на доверена сертифицираща организация и съвкупност от ограничения, с които се задава множеството от пътища, което може да бъде проверявано. Обикновено тази съвкупност от ограничения не се задава и по подразбиране няма такива ограничения. За простота можем да разглеждаме един `TrustAnchor` обект като доверен Root-сертификат, който се използва при верификация на сертификационни вериги.

java.security.cert.PKIXParameters – представлява помощен клас, който описва параметрите на алгоритъма `PKIX`, използван за верификация на сертификационни вериги. Тези параметри включват списък от крайните точки на доверие за верификацията (множество от `TrustAnchor` обекти), датата, към която се верифицира сертификационната верига, указание дали да се използват CRL списъци и различни други настройки на алгоритъма.

java.security.cert.CertPathValidator – предоставя функционалност за верификация на сертификационни вериги. При създаването на инстанция на този клас се задава алгоритъма за верификация, който да се използва. Най-често се използва алгоритъма `PKIX`, който е описан в RFC-3280. При проверка на сертификационни вериги алгоритъмът `PKIX` започва от първия сертификат във веригата (сертификата на потребителя), продължава със следващия и завършва с последния. Необходимо е този последен сертификат от веригата да е подписан от сертификат, който е в списъка на крайните точки на доверие за верификацията (множеството от `TrustAnchor` обекти). По спецификация веригата, която се проверява, не трябва да съдържа крайната точка от проверката, т.е. не трябва да завършва с Root-сертификата на някоя сертифицираща организация, а с предходния преди него. Поради тази причина, когато се прочете една сертификационна верига от някакво защитено хранилище за ключове и сертификати, преди да се започне проверката ѝ е необходимо от нея да се премахне последния сертификат. В противен случай е възможно проверката да пропадне, независимо, че веригата е валидна. Методът за проверка на сертификационна верига `validate()` изисква като входни параметри веригата, която ще се проверява (от която е премахнат последния сертификат) и параметрите на алгоритъма за проверка. За алгоритъма `PKIX` тези параметри представляват обект от класа `PKIXParameters`, който съдържа списъка от доверени Root-сертификати, които да бъдат използвани.

Проблеми при подписването на документи в Web-базирани системи

Да си представим, че разработваме някаква информационна система с Web-базиран потребителски интерфейс, която е достъпна от Интернет. Потребителите на тази система трябва да могат да изпращат на сървъра различни документи. Документите могат да бъдат файлове с най-разнообразни формати – MS Word .DOC файлове, Adobe Acrobat .PDF файлове, MS Excel .XLS файлове, JPEG и GIF изображения, текстови файлове и др. За да се следи самоличността на изпращачите и за да се гарантира, че никой не може да промени документите след тяхното изпращане, е необходимо системата да дава възможност на потребителите да подписват с цифров подпис изпращаните файлове. Да разгледаме проблемите, които стоят пред реализацията на такава система.

За целите на цифровия подпис системата трябва да използва криптография, базирана на публични ключове, а за удостоверяване на самоличността на потребителите, които подписват документи, е най-удобно да бъдат използвани цифрови сертификати.

Сертификатите могат да бъдат или собственоръчно-подписани (self-signed) или да бъдат издадени от някоя сертифицираща организация. Ако потребителите собственоръчно си издават сертификати и подписват с тях изпращаните документи, няма гаранция че някой злонамерен потребител няма да си издаде сертификат с чуждо име и да подпише някакви документи от името на друг потребител.

За да се избегне този проблем е възможен друг вариант – лицето, което поддържа системата (системният администратор) да издава цифрови сертификати на потребителите. За това има две възможни схеми – системният администратор да генерира публичните и личните ключове на потребителите или потребителите да си ги генерират сами. Ако администраторът генерира личните ключове на потребителите, той потенциално може да злоупотреби с тях. Остава възможността потребителите сами да генерират публичния и личния си ключ, след което да изпращат по някакъв начин публичния си ключ заедно с информация за самоличността си на системния администратор, а той да им издава сертификат. Тази възможност не гарантира по никакъв начин, че някой потребител няма да

изпрати фалшива информация за самоличността си. Остава варианта всеки потребител да носи лично на администратора публичния си ключ и той да му издава сертификат след внимателна проверка на документите му за самоличност. Този вариант е много по-приемлив, но реално системният администратор поема ролята на сертифицираща организация и може евентуално да злоупотреби с правата си да издава сертификати.

За да се решат всички тези проблеми, се използват услугите на утвърдените сертифициращи организации. Всеки потребител на системата закупува сертификат от някоя сертифицираща организация и тази организация гарантира, че издаденият сертификат е наистина на лицето, на което е издаден. Личният ключ, съответстващ на този сертификат, остава достъпен единствено за неговия собственик и за никой друг. Благодарение на това когато даден потребител подпише даден документ със сертификата, издаден му от някоя сертифицираща организация, има гаранция, че подписът е положен наистина от него. Фалшификация е възможна само ако злонамерени лица се сдобият с личния ключ на някой потребител, за което отговорност носи самият този потребител.

Закупуването на цифров сертификат е свързано с известни разходи, които всеки потребител трябва да направи, но това е единственият надежден начин да се гарантира сигурността. Обикновено при закупуване на цифров сертификат потребителят може да се сдобие с PFX файл, който съдържа издадения му сертификат и личния му ключ, защитени с парола за достъп. Ако сертифициращата организация при издаване на сертификат на своя клиент го инсталира директно в неговия Web-браузър, клиентът може да го експортира от браузъра си в PFX файл (във формат PKCS#12) и да използва по-нататък този PFX файл за подписване на документи.

Подписването на документи изисква достъп до личния ключ на потребителя, който извършва подписването. Понеже личният ключ на един потребител е достъпен само от самия него, е необходимо подписването да се извършва физически на неговата машина. В противен случай потребителят ще трябва да изпраща на сървъра личния си ключ, а това създава потенциална опасност този ключ да бъде откраднат от злонамерени лица. При Web-приложенията подписването на документи на машината на клиента не е никак лесна задача, защото клиентът разполага единствено със стандартен Web-браузър, който няма вградени функции за подписване на документи.

Един възможен подход за подписване на документи на машината на потребителя е всеки потребител да си инсталира за целта някакъв специален софтуер, но използването на такъв подход създава известни проблеми. Единият от тях е, че софтуерът за подписване трябва да има отделни версии за различните операционни системи, които потребителят би могъл да използва. Освен това поддръжката на такъв софтуер е трудна, защото при всяка промяна в него е необходимо да се заставят всички потребители да си изтеглят и инсталират променената версия. Интеграцията на такъв софтуер с Web-интерфейса на системата също не е лесна задача, а ако този софтуер не е добре интегриран, използването му може да бъде неудобно за потребителя. От съображения за сигурност има и вероятност потребителите да не са съгласни да инсталират специален софтуер на компютъра си, а ако използват чужд компютър, може и да нямат физическата възможност да направят това.

Всички тези проблеми ни насочват към търсене на друго решение на проблема с подписването на документи на машината на клиента. Една друга възможност е да се използва JavaScript или някаква технология за изпълнение на програмен код при клиента (client-side scripting), като ActiveX, Macromedia Flash, .NET Windows Forms Controls или Java-аплети. Проблемът с езика JavaScript е, че не поддържа стандартно функционалност за работа с цифрови подписи и сертификати. Освен това JavaScript не може да осъществява достъп нито до сертификатите, инсталирани в Web-браузъра на потребителя, нито до външни хранилища за ключове и сертификати. Технологията на Macromedia Flash също не поддържа цифрови подписи и сертификати, а освен това не може да осъществява достъп до локалната файлова система, а това е необходимо, защото за да бъде подписан един файл, съдържанието му трябва да бъде прочетено. ActiveX контролите могат технически да решат проблема, но работят само под Internet Explorer върху операционна система Windows. Windows Forms контролите работят само под Internet Explorer и допълнително изискват на машината на клиента има инсталиран Microsoft .NET Framework. Остава една, последна, възможност – да се използват Java-аплети. Те имат предимството, че могат да работят на всички по-известни Web-браузъри и всички операционни системи. Проблемът с тях е, че по принцип нямат достъп до локалната файлова система на машината, на която се изпълняват, но това ограничение може да се преодолее, ако се използват подписани Java-аплети.

Java-аплетите представляват компилирани програми на Java, които се вграждат като обекти в HTML документи и се изпълняват от Web-браузъра, когато тези документи бъдат отворени. Вграждането на аплет в една Web-страница става по начин много подобен на вграждането на картинки, но за разлика от картинките аплетите не са просто графични изображения. Те представляват програми, които използват за графичния си потребителски интерфейс правоъгълна област от страницата, в която са разположени. Аплетите се състоят от един компилиран Java клас или от съвкупност от компилирани Java класове, записани в JAR файл. Както всички програми на Java, аплетите се изпълняват от **виртуалната машина на Java** (JVM) и затова всички Web-браузъри, които поддържат аплети, имат вградена в себе си или допълнително инсталирана виртуална машина. При отварянето на HTML документ, съдържащ аплет, браузърът зарежда Java виртуалната си машина и стартира аплета в нея. За да се осигури безопасността на потребителя, на аплетите не е позволено да извършват операции, които биха могли да осъществят достъп до някаква потребителска информацията на машината, на която се изпълняват. Един аплет стандартно няма достъп до локалната файлова система, което затруднява използването на тази технология за целите на цифрово подписване на документи.

Подписаните Java-аплети предоставят механизъм за повишаване на правата на обикновените Java-аплети. Те, също като обикновените аплети, представляват JAR архиви, които съдържат съвкупност от компилирани Java класове, но в допълнение към тях съдържат цифрови подписи на всички тези класове и цифров сертификат (заедно с пълната му сертификационна верига), с който тези подписи могат да се верифицират. При изпълнението на подписан аплет Web-браузърът показва информацията от сертификата, с който този аплет е подписан и пита потребителя дали вярва на този сертификат. Ако потребителят му се довери, аплетът се изпълнява без никакви ограничения на правата и може свободно да осъществява пълен достъп до локалната файлова система на машината, на която се изпълнява. Доверяването на аплети, които са подписани от непознат производител или са подписани със сертификат, който не може да бъде верифициран, е много рисковано, защото такива аплети имат пълен достъп до машината и могат да изпълнят всякакъв злонамерен код, като например вируси и троянски коне.

Да разгледаме конкретните проблеми, които възникват при реализацията на Java-аплет, който служи за подписване на документи.

Аплетът, за да осъществява достъп до локалната файлова система, трябва да е подписан. Да предположим, че сме написали кода на аплета и след като сме го компилирали, сме получили файла `Applet.jar`. Необходимо е да подпишем този JAR файл. Можем да използваме за целта програмката **`jarsigner`**, която се разпространява стандартно с JDK 1.4. Ето един пример:

```
jarsigner -storetype pkcs12 -keystore keystore.pfx -storepass store_password -keypass
private_key_password Applet.jar signFilesAlias
```

Посочената команда подписва JAR файла `Applet.jar` с личния ключ, записан под име `signFilesAlias` в хранилището за сертификати и ключове `keystore.pfx` като използва пароли за достъп до хранилището и сертификата съответно `store_password` и `private_key_password`. Програмката `jarsigner` поддържа два типа хранилища за сертификати и ключове – Java Key Store (.JKS файлове) и PKCS#12 (.PFX файлове). За да я използваме, трябва или да имаме сертификат, издаден от някоя сертифицираща организация (PFX файл, съдържащ сертификата и съответния му личен ключ) или трябва да си генерираме собственоръчно-подписан сертификат. Това можем да направим с програмката **`keytool`**, която също върви стандартно с JDK 1.4. Ето една примерна команда за генериране на собственоръчно-подписан сертификат:

```
keytool -genkey -alias signFiles -keystore SignApplet.jks -keypass !secret -dname
"CN=My Company" -storepass !secret
```

Посочената команда генерира X.509 сертификат и съответен на него личен ключ и ги записва под име `signFiles` в хранилището за ключове и сертификати `SignApplet.jks`. В сертификата се записва, че собственикът му е "`My Company`", а за парола за достъп до хранилището и до личния ключ се използва "`!secret`". По подразбиране програмката `keytool` използва формата за хранилища JKS (Java Key Store).

Можем да използваме генерирания с горната команда сертификат за да подпишем нашия аплет по следния начин:

```
jarsigner -keystore SignApplet.jks -storepass !secret -keypass !secret Applet.jar
signFiles
```

Тази команда подписва аплета `Applet.jar` с личния ключ, записан под име "`signFiles`" в хранилището `SignApplet.jks`, използвайки парола за достъп "`!secret`". В резултат се получава подписан JAR файл, който съдържа всички файлове от архива `Applet.jar`, заедно с цифровите сигнатури на тези файлове и сертификата от хранилището `SignApplet.jks` заедно с пълната му сертификационна верига. Ако не се зададе име на файл, в който да се запише резултата, както е в посочения пример, за изходен JAR файл се използва входният JAR файл.

Кодът, с който един подписан аплет се вгражда в един HTML документ, не се различава от HTML кода, с който се вгражда обикновен аплет. Все пак, когато се използват подписани аплети, не се препоръчва да се ползва остарелият таг `<applet>`, защото при него няма начин да се укаже минималната версия на JDK, която е необходима за нормалната работа на аплета. Някои Web-браузъри (например Internet Explorer) стандартно поддържат JDK версия 1.1 и ако не се укаже, че подписаният аплет изисква по-висока версия на виртуалната машина, този аплет или стартира с ограничени права и съответно не работи правилно или въобще не стартира. За да се избегнат такива проблеми се препоръчва да се използват таговете `<object>` в Internet Explorer или `<embed>` в останалите браузъри и в тях да се укаже минималната версия на JDK, която е необходима на аплета. За автоматично преобразуване на тага `<applet>` към по-новите тагове за вграждане на аплети към JDK 1.4 има специална помощна програмка `HtmlConverter.exe`.

Средата, която изпълнява аплети в Web-браузъра на клиента (обикновено това е Java Plug-In), има грижата да прецени дали даден аплет е подписан или не. Ако един аплет е подписан, при зареждането му се появява диалог, който предупреждава, че е зареден подписан аплет, който изисква пълни права върху клиентската система, за да работи нормално. Средата дава подробна информация за сертификата, с който този аплет е подписан, съобщава дали е валиден, след което пита потребителя дали да изпълни аплета без ограничения на правата. Ако потребителят се съгласи, аплетът се стартира с пълни права, а в противен случай се изпълнява като нормален (неподписан) аплет.

Нека сега разгледаме един друг проблем. Аплетът, който трябва да подписва документи, трябва по някакъв начин да изпраща на сървъра изчислената цифрова сигнатура. Това може да се реализира по няколко начина – аплетът или отваря сокет към сървъра и му изпраща сигнатурата през този сокет, или изпраща информацията чрез заявка за достъп до някое сървърско URL или си комуникира с Web-браузъра и изпраща информацията към него, а той я препраща към сървъра. Последната възможност е най-удобна, защото изисква най-малко усилия от страна на програмиста, за да бъде изпратен и приет един подписан файл. В този случай сървърът може да получава файла заедно с подписа наведнъж с една единствена заявка от браузъра без да са необходими никакви други действия.

Да предположим, че имаме обикновена HTML форма, с която се изпращат файлове към дадено Web-приложение без да бъдат подписвани. Ако искаме да разширим тази форма, така че да поддържа и цифрови подписи, можем да интегрираме в нея Java-аплет за подписване на файлове. Ако имаме аплет, който изчислява цифрова сигнатура на даден файл и поставя тази сигнатура в някакво поле на тази HTML форма, усилията необходими за изпращане на цифровия подпис към сървъра ще са минимални. Web-браузърът, когато изпраща HTML формата, ще изпрати заедно с нея и цифровия подпис и така няма да има нужда Java-аплетът да се занимава с комуникация между клиента и сървъра. Стандартно Java-аплетите могат да осъществяват достъп до HTML страницата, от която са заредени. Тази възможност може да бъде използвана за да се вземе от HTML формата името на файла, който потребителят ще изпраща, за да бъде прочетено и подписано съдържанието на този файл. Резултатът от подписването може да бъде върнат в някое поле от същата HTML форма. Технически такова взаимодействие между аплет и Web-браузър може да се реализира чрез стандартния клас `netscape.javascript.JSObject`, който е достъпен от всички браузъри, поддържащи аплети. Този клас предоставя функционалност за достъп до всички свойства на текущия прозорец на Web-браузъра, от който е бил зареден аплета, а това означава, че от аплет можем да имаме достъп до HTML документа зареден в този прозорец, до HTML формите в него, до полетата в тези форми и въобще до всичко, до което можем да имаме достъп с JavaScript. Някои браузъри позволяват на аплетите да изпълняват JavaScript и да осъществяват достъп до HTML документа, от който са заредени, само ако това е изрично указано чрез параметри на тага, с който те се вграждат в документа. Такива са параметрите "`mayscript`" и "`scriptable`" и те трябва да имат стойност "`true`".

По принцип има и още една възможност за осъществяване на комуникацията между аплета и браузъра – вместо аплетът да записва в поле от формата резултата от изчислението на цифровата сигнатура на изпращания файл, от JavaScript функция би могъл да се извиква някакъв метод на аплета, който да връща цифровия подпис и след това също с JavaScript да се записва този подпис в някое поле от формата. Този подход не работи, защото в повечето браузъри JavaScript кодът се изпълнява с такива права, че не може да осъществява достъп до файловата система. Независимо, че аплетът е подписан и може да чете локални файлове, ако бъде извикан от JavaScript тази негова възможност се отнема. Затова вместо да извикваме Java функция за подписване на файл от JavaScript, можем да направим аплет с формата на бутон, който при натискане да подписва избрания от потребителя файл и да записва изчислената сигнатура в определено поле на HTML формата. По желание може да се накара аплетът чрез JavaScript автоматично да изпраща HTML формата след изчисляване на сигнатурата за да не е възможно потребителят да промени нещо по тази форма след извършване на подписването. В такъв случай може да е удобно HTML формата да няма бутон за изпращане и изпращането й да бъде възможно единствено от аплета и то само след успешно подписване. Така потребителят няма да има възможност да изпраща неподписани или грешно подписани файлове.

При подписване на документ е необходимо на сървъра да се изпраща не само документа и изчисления от него цифров подпис, но също и сертификата, използван при подписването, придружен от цялата му сертификационна верига (ако е налична). Сертификатът е необходим, защото в него се съхранява публичният ключ на потребителя, извършил подписването, а без него не може да се верифицира подписа. Освен това сертификатът свързва този публичен ключ с конкретно лице, извършило подписването. Ако изпращаме на сървъра само документа, сигнатурата и публичния ключ, ще можем да проверим валидността сигнатурата, но няма да имаме информация кое е лицето, което притежава този публичния ключ, освен ако сървърът няма някакъв списък от публичните ключове на всички свои клиенти. В общия случай най-удобно е на сървъра да се изпраща сертификата на потребителя заедно със сертификационната му верига. Самият процес на подписване, който започва при натискане на бутона от аплета за подписване, може да се извърши по следния начин:

1. Подканва се потребителят да избере от локалната си файлова система защитено хранилище (PFX файл), съдържащо цифровия му сертификат и съответните на него личен ключ и сертификационна верига. Изисква се от потребителя да въведе паролата си за достъп до информацията в избраното защитено хранилище.
2. Зарежда се избрания PFX файл и от него се изваждат сертификата на потребителя, съответния му личен ключ и цялата сертификационна верига, свързана с този сертификат.
3. Взема се името на файла за подписване от HTML формата. Извършва се подписване на този файл с личния ключ на потребителя.
4. Записват се резултата от подписването и сертификата на потребителя заедно с цялата му сертификационна верига в определени полета от HTML формата.

Сървърът, който посреща подписания файл, има грижата да провери дали файлът е коректно подписан с личния ключ, съответстващ на изпратения сертификат. Освен това сървърът трябва на даден етап да проверява дали използваният сертификат е валиден. Това може да става веднага при получаването на файла или на по-късен етап, ако е необходимо да се установи от кого е подписан даден документ.

Освен аплета, който подписва изпращаните от потребителя файлове, нашата Web-базирана информационна система трябва да реализира и функционалност за посрещане на тези подписани файлове заедно с проверка на получената сигнатура. Необходимо е още получените клиентски сертификати и сертификационни вериги също да бъдат проверявани, за да е ясно кой всъщност подписва получените файлове. Да разгледаме проблемите, свързани с тези проверки.

Проверката на цифровия подпис има за цел да установи дали изпратената сигнатура съответства на изпратения файл и на изпратения сертификат, т.е. дали тя е истинска. Тази проверка се осъществява по стандартния начин за верификация на сигнатури – от сертификата на изпращача се извлича публичния му ключ и се проверява дали сигнатурата на получения документ е направена със съответния му личен ключ. За проверката на цифрови сигнатури има стандартни класове в Java Cryptography Architecture.

Проверката на получения сертификат има за цел да установи дали публичният ключ, записан в него, е наистина собственост на лицето, на което сертификатът е издаден, т.е. дали потребителят е наистина този, за когото се представя при подписването. Тази проверка е по-сложна и изисква предварителна подготовка.

Има няколко механизма за проверката на цифрови сертификати. Ние ще разгледаме два от тях. Класическият механизъм за проверка на сертификат изисква да се провери сертификационната му верига. За да се провери една сертификационна верига, всички сертификати, които я изграждат трябва да са налични. В противен случай проверката не може да се извърши.

В нашата система потребителят използва при подписването стандартни защитени хранилища за ключове и сертификати, записани в PFX файлове. Както знаем, тези файлове обикновено съдържат сертификат и съответстващ му личен ключ. В повечето случаи сертификатът е придружен от пълната му сертификационна верига, но понякога такава верига не е налична. Например, когато потребителят използва за подписване на документи собственоръчно-подписан сертификат, този сертификат няма сертификационна верига. Следователно е възможно при получаването на подписан файл на сървъра да се получи сертификата на потребителя заедно с пълната му сертификационна верига, но е възможно също да се получи само сертификата без никаква сертификационна верига.

Ако е налична сертификационната верига на използвания при подписването сертификат, тази верига може да се провери по класическия начин – чрез проверка на всеки от сертификатите, които я изграждат и проверка на валидността на връзките между тях. За целта може да се използват средствата на Java Certification Path API и алгоритъма `PKIX`, който е реализиран стандартно в JDK 1.4. Необходимо е единствено приложението да поддържа множество от Root-сертификати на сертифициращи организации от първо ниво, на които безусловно вярва (trusted CA root certificates).

В случай, че сертификационната верига на използвания за подписването сертификат не е налична, може да се използва друг, макар и малко по-неудобен, механизъм за верификация – директна верификация на сертификата. Вместо да се изгражда и верифицира сертификационната верига на даден сертификат, той може да се проверява само дали е подписан директно от сертификат, на който имаме доверие. Системата може да поддържа съвкупност от сертификати, на които има доверие (trusted certificates). При проверка на даден сертификат може да се търси сертификат от списъка на доверените сертификати, който се явява негов директен издател. Ако се намери такъв доверен сертификат, проверяваният сертификат, може да се счита за валиден, ако не е с изтекъл срок на годност.

Главното неудобство на тази схема за проверка на сертификати е, че е необходимо системата да разполага със сертификатите на всички междинни сертифициращи организации, които потребителите биха могли да използват. Ако за даден потребителски сертификат системата не разполага със сертификата, с който сертифициращата организация го е подписала при неговото издаване, този потребителски сертификат няма да бъде успешно верифициран, дори ако е валиден. Все пак тази схема за проверка може да бъде полезна, когато потребителите използват сертификати, които не са придружени от сертификационна верига.

Разгледахме основните проблеми, които стоят пред цифровото подписване на документи в Web-приложения и предложихме конкретни идеи за тяхното решаване чрез използване на подписан Java-аплет. Анализирахме и проблемите по проверката на цифрови подписи, сертификати и сертификационни вериги. Нека сега разгледаме една система, в която за изложените проблеми са реализирани конкретни решения:

Система за подписване на документи в Web-приложения NakovDocumentSigner

NakovDocumentSigner представлява безплатна система (framework) за цифрово подписване на документи в Java-базирани Web-приложения, разработена в Софийския университет “Св. Климент Охридски” от екипа на Светлин Наков. Системата се състои от следните компоненти:

- Подписан Java-аплет, който служи за цифрово подписване на файлове преди изпращането им.
- Примерно Web-приложение, което посреща подписаните файлове и проверява дали полученият цифров подпис отговаря на получения файл и сертификат.

- Проста подсистема за верификация на сертификати и сертификационни вериги, реализирана като част от примерното Web-приложение.

Пълният сорс-код на системата NakovDocumentSigner, заедно с инструкции за компилиране и използване, е достъпен от нейния сайт – http://www.nakov.com/documents-signing/.

Подписаният Java-аплет изисква инсталиран Java Plug-In версия 1.4 или по-нова на машината на клиента. Това е необходимо, защото аплетът използва Java Cryptography Architecture, която не е достъпна при по-ниските версии на Java Plug-In. Аплетът не работи със стандартната виртуална машина, която идва с някои версии на Internet Explorer. Той е подписан, за да може да осъществява достъп до локалната файлова система на потребителя и работи правилно само ако потребителят му позволи да бъде изпълнен с пълни права.

Аплетът следва твърдо описаните в предходната точка стъпки за подписване на документи и представлява сам по себе си един бутон, който се поставя в HTML формата за изпращане на файлове. Като параметри му се подават името на полето, от което се взима файла за подписване и имената на полетата, в които се записват изчислената сигнатура и използвания цифров сертификат, заедно с цялата му сертификационна верига.

Очаква се клиентът да притежава цифров сертификат и съответен на него личен ключ, записани в PFX файл, като паролата за достъп до този файл трябва да съвпада с паролата за достъп до личния ключ. Такъв PFX файл може да се придобие при закупуването на сертификат от някоя сертифицираща организация.

За тестови цели могат да бъдат използвани демонстрационни цифрови сертификати, които някои известни сертификационни организации като Thawte, VeriSign и GlobalSign издават на потенциални клиенти от своите сайтове. Срещу валиден e-mail адрес Thawte издават напълно безплатно сертификати за подписване на електронна поща. Те могат да бъдат получени само за няколко минути от адрес http://www.thawte.com/html/COMMUNITY/personal/index.html. VeriSign издават на потенциални клиенти демонстрационни сертификати със срок на валидност 60 дни също срещу валиден e-mail адрес от http://www.verisign.com/client/enrollment/index.html. GlobalSign също предлагат демонстрационни сертификати срещу валиден e-mail адрес от сайта си http://secure.globalsign.net/, но техните са с валидност 30 дни. И трите сертификационни организации издават сертификатите си по Интернет и в резултат потребителите ги получават инсталирани директно в техните Web-браузъри. За да бъдат използвани такива сертификати в NakovDocumentSigner, те трябва да бъдат експортирани заедно с личния им ключ в .PFX или .P12 файл.

Сорс-кодът на аплета се състои от няколко файла, достъпни от сайта на NakovDocumentSigner, които са дадени по-долу:

DigitalSignerApplet.java

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.util.Arrays;
import java.util.Enumeration;
import java.util.List;
import java.security.GeneralSecurityException;
import java.security.KeyStore;
import java.security.PrivateKey;
import java.security.Signature;
import java.security.cert.CertPath;
import java.security.cert.Certificate;
import java.security.cert.CertificateException;
import java.security.cert.CertificateFactory;
import netscape.javascript.JSException;
import netscape.javascript.JSObject;

/**
```

```
 * Applet for digital signing documents. The applet is intended to be placed in a HTML document
 * containing a single HTML form that is used for applet input/output. The applet accepts several
 * parameters - the name of the field in the HTML form that contains the file name to be signed
 * and the names of the fields in the HTML form, where the certificate chain and signature should
 * be stored. If the signing process is sucecssfull, the signature and certificate chain fields
 * in the HTML form are filled. Otherwise an error message explaining the failure reason is shown
 * by the applet. The applet asks the user to locate in his local file system a PFX file (PKCS#12
 * keystore), that holds his certificate (with the corresponding certificate chain) and his
 * private key. Also the applet asks the user to enter his password for accessing the keystore
 * and the private key. If the specified file contains a certificate and a corresponding private
 * key that is accessible with supplied password, the signature of the file is calculated and is
 * placed in the HTML form. The applet consider the password for the keystore and the password
 * for the private key in it are the same (this is typical for PFX files). In addition to the
 * calculated signature the certificate chain is extracted from the PFX file and is placed in
 * the HTML form too. The digital signature is placed as Base64-encoded sequence of bytes. The
 * certificate chain is placed as ASN.1 DER-encoded sequence of bytes, additionally encoded in
 * Base64. In case the PFX file contains only one certificate without its full certificate chain,
 * a chain consisting of this single certificate is extracted and stored in the HTML form instead
 * of a full certificate chain. Digital singature algorithm used is SHA1withRSA. The length of
 * the private key and respectively the length of the calculated singature depend on the length
 * of the private key in the PFX file. The applet should be able to access the local machine's
 * file system for reading and writing. Reading the local file system is required for the applet
 * to access the file that should be signed and the PFX keystore file. Writing the local file
 * system is required for the applet to save its settings in the user's home directory. Accessing
 * the local file system is not possible by default, but if the applet is digitally signed (with
 * jarsigner), it runs with no security restrictions and can do anything. This applet should be
 * signed in order to run. A JRE version 1.4 or hihger is required for accessing the cryptography
 * functionality, so the applet will not run in any other runtime environment.
 *
 * This file is part of NakovDocumentSigner digital document
 * signing framework for Java-based Web applications:
 * http://www.nakov.com/documents-signing/
 *
 * Copyright (c) 2003 by Svetlin Nakov - http://www.nakov.com
 * All rights reserved. This code is freeware. It can be used
 * for any purpose as long as this copyright statement is not
 * removed or modified.
 */
public class DigitalSignerApplet extends Applet {

    private static final String FILE_NAME_FIELD_PARAMETER = "fileNameField";
    private static final String CERTIFICATE_CHAIN_FIELD_PARAMETER = "certificateChainField";
    private static final String SIGNATURE_FIELD_PARAMETER = "signatureField";
    private static final String SIGN_BUTTON_CAPTION_PARAMETER = "signButtonCaption";

    private static final String PKCS12_KEYSTORE_TYPE = "PKCS12";
    private static final String X509_CERTIFICATE_TYPE = "X.509";
    private static final String CERTIFICATE_CHAIN_ENCODING = "PkiPath";
    private static final String DIGITAL_SIGNATURE_ALGORITHM_NAME = "SHA1withRSA";

    private Button mSignButton;

    /**
     * Initializes the applet - creates and initializes its graphical user interface.
     * In practice the applet consists of a single button, that fills its all surface.
     * The button's caption is taken from the applet parameter SIGN_BUTTON_CAPTION_PARAMETER.
     */
    public void init() {
        String signButtonCaption = this.getParameter(SIGN_BUTTON_CAPTION_PARAMETER);
        mSignButton = new Button(signButtonCaption);
        mSignButton.setLocation(0, 0);
        Dimension appletSize = this.getSize();
        mSignButton.setSize(appletSize);
        mSignButton.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e) {
                signSelectedFile();
            }
        });
        this.setLayout(null);
        this.add(mSignButton);
    }

    /**
     * Signs the selected file. The file name is taken from a field in the HTML document.
```

```
     * The result consists of the calculated digital signature and certificate chain,
     * both placed in fields in the HTML document, encoded in Base64 format. The HTML
     * document should contain only one HTML form. The name of the field, that contains
     * the name of the file to be signed is obtained from FILE_NAME_FIELD_PARAMETER applet
     * parameter. The names of the output fields for the signature and the certificate chain
     * are obtained from the applet parameters CERTIFICATE_CHAIN_FIELD_PARAMETER and
     * SIGNATURE_FIELD_PARAMETER.
     */
    private void signSelectedFile() {
        try {
            // Get the file name to be signed from the form in the HTML document
            JSObject browserWindow = JSObject.getWindow(this);
            JSObject document = (JSObject) browserWindow.getMember("document");
            JSObject forms = (JSObject) document.getMember("forms");
            JSObject mainForm = (JSObject) forms.getSlot(0);
            String fileNameFieldName = this.getParameter(FILE_NAME_FIELD_PARAMETER);
            JSObject fileNameField = (JSObject) mainForm.getMember(fileNameFieldName);
            String fileName = (String) fileNameField.getMember("value");

            // Perform file signing
            CertificateChainAndSignatureBase64Encoded signingResult = signFile(fileName);

            if (signingResult != null) {
                // Document successfully signed. Fill the certificate and signature fields
                String certChainFieldName = this.getParameter(CERTIFICATE_CHAIN_FIELD_PARAMETER);
                JSObject certChainField = (JSObject) mainForm.getMember(certChainFieldName);
                certChainField.setMember("value", signingResult.mCertificateChain);
                String signatureFieldName = this.getParameter(SIGNATURE_FIELD_PARAMETER);
                JSObject signatureField = (JSObject) mainForm.getMember(signatureFieldName);
                signatureField.setMember("value", signingResult.mSignature);
            } else {
                // User canceled signing
            }
        }
        catch (DocumentSignException dse) {
            // Document signing failed. Display error message
            String errorMessage = dse.getMessage();
            JOptionPane.showMessageDialog(this, errorMessage);
        }
        catch (SecurityException se) {
            JOptionPane.showMessageDialog(this,
                "Unable to access the local file system.\n" +
                "This applet should be started with full security permissions.\n" +
                "Please accept to trust this applet when the Java Plug-In ask you.");
        }
        catch (JSException jse) {
            JOptionPane.showMessageDialog(this,
                "Unable to access some of the fields in the\n" +
                "HTML form. Please check applet parameters.");
        }
        catch (Exception e) {
            e.printStackTrace();
            JOptionPane.showMessageDialog(this, "Unexpected error: " + e.getMessage());
        }
    }

    /**
     * Signs given local file. The certificate chain and private key to be used for signing
     * are specified by the local user who choose a PFX file and password for accessing it.
     * @param aFileName the name of the file to be signed.
     * @return the digital signature of the given file and the certificate chain of the
     * certificate used for signing the file, both Base64-encoded or null if the signing
     * process is canceled by the user
     * @throws DocumentSignException when a problem arised during the singing process
     * (e.g. invalid file format, invalid certificate, invalid password, etc.)
     */
    private CertificateChainAndSignatureBase64Encoded signFile(String aFileName)
    throws DocumentSignException {

        // Load the file for signing
        byte[] documentToSign = null;
        try {
            documentToSign = readFileInByteArray(aFileName);
        } catch (IOException ioex) {
```

```
            String errorMessage = "Can not read the file to be signed " + aFileName + ".";
            throw new DocumentSignException(errorMessage, ioex);
        }

        // Show a dialog for selecting PFX file and password
        CertificateFileAndPasswordDialog certFileAndPasswdDlg =
            new CertificateFileAndPasswordDialog();
        if (certFileAndPasswdDlg.run()) {

            // Load the keystore from specified file using the specified password
            String keyStoreFileName = certFileAndPasswdDlg.getCertificateFileName();
            String password = certFileAndPasswdDlg.getCertificatePassword();
            KeyStore userKeyStore = null;
            try {
                userKeyStore = loadKeyStoreFromPFXFile(keyStoreFileName, password);
            } catch (Exception ex) {
                String errorMessage = "Can not read certificate keystore file (" +
                    keyStoreFileName + ").\nThe file is either not in PKCS#12 format (.P12 " +
                    "or .PFX) or is corrupted or the password you entered is invalid.";
                throw new DocumentSignException(errorMessage, ex);
            }

            // Get the private key and its certificate chain from the keystore
            PrivateKeyAndCertificateChain privateKeyAndCertificateChain = null;
            try {
                privateKeyAndCertificateChain =
                    getPrivateKeyAndCertChain(userKeyStore, password);
            } catch (GeneralSecurityException gsex) {
                String errorMessage = "Can not extract private key and certificate chain " +
                    "from specified file with given password. Probably incorrect password.";
                throw new DocumentSignException(errorMessage, gsex);
            }

            // Check if private key is available
            PrivateKey privateKey = privateKeyAndCertificateChain.mPrivateKey;
            if (privateKey == null) {
                String errorMessage = "Can not find private key in the specified file " +
                    keyStoreFileName + ".";
                throw new DocumentSignException(errorMessage);
            }

            // Check if X.509 certificate chain is available
            Certificate[] certChain = privateKeyAndCertificateChain.mCertificateChain;
            if (certChain == null) {
                String errorMessage = "Can not find neither certificate nor certificate " +
                    "chain in the specified file " + keyStoreFileName + ".";
                throw new DocumentSignException(errorMessage);
            }

            // Create the result object
            CertificateChainAndSignatureBase64Encoded signingResult =
                new CertificateChainAndSignatureBase64Encoded();

            // Save X.509 certificate chain in the result encoded in Base64
            try {
                signingResult.mCertificateChain = encodeX509CertChainToBase64(certChain);
            }
            catch (CertificateException cee) {
                String errorMessage = "Invalid certificate chain found in the file " +
                    keyStoreFileName + ".";
                throw new DocumentSignException(errorMessage);
            }

            // Calculate the digital signature of the file,
            // encode it in Base64 and save it in the result
            try {
                byte[] digitalSignature = signDocument(documentToSign, privateKey);
                signingResult.mSignature = Base64Utils.base64Encode(digitalSignature);
            } catch (GeneralSecurityException gsex) {
                String errorMessage = "Error signing file " + aFileName + ".";
                throw new DocumentSignException(errorMessage, gsex);
            }

            // Document signing completed succesfully
```

```
            return signingResult;
        }
        else {
            // Document signing canceled by the user
            return null;
        }
    }

    /**
     * Loads a keystore from .PFX or .P12 file (file format should be PKCS#12)
     * using given keystore password.
     */
    private KeyStore loadKeyStoreFromPFXFile(String aFileName, String aKeyStorePassword)
    throws GeneralSecurityException, IOException {
        KeyStore keyStore = KeyStore.getInstance(PKCS12_KEYSTORE_TYPE);
        FileInputStream keyStoreStream = new FileInputStream(aFileName);
        char[] password = aKeyStorePassword.toCharArray();
        keyStore.load(keyStoreStream, password);
        return keyStore;
    }

    /**
     * @return private key and certificate chain for this private key extracted from given
     * keystore using given password to access the keystore and the same password to access
     * the private key in it. The keystore is considered to have only one entry that contains
     * both certificate chain and the corresponding private key. If the certificate has no
     * entries, null is returned.
     */
    private PrivateKeyAndCertificateChain getPrivateKeyAndCertChain(
        KeyStore aKeyStore, String aKeyPassword)
    throws GeneralSecurityException {
        char[] password = aKeyPassword.toCharArray();
        Enumeration aliasesEnum = aKeyStore.aliases();
        while (aliasesEnum.hasMoreElements()) {
            String alias = (String)aliasesEnum.nextElement();
            Certificate[] certificateChain = aKeyStore.getCertificateChain(alias);
            PrivateKey privateKey = (PrivateKey) aKeyStore.getKey(alias, password);
            PrivateKeyAndCertificateChain result = new PrivateKeyAndCertificateChain();
            result.mPrivateKey = privateKey;
            result.mCertificateChain = certificateChain;
            return result;
        }
        return null;
    }

    /**
     * @return Base64-encoded ASN.1 DER representation of given X.509 certificate chain.
     */
    private String encodeX509CertChainToBase64(Certificate[] aCertificateChain)
    throws CertificateException {
        List certList = Arrays.asList(aCertificateChain);
        CertificateFactory certFactory = CertificateFactory.getInstance(X509_CERTIFICATE_TYPE);
        CertPath certPath = certFactory.generateCertPath(certList);
        byte[] certPathEncoded = certPath.getEncoded(CERTIFICATE_CHAIN_ENCODING);
        String base64encodedCertChain = Base64Utils.base64Encode(certPathEncoded);
        return base64encodedCertChain;
    }

    /**
     * Reads the specified file into a byte array.
     */
    private byte[] readFileInByteArray(String aFileName)
    throws IOException {
        File file = new File(aFileName);
        FileInputStream fileStream = new FileInputStream(file);
        try {
            int fileSize = (int) file.length();
            byte[] data = new byte[fileSize];
            int bytesRead = 0;
            while (bytesRead < fileSize) {
                bytesRead += fileStream.read(data, bytesRead, fileSize-bytesRead);
            }
            return data;
        }
```

```
        finally {
            fileStream.close();
        }
    }

    /**
     * Signs given document with a given private key.
     */
    private byte[] signDocument(byte[] aDocument, PrivateKey aPrivateKey)
    throws GeneralSecurityException {
        Signature signatureAlgorithm = Signature.getInstance(DIGITAL_SIGNATURE_ALGORITHM_NAME);
        signatureAlgorithm.initSign(aPrivateKey);
        signatureAlgorithm.update(aDocument);
        byte[] digitalSignature = signatureAlgorithm.sign();
        return digitalSignature;
    }

    /**
     * Data structure that holds a pair of private key and
     * certificate chain corresponding to this private key.
     */
    static class PrivateKeyAndCertificateChain {
        public PrivateKey mPrivateKey;
        public Certificate[] mCertificateChain;
    }

    /**
     * Data structure that holds a pair of Base64-encoded
     * certificate chain and digital signature.
     */
    static class CertificateChainAndSignatureBase64Encoded {
        public String mCertificateChain = null;
        public String mSignature = null;
    }

    /**
     * Exception class used for document signing errors.
     */
    static class DocumentSignException extends Exception {
        public DocumentSignException(String aMessage) {
            super(aMessage);
        }

        public DocumentSignException(String aMessage, Throwable aCause) {
            super(aMessage, aCause);
        }
    }

}
```

CertificateFileAndPasswordDialog.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.filechooser.FileFilter;
import java.io.*;
import java.util.Properties;

/**
 * Dialog for choosing certificate file name and password for it. Allows the user to select a PFX
 * file and enter a password for accessing it. The last used PFX file is remembered in the config
 * file called ".digital_signer_applet.config", located in the user home directory in order to be
 * automatically shown the next time when the same user access this dialog.
 *
 * This file is part of NakovDocumentSigner digital document
 * signing framework for Java-based Web applications:
 * http://www.nakov.com/documents-signing/
 *
 * Copyright (c) 2003 by Svetlin Nakov - http://www.nakov.com
 * All rights reserved. This code is freeware. It can be used
 * for any purpose as long as this copyright statement is not
 * removed or modified.
```

```
 */
public class CertificateFileAndPasswordDialog extends JDialog {

    private static final String CONFIG_FILE_NAME = ".digital_signer_applet.config";
    private static final String PERSONAL_KEYSTORE_FILE_NAME_KEY = "last-PFX-file-name";

    private JButton mBrowseForCertButton = new JButton();
    private JTextField mCertFileNameTextField = new JTextField();
    private JLabel mChooseCertFileLabel = new JLabel();
    private JTextField mPasswordTextField = new JPasswordField();
    private JLabel mEnterPasswordLabel = new JLabel();
    private JButton mSignButton = new JButton();
    private JButton mCancelButton = new JButton();

    private boolean mResult = false;

    /**
     * Initializes the dialog - creates and initializes its GUI controls.
     */
    public CertificateFileAndPasswordDialog() {
        // Initialize the dialog
        this.getContentPane().setLayout(null);
        this.setSize(new Dimension(426, 165));
        this.setBackground(SystemColor.control);
        this.setTitle("Select digital certificate");
        this.setResizable(false);

        // Center the dialog in the screen
        Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
        Dimension dialogSize = this.getSize();
        int centerPosX = (screenSize.width - dialogSize.width) / 2;
        int centerPosY = (screenSize.height - dialogSize.height) / 2;
        setLocation(centerPosX, centerPosY);

        // Initialize certificate keystore file label
        mChooseCertFileLabel.setText(
            "Please select your certificate keystore file (.PFX / .P12) :");
        mChooseCertFileLabel.setBounds(new Rectangle(10, 5, 300, 15));
        mChooseCertFileLabel.setFont(new Font("Dialog", 0, 12));

        // Initialize certificate keystore file name text field
        mCertFileNameTextField.setBounds(new Rectangle(10, 25, 315, 20));
        mCertFileNameTextField.setFont(new Font("DialogInput", 0, 12));
        mCertFileNameTextField.setEditable(false);
        mCertFileNameTextField.setBackground(SystemColor.control);

        // Initialize browse button
        mBrowseForCertButton.setText("Browse");
        mBrowseForCertButton.setBounds(new Rectangle(330, 25, 80, 20));
        mBrowseForCertButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                browseForCertButton_actionPerformed();
            }
        });

        // Initialize password label
        mEnterPasswordLabel.setText("Enter the password for your private key:");
        mEnterPasswordLabel.setBounds(new Rectangle(10, 55, 220, 15));
        mEnterPasswordLabel.setFont(new Font("Dialog", 0, 12));

        // Initialize password text field
        mPasswordTextField.setBounds(new Rectangle(10, 75, 400, 20));
        mPasswordTextField.setFont(new Font("DialogInput", 0, 12));

        // Initialize sign button
        mSignButton.setText("Sign");
        mSignButton.setBounds(new Rectangle(110, 105, 75, 25));
        mSignButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                signButton_actionPerformed();
            }
        });

        // Initialize cancel button
```

```
        mCancelButton.setText("Cancel");
        mCancelButton.setBounds(new Rectangle(220, 105, 75, 25));
        mCancelButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                cancelButton_actionPerformed();
            }
        });

        // Add the initialized components into the dialog's content pane
        this.getContentPane().add(mChooseCertFileLabel, null);
        this.getContentPane().add(mCertFileNameTextField, null);
        this.getContentPane().add(mBrowseForCertButton, null);
        this.getContentPane().add(mEnterPasswordLabel, null);
        this.getContentPane().add(mPasswordTextField, null);
        this.getContentPane().add(mSignButton, null);
        this.getContentPane().add(mCancelButton, null);
        this.getRootPane().setDefaultButton(mSignButton);

        // Add some functionality for focusing the most appropriate
        // control when the dialog is shown
        this.addWindowListener(new WindowAdapter() {
            public void windowOpened(WindowEvent windowEvent) {
                String certFileName = mCertFileNameTextField.getText();
                if (certFileName != null && certFileName.length() != 0)
                    mPasswordTextField.requestFocus();
                else
                    mBrowseForCertButton.requestFocus();
            }
        });
    }

    /**
     * Called when the browse button is pressed.
     * Shows file choose dialog and allows the user to locate a PFX file.
     */
    private void browseForCertButton_actionPerformed() {
        JFileChooser fileChooser = new JFileChooser();
        PFXFileFilter pfxFileFilter = new PFXFileFilter();
        fileChooser.addChoosableFileFilter(pfxFileFilter);
        String certFileName = mCertFileNameTextField.getText();
        File directory = new File(certFileName).getParentFile();
        fileChooser.setCurrentDirectory(directory);
        if (fileChooser.showOpenDialog(this) == JFileChooser.APPROVE_OPTION) {
            String selectedCertFileName = fileChooser.getSelectedFile().getAbsolutePath();
            mCertFileNameTextField.setText(selectedCertFileName);
        }
    }

    /**
     * Called when the sign button is pressed. Closses the dialog and sets the result flag to
     * true to indicate that the user is confirmed the information entered in the dialog.
     */
    private void signButton_actionPerformed() {
        mResult = true;
        hide();
    }

    /**
     * Called when the cancel button is pressed. Closses the dialog and sets the
     * result flag to false that indicates that the dialog is canceled.
     */
    private void cancelButton_actionPerformed() {
        mResult = false;
        hide();
    }

    /**
     * @return the file name with full path to it, where the dialog settings are stored.
     */
    private String getConfigFileName() {
        String configFileName =
            System.getProperty("user.home") + System.getProperty("file.separator") +
            CONFIG_FILE_NAME;
        return configFileName;
```

```
    }

    /**
     * Loads the dialog settings from the dialog configuration file. These settings
     * consist of a single value - the last used PFX file name with its full path.
     */
    private void loadSettings()
    throws IOException {
        // Load settings file
        String configFileName = getConfigFileName();
        FileInputStream configFileStream = new FileInputStream(configFileName);
        Properties configProps = new Properties();
        configProps.load(configFileStream);
        configFileStream.close();

        // Apply setings from the config file
        String lastCertificateFileName =
            configProps.getProperty(PERSONAL_KEYSTORE_FILE_NAME_KEY);
        if (lastCertificateFileName != null)
            mCertFileNameTextField.setText(lastCertificateFileName);
        else
            mCertFileNameTextField.setText("");
    }

    /**
     * Saves the dialog settings to the dialog configuration file. These settings
     * consist of a single value - the last used PFX file name with its full path.
     */
    private void saveSettings()
    throws IOException {
        // Create a list of settings to store in the config file
        Properties configProps = new Properties();
        String currentCertificateFileName = mCertFileNameTextField.getText();
        configProps.setProperty(PERSONAL_KEYSTORE_FILE_NAME_KEY, currentCertificateFileName);

        // Save the settings in the config file
        String configFileName = getConfigFileName();
        FileOutputStream configFileStream = new FileOutputStream(configFileName);
        configProps.store(configFileStream, "");
        configFileStream.close();
    }

    /**
     * @return the PFX file selected by the user.
     */
    public String getCertificateFileName() {
        String certFileName = mCertFileNameTextField.getText();
        return certFileName;
    }

    /**
     * @return the password entered by the user.
     */
    public String getCertificatePassword() {
        String password = mPasswordTextField.getText();
        return password;
    }

    /**
     * Shows the dialog and allow the user to choose PFX file and enter a password.
     * @return true if the user click sign button or false if the user cancel the dialog.
     */
    public boolean run() {
        try {
            loadSettings();
        } catch (IOException ioex) {
            // Loading settings failed. It is not important. Do nothing
        }

        setModal(true);
        show();

        try {
            if (mResult)
```

```
                saveSettings();
            } catch (IOException ioex) {
                // Saving settings failed. It is not important. Do nothing.
            }

            return mResult;
        }

        /**
         * File filter class, intended to accept only .PFX and .P12 files.
         */
        private static class PFXFileFilter extends FileFilter {
            public boolean accept(File aFile) {
                if (aFile.isDirectory()) {
                    return true;
                }

                String fileName = aFile.getName().toUpperCase();
                boolean accepted =
                    (fileName.endsWith(".PFX") || fileName.endsWith(".P12"));
                return accepted;
            }

            public String getDescription() {
                return "PKCS#12 certificate keystore file with private key (.PFX, .P12)";
            }
        }

}
```

Base64Utils.java

```
/**
 * Provides utilities for Base64 encode/decode of binary data.
 */
public class Base64Utils {

    private static byte[] mBase64EncMap, mBase64DecMap;

    /**
     * Class initializer. Initializes the Base64 alphabet (specified in RFC-2045).
     */
    static {
        byte[] base64Map = {
            (byte)'A', (byte)'B', (byte)'C', (byte)'D', (byte)'E', (byte)'F',
            (byte)'G', (byte)'H', (byte)'I', (byte)'J', (byte)'K', (byte)'L',
            (byte)'M', (byte)'N', (byte)'O', (byte)'P', (byte)'Q', (byte)'R',
            (byte)'S', (byte)'T', (byte)'U', (byte)'V', (byte)'W', (byte)'X',
            (byte)'Y', (byte)'Z',
            (byte)'a', (byte)'b', (byte)'c', (byte)'d', (byte)'e', (byte)'f',
            (byte)'g', (byte)'h', (byte)'i', (byte)'j', (byte)'k', (byte)'l',
            (byte)'m', (byte)'n', (byte)'o', (byte)'p', (byte)'q', (byte)'r',
            (byte)'s', (byte)'t', (byte)'u', (byte)'v', (byte)'w', (byte)'x',
            (byte)'y', (byte)'z',
            (byte)'0', (byte)'1', (byte)'2', (byte)'3', (byte)'4', (byte)'5',
            (byte)'6', (byte)'7', (byte)'8', (byte)'9', (byte)'+', (byte)'/' };
        mBase64EncMap = base64Map;
        mBase64DecMap = new byte[128];
        for (int i=0; i<mBase64EncMap.length; i++)
            mBase64DecMap[mBase64EncMap[i]] = (byte) i;
    }

    /**
     * This class isn't meant to be instantiated.
     */
    private Base64Utils() {
    }

    /**
     * Encodes the given byte[] using the Base64-encoding,
     * as specified in RFC-2045 (Section 6.8).
     *
     * @param aData the data to be encoded
```

```
     * @return the Base64-encoded <var>aData</var>
     * @exception IllegalArgumentException if NULL or empty array is passed
     */
    public static String base64Encode(byte[] aData) {
        if ((aData == null) || (aData.length == 0))
            throw new IllegalArgumentException("Can not encode NULL or empty byte array.");

        byte encodedBuf[] = new byte[((aData.length+2)/3)*4];

        // 3-byte to 4-byte conversion
        int srcIndex, destIndex;
        for (srcIndex=0, destIndex=0; srcIndex < aData.length-2; srcIndex += 3) {
            encodedBuf[destIndex++] = mBase64EncMap[(aData[srcIndex] >>> 2) & 077];
            encodedBuf[destIndex++] = mBase64EncMap[(aData[srcIndex+1] >>> 4) & 017 |
                        (aData[srcIndex] << 4) & 077];
            encodedBuf[destIndex++] = mBase64EncMap[(aData[srcIndex+2] >>> 6) & 003 |
                        (aData[srcIndex+1] << 2) & 077];
            encodedBuf[destIndex++] = mBase64EncMap[aData[srcIndex+2] & 077];
        }

        // Convert the last 1 or 2 bytes
        if (srcIndex < aData.length) {
            encodedBuf[destIndex++] = mBase64EncMap[(aData[srcIndex] >>> 2) & 077];
            if (srcIndex < aData.length-1) {
                encodedBuf[destIndex++] = mBase64EncMap[(aData[srcIndex+1] >>> 4) & 017 |
                    (aData[srcIndex] << 4) & 077];
                encodedBuf[destIndex++] = mBase64EncMap[(aData[srcIndex+1] << 2) & 077];
            }
            else {
                encodedBuf[destIndex++] = mBase64EncMap[(aData[srcIndex] << 4) & 077];
            }
        }

        // Add padding to the end of encoded data
        while (destIndex < encodedBuf.length) {
            encodedBuf[destIndex] = (byte) '=';
            destIndex++;
        }

        String result = new String(encodedBuf);
        return result;
    }


    /**
     * Decodes the given Base64-encoded data,
     * as specified in RFC-2045 (Section 6.8).
     *
     * @param aData the Base64-encoded aData.
     * @return the decoded <var>aData</var>.
     * @exception IllegalArgumentException if NULL or empty data is passed
     */
    public static byte[] base64Decode(String aData) {
        if ((aData == null) || (aData.length() == 0))
            throw new IllegalArgumentException("Can not decode NULL or empty string.");

        byte[] data = aData.getBytes();

        // Skip padding from the end of encoded data
        int tail = data.length;
        while (data[tail-1] == '=')
            tail--;

        byte decodedBuf[] = new byte[tail - data.length/4];

        // ASCII-printable to 0-63 conversion
        for (int i = 0; i < data.length; i++)
            data[i] = mBase64DecMap[data[i]];

        // 4-byte to 3-byte conversion
        int srcIndex, destIndex;
        for (srcIndex = 0, destIndex=0; destIndex < decodedBuf.length-2;
                srcIndex += 4, destIndex += 3) {
            decodedBuf[destIndex] = (byte) ( ((data[srcIndex] << 2) & 255) |
```

```
                ((data[srcIndex+1] >>> 4) & 003) );
            decodedBuf[destIndex+1] = (byte) ( ((data[srcIndex+1] << 4) & 255) |
                ((data[srcIndex+2] >>> 2) & 017) );
            decodedBuf[destIndex+2] = (byte) ( ((data[srcIndex+2] << 6) & 255) |
                (data[srcIndex+3] & 077) );
        }

        // Handle last 1 or 2 bytes
        if (destIndex < decodedBuf.length)
            decodedBuf[destIndex] = (byte) ( ((data[srcIndex] << 2) & 255) |
                ((data[srcIndex+1] >>> 4) & 003) );
        if (++destIndex < decodedBuf.length)
            decodedBuf[destIndex] = (byte) ( ((data[srcIndex+1] << 4) & 255) |
                ((data[srcIndex+2] >>> 2) & 017) );

        return decodedBuf;
    }

}
```

Аплетът използва класа `netscape.javascript.JSObject` за достъп до полетата на HTML формата, взема от нея избрания от потребителя файл и го подписва. При подписването първо се прочита съдържанието на файла, след което се показва на потребителя диалога за избор на PFX файл и парола за достъп до него. След като потребителят избере PFX файл, този файл се прочита и от него се извлича личния ключ и съответната му сертификационна верига. Тази верига винаги започваща със сертификата на потребителя, но е възможно да се състои единствено от него, т.е. да не съдържа други сертификати. Ако извличането на личния ключ и сертификационната верига от PFX файла е успешно, сертификационната верига се кодира по подходящ начин в текстов вид, за да може да бъде пренесена през текстово поле на HTML формата. Използва се стандартното кодиране `PkiPath`, което представлява последователност от ASN.1 DER-кодирани сертификати. Получената кодирана сертификационна верига се кодира допълнително с Base64 за да добие текстов вид. След това се извършва самото подписване на документа с личния ключ, прочетен от PFX файла. Получената цифрова сигнатура се кодира в текстов вид с Base64 кодиране. Накрая текстовите стойности на извлечената от PFX файла сертификационна верига и получената сигнатура се записват в определени полета на HTML формата. Имената на тези полета, както и името на полето, съдържащо името на файла за подписване, се взимат от параметри, подадени на аплета. Очаква се HTML документът, в който е разположен аплета за подписване да съдържа точно една HTML форма. Ако възникване грешка на някоя от описаните стъпки, на потребителя се показва подходящо съобщение за грешка. Грешка може да възникне при много ситуации – при невъзможност да бъде прочетен файла за подписване, при невъзможност да бъде прочетен PFX файла, при невалиден формат на PFX файла, при липса на личен ключ, при липса на сертификат, при невалиден формат на сертификата, поради грешка при самото подписване на прочетения файл, поради невъзможност за достъп до файловата система, поради невъзможност за достъп до някое поле от формата, което е необходимо и в други необичайни ситуации.

Възможно е някой да остане с впечатлението, че аплетът записва в HTML формата само сертификационната верига на потребителския сертификат, без него самия, но това не е точно така. По спецификация една сертификационна верига винаги започва със сертификата, към който принадлежи, дори в частност може да се състои единствено от този сертификат. Поради тази причина записването на сертификационната вериги на даден сертификат в HTML формата означава, че този сертификат също се записва в нея като част от тази верига.

Диалогът за избор на PFX файл и парола дава възможност за избор само измежду PFX и P12 файлове. Последният използван PFX файл се запомня заедно с пълния път до него в конфигурационен файл, намиращ се home директорията на потребителя, за да бъде използван при следващо показване на същия диалог.

Графичният потребителски интерфейс на аплета е базиран на библиотеките AWT и JFC/Swing, които се доставят стандартно с JDK 1.4.

Реализацията на всички използвани криптографски алгоритми, се осигурява от доставчика на криптографски услуги по подразбиране SunJSSE, който също е част от JDK 1.4.

За подписването на файлове се използва алгоритъма за цифрови подписи `SHA1withRSA`. Това означава, че за изчисляване на хеш-стойността на документа се използва алгоритъм SHA1, а за шифрирането на тази хеш-стойност и получаване на цифровия подпис се използва алгоритъм RSA. Алгоритъмът `SHA1withRSA` е достъпен от Java Cryptography Architecture (JCA) посредством класа `java.security.Signature`. Дължината на ключовете, използвани от RSA алгоритъма зависи от подадения от потребителя PFX файл и обикновено е 512 или 1024 бита. В зависимост от дължината на RSA ключа се получава цифрова сигнатура със същата дължина, обикновено 512 или 1024 бита (64 или 128 байта). След кодиране с Base64 тази сигнатура се записва съответно с 88 или 172 текстови символа.

Дължината на сертификационната верига зависи от броя сертификати, които я съставят и от съдържанието на всеки от тези сертификати. Обикновено след кодиране с Base64 тази верига има дължина от 200-300 до 8000-10000 текстови символа.

За осъществяването на достъп до защитеното хранилище за ключове и сертификати (PFX файла, който потребителят избира) се използва класа `java.security.KeyStore`. Аплетът очаква хранилището да бъде във формат PKCS#12 и да съдържа само един запис (alias), в който са записани личния ключ на потребителя и сертификационната верига на неговия сертификат. В частност тази сертификационна верига може да се състои само от един сертификат – сертификата на потребителя. Очаква се паролата за достъп до хранилището да съвпада са паролата за достъп до личния ключ в него.

За да работи правилно аплета, е необходимо той да бъде подписан. За подписването му може да се използва сертификат, издаден от някоя сертифицираща организация (PFX файл) или собственоръчно-подписан сертификат. Можем да използваме следния скрипт за да си генерираме собственоръчно-подписан сертификат:

generate-certificate.bat

```
del DigitalSignerApplet.jks
keytool -genkey -alias signFiles -keystore DigitalSignerApplet.jks -keypass !secret -dname
"CN=Your Company" -storepass !secret
pause
```

Резултатът е файла `DigitalSignerApplet.jks`, който представлява хранилище за ключове и сертификати, съдържащо генерирания сертификат и съответния му личен ключ, записани под име “`signFiles`”, достъпни с парола “`!secret`”. Форматът на изходния файл не е PFX, а JKS (Java KeyStore), който се използва по подразбиране от програмката `keytool`.

За компилирането на сорс-кода на аплета, получаването на JAR архив и подписването на този архив можем да използваме следния скрипт:

build-script.bat

```
del *.class
javac -classpath .;"%JAVA_HOME%\jre\lib\jaws.jar" *.java

del *.jar
jar -cvf DigitalSignerApplet.jar *.class

jarsigner -keystore DigitalSignerApplet.jks -storepass !secret -keypass !secret
DigitalSignerApplet.jar signFiles

pause
```

Посочената последователност от команди изтрива всички компилирани `.class` файлове, компилира всички `.java` файлове, които съставят аплета, пакетира получените `.class` файлове в JAR архив и подписва този архив с генерирания преди това собственоръчно-подписан сертификат (намиращ се в хранилището `DigitalSignerApplet.jks`). За компилирането на сорс-файловете на аплета е необходим JDK 1.4 или по-нова версия. Очаква се променливата на средата JAVA_HOME да има стойност, която отговаря на пълния път до директорията, в която е инсталиран JDK, а също в текущия път да е присъства bin директорията на JDK инсталацията.

Подписаният аплет можем да тестваме с примерен HTML документ, който съдържа подходяща HTML форма:

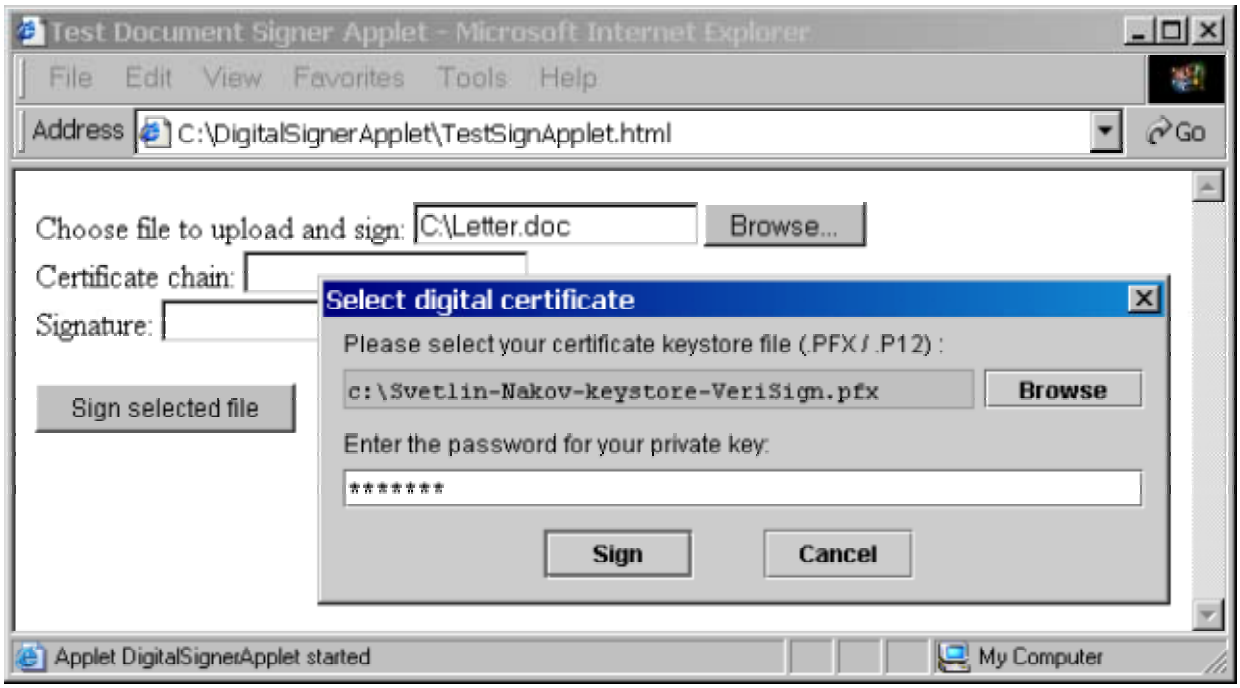
TestSignApplet.html

```
<html>

<head>
    <title>Test Document Signer Applet</title>
</head>

<body>
    <form name="mainForm" method="post" action="FileUploadServlet">
        Choose file to upload and sign:
        <input type="file" name="fileToBeSigned">
        <br>
        Certificate chain:
        <input type="text" name="certificateChain">
        <br>
        Signature:
        <input type="text" name="signature">
    </form>

    <object
        classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
        codebase="http://java.sun.com/products/plugin/autodl/jinstall-1_4-windows-
i586.cab#Version=1,4,0,0"
        width="130" height="25" mayscript="true">
        <param name="type" value="application/x-java-applet;version=1.4">
        <param name="code" value="DigitalSignerApplet">
        <param name="archive" value="DigitalSignerApplet.jar">
        <param name="mayscript" value="true">
        <param name="scriptable" value="true">
        <param name="fileNameField" value="fileToBeSigned">
        <param name="certificateChainField" value="certificateChain">
        <param name="signatureField" value="signature">
        <param name="signButtonCaption" value="Sign selected file">
        <comment>
        <embed
            type="application/x-java-applet;version=1.4"
            pluginspage="http://java.sun.com/products/plugin/index.html#download"
            code="DigitalSignerApplet" archive="DigitalSignerApplet.jar"
            width="130" height="25"
            mayscript="true" scriptable="true"
            uploadForm = "mainForm"
            fileNameField="fileToBeSigned"
            certificateChainField="certificateChain"
            signatureField="signature"
            signButtonCaption="Sign selected file">
            <noembed>
                Document signing applet can not be started because
                Java Plugin 1.4 is not installed.
            </noembed>
        </embed>
        </comment>
    </object>

</body>

</html>
```

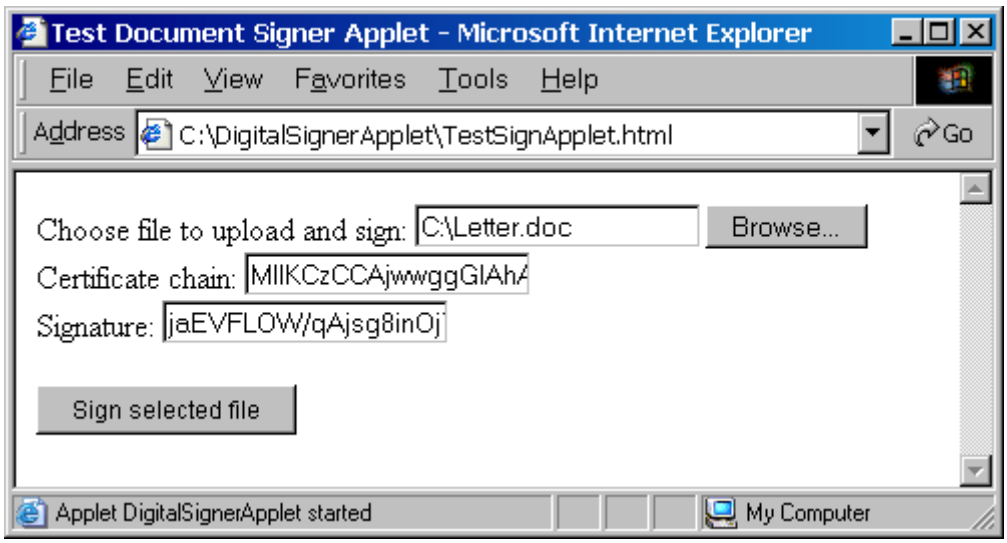
Важно е да отбележим, че не използваме остарелият таг `<applet>`, защото при него няма начин да укажем на Web-браузъра, че аплетът може да работи само с JDK версия 1.4 или по-висока. Ако използваме `<applet>` тага, нашият аплет ще стартира на всяко JDK, но няма винаги да работи правилно.

При отваряне на тестовият HTML документ от примера по-горе първо се проверява дали на машината има инсталиран Java Plug-In 1.4 или по-висока версия. Ако съответната версия на Java Plug-In не бъде намерена, потребителят се препраща към сайта, от който тя може да бъде изтеглена. Ако на машината е инсталиран Java Plug-In 1.4 и Web-браузърът на потребителя е правилно конфигуриран да го използва за изпълнение на Java-аплети, при зареждане на тестовия HTML документ се появява диалог, който пита дали да се позволи на подписания аплет, намиращ се в заредената HTML страница, да бъде изпълнен с пълни права. Ако потребителят се съгласи, аплетът стартира нормално.

Целта на тестовата HTML страница от примера по-горе е да демонстрира подписването на файлове на машината на клиента. Примерът представлява HTML форма, съдържаща три полета, аплет за подписване и бутон за активиране на подписването, който всъщност се намира вътре в аплета. Трите полета във формата служат съответно за избор на файл за изпращане, за записване на сертификационната верига, използвана при подписването и за записване на получената сигнатура. При натискане на бутона за подписване на потребителя се дава възможност да избере PFX файл и парола. След това аплетът подписва избрания от HTML формата файл с личния ключ от избрания PFX файл. В резултат от подписването изчислената сигнатура и сертификационната верига се записват в съответното поле от HTML формата. Ето как изглеждат HTML формата и диалога за избор на PFX файл от примера:



След успешно подписване на избрания файл HTML формата изглежда по следния начин:



Полетата за сертификационна верига и сигнатура са попълнени от аплета със стойности, кодирани текстово във формат Base64.

Като част от системата за подписване на файлове NakovDocumentSigner е разработено и примерно Java-базирано Web-приложение, което демонстрира в пълнота нейните възможности. Приложението дава възможност на потребителя да подписва и изпраща файлове, след което проверява цифровия подпис и сертификата на изпращача. Реализирана е проверка на цифровия подпис на получения файл, директна проверка на получения сертификат и проверка на получената сертификационна верига за случаите, в които такава е налична. Проверката на цифровия подпис установява дали изпратената сигнатура съответства на изпратения файл и изпратения сертификат. Директната проверка на сертификата установява дали на този сертификат може да се има доверие, без да се проверява сертифика-

ционната му верига. Проверката на сертификационната верига на сертификата на изпращача, когато е налична, има за цел да потвърди неговата самоличност.

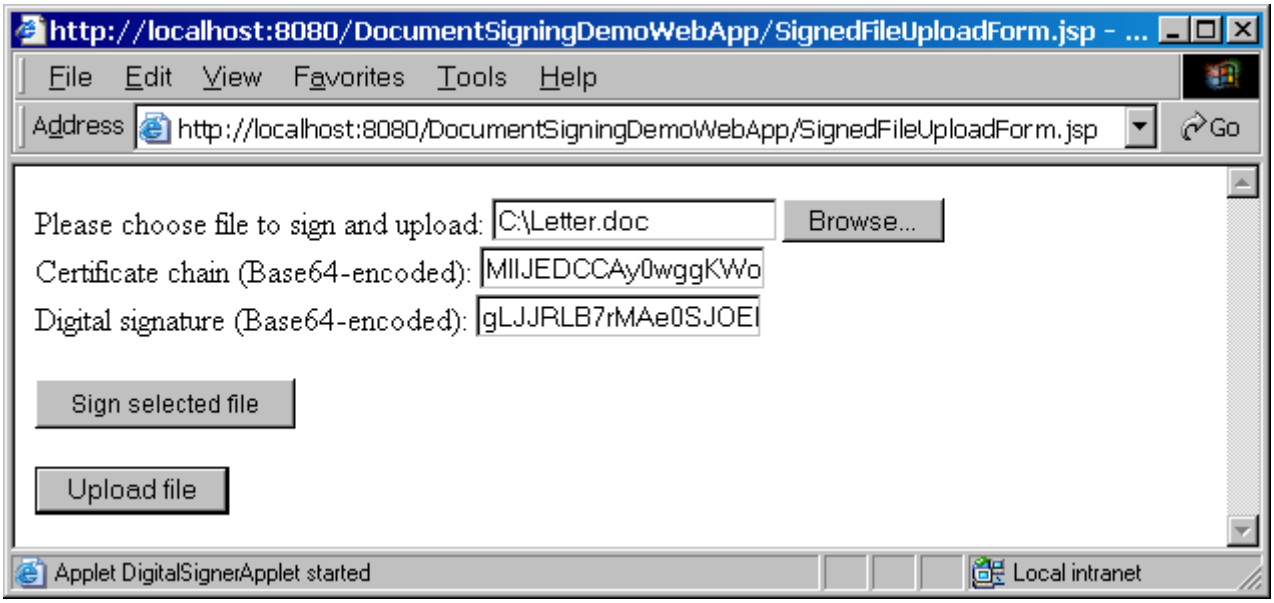
Проверката на получената сигнатура се извършва по най-стандартния начин с използване на класа `java.security.Signature`.

За директната проверка на сертификат примерното Web-приложение поддържа множество от сертификати, на които има доверие и проверява дали сертификатът на потребителя е подписан с някой от тях. Доверените сертификати се записват като .CER файлове в специална директория на приложението, името на която се указва от константа в страницата за проверка на потребителския сертификат.

За проверката на сертификационната верига на потребителски сертификат в примерното приложение се използва множество от доверени Root-сертификати на сертифициращи организации от първо ниво. От това множество се построява съвкупността от крайни точки на доверие (trust anchors), която е необходима за проверката на сертификационната верига. Доверените Root-сертификати се записват като .CER файлове в специална директория на приложението, името на която се указва от константа в страницата за проверка на сертификационната верига на потребителския сертификат.

Примерното Web-приложение е базирано на платформата J2EE (Java 2 Enterprise Edition) и широко-разпространения framework за Web-приложения Struts. Приложението се състои една Struts-базирана HTML форма, Struts action клас за обработка на тази форма, Struts action form клас за съхранение на данните от формата, клас за проверка на цифрови подписи и сертификати, страница за обработка на изпратения подписан файл и няколко конфигурационни файла. Пълният сорс-код на приложението е достъпен от сайта на NakovDocumentSigner – http://www.nakov.com/documents-signing/.

Началната форма на приложението се състои от три полета, съответно за името на файла, сертификата и сигнатурата и съдържа още аплет за подписване и бутон за изпращане:



Тази форма представлява JSP страница, базирана на библиотеката от тагове “struts-html” и използва Struts-тага за изпращане на файл `<html:file>`:

SignedFileUploadForm.jsp

```
<%
/**
 * A JSP that contains the form for signing and uploading a file. The form contains 3 fields -
 * the file to be uploaded, the certificate chain and the digital signature. The form contains
 * the DigitalSignatureApplet that signs the file on the client's machine.
 *
 * This file is part of NakovDocumentSigner digital document
 * signing framework for Java-based Web applications:
 * http://www.nakov.com/documents-signing/
 *
 * Copyright (c) 2003 by Svetlin Nakov - http://www.nakov.com
 * All rights reserved. This code is freeware. It can be used
```

```
 * for any purpose as long as this copyright statement is not
 * removed or modified.
 */
%>

<%@ taglib uri="/WEB-INF/taglibs/struts-html.tld" prefix="html" %>

<html>
<body>

<html:form type="demo.SignedFileUploadActionForm" action="/SignedFileUpload"
        method="post" enctype="multipart/form-data">
    Please choose file to sign and upload: <html:file property="uploadFile"/> <br>
    Certificate chain (Base64-encoded): <html:text property="certificateChain" value=""/> <br>
    Digital signature (Base64-encoded): <html:text property="signature" value=""/> <br>

    <br>

    <object
        classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
        codebase="http://java.sun.com/products/plugin/autodl/jinstall-1_4-windows-
i586.cab#Version=1,4,0,0"
        width="130" height="25" mayscript="true">
        <param name="type" value="application/x-java-applet;version=1.4">
        <param name="code" value="DigitalSignerApplet">
        <param name="archive" value="DigitalSignerApplet.jar">
        <param name="mayscript" value="true">
        <param name="scriptable" value="true">
        <param name="fileNameField" value="uploadFile">
        <param name="certificateChainField" value="certificateChain">
        <param name="signatureField" value="signature">
        <param name="signButtonCaption" value="Sign selected file">
        <comment>
            <embed
                type="application/x-java-applet;version=1.4"
                pluginspage="http://java.sun.com/products/plugin/index.html#download"
                code="DigitalSignerApplet" archive="DigitalSignerApplet.jar"
                width="130" height="25"
                mayscript="true" scriptable="true"
                uploadForm = "mainForm"
                fileNameField="uploadFile"
                certificateChainField="certificateChain"
                signatureField="signature"
                signButtonCaption="Sign selected file">
            <noembed>
                Document signing applet can not be started because
                Java Plug-In version 1.4 or later is not installed.
            </noembed>
        </embed>
        </comment>
    </object>

    <br>
    <br>

    <html:submit property="submit" value="Upload file"/>
</html:form>

</body>
</html>
```

При преноса на данните от формата към сървъра се използва кодиране “multipart/form-data”, което позволява изпращане на файлове и друга обемиста информация. Аплетът за подписване се вгражда с таговете `<object>` и `<embed>`. Всъщност тага `<object>` се разпознава само от Internet Explorer, а `<embed>` от всички останали браузъри и за да работи нашето приложение на всички браузъри, двата тага се комбинират. При зареждане на страницата, ако на машината има инсталиран Java Plug-In 1.4 или по-нов, се появява предупреждението, че аплетът е подписан и иска да работи с пълни права, а ако необходимата версия на Java Plug-In не е налична, потребителският браузър се препраща към място от където тя може да бъде изтеглена. Когато потребителят позволи на заредения аплет да се изпълни с пълни права, той се стартира и вече може да бъде използван за подписването на файлове.

Зад Struts-формата за изпращане на подписан файл стои съответен Struts action form клас, който се използва за съхранение на данните от нея:

SignedFileUploadActionForm.java

```
package demo;

import org.apache.struts.action.ActionForm;
import org.apache.struts.upload.FormFile;

/**
 * Struts action form class that maps to the form for uploading signed files
 * (SignedFileUploadForm.jsp). It is actually a data structure that consist of
 * the uploaded file, the sender's certificate chain and the digital signature
 * of the uploaded file.
 *
 * This file is part of NakovDocumentSigner digital document
 * signing framework for Java-based Web applications:
 * http://www.nakov.com/documents-signing/
 *
 * Copyright (c) 2003 by Svetlin Nakov - http://www.nakov.com
 * All rights reserved. This code is freeware. It can be used
 * for any purpose as long as this copyright statement is not
 * removed or modified.
 */
public class SignedFileUploadActionForm extends ActionForm {

    private FormFile mUploadFile;
    private String mCertificateChain;
    private String mSignature;

    public FormFile getUploadFile() {
        return mUploadFile;
    }

    public void setUploadFile(FormFile aUploadFile) {
        mUploadFile = aUploadFile;
    }

    public String getCertificateChain() {
        return mCertificateChain;
    }

    public void setCertificateChain(String aCertificateChain) {
        mCertificateChain = aCertificateChain;
    }

    public String getSignature() {
        return mSignature;
    }

    public void setSignature(String aSignature) {
        mSignature = aSignature;
    }

}
```

Този клас не представлява нищо повече от прост Java bean, в който са дефинирани свойства, точно съответстващи на полетата от формата. Когато потребителят попълни полетата на формата и я изпрати, Struts framework автоматично създава обект от този клас и прехвърля получените от формата данни в свойствата на този обект.

Получените данни, записани в action form обекта се обработват от събитието `/SignedFileUpload`, което съответства на Struts action класа:

SignedFileUploadAction.java

```
package demo;

import javax.servlet.http.*;
import org.apache.struts.action.*;
```

```
/**
 * Struts action class for handling the results of submitting SignedFileUploadForm.jsp form.
 * It gets the data from the form as SignedFileUploadActionForm object and puts this object in
 * the current user's session with key "signedFileUploadActionForm". After that this struts
 * action redirects the user's Web browser to ShowSignedFileUploadResults.jsp that is used to
 * display the received file, certificate, certificate chain and digital signature.
 *
 * This file is part of NakovDocumentSigner digital document
 * signing framework for Java-based Web applications:
 * http://www.nakov.com/documents-signing/
 *
 * Copyright (c) 2003 by Svetlin Nakov - http://www.nakov.com
 * All rights reserved. This code is freeware. It can be used
 * for any purpose as long as this copyright statement is not
 * removed or modified.
 */
public class SignedFileUploadAction extends Action {

    public ActionForward perform(ActionMapping aActionMapping, ActionForm aActionForm,
            HttpServletRequest aRequest, HttpServletResponse aResponse) {
        SignedFileUploadActionForm signedFileUploadActionForm =
            (SignedFileUploadActionForm) aActionForm;
        HttpSession session = aRequest.getSession();
        session.setAttribute("signedFileUploadActionForm", signedFileUploadActionForm);

        return aActionMapping.findForward("ShowSignedFileUploadResults");
    }

}
```

Това събитие записва получения action form клас с данните, пристигнали от клиентския Web-браузър, в потребителската сесия под ключ “signedFileUploadActionForm” и след това пренасочва Web-приложението към страницата за анализ на получения подписан файл:

ShowSignedFileUploadResults.jsp

```
<%
/**
 * A JSP for verifying digital signature, certificate and certificate chain of received
 * signed file. It assumes that the data, received by submitting SignedFileUploadForm.jsp
 * stays in the user's session in SignedFileUploadActionForm object stored with key
 * "signedFileUploadActionForm".
 *
 * The trusted certificates used for direct certificate verification should be located
 * in a directory whose name stays in the CERTS_FOR_DIRECT_VALIDATION_DIR constant
 * (see the code below).
 *
 * The trusted root CA certificates used for certificate chain verification should be
 * located in a directory whose name stays in the TRUSTED_CA_ROOT_CERTS_DIR constant
 * (see the code below).
 *
 * This file is part of NakovDocumentSigner digital document
 * signing framework for Java-based Web applications:
 * http://www.nakov.com/documents-signing/
 *
 * Copyright (c) 2003 by Svetlin Nakov - http://www.nakov.com
 * All rights reserved. This code is freeware. It can be used
 * for any purpose as long as this copyright statement is not
 * removed or modified.
 */
%>

<%@page import="demo.*,
                java.io.*,
                java.util.*,
                java.security.*,
                java.security.cert.*,
                org.apache.struts.upload.FormFile,
                javax.servlet.jsp.JspWriter" %>

<%!
    public static final String CERTS_FOR_DIRECT_VALIDATION_DIR =
```

```
        "/WEB-INF/certs-for-direct-validation";
    public static final String TRUSTED_CA_ROOT_CERTS_DIR =
        "/WEB-INF/trusted-CA-root-certs";

    private static final int KEY_USAGE_DIGITAL_SIGNATURE = 0;
    private static final int KEY_USAGE_NON_REPUDIATION = 1;
    private static final int KEY_USAGE_KEY_ENCIPHERMENT = 2;
    private static final int KEY_USAGE_DATA_ENCIPHERMENT = 3;
    private static final int KEY_USAGE_KEY_AGREEMENT = 4;
    private static final int KEY_USAGE_CERT_SIGN = 5;
    private static final int KEY_USAGE_CRL_SIGN = 6;
    private static final int KEY_USAGE_ENCIPHER_ONLY = 7;
    private static final int KEY_USAGE_DECIPHER_ONLY = 8;

    private ServletContext mApplicationContext = null;
    private JspWriter mOut = null;
    private SignedFileUploadActionForm mActionForm = null;
    private FormFile mReceivedFile = null;
    private byte[] mReceivedFileData = null;
    private CertPath mCertPath = null;
    private X509Certificate[] mCertChain = null;
    private X509Certificate mCertificate = null;
    private byte[] mSignature = null;
    private String mSignatureBase64Encoded;
%>

<html>
<body>
    <%
        mApplicationContext = application;
        mOut = out;
        mActionForm = (SignedFileUploadActionForm)
            session.getAttribute("signedFileUploadActionForm");

        if (mActionForm == null) {
            // User session does not contain the SignedFileUploadActionForm object
    %>
            Please choose file for signing and uploading from
            <a href="SignedFileUploadForm.jsp">SignedFileUploadForm.jsp</a>
    <%
        } else {
            try {
                // Analyse received signed file and display information about it
                processReceivedFile();
                displayFileInfo(mReceivedFile, mReceivedFileData);
                mOut.println("<hr>");

                // Analyse received certificate chain
                processReceivedCertificateChain();

                // Analyse received digital signature
                processReceivedSignature();

                // Display signature, veirfy it and display the verification results
                displaySignature(mSignatureBase64Encoded);
                verifyReceivedSignature();
                mOut.println("<hr>");

                // Display certificate, verify it and display the verification results
                displayCertificate(mCertificate);
                verifyReceivedCertificate();
                mOut.println("<hr>");

                // Display certificate chain, verify it and display the verification results
                displayCertificateChain(mCertChain);
                verifyReceivedCertificateChain();
            } catch (Exception e) {
                // Error occurred. Display the exception with its full stack trace
                out.println("<pre>Error: ");
                e.printStackTrace(new PrintWriter(out));
                out.println("</pre>");
            }
    %>
            <br> <a href="SignedFileUploadForm.jsp"> Back to SignedFileUploadForm.jsp </a>
```

```
    <%
        }
    %>
</body>
</html>

<%!
    /**
     * Extracts the received file and its data from the received HTML form data. The
     * extracted file and its content are stored in the member variables mReceivedFile
     * and mReceivedFileData.
     * @throws Exception if no file is received.
     */
    private void processReceivedFile()
    throws Exception {
        mReceivedFile = mActionForm.getUploadFile();
        if (mReceivedFile == null) {
            throw new Exception("No file received. Please upload some file.");
        }
        mReceivedFileData = mReceivedFile.getFileData();
    }

    /**
     * Displays information about give file. Displays its file name and file size.
     */
    private void displayFileInfo(FormFile aFile, byte[] aFileData)
    throws Exception {
        String fileName = aFile.getFileName();
        mOut.println("Signed file successfully uploaded. <br>");
        mOut.println("File name: " + fileName + " <br>");
        mOut.println("File size: " + aFileData.length + " bytes. <br>");
    }

    /**
     * Analyses received certificate chain and extracts the certificates that construct it.
     * The certificate chain should be PkiPath-encoded (ASN.1 DER formatted), stored as
     * Base64-string. The extracted chain is stored in the member variables mCertPath as
     * certPath and in mCertChain as array of X.509 certificates. Also the certificate used
     * for signing the received file is extracted in the member variable mCertificate.
     * @throws Exception if received certificate chain can not be decoded (i.e. its
     * encoding or internal format is invalid).
     */
    private void processReceivedCertificateChain()
    throws Exception {
        String certChainBase64Encoded = mActionForm.getCertificateChain();
        try {
            mCertPath = DigitalSignatureUtils.loadCertPathFromBase64String(
                certChainBase64Encoded);
            List certsInChain = mCertPath.getCertificates();
            mCertChain = (X509Certificate[]) certsInChain.toArray(new X509Certificate[0]);
            mCertificate = mCertChain[0];
        }
        catch (Exception e) {
            throw new Exception("Invalid certificate chain received.", e);
        }
    }

    /**
     * Displays given certificate chain. Displays the length of the chain and the
     * subject distinguished names of all certificates in the chain, starting from
     * the first and finishing to the last.
     */
    private void displayCertificateChain(X509Certificate[] aCertChain)
    throws IOException {
        mOut.println("Certificate chain length: " + aCertChain.length + " <br>");
        for (int i=0; i<aCertChain.length; i++) {
            Principal certPrincipal = aCertChain[i].getSubjectDN();
            mOut.println("certChain[" + i + "] = " + certPrincipal + " <br>");
        }
    }

    /**
     * Analyses received Base64-encoded digital signature, decodes it and stores it
     * in the member variable mSignature.
```

```
     * @throws Exception if the received signature can not be decoded.
     */
    private void processReceivedSignature()
    throws Exception {
        mSignatureBase64Encoded = mActionForm.getSignature();
        try {
            mSignature = Base64Utils.base64Decode(mSignatureBase64Encoded);
        } catch (Exception e) {
            throw new Exception("Invalid signature received.", e);
        }
    }

    /**
     * Displays given Base64-encoded digital signature.
     */
    private void displaySignature(String aSignatureBase64Encoded)
    throws IOException {
        mOut.println("Digital signature (Base64-encoded): " + aSignatureBase64Encoded);
    }

    /**
     * Verifies received signature using the received file data and certificate and
     * displays the verification results. The received document, certificate and
     * signature are taken from the member variables mReceivedFileData, mCertificate
     * and mSignature respectively.
     */
    private void verifyReceivedSignature()
    throws IOException {
        mOut.println("Digital signature status: <b>");
        try {
            boolean signatureValid = DigitalSignatureUtils.verifyDocumentSignature(
                mReceivedFileData, mCertificate, mSignature);
            if (signatureValid)
                mOut.println("Signature is verified to be VALID.");
            else
                mOut.println("Signature is INVALID!");
        } catch (Exception e) {
            e.printStackTrace();
            mOut.println("Signature verification failed due to exception: " + e.toString());
        }
        mOut.println("</b>");
    }

    /**
     * Displays information about given certificate. This information includes the
     * certificate subject distinguished name and its purposes (public key usages).
     */
    private void displayCertificate(X509Certificate aCertificate)
    throws IOException {
        String certificateSubject = aCertificate.getSubjectDN().toString();
        mOut.println("Certificate subject: " + certificateSubject + " <br>");

        boolean[] certKeyUsage = aCertificate.getKeyUsage();
        mOut.println("Certificate purposes (public key usages): <br>");
        if (certKeyUsage != null) {
            if (certKeyUsage[KEY_USAGE_DIGITAL_SIGNATURE])
                mOut.println("[digitalSignature] - verify digital signatures <br>");
            if (certKeyUsage[KEY_USAGE_NON_REPUDIATION])
                mOut.println("[nonRepudiation] - verify non-repudiation signatures <br>");
            if (certKeyUsage[KEY_USAGE_KEY_ENCIPHERMENT])
                mOut.println("[keyEncipherment] - encipher crypto keys for transport <br>");
            if (certKeyUsage[KEY_USAGE_DATA_ENCIPHERMENT])
                mOut.println("[dataEncipherment] - encipher user data, but not keys <br>");
            if (certKeyUsage[KEY_USAGE_KEY_AGREEMENT])
                mOut.println("[keyAgreement] - use for key agreement <br>");
            if (certKeyUsage[KEY_USAGE_CERT_SIGN])
                mOut.println("[keyCertSign] - verify signatures on certificates <br>");
            if (certKeyUsage[KEY_USAGE_CRL_SIGN])
                mOut.println("[cRLSign] - verify signatures on CRLs <br>");
            if (certKeyUsage[KEY_USAGE_ENCIPHER_ONLY])
                mOut.println("[encipherOnly] - encipher data during the key agreement <br>");
            if (certKeyUsage[KEY_USAGE_DECIPHER_ONLY])
                mOut.println("[decipherOnly] - decipher data during the key agreement <br>");
        } else {
```

```
            mOut.println("[No purposes defined] <br>");
        }
    }

    /**
     * Verifies received certificate directly and displays the verification results. The
     * certificate for verification is taken form mCertificate member variable. Trusted
     * certificates are taken from the CERTS_FOR_DIRECT_VALIDATION_DIR directory. This
     * directory should be relative to the Web application root directory and should
     * contain only .CER files (DER-encoded X.509 certificates).
     */
    private void verifyReceivedCertificate()
    throws IOException, GeneralSecurityException {
        // Create the list of the trusted certificates for direct validation
        X509Certificate[] trustedCertificates =
            getCertificateList(mApplicationContext, CERTS_FOR_DIRECT_VALIDATION_DIR);

        // Verify the certificate and display the verification results
        mOut.println("Certificate direct verification status: <b>");
        try {
            DigitalSignatureUtils.verifyCertificate(mCertificate, trustedCertificates);
            mOut.println("Certificate is verified to be VALID.");
        } catch (CertificateExpiredException cee) {
            mOut.println("Certificate is INVALID (validity period expired)!");
        } catch (CertificateNotYetValidException cnyve) {
            mOut.println("Certificate is INVALID (validity period not yet started)!");
        } catch (DigitalSignatureUtils.CertificateValidationException cve) {
            mOut.println("Certificate is INVALID! " + cve.getMessage());
        }
        mOut.println("</b>");
    }

    /**
     * Verifies received certificate chain and displays the verification results.
     * The chain for verification is taken form mCertPath member variable. Trusted CA
     * root certificates are taken from the TRUSTED_CA_ROOT_CERTS_DIR directory. This
     * directory should be relative to the Web application root directory and should
     * contain only .CER files (DER-encoded X.509 certificates).
     */
    private void verifyReceivedCertificateChain()
    throws IOException, GeneralSecurityException {
        // Create the most trusted CA set of trust anchors
        X509Certificate[] trustedCACerts =
            getCertificateList(mApplicationContext, TRUSTED_CA_ROOT_CERTS_DIR);

        // Verify the certificate chain and display the verification results
        mOut.println("Certificate chain verification: <b>");
        try {
            DigitalSignatureUtils.verifyCertificateChain(mCertPath, trustedCACerts);
            mOut.println("Certificate chain verified to be VALID.");
        } catch (CertPathValidatorException cpve) {
            mOut.println("Certificate chain is INVALID! Validation failed on certificate " +
                "[" + cpve.getIndex() + "] from the chain: " + cpve.toString());
        }
        mOut.println("</b> <br>");
    }

    /**
     * @return a list of X509 certificates, obtained by reading all files from the given
     * directory. The supplied directory should be a given as a relative path from the
     * Web appication root (e.g. "/WEB-INF/test") and should contain only .CER files
     * (DER-encoded X.509 certificates).
     */
    private X509Certificate[] getCertificateList(ServletContext aServletContext,
        String aCertificatesDirectory)
    throws IOException, GeneralSecurityException {
        // Get a list of all files in the given directory
        Set trustedCertsResourceNames =
            aServletContext.getResourcePaths(aCertificatesDirectory);

        // Allocate an array for storing the certificates
        int count = trustedCertsResourceNames.size();
        X509Certificate[] trustedCertificates = new X509Certificate[count];
```

```
        // Read all X.509 certificate files one by one into an array
        int index = 0;
        Iterator trustedCertsResourceNamesIterator = trustedCertsResourceNames.iterator();
        while (trustedCertsResourceNamesIterator.hasNext()) {
            String certResourceName = (String) trustedCertsResourceNamesIterator.next();
            InputStream certStream = aServletContext.getResourceAsStream(certResourceName);
            X509Certificate trustedCertificate =
                DigitalSignatureUtils.loadX509CertificateFromStream(certStream);
            certStream.close();
            trustedCertificates[index] = trustedCertificate;
            index++;
        }

        return trustedCertificates;
    }

%>
```

Страницата за анализ на подписан файл взима изпратените от потребителя данни от сесията, верифицира цифровия подпис, сертификата и сертификационната верига и показва информация за тях. Тези действия се извършват на няколко стъпки.

Първоначално се проверява дали в сесията е записан action form обекта, който съдържа изпратените от потребителя данни. Ако такъв обект няма, това означава, че данни не са получени и потребителят се препраща към страницата за подписване и изпращане на файл. Ако action form обекта е намерен, от него се взимат името на файла и дължината му и се отпечатват.

След това се декодира получената от клиента сертификационна верига. Понеже тя е била кодирана в текстов вид за да може да бъде пренесена през поле от HTML формата, първо се получава оригиналният й бинарен вид чрез Base64 декодиране. След това от този бинарен вид се възстановява оригиналната последователност от X.509 сертификати като се има предвид, че кодирането е било извършено във формат `PkiPath`. От декодираната сертификационна верига се изважда сертификата на потребителя, използван при подписването. Този сертификат е първият сертификат от тази верига.

Следва декодиране на получената цифрова сигнатура за да се възстанови оригиналният й бинарен вид, след което се проверява дали тя е валидна. От сертификата на клиента се получава публичният му ключ и с него се проверява дали получената сигнатура съответства на получения документ. На потребителя се отпечатва сигнатурата, в текстов вид, както е получена, заедно с резултата от нейната проверка.

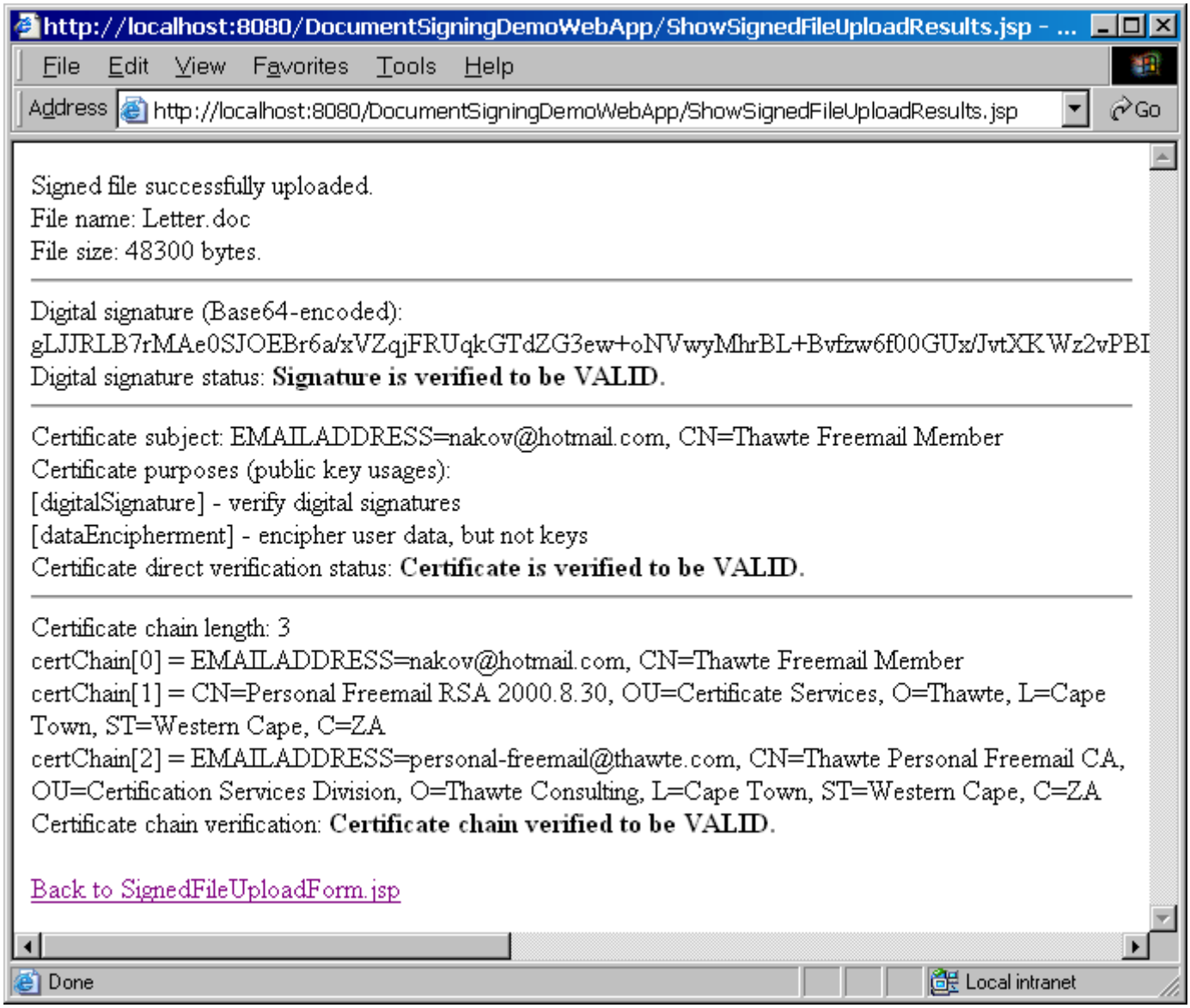
След като се установи, че получената сигнатура е истинска, се преминава към проверка на получения сертификат. Първоначално се отпечатва информация за сертификата – кой е негов собственик и за какво е предназначен да бъде използван. След това този сертификат се проверява дали е валиден. Проверката се извършва директно, без да се използва сертификационната му верига. Това може да бъде особено полезно в случаите, когато сертификационна верига липсва. Директната проверка използва съвкупност от доверени сертификати, която се прочита от специална директория, името на която се взима от константа в страницата. Тази директория трябва да е поддиректория на Web-приложението и трябва да съдържа само файлове със сертификати (.CER файлове). Ако се установи, че сертификатът на потребителя е директно подписан от някой от тези сертификати, той се счита за валиден. На потребителя се отпечатва резултатът от директната верификация.

Следва проверка на получената сертификационна верига. Преди да започне самата проверка веригата се отпечатва на потребителя във вид на последователност от сертификати, всеки на отделен ред. За всеки сертификат се отпечатва поредният му номер във веригата и информация за собственика му. Проверката започва със зареждане на сертификатите, които ще бъдат използвани като крайни точки на доверие. Тези доверени Root-сертификати стоят във вид на .CER файлове в специална директория на Web-приложението, името на която се указва от константа. Проверката използва средствата на Java Certification Path API и установява дали подадената верига е валидна. Ако веригата се състои от само един сертификат (верига няма), тя се счита за невалидна. Ако веригата се състои от повече от един сертификат, от нея се премахва последния сертификат и се изпълнява `PKIX` алгоритъма. Резултатът от извършената верификация се отпечатва на потребителя. Ако се установи, че веригата е невалидна, на потребителя се отпечатва причината за това и поредният номер на сертификата, при проверката на който е възникнал проблема.

За четенето на всички файлове от дадена директория се използват методите `getResourcePaths()` и `getResourceAsStream()` на класа `ServletContext`.

На всяка една стъпка от обработката на получения подписан файл ако възникне проблемна ситуация, при която приложението не може да продължи работата си, на потребителя се отпечатва съобщение за грешка, придружено от пълния вид на полученото изключение. Възможни са различни проблеми ситуации като например липса на файл, липса на сертификационна верига, липса на сигнатура, невалидно кодиране на получената сертификационна верига или сигнатура, липса на някоя от директориите с доверени сертификати, невалиден формат на сертификат и други.

Резултатът от всички описани проверки, които се извършват при получаване на подписан файл, изглежда по следния начин:



Основната функционалност по верификацията на сигнатури, сертификати и сертификационни вериги се намира в класа:

DigitalSignatureUtils.java

```
package demo;

import java.security.PublicKey;
import java.security.Signature;
import java.security.GeneralSecurityException;
import java.security.cert.*;
import java.io.*;
import java.util.List;
import java.util.HashSet;
```

```
/**
 * Utility class for digital signatures and certificates verification.
 *
 * Verification of digital signature aims to confirm or deny that given signature is
 * created by signing given document with the private key corresponding to given
 * certificate. Verification of signatures is done with the standard digital signature
 * verification algorithm, provided by Java Cryptography API:
 *    1. The message digest is calculated from given document.
 *    2. The original message digest is obtained by decrypting the signature with the
 * public key of the signer (this public key is taken from the signer's certificate).
 *    3. Values calculated in step 1. and step 2. are compared.
 *
 * Verification of a certificate aims to check if the certificate is valid wihtout
 * inspecting its certificate chain (sometimes it is unavailable). The certificate
 * verification is done in two steps:
 *    1. The certificate validity period is checked against current date.
 *    2. The certificate is checked if it is directly signed by some of the trusted
 * certificates that we have. A list of trusted certificates is supported for this
 * direct certificate verification process. If we want to successfully validate the
 * certificates issued by some certification authority (CA), we need to add the
 * certificate of this CA in our trusted list. Note that some CA have several
 * certificates and we should add only that of them, which the CA directly uses for
 * issuing certificates to its clients.
 *
 * Verification of a certificate chains aims to check if given certificate is valid
 * by analysing its certificate chain. A certificate chain always starts with the user
 * certificate that should be verified, then several intermediate CA certificates
 * follow and at the end of the chain stays some root CA certificate. The verification
 * process includes following steps (according to PKIX algorithm):
 *    1. Check the certificate validity period against current date.
 *    2. Check if each certificate in the chain is signed by the previous.
 *    3. Check if all the certificates in the chain, except the first, belong to some
 * CA, i.e. if they are authorized to be used for signing other certificates.
 *    4. Check if the root CA certificate in the end of the chain is trusted, i.e. if
 * is it in the list of trusted root CA certificates.
 * The verification process uses PKIX algorithm, defined in RFC-3280, but don't use
 * CRL lists.
 *
 * This file is part of NakovDocumentSigner digital document
 * signing framework for Java-based Web applications:
 * http://www.nakov.com/documents-signing/
 *
 * Copyright (c) 2003 by Svetlin Nakov - http://www.nakov.com
 * All rights reserved. This code is freeware. It can be used
 * for any purpose as long as this copyright statement is not
 * removed or modified.
 */
public class DigitalSignatureUtils {

    private static final String X509_CERTIFICATE_TYPE = "X.509";
    private static final String CERTIFICATE_CHAIN_ENCODING = "PkiPath";
    private static final String DIGITAL_SIGNATURE_ALGORITHM_NAME = "SHA1withRSA";
    private static final String CERT_CHAIN_VALIDATION_ALGORITHM = "PKIX";

    /**
     * Loads X.509 certificate from DER-encoded binary stream.
     */
    public static X509Certificate loadX509CertificateFromStream(InputStream aCertStream)
    throws GeneralSecurityException {
        CertificateFactory cf = CertificateFactory.getInstance(X509_CERTIFICATE_TYPE);
        X509Certificate cert = (X509Certificate)cf.generateCertificate(aCertStream);
        return cert;
    }

    /**
     * Loads X.509 certificate from DER-encoded binary file (.CER file).
     */
    public static X509Certificate loadX509CertificateFromCERFile(String aFileName)
    throws GeneralSecurityException, IOException {
        FileInputStream fis = new FileInputStream(aFileName);
        X509Certificate cert = null;
        try {
            cert = loadX509CertificateFromStream(fis);
        } finally {
```

```
            fis.close();
        }
        return cert;
    }

    /**
     * Loads a certificate chain from given Base64-encoded string, containing
     * ASN.1 DER formatted chain, stored with PkiPath encoding.
     */
    public static CertPath loadCertPathFromBase64String(String aCertChainBase64Encoded)
    throws CertificateException, IOException {
        byte[] certChainEncoded = Base64Utils.base64Decode(aCertChainBase64Encoded);
        CertificateFactory cf = CertificateFactory.getInstance(X509_CERTIFICATE_TYPE);
        ByteArrayInputStream certChainStream = new ByteArrayInputStream(certChainEncoded);
        CertPath certPath;
        try {
            certPath = cf.generateCertPath(certChainStream, CERTIFICATE_CHAIN_ENCODING);
        } finally {
            certChainStream.close();
        }
        return certPath;
    }

    /**
     * Verifies given digital singature. Checks if given signature is obtained by
     * signing given document with the private key, corresponing to given public key.
     */
    public static boolean verifyDocumentSignature(byte[] aDocument,
        PublicKey aPublicKey, byte[] aSignature)
    throws GeneralSecurityException {
        Signature signatureAlgorithm = Signature.getInstance(DIGITAL_SIGNATURE_ALGORITHM_NAME);
        signatureAlgorithm.initVerify(aPublicKey);
        signatureAlgorithm.update(aDocument);
        boolean valid = signatureAlgorithm.verify(aSignature);
        return valid;
    }

    /**
     * Verifies given digital singature. Checks if given signature is obtained by
     * signing given document with the private key, corresponing to given certificate.
     */
    public static boolean verifyDocumentSignature(byte[] aDocument,
        X509Certificate aCertificate, byte[] aSignature)
    throws GeneralSecurityException {
        PublicKey publicKey = aCertificate.getPublicKey();
        boolean valid = verifyDocumentSignature(aDocument, publicKey, aSignature);
        return valid;
    }

    /**
     * Verifies a certificate. Checks its validity period and tries to find a trusted
     * certificate from given list of trusted certificates that is directly signed given
     * certificate. The certificate is valid if no exception is thrown.
     *
     * @param aCertificate the certificate to be verified.
     * @param aTrustedCertificates a list of trusted certificates to be used in
     * the verification process.
     *
     * @throws CertificateExpiredException if the certificate validity period is expired.
     * @throws CertificateNotYetValidException if the certificate validity period is not
     * yet started.
     * @throws CertificateValidationException if the certificate is invalid (can not be
     * validated using the given set of trusted certificates.
     */
    public static void verifyCertificate(X509Certificate aCertificate,
        X509Certificate[] aTrustedCertificates)
    throws GeneralSecurityException {
        // First check certificate validity period
        aCertificate.checkValidity();

        // Check if the certificate is signed by some of the given trusted certificates
        for (int i=0; i<aTrustedCertificates.length; i++) {
            X509Certificate trustedCert = aTrustedCertificates[i];
            try {
```

```
                aCertificate.verify(trustedCert.getPublicKey());
                // Found parent certificate. Certificate is verified to be valid
                return;
            }
            catch (GeneralSecurityException ex) {
                // Certificate is not signed by current trustedCert. Try the next one
            }
        }

        // Certificate is not signed by any of the trusted certificates, so it is invalid
        throw new CertificateValidationException(
            "Can not find trusted parent certificate.");
    }

    /**
     * Verifies certificate chain using "PKIX" algorithm, defined in RFC-3280. It is
     * considered that the given certificate chain start with the target certificate
     * and finish with some root CA certificate. The certificate chain is valid if no
     * exception is thrown.
     *
     * @param aCertChain the certificate chain to be verified.
     * @param aTrustedCACertificates a list of most trusted root CA certificates.
     * @throws CertPathValidatorException if the certificate chain is invalid.
     */
    public static void verifyCertificateChain(CertPath aCertChain,
        X509Certificate[] aTrustedCACertificates)
    throws GeneralSecurityException {
        int chainLength = aCertChain.getCertificates().size();
        if (chainLength < 2) {
            throw new CertPathValidatorException("The certificate chain is too short. " +
                "It should consist of at least 2 certiicates.");
        }

        // Create a set of trust anchors from given trusted root CA certificates
        HashSet trustAnchors = new HashSet();
        for (int i = 0; i < aTrustedCACertificates.length; i++) {
            TrustAnchor trustAnchor = new TrustAnchor(aTrustedCACertificates[i], null);
            trustAnchors.add(trustAnchor);
        }

        // Create a certificate chain validator and a set of parameters for it
        PKIXParameters certPathValidatorParams = new PKIXParameters(trustAnchors);
        certPathValidatorParams.setRevocationEnabled(false);
        CertPathValidator chainValidator =
            CertPathValidator.getInstance(CERT_CHAIN_VALIDATION_ALGORITHM);

        // Remove the root CA certificate from the end of the chain. This is required by
        // the validation algorithm because by convention the trust anchor certificates
        // should not be a part of the chain that is validated
        CertPath certChainForValidation = removeLastCertFromCertChain(aCertChain);

        // Execute the certificate chain validation
        chainValidator.validate(certChainForValidation, certPathValidatorParams);
    }

    /**
     * Removes the last certificate from given certificate chain.
     * @return given cert chain without the last certificate in it.
     */
    private static CertPath removeLastCertFromCertChain(CertPath aCertChain)
    throws CertificateException {
        List certs = aCertChain.getCertificates();
        int certsCount = certs.size();
        List certsWithoutLast = certs.subList(0, certsCount-1);
        CertificateFactory cf = CertificateFactory.getInstance(X509_CERTIFICATE_TYPE);
        CertPath certChainWithoutLastCertificate = cf.generateCertPath(certsWithoutLast);
        return certChainWithoutLastCertificate;
    }

    /**
     * Exception class for certificate validation errors.
     */
    public static class CertificateValidationException extends GeneralSecurityException {
        public CertificateValidationException(String aMessage) {
```

```
            super(aMessage);
        }
    }

}
```

Класът започва с методи за зареждане на сертификат от поток и от файл, с които се прочитат файловете с доверените сертификати, използвани при проверката на сертификати и сертификационни вериги. Очаква се тези файлове да бъдат в стандартния .CER формат (ASN.1 DER-кодирани). Следва метод за зареждане на сертификационна верига, представена във формат `PkiPath` и кодирана в текстов вид с кодиране Base64. Класът предлага още функционалност за проверка на цифрови сигнатури, която използва алгоритъма `SHA1withRSA` – същият, който се използва от аплета за подписване. Предоставят се още методи за директна верификация на сертификат и верификация на сертификационни вериги. Методът за директна верификация на сертификат като параметър очаква сертификата и множество от доверени сертификати, сред които да търси издателя на проверявания сертификат. Верификацията е успешна, ако методът завърши изпълнението си без да генерира изключение.

Методът за проверката на сертификационни вериги е малко по-сложен. Той приема като вход сертификационна верига и множество от доверени Root-сертификати. При проверката на подадената верига първоначално се проверява дали тя се състои от поне 2 сертификата. Верига от 1 потребителски сертификат не може да бъде валидна, защото тя трябва да завършва с Root-сертификата на някоя сертифицираща организация от първо ниво, а в същото време тя започва с потребителския сертификат. Проверката започва с построяване на множество от крайни точки на доверие (`TrustAnchor` обекти) от зададените доверени Root-сертификати. След това се създава обект, съдържащ множеството от параметри на алгоритъма за проверка. От тези параметри се указва на алгоритъма да не използва списъци от анулирани сертификати (CRLs). Използването на такива CRL списъци не се прилага, защото е сложно за реализация и изисква допълнителни усилия за извличане на тези списъци от сървърите на сертифициращите организации, които ги издават и разпространяват. След създаването на крайните точки на доверите и инициализирането на параметрите на алгоритъма за верификация има още една важна стъпка преди самата верификация. Алгоритъмът `PKIX`, който се използва за верификацията има една особеност. Той очаква да му се подаде сертификационна верига, която не завършва с Root-сертификат на някоя сертифицираща организация, а със сертификата, който стои непосредствено преди него, т.е. очаква от сертификационната верига да бъде отстранен последният сертификат. Ако последният сертификат от веригата не бъде отстранен преди проверката, е възможно валидни сертификационни вериги да бъдат приети за невалидни. Ако веригата не е валидна, се генерира изключение, в което се указва поредния номер на сертификата, в който е възникнал проблемът.

Както и при аплета, за работата с цифрови подписи, сертификати и сертификационни вериги се използват средствата на Java Cryptography Architecture, а реализацията на всички криптографски алгоритми се осигурява от доставчика на криптографски услуги по подразбиране `SunJSSE`, който е част от JDK 1.4.

За работа с Base64-кодирана информация в Web-приложението се използва същия клас `Base64Utils`, който се използва и при аплета за подписване на файлове.

Съгласно стандартите на платформата J2EE за работата на демонстрационното Web-приложение е необходим още конфигурационният файл:

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>

    <servlet>
        <servlet-name>action</servlet-name>
        <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
        <init-param>
            <param-name>config</param-name>
            <param-value>/WEB-INF/struts-config.xml</param-value>
        </init-param>
```

```
        <load-on-startup>2</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>action</servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>

    <welcome-file-list>
        <welcome-file>/SignedFileUploadForm.jsp</welcome-file>
    </welcome-file-list>

</web-app>
```

Този файл съдържа настройките на приложението.

За да се използва Struts framework е необходим и неговият конфигурационен файл:

struts-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD Struts Configuration 1.0//EN"
"http://jakarta.apache.org/struts/dtds/struts-config_1_0.dtd">

<struts-config>

    <form-beans>
        <form-bean name="SignedFileUploadActionForm" type="demo.SignedFileUploadActionForm" />
    </form-beans>

    <action-mappings>
        <action name="SignedFileUploadActionForm" type="demo.SignedFileUploadAction"
                input="/SignedFileUploadForm.jsp" scope="request" path="/SignedFileUpload">
            <forward name="ShowSignedFileUploadResults" path="/ShowSignedFileUploadResults.jsp"
                     redirect="true"/>
        </action>
    </action-mappings>

</struts-config>
```

Този файл конфигурира Struts-формите и Struts action-ите, които се използват от приложението, заедно с всички техни параметри.

Съвкупността от всички описани файлове съставя демонстрационното Web-приложение. За да го компилираме и подготвим за изпълнение можем да използваме следния Apache Ant скрипт:

build.xml

```
<?xml version="1.0" encoding="iso-8859-1"?>

<project name="DocumentSigningDemoWebApp" default="build" basedir=".">

    <target name="init">
        <property name="app-name" value="DocumentSigningDemoWebApp"/>
        <property name="webapp-name" value="${app-name}.war"/>
        <property name="src-dir" value="src"/>
        <property name="www-dir" value="wwwroot"/>
        <property name="classes-dir" value="${www-dir}/WEB-INF/classes"/>
        <property name="web-xml" value="${www-dir}/WEB-INF/web.xml"/>
        <property name="lib-dir" value="${www-dir}/WEB-INF/lib"/>
        <property name="deploy-dir" value="deploy"/>
    </target>

    <target name="clean" depends="init">
        <delete dir="${classes-dir}"/>
        <mkdir dir="${classes-dir}"/>
        <delete dir="${deploy-dir}"/>
        <mkdir dir="${deploy-dir}"/>
    </target>

    <target name="compile" depends="init">
        <javac srcdir="src"
```

```
            destdir="wwwroot/WEB-INF/classes"
            debug="on">
            <classpath>
                <fileset dir="${lib-dir}">
                    <include name="**/*.jar"/>
                    <include name="**/*.zip"/>
                </fileset>
            </classpath>
        </javac>
    </target>

    <target name="war" depends="init">
        <war compress="true" destfile="${deploy-dir}/${webapp-name}" webxml="${web-xml}" >
            <fileset dir="${www-dir}">
                <include name="**/*.*"/>
            </fileset>
        </war>
    </target>

    <target name="build">
        <antcall target="clean"/>
        <antcall target="compile"/>
        <antcall target="war"/>
    </target>

</project>
```

Резултатът от изпълнението на този скрипт е файлът `DocumentSigningDemoWebApp.war`, който съдържа компилираното приложение във вид готов за изпълнение, отговарящ на спецификациите за Web-приложения на платформата J2EE. За да го изпълним е необходимо да го deploy-нем на някой J2EE сървър или сървър за Java Web-приложения (Servlet container).

По време на тестовете на системата NakovDocumentSigner използвахме сървъра за Web-приложения Apache Tomcat версия 4.0 и Struts framework версия 1.0. От страна на клиента системата успешно тествахме с Java Plug-In 1.4 в средата на Web-браузърите Internet Explorer 5.0, Internet Explorer 6.0, Mozilla 1.3, Netscape Communicator 4.5 и Netscape 6.1, работещи върху операционни системи Windows 98, Windows 2000, Windows XP и Red Hat Linux 8.0 (с графична среда KDE 3.0).

Системата NakovDocumentSigner е работещ пример, който илюстрира един подход за използване на цифрови подписи от Java-базирани Web-приложения. Тя решава проблемите, които възникват при подписване на документи на машината на клиента и демонстрира как със средствата на Java платформата могат да се верифицират цифрови подписи, сертификати и сертификационни вериги. Системата е напълно безплатна и може да бъде използвана в чист или променен вид за всякакви цели, включително и в комерсиални продукти.

Нуждата от информационна сигурност при Web-приложенията непрекъснато нараства и това неизбежно води до развитие и усъвършенстване на свързаните с нея на технологии. Напълно е възможно в някои бъдещи версии на най-разпространените Web-браузъри да се появят вградени средства за подписване на документи и HTML форми, използващи за целта сертификатите, инсталирани в тези браузъри, но докато тези средства се появят и се утвърдят, NakovDocumentSigner вероятно ще си остане едно от малкото безплатни цялостни решения в това направление.

Използвана литература

1. Общи условия на публични сертификационни услуги на GlobalSign, раздел 13.1 (дефиниции) – http://www.bia-bg.com/globalsign.bg/cps_chapter13.htm
2. Работа с PGP – http://www.ecs.ru.acad.bg/rk2/StProj/4/pgp/overview.htm
3. Java Glossary – Certificates – http://mindprod.com/jgloss/certificate.html
4. Digital Signatures: How They Work – http://www.silicontrust.com/background/sp_digital-sig-2.asp
5. Java Cryptography Architecture (JCA) - API Specification & Reference – http://java.sun.com/products/jdk/1.2/docs/guide/security/CryptoSpec.html

6. Trail: Security in Java 2 SDK 1.2 - The Java Tutorial – http://java.sun.com/docs/books/tutorial/security1.2/index.html
7. Java Certification Path API Programmer's Guide – http://java.sun.com/j2se/1.4.2/docs/guide/security/certpath/CertPathProgGuide.html
8. Java security evolution and concepts, Part 5 - Java CertPath API – http://www.javaworld.com/javaworld/jw-12-2001/jw-1221-jdk4security-p2.html
9. Java 2 Platform, Standard Edition, v 1.4.2 - API Specification – http://java.sun.com/j2se/1.4.2/docs/api/
10. How to Sign Applets Using RSA-Signed Certificates – http://java.sun.com/j2se/1.4.1/docs/guide/plugin/developer_guide/rsa_signing.html
11. Java-to-Javascript Communication – http://java.sun.com/j2se/1.4.1/docs/guide/plugin/developer_guide/java_js.html
12. NakovDocumentSigner - Digital Document Signing Framework for Java-based Web Applications – http://www.nakov.com/documents-signing/
13. Struts Framework – http://jakarta.apache.org/struts/
14. Apache Ant – http://ant.apache.org/
15. Apache Tomcat – http://jakarta.apache.org/tomcat/