Светлин Наков

Java за цифрово подписване на документи в уеб

НАПС
НАЦИОНАЛНА АКАДЕМИЯ ПО РАЗРАБОТКА НА СОФТУЕР

Java за цифрово подписване на документи в уеб

Светлин Наков

Софийски университет „Св. Климент Охридски”
Българска асоциация на разработчиците на софтуер
Национална академия по разработка на софтуер

София, 2005

Java за цифрово подписване на документи в уеб

Web-site: www.nakov.com

Официален сайт:

www.nakov.com/books/signatures/

ISBN 954-775-504-8

НАПС
НАЦИОНАЛНА АКАДЕМИЯ ПО РАЗРАБОТКА НА СОФТУЕР

НАЦИОНАЛНА АКАДЕМИЯ ПО РАЗРАБОТКА НА СОФТУЕР

Лекторите

» Светлин Наков е преподавател по съвременни софтуерни технологии в СУ "Св. Климент Охридски".

Той е автор на десетки научни и технически публикации и няколко книги, свързани с разработката на софтуер, заради което е търсен лектор и консултант.

През 2004 г. получава наградата "**Джон Атанасов**" от президента на България Георги Първанов за приноса му към развитието на информационните технологии и информационното общество.

» Мартин Кулов е изпълнителен директор във фирма "Код Атест", където разработва проекти за повишаване качеството на софтуерните продукти в България чрез автоматизация на процесите и внедряване на системи за управление на качеството.

Мартин е опитен лектор и сертифициран от Майкрософт разработчик по програмите MCSD и MCSD.NET.

» Други инструктори с опит като преподаватели и програмисти.

Академията

» **Национална академия по разработка на софтуер** (**НАПС**) е център за професионално обучение на софтуерни специалисти.

» **НАПС** провежда задълбочени курсове по разработка на софтуер и съвременни софтуерни технологии.

» Предлагани специалности:

.NET Enterprise Developer

Java Enterprise Developer

» **Качествено обучение** с много **практически упражнения**

» Завършвате само за 3 месеца.

» **Гарантирана работа** след успешно завършване!

» Професионална сертификация!

» **БЕЗПЛАТНО!**

Учите безплатно, плащате след като завършите и започнете работа.

Стипендии от софтуерни фирми.

http://academy.devbg.org

www.devbg.org

Българската асоциация на разработчиците на софтуер е нестопанска организация, която подпомага професионалното развитие на българските софтуерни разработчици чрез различни дейности, насърчаващи обмяната на опит между тях и усъвършенстването на техните знания и умения в областта на проектирането и разработването на софтуер.

Асоциацията организира конференции, семинари, симпозиуми, работни срещи и курсове за обучение по разработка на софтуер и софтуерни технологии.

Съдържание

Увод

Настоящата разработка има за цел да запознае читателя с проблемите, свързани с цифровото подписване на документи в уеб среда и да предложи конкретни подходи за тяхното решаване. След кратък анализ на съществуващите решения се прави преглед на технологиите, проектира се и се имплементира Java-базирана система за цифрово подписване на документи в уеб и верификация на цифрови подписи и сертификати.

Проблемът с цифровите подписи в уеб среда

Днешните уеб браузъри нямат стандартна функционалност за подписване на прикачени файлове при изпращането им от клиента към уеб сървъра. Това води до проблеми при реализацията на някои специфични приложения, в които потребителят трябва да удостовери по надежден начин, че той е изпратил даден документ. Примери за такива приложения са взаимодействието с електронното правителство, електронно банкиране, някои финансови системи и др.

Цели

Целта, която си поставяме, е да се разработи технология за цифрово подписване на документи в уеб среда, основана на инфраструктурата на публичния ключ (PKI) и сървърен софтуер, който посреща подписаните документи и проверява валидността на сигнатурата им и използвания цифров сертификат.

Разработената технология трябва да е независима от операционната система и от уеб браузъра на потребителя.

Потенциални потребители

С въвеждането на електронното правителство и увеличаването на услугите, извършвани по електронен път, нуждата от средства за цифрово подписване на документи нараства. Нараства и необходимостта този процес да се извършва в уеб среда за да максимално лесно достъпен за обикновения гражданин.

Като потенциални потребители на разработената технология можем да идентифицираме уеб разработчиците, които в своите приложения трябва да прилагат технологията на цифровия подпис. Предложеното решение е базирано на Java технологиите, но може да бъде използвано и от други уеб разработчици, тъй като използва отворени стандарти, необвързани с Java платформата.

Цифрови подписи и сертификати

В първа глава е направен обзор на терминологията, проблемите и технологиите, свързани с използването на цифров подпис в уеб среда.

Изясняват са понятията, свързани с цифровия подпис, моделите на доверие между непознати страни и инфраструктурата на публичния ключ: публичен ключ, личен ключ, цифров сертификат, сертификационен орган, 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. Изяснява се технологията на смарт картите като сигурен и надежден начин за съхранение на ключове и сертификати.

Прави се преглед на съществуващите технологии за подписване на документи в уеб среда. Анализират се техните силни и слаби страни. Разглеждат се начините за реализация на система за подписване на документи в клиентския уеб браузър, работеща върху всички операционни системи и всички масово разпространени уеб браузъри. Обосновава се необходимостта от използване на подписан Java аплет, който подписва файловете преди изпращането им от клиента към сървъра.

Работа с цифрови подписи и сертификати в Java

Във втора глава се разглеждат 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. Разглеждат се и средствата за достъп до смарт карти от Java.

Дават се конкретни насоки как разгледаните класове могат да се използват за полагане на цифров подпис, верификация на цифров подпис и верификация на цифрови сертификати и сертификационни вериги.

Проектиране на система за цифрово подписване в уеб среда

В трета глава се анализират и решават конкретните проблеми, свързани с реализацията на Java-базирана система за цифрово подписване на документи в уеб среда. Проектират се отделните компоненти на системата –Java аплет, който подписва документите в клиентския уеб браузър (във вариант за работа с локален сертификат и със смарт карта), уеб приложение, което посреща подписаните документи с подсистема за верификация на цифровия подпис и подсистема за верификация на сертификата на потребителя.

Обосновава се необходимостта Java аплетът, който подписва документи, да работи с повишени права. Разглежда се технологията за подписване на Java аплети с цел повишаване на правата, с които се изпълняват. Разглеждат се и средствата за комуникация между аплет и уеб браузър.

От страна на сървъра се разглежда проблемът с верификацията на цифровия подпис и начините за верификация на цифрови сертификати. Разглеждат се двата подхода за верификацията на сертификата на потребителя, които имат приложение при различни сценарии за използване на цифров подпис. Единият подход е директна верификация, при която се проверява дали сертификатът на клиента е директно издаден от даден сертификат, на който

сървърът има доверие. Другият подход е класическият – проверка за валидност на цялата сертификационната верига, която започва от клиентския сертификат и трябва да завършва със сертификат, на който сървъра има доверие.

Реализация на системата за цифрово подписване в уеб среда

В четвърта глава се преминава през стъпките за конкретната реализация на вече проектираната система за подписване на документи в уеб среда.

Реализацията е наречена NakovDocumentSigner и предоставя напълно функционално рамково приложение (framework) за цифрово подписване на документи в Java-базирани уеб приложения.

Системата демонстрира как със средствата на Java 2 платформата могат да се подписват документи в уеб среда и да се верифицират цифрови подписи, сертификати и сертификационни вериги. Включен е пълният сорс код на отделните компоненти на системата, придружен с разяснения за тяхната работа. Приложени са и скриптове и инструкции за компилиране, настройка и внедряване на системата.

Тестване, оценка и усъвършенстване

В пета глава е направена критична оценка на реализираната система за подписване на документи в уеб среда. Системата е тествана под различни операционни системи и уеб браузъри и оценена е нейната работоспособност. Анализирани са недостатъците на системата и са описани посоките за нейното бъдещо развитие и възможностите за подобрение.

За автора на настоящата разработка

Светлин Наков е преподавател по съвременни софтуерни технологии в СУ "Св. Климент Охридски", където води курсове по Компютърни алгоритми, Интернет програмиране с Java, Мрежова сигурност, Програмиране за .NET Framework и Конструиране на качествен програмен код.

Той има сериозен професионален опит като софтуерен разработчик и консултант. Неговите интереси обхващат Java технологиите, .NET платформата и информационната сигурност. Работил е по образователни проекти на Microsoft Research в областта на .NET Framework и по изграждането на големи и сложни информационни системи за държавната администрация.

Светлин е завършил Факултета по математика и информатика на СУ "Св. Климент Охридски". Като ученик и студент той е победител в десетки национални състезания по програмиране и е носител на 4 медала от международни олимпиади по информатика.

Светлин има десетки научни и технически публикации, свързани с разработката на софтуер, в български и чуждестранни списания и е автор на книгата "Интернет програмиране с Java". Той е автор и на серия статии, свързани с използването на цифров подпис в уеб среда в престижното електронно издание „developer.com”.

През 2003 г. Светлин Наков е носител на наградата "Джон Атанасов" на фондация Еврика. През 2004 г. получава най-високото българско отличие за ИТ – награда "Джон Атанасов" от президента на България Георги Първанов за приноса му към развитието на информационните технологии и информационното общество.

В момента Светлин е директор и водещ преподавател в Национална академия по разработка на софтуер, където обучава софтуерни специалисти за практическа работа в ИТ индустрията.

Повече информация за Светлин Наков може да се намери на неговия личен уеб сайт: www.nakov.com.

Благодарности

Авторът благодари на Николай Недялков, президентът на Асоциация за информационна сигурност (ISECA) за съдействието му при разрешаване на технически проблеми, свързани с използването на смарт карти, и за осигурения хардуер за провеждане на изследванията и тестването на реализираните системи.

НАПС
НАЦИОНАЛНА АКАДЕМИЯ ПО РАЗРАБОТКА НА СОФТУЕР

НАЦИОНАЛНА АКАДЕМИЯ ПО РАЗРАБОТКА НА СОФТУЕР

Лекторите

» Светлин Наков е преподавател по съвременни софтуерни технологии в СУ "Св. Климент Охридски".

Той е автор на десетки научни и технически публикации и няколко книги, свързани с разработката на софтуер, заради което е търсен лектор и консултант.

През 2004 г. получава наградата "**Джон Атанасов**" от президента на България Георги Първанов за приноса му към развитието на информационните технологии и информационното общество.

» Мартин Кулов е изпълнителен директор във фирма "Код Атест", където разработва проекти за повишаване качеството на софтуерните продукти в България чрез автоматизация на процесите и внедряване на системи за управление на качеството.

Мартин е опитен лектор и сертифициран от Майкрософт разработчик по програмите MCSD и MCSD.NET.

» Други инструктори с опит като преподаватели и програмисти.

Академията

» **Национална академия по разработка на софтуер** (**НАПС**) е център за професионално обучение на софтуерни специалисти.

» **НАПС** провежда задълбочени курсове по разработка на софтуер и съвременни софтуерни технологии.

» Предлагани специалности:

.NET Enterprise Developer

Java Enterprise Developer

» **Качествено обучение** с много **практически упражнения**

» Завършвате само за 3 месеца.

» **Гарантирана работа** след успешно завършване!

» Професионална сертификация!

» **БЕЗПЛАТНО!**

Учите безплатно, плащате след като завършите и започнете работа.

Стипендии от софтуерни фирми.

http://academy.devbg.org

Глава 1. Цифрови подписи и сертификати

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. Най-разпространеният от подходите за удостоверяване на произхода на документи и файлове е чрез използването на т. нар. цифров подпис (електронен подпис).

1.1. Цифров подпис

Да разгледаме понятията, свързани с цифровия подпис, сценариите за неговата употреба и алгоритмите за подписване и верификация на подпис.

1.1.1. Основни понятия

Симетрични и несиметрични кодиращи схеми

В света на компютърната криптографията има два вида шифриращи алгоритмични схеми – симетрични и асиметрични [Johner, 2000, стр. 5-9].

При **симетричните схеми** за шифриране се използва един и същ ключ при шифриране и дешифриране на информацията. Примери за симетрични шифриращи схеми са алгоритмите за блоков шифър DES, 3DES, IDEA, Blowfish, Rijndael и RC5 и алгоритмите за поточен шифър – RC4 и SEAL.

При **асиметричните схеми** се използва един ключ за шифрирането на информацията и друг (различен от първия) ключ за дешифрирането. Примери за асиметрични кодиращи схеми са алгоритмите RSA, DSA и ElGamal.

Цифровото подписване на документи използва като математическа основа криптографията, базирана на публични ключове (асиметричните схеми).

Криптография, 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 съобщения (документи), проверка на цифров подпис, шифриране и дешифриране на съобщения (документи).

Публични и лични ключове

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. При това от даден публичен ключ не може да се извлече съответният му личен ключ и

обратното. Това прави двойките публичен/личен ключ подходящи за шифриране на съобщения без да е необходимо двете страни, които обменят информация, да си знаят ключовете една на друга.

За да използва дадено лице криптография с публични ключове, то трябва да притежава двойка публичен ключ и съответен на него личен ключ. Съществуват алгоритми, които могат да генерират такива двойки ключове по случаен начин.

Публичен 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 посредством криптография с публични ключове.

Личен ключ

Личният ключ (**private key**) е число (последователност от битове), известно само на притежателя му. С личния си ключ дадено 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 широкоразпространено 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, се компрометира и 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.

Криптография и криптоанализ

Математическата наука, която се занимава с изследването на алгоритми и схеми за шифриране и дешифриране на съобщения, се нарича **криптография**. Науката, която се занимава с анализирането и разбиването на

криптографски ключове, кодове и схеми за конфиденциална обмяна на съобщения, се нарича **криптоанализ**.

За сигурността на публичните и личните ключове

За целите на криптографията с публични ключове се използват такива криптографски алгоритми, че практически не е по силите на съвременния криптоанализ и съвременната изчислителна техника да се открие личният ключ на лице, чийто публичен ключ е известен.

В практиката най-често се използват ключове с големина между 128 и 2048 бита, поради което методът на грубата сила (brute force) не може да бъде приложен за отгатването на таен ключ.

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 [Wikipedia-PKC, 2005].

Хеширане и хеш-стойност

Хеш-стойността на едно входно съобщение представлява последователност от битове, обикновено с фиксирана дължина (например 160 или 256 бита), извлечено по някакъв начин от съобщението. Алгоритмите, които изчисляват хеш-стойност, се наричат **хеширащи алгоритми**.

Криптографски силни хеширащи алгоритми

Съществуват различни хеширащи алгоритми, но само някои от тях са криптографски силни (cryptographically secure hash algorithms). При криптографски силните хеширащи алгоритми е изключително трудно (почти невъзможно) по дадена хеш-стойност да се намери входно съобщение, на което тя съответства [Steffen, 1998].

Криптографски силните алгоритми за изчисление на хеш-стойност имат свойството, че при промяна дори само на 1 бит от входното съобщение се получава тотално различна хеш-стойност. Това поведение ги прави изключително устойчиви на криптоаналитични атаки.

Невъзможността за възстановяване на входното съобщение по дадена хеш-стойност е съвсем логична, като се има предвид, че хеш-стойността на едно съобщение може да е със стотици пъти по-малък размер от него самото. Това, обаче, не изключва възможността две напълно различни съобщения да имат една и съща хеш-стойност (**колизия**), но вероятността това да се случи е много малка.

При криптографски силните хеширащи алгоритми съществуват колизии, но ако дадено (оригинално) съобщение има дадена хеш-стойност, всяко друго съобщение със същата хеш-стойност се различава значително по дължина от оригиналното.

Някои по-известни криптографски алгоритми за хеширане, широко използвани в практиката, са MD4, MD5, RIPEMD160, SHA1, SHA-256 и др.

Цифрово подписване

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 ключ. На по-късен етап тази допълнителна информация може да се използва за да се провери произходът на подписания документ.

Цифров подпис

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 ключа, с който е извършено подписването.

Цифровият подпис позволява на получателя на дадено съобщение да провери истинския му произход и неговата цялостност (интегритет).

При цифровото подписване се използват асиметрични схеми за криптиране, като подписът се полага с личния ключ на лицето, което подписва, а проверката на подписа става с неговия публичен ключ. Така ако дадено съобщение бъде подписано от дадено лице, всеки, който има неговия публичен ключ, ще може да провери (верифицира) цифровия подпис.

След като е веднъж подписано, дадено съобщение не може да бъде променяно, защото това автоматично прави невалиден цифровия подпис.

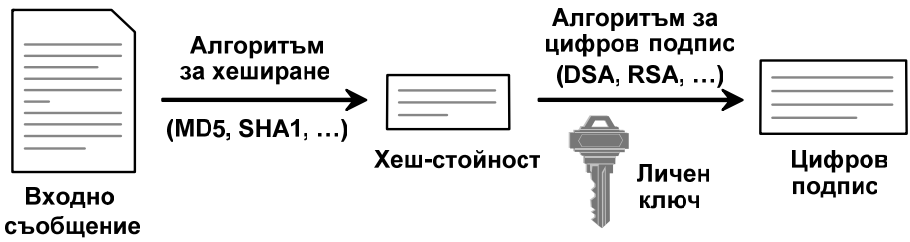
От математическа гледна точка е практически невъзможно един документ да бъде подписан без да е известен личният ключ на лицето, което го подписва. Това се дължи на технологията на цифровия подпис, която съчетава хеширане и шифриране с надеждни алгоритми, устойчиви на криптоаналитични атаки.

1.1.2. Технология на цифровия подпис

Криптографията, базирана на публични ключове осигурява надежден метод за цифрово подписване, при който се използват двойки публични и лични ключове [Johner, 2000, стр. 10-12].

Полагане на цифров подпис

Едно лице полага цифров подпис под дадено електронно съобщение (символен низ, файл, документ, e-mail и др.) чрез личния си ключ. Технически цифровото подписване на едно съобщение се извършва на две стъпки, както е показано на фигура 1-1:



Фигура 1-1. Цифрово подписване на съобщение

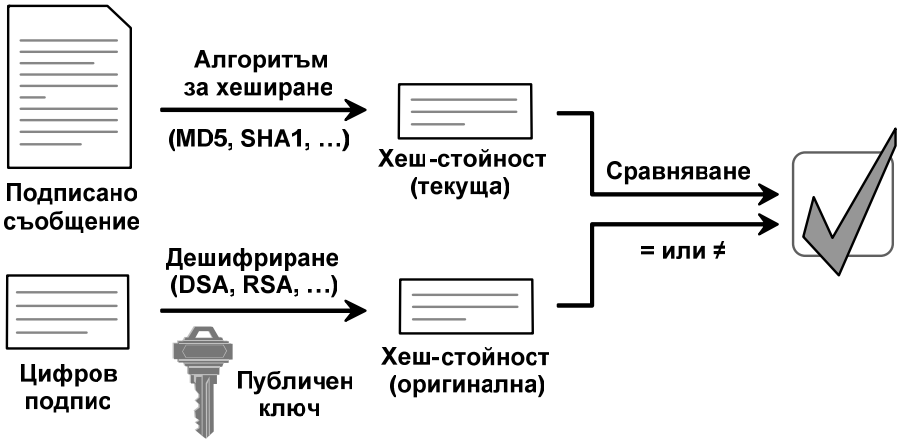
1. На първата стъпка се изчислява **хеш-стойност** на съобщението (**message digest**) по някакъв криптографски силен **алгоритъм за хеширане** (например MD4, MD5, SHA1 или друг).
2. На втората стъпка от цифровото подписване получената в първата стъпка хеш-стойност на съобщението се шифрира с личния ключ, с който се извършва подписването. За целта се използва някакъв математически алгоритъм за цифров подпис (**digital signature algorithm**), който използвайки личния ключ преобразува хеш-стойността в шифрирана хеш-стойност, наричана още **цифров подпис**. Най-често се използват алгоритмите за цифров подпис **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.

Верификация на цифров подпис

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 проверката на цифров подпис се извършва на три стъпки, както е показано на фигура 1-2:



Фигура 1-2. Верификация на цифров подпис

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

2. На втората стъпка се дешифрира цифровият подпис, който трябва да бъде проверен. Дешифрирането става с публичния ключ, съответстващ на личния ключ, който е използван при подписването на съобщението, и със същия алгоритъм, който е използван при шифрирането. В резултат се получава оригиналната хеш-стойност, която е била изчислена при хеширането на оригиналното съобщение в първата стъпка от процеса на подписването му.

3. 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 съобщението е било подписано 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 съобщението е било променяно (подправено) след подписването му, текущата хеш-стойност, изчислена от подправеното съобщение, ще бъде различна от оригиналната хеш-стойност, защото на различни съобщения съответстват различни хеш-стойности.

   - Ако публичният ключ не съответства на личния ключ, който е бил използван за подписването, оригиналната хеш-стойност, получената

от цифровия подпис при дешифрирането с неправилен ключ, няма да е вярната.

Причини за невалиден цифров подпис

Ако верификацията се провали, независимо по коя от трите причини, това доказва само едно: подписът, който се верифицира, не е получен при подписването на съобщението, което 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 и системата да прави опит за верифициране на подписани от това лице съобщения 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 документът и цифровият му подпис, се изпраща и цифров сертификат на лицето, което е положило този подпис. Цифровите сертификати ще разгледаме в детайли в следващата точка, но засега можем да считаме, че те свързват дадено лице с даден публичен ключ.

При верификацията на подписан документ, придружен от цифров сертификат, се използва публичният ключ, съдържащ се в сертификата и ако верификацията е успешна, се счита, че документът е изпратен от лицето, описано в сертификата. Разбира се, преди това трябва да се провери валидността на сертификата.

1.2. Цифрови сертификати и PKI

Нека сега разгледаме технологията на цифровите сертификати и инфраструктурата на публичния ключ, които дават възможност да се установи доверие между непознати страни.

1.2.1. Модели на доверие между непознати страни

Когато непознати страни трябва да обменят информация по сигурен начин, това може да стане чрез криптография с публични и лични ключове, но има един съществен проблем. По какъв начин двете страни да обменят първоначално публичните си ключове по несигурна преносна среда?

За съжаление математическо решение на този проблем няма. Съществуват алгоритми за обмяна на ключове, например алгоритъмът на Diffe-Helman, но те не издържат на атаки от тип „човек по средата” (main-in-the-middle).

За да започнат две страни сигурна комуникация по несигурна преносна среда, е необходимо по някакъв начин те да обменят публичните си ключове. Един от подходите за това е двете страни да се доверят на **трета,**

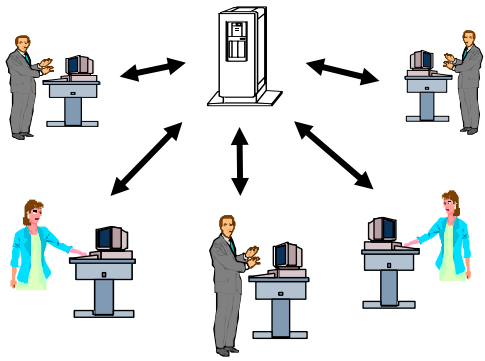
независима страна, на която вярват, и с нейна помощ да обменят ключовете си. Тази трета страна е в основата на инфраструктурата на публичния ключ, но има различни подходи, по които се избира такава трета страна и различни принципи, на които се изгражда доверието. Тези подходи и принципи се дефинират от т. нар. „**модели на доверие**” [PGP, 1999, стр. 29-33], [NZSSC, 2005].

Разпространени са различни модели на доверие между непознати страни – модел с единствена доверена страна, Web of trust, йерархичен модел, модел на кръстосаната сертификация (cross certification) и др.

Модел с единствена доверена страна

Моделът с единствена доверена страна (директно доверие) е най-простият. При него има единствена точка, централен сървър, на който всички вярват. Сървърът може да съхранява публичните ключове на всички участници в комуникацията или просто да ги подписва със своя личен ключ, с което да гарантира истинността им.

На фигура 1-3 е показан схематично моделът на директно доверие:



Фигура 1-3. Директно доверие

Когато две непознати страни искат да установят доверие по между си, всяка от тях извлича публичния ключ на другата от централния сървър или просто проверява дали той е подписан с ключа на сървъра. Ако някой участник бъде компрометиран, сървърът лесно може да анулира доверието си към него и респективно всички участници да спрат да му вярват.

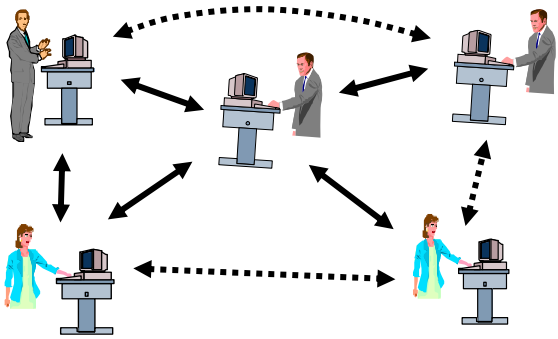
Този модел е приложим за малки и средни по големина организации. При големи организации настъпват трудности със скалируемостта, защото един сървър не може да обслужи прекалено много клиенти. Понеже сървърът е единствен, рискът е съсредоточен в единствена точка. Така, ако по някакъв начин сървърът бъде компрометиран, щетите ще са непоправими.

Модел „Web of trust”

Моделът “Web of trust” е приложим при малки групи участници, между някои от които има предварително установено доверие. При този модел всеки

участник знае истинските публични ключове на част от останалите (има доверие на своите приятели). За установяване на доверие между непознати участници не е необходим централен сървър, който да съхранява ключовете на всички, а доверието се трансферира транзитивно от приятел на приятел. Възможно даден участник да подписва цифрово публичните ключове на всички участници, на които има директно доверие.

На фигура 1-4 е показан моделът „Web of trust" в действие:



Фигура 1-4. Модел на доверие "Web of trust"

Ако даден участник A има директно доверие на друг участник B, а участник B има директно доверие на участник C, то може да се счита, че участник A има индиректно доверие на участник C. По този начин всеки участник може транзитивно да изгради своя мрежа на доверие.

По модела на Web of Trust работи една от най-големите световни системи за подписване на електронна поща и други документи – PGP (Pretty Good Privacy).

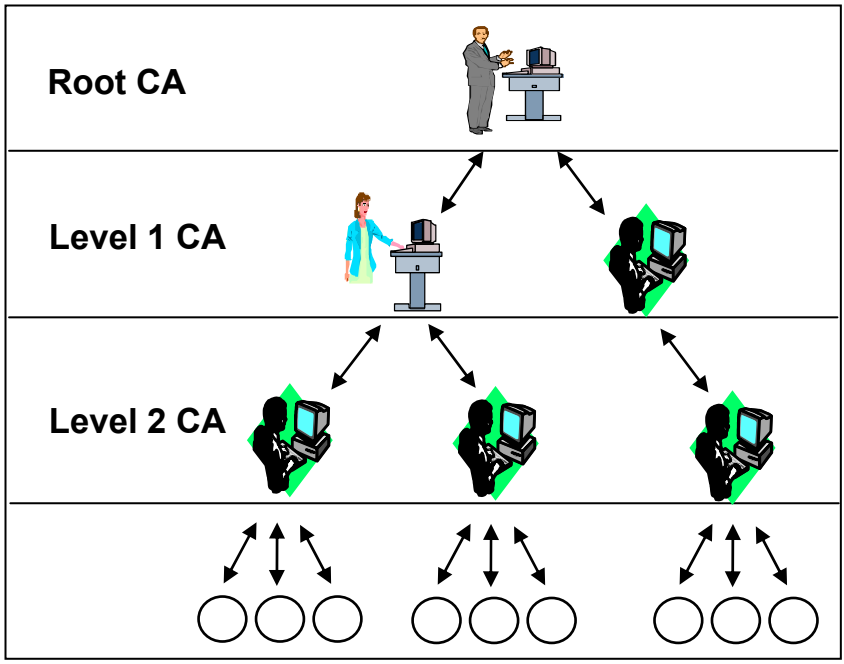
Проблемите с този модел са, че индиректното доверие много лесно може да бъде компрометирано, защото някой компрометиран участник може да изгради фалшиво доверие към други участници и така транзитивно да компрометира голяма част от цялата мрежа на доверие. При някои имплементации, за по-голяма сигурност при изграждане на индиректно доверие се изисква двойна индиректна връзка, т. е. две независими страни да потвърдят, че вярват на публичния ключ на даден участник.

Йерархичен модел на доверие

При йерархичния модел на доверие всички участници вярват в няколко доверени трети страни, наречени сертификационни органи (certification authorities – CA). Сертификационните органи от своя страна трансферират доверието на свои подчинени сертификационни органи, а те също могат да го трансферират или да потвърдят доверието към някой конкретен участник в комуникацията.

На сертификационните органи от първо ниво (root CA) се вярва безусловно. Техните публични ключове са известни предварително на всички участници. На сертификационните органи от междинно ниво се вярва индиректно

(транзитивно). Техните публични ключове са подписани от сертификационните органи на предходното ниво. На конкретните участници в комуникацията се вярва също транзитивно. Техните публични ключове са подписани от междинните сертификационни органи (фигура 1-5):



Фигура 1-5. Йерархичен модел на доверие

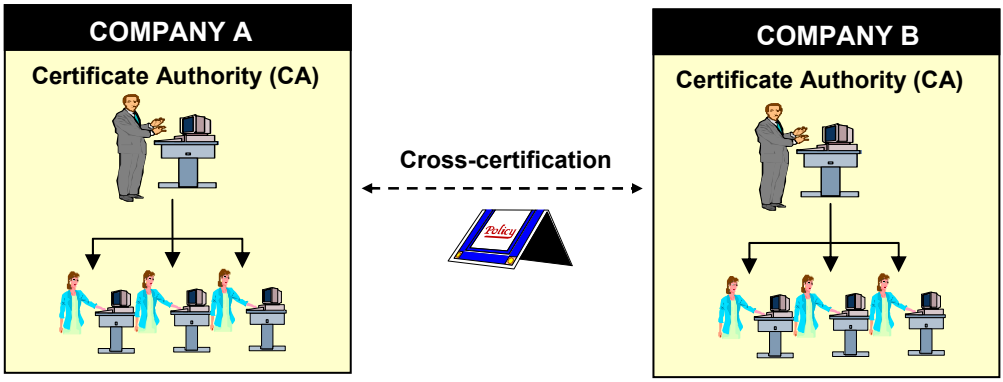
Йерархичният модел на доверие е силно разпространен в Интернет, при финансови, e-commerce и e-goverment приложения. Той е мощен и работи добре при голям брой участници.

При компрометиране на някой сертификационен орган, се компрометират и всички участници, които се намират по-надолу от него. При компрометиране на обикновен участник, доверието към него се анулира от сертификационния орган, който му го е дал.

Хибриден модел „кръстосана сертификация”

Хибридният модел на кръстосана сертификация се базира на йерархии от сертификационни органи и участници в комуникацията (както при йерархичния модел), но позволява транзитивно прехвърляне на доверие между отделните йерархии (както в „Web of trust” модела). Използва се при нужда от обединяване на няколко йерархични инфраструктури на доверие.

На фигура 1-6 моделът е показан схематично:



Фигура 1-6. Хибриден модел на кръстосана сертификация

Кой модел да използваме?

Няма универсален най-добър модел. При различни сценарии различни модели могат да бъдат най-подходящи. В нашата работа ще наблегнем на йерархичния модел, като най-широко разпространен за масова употреба в Интернет среда.

1.2.2. Цифрови сертификати и инфраструктура на публичния ключ

Инфраструктурата на публичния ключ разчита на цифрови сертификати и сертификационни органи за да осигури доверие между непознати страни. Нека разгледаме по-детайлно нейната организация.

Цифрови сертификати

Цифровите сертификати свързват определен публичен ключ с определено лице. Можем да ги възприемаме като електронни документи, удостоверяващи, че даден публичен ключ е собственост на дадено лице [PGP, 1999, стр 21-27].

Сертификатите могат да имат различно ниво на доверие. Те могат да бъдат саморъчно-подписани (self-signed) или издадени (подписани) от някого. За по-голяма сигурност сертификатите се издават от специални институции, на които се има доверие (т. нар. **сертификационни органи**) при строги мерки за сигурност, които гарантират тяхната достоверност.

Съществуват различни стандарти за цифрови сертификати, например PGP (Pretty Good Privacy), SPKI/SDSI (Simple Public Key Infrastructure / Simple Distributed Security Infrastructure) и X.509. В практиката за целите на цифровия подпис най-масово се използват X.509 сертификатите. Те са ориентирани към йерархичния модел на доверие.

Стандартът X.509

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 сертификационния орган, който е издал сертификата, информация за срока му на валидност, информация за използваните криптографски алгоритми и различни други детайли.

Инфраструктура на публичния ключ

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 [GlobalSign, 1999].

PKI използва т. нар. сертификационни органи (certification authorities), които управляват и подпомагат процесите по издаване, анулиране, съхраняване и верификация на цифрови сертификати.

Доверието в рамките на PKI инфраструктурата между непознати страни се базира на цифрови сертификати, чрез които даден сертификационен орган удостоверява кой е собственикът на даден публичен ключ.

Има различни типове PKI инфраструктура, но ние ще разгледаме само най-разпространения тип – този с йерархичен модел на доверие, който широко се използва в Интернет (например при HTTPS протокола за сигурна връзка между уеб браузър и уеб сървър).

Услуги на PKI инфраструктурата

PKI инфраструктурата има за цел да осигури следните услуги на своите потребители:

- **Конфиденциалност** при пренос на информация. Осъществява се чрез шифриране на информацията с публичен ключ и дешифриране със съответния му личен ключ.
- **Интегритет** (неизменимост) на пренасяната информация. Осъществява се чрез технологията на цифровия подпис.
- **Идентификация и автентикация**. Цифровите сертификати осигуряват идентификация на даден участник в комуникацията и позволяват той да бъде автентикиран по сигурен начин. Сертификационните органи гарантират с цифров подпис, че даден публичен ключ е на даден участник.
- **Невъзможност за отказ** (non-repudiation) от извършено действие. Осъществява се чрез цифрови подписи, които идентифицират участника, извършил дадено действие, за което се е подписал.

Сертификационни органи

За издаването и управлението на цифрови сертификати инфраструктурата на публичния ключ разчита на т. нар. сертификационни органи (доставчици на удостоверителни услуги), които позволяват да се изгради доверие между

непознати страни, участнички в защитена комуникация, базирана на публични и лични ключове.

Сертификационен орган (certification authority – CA) е институция, която е упълномощена да издава цифрови сертификати и да ги подписва със своя личен ключ [Raina, 2003, стр 30-31]. Целта на сертификатите е да потвърдят, че даден публичен ключ е притежание на дадено лице, а целта на сертификационните органи е да потвърдят, че даден сертификат е истински и може да му се вярва. В този смисъл сертификационните органи се явяват безпристрастна доверена трета страна, която осигурява висока степен на сигурност при компютърно-базирания обмен на информация.

Сертификационните органи се наричат още **удостоверяващи органи** или **доставчици на удостоверителни услуги**.

Ако един сертификационен орган е издал цифров сертификат на дадено лице и се е подписал, че този сертификат е наистина на това лице, можем да вярваме, че публичният ключ, който е записан в сертификата, е наистина на това лице, стига да имаме доверие на този сертификационен орган.

Регистрационен орган

Регистрационният орган (registration authoruty – RA) е орган, който е упълномощен от даден сертификационния орган да проверява самоличността и автентичността на заявителя при заявка за издаване на цифров сертификат. Сертификационните органи издават цифрови сертификати на дадено лице след като получат потвърждение от регистрационния орган за неговата самоличност [Raina, 2003, стр. 33-35].

Регистрационните органи се явяват на практика междинни звена в процеса на издаване на цифрови сертификати. Регистрационен орган може да бъде както дадено физическо лице, например системният администратор при малка корпоративна мрежа, така и дадена институция, например орган упълномощен от държавата.

Ниво на доверие в сертификатите

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 регистрационния орган.

Ниво на доверие в сертификационните органи

Не на всички сертификационни органи може да се има доверие, защото е възможно злонамерени лица да се представят за сертификационен орган, който реално не съществува или е фалшив.

За да се вярва на един сертификационен орган, той трябва да е **глобален** и съответно световно признат и утвърден или **локален и** утвърден по силата на локалните закони в дадена държава.

В света на цифровата сигурност утвърдените сертификационни органи 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 органи задължително използват специален хардуер, който гарантира невъзможността за изтичане на важна информация, като например лични ключове.

Световно утвърдени сертификационни органи

Сред най-известните утвърдени глобални световни сертификационни органи са компаниите: GlobalSign NV/SA (www.globalsign.net), Thawte Consulting (www.thawte.com), VeriSign Inc. (www.verisign.com), TC TrustCenter AG (www.trustcenter.de), Entrust Inc. (www.entrust.com) и др.

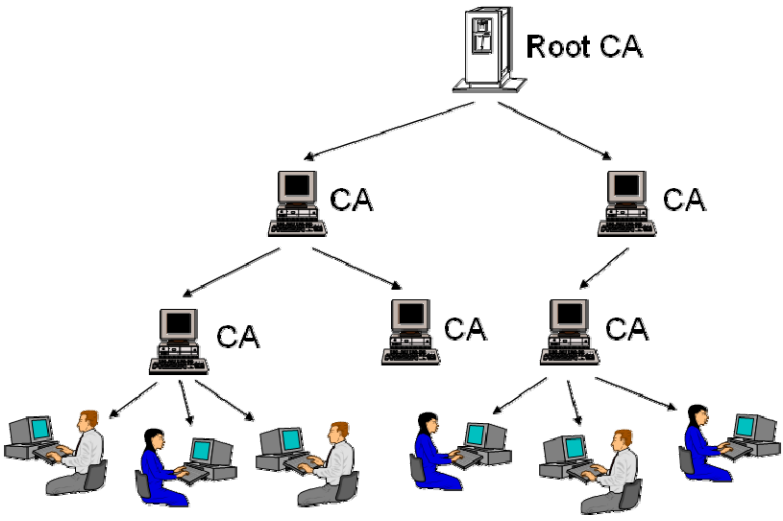
Български сертификационни органи

В България има няколко сертификационни органа, утвърдени от закона за универсалния електронен подпис и комисията за регулиране на съобщенията (www.crc.bg). Сред тях са фирмите: "Информационно обслужване" АД (www.stampit.org) и "Банксервиз" АД (www.b-trust.org).

Йерархична структура на сертификационните органи

Всеки сертификационен орган има свой сертификат и съответстващ на него личен ключ (който се съхранява при много строги мерки за сигурност), с който подписва сертификатите, които издава на своите клиенти.

Един сертификационен орган може да бъде от **първо ниво** (top-level certification authority; Root CA) или да бъде от някое следващо ниво (вж. фигура 1-7):



Фигура 1-7. Йерархия на сертификационните органи

Сертификационните органи от първо ниво при започването на своята дейност издават сами на себе си сертификат, който подписват с него самия. Тези сертификати са саморъчно-подписани и се наричат **Root-сертификати**.

Root-сертификатите на утвърдените световни сертификационни органи са публично достъпни и могат да се използват за верификация на други сертификати. Сертификационните органи, които не са на първо ниво, разчитат на някой орган от по-горно ниво да им издаде сертификат, с който имат право да издават и подписват сертификати на свои клиенти.

Най-често регистрационните органи, които проверяват самоличността и автентичността на заявителя преди издаване на цифров сертификат, се явяват сертификационни органи от междинно ниво. Те са упълномощени от даден сертификационен органи от първо ниво да препродават неговите удостоверителни услуги на крайни клиенти.

Подписване на сертификати

Технически е възможно всеки сертификат да бъде използван за да се подпише с него всеки друг сертификат, но на практика възможността за подписване на други сертификати е силно ограничена. Всеки сертификат съдържа в себе си неизменими параметри, които указват дали може да бъде използван за подписване на други сертификати.

Сертификационните органи издават на своите клиенти (крайните потребители) сертификати, които не могат да бъдат използвани за подписване на други сертификати. Ако един потребител си купи сертификат от някой сертификационен орган и подпише с него друг сертификат, новоподписаният сертификат няма да е валиден.

Един сертификационен орган издава сертификати, с които могат да бъдат подписвани други сертификати, само на други сертификационни органи, като по този начин ги прави непосредствено подчинени на себе.

Саморъчно подписани сертификати

Един сертификат може да бъде подписан от друг сертификат (най-често собственост на някой сертификационен орган) 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 **саморъчно-подписани сертификати** (self-signed certificates). В частност Root-сертификатите на сертификационните органи на първо ниво се явяват саморъчно-подписани сертификати [Wikipedia, SSC, 2005].

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.

Саморъчно подписани сертификати и модел на директно доверие

Въпреки, че не могат да се възприемат с доверие, саморъчно-подписаните сертификати все пак имат свое приложение. Например в рамките на една вътрешнофирмена инфраструктура, където е възможно сертификатите да се разпространят физически по сигурен начин между отделните служители и вътрешнофирмените системи, self-signed сертификатите могат успешно да заместят сертификатите, издавани от сертификационните органи. В такива вътрешнофирмени среди не е необходимо някой сертификационен орган да потвърждава, че даден публичен ключ е на дадено лице, защото това може да се гарантира от начина издаване и пренасяне на сертификатите. Всъщност тази схема използва директния модел на доверие, при който цялата организация вярва на една единствена трета страна (някакъв централен сървър).

Локален сертификационен орган

Описаната схема за сигурност, базирана на self-signed сертификати може да се подобри, като фирмата изгради свой собствен **локален сертификационен орган** за служителите си. За целта фирмата трябва първоначално да си издаде саморъчно-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 локален сертификационен орган от първо ниво.

Изграждането на локален сертификационен орган се поддържа стандартно от някои операционни системи, например от Windows 2000 Server и Windows 2003 Server, които предоставят т. нар. certificate services.

Рисковете на директното доверие и локалната сертификация

И при двете описани схеми за сигурност (чрез директно доверие и чрез локален сертификационен орган) 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 вътрешнофирмени процедури по издаването и управлението на сертификатите и с помощта на специализиран хардуер, но пълна сигурност не е гарантирана.

Използване на сертификати в Интернет

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 сертификационни органи. Например, ако един уеб сървър в Интернет трябва да комуникира с уеб браузъри по защитен комуникационен канал (SSL), той непременно трябва да притежава сертификат, издаден от някой общопризнат сертификационен орган. В противен случай е възможно шифрираните връзки между клиентите и този уеб сървър да се подслушват от трети лица.

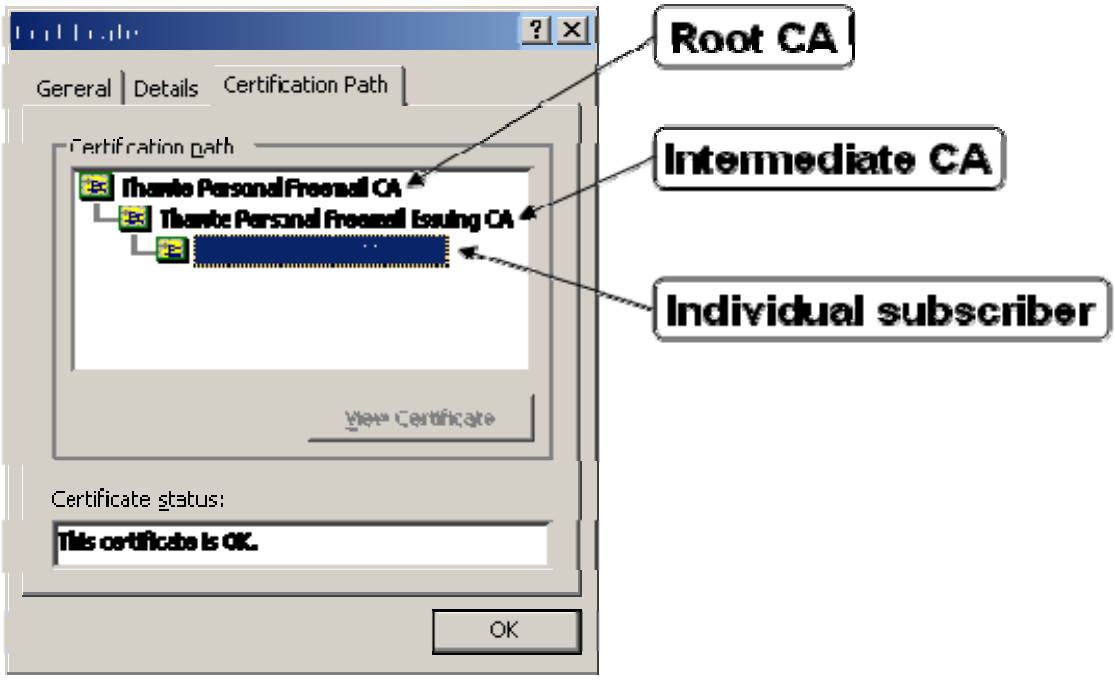
Сертификатите, издадени от утвърдените сертификационни органи дават по-голяма степен на сигурност на комуникацията, независимо дали се използват в частна корпоративна мрежа или в Интернет. Въпреки това self-signed сертификати често се използват, защото сертификатите, издадени от сертификационните органи, 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.

Сертификационни вериги

Когато един сертификационен орган от първо ниво издава сертификат на свой клиент, той го подписва със своя Root-сертификат. По този начин се създава верига от сертификати, състояща се от два сертификата – този на сертификационния орган, предхождан от този на неговия клиент.

Вериги от сертификати (сертификационни вериги, certificate chains, certification paths) 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-сертификатът на някой сертификационен орган. В средата на веригата стоят сертификатите на някакви междинни сертификационни органи [RFC 3280, раздел 3.2].

На фигура 1-8 е даден пример за сертификационна верига:



Фигура 1-8. Сертификационна верига

Тя започва от сертификата на крайния клиент (Thawte Freemail Member), който е подписан от сертификата на междинен сертификационен орган (Thawte Personal Freemail Issuing CA) и завършва с root-сертификата на сертификационен орган от първо ниво (Thawte Personal Freemail CA).

Общовъзприетата практика е сертификационните органи от първо ниво да издават сертификати на междинните сертификационни органи, като отбелязват в тези сертификати, че могат да бъдат използвани за издаване на други сертификати. Междинните сертификационни органи издават сертификати на свои клиенти или на други междинни сертификационни органи. 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 сертификационни органи не се налага такова ограничение.

Верификация на цифрови сертификати

Има няколко подхода за проверка на валидността на цифрови сертификати. В зависимост от модела на доверие (модел с единствена доверена страна, йерархичен модел, модел Web of trust и т. н.) сертификатите могат да се проверяват по различни начини.

Ние ще разгледаме два начина за проверка на цифрови сертификати – директна верификация и верификация на сертификационна верига.

Директна верификация на сертификати

При директната верификация се проверява срокът на валидност на сертификата и се проверява дали той е издаден директно от сертификационен орган, на който имаме доверие. Обикновено директната верификация се използва при модел на доверие с единствена доверена страна (локален сертификационен орган).

Реализацията е много лесна – просто се проверява дали даденият сертификат е директно подписан от сертификат, на който имаме доверие. За целта системата, която извършва такава проверка, трябва да поддържа множество от **доверени сертификати за директна проверка**.

Верификация на сертификационна верига

При йерархичния модел на доверие, когато трябва да се верифицира даден сертификат, се проверява последователно доверието към сертификатите от цялата негова сертификационна верига. Ако сертификационната верига не е налична, тя трябва да се построи по някакъв начин.

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. За всеки от сертификатите се проверява допълнително срокът му на валидност и дали има право да подписва други сертификати [RFC 3280, раздел 6].

Доверени Root-сертификати

Всеки софтуер за проверка на сертификационни вериги поддържа списък от **доверени Root-сертификати** (trusted root CA certificates), на които вярва безусловно. Това са най-често Root-сертификатите на утвърдените глобални (световни) сертификационни органи.

Например уеб браузърът Microsoft Internet Explorer идва стандартно със списък от около 150 доверени Root-сертификата, а браузърът Mozilla при първоначална инсталация съдържа около 70 доверени сертификата.

Процесът на верификация на сертификационна верига

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). На процеса по анулиране на сертификати ще се спрем малко по-късно.

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 по описания начин, ако не е налична цялата му сертификационна верига или ако Root-сертификатът, 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 участват в нея.

1.2.3. Хранилища за сертификати

В системите за електронно подписване на документи се използват **защитени хранилища за ключове и сертификати** (protected keystores). Едно такова хранилище може да съдържа три типа елементи – сертификати, сертификационни вериги и лични ключове.

Хранилищата са защитени с пароли

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 осъществяване на неправомерен достъп до дадено 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.

Стандарти за защитени хранилища

Има няколко разработени стандарта за защитени хранилищата за ключове и сертификати, например JKS (Java Key Store), PKCS#12 и смарт-карти (PKCS#11).

Стандартът PKCS#12

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). Едно PKCS#12 хранилище обикновено съдържа един сертификат, съответен на него личен ключ и сертификационна верига, удостоверяваща автентичността на сертификата [PKCS#12]. 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.

Смарт карти

Смарт картите (smart card) представляват магнитни или чип-карти, които съхраняват чувствителна информация (лични ключове, сертификати и др.) и защитават много по-сигурно информацията, отколкото файловете с PFX хранилища.

Повечето съвременни смарт карти притежават криптопроцесор и защитена област за данни, която не може да бъде прекопирана. Всъщност достъп до защитената област от смарт картата има само нейният криптопроцесор, който може да криптира и подписва данни. Потребителите реално не извличат личните ключове от картата, а ползват услугите на вградения в нея криптопроцесор.

Невъзможността за извличане на защитената информация от смарт карта я прави изключително надеждна, защото достъпът до личните ключове в нея е възможен, само ако тя е физически налична. Ако картата бъде откраднатa, собственикът й може да установи това и да сигнализира на издателя й да я анулира.

При използването на PKCS#12 хранилища, съхранявани във вид на файл, има риск някой да прекопира този файл и да подслуша по някакъв начин паролите за достъп до него, след което да извлече личните ключове без знанието на собственика на хранилището. При използването на смарт карта с криптопроцесор не е възможно личните ключове да бъдат извлечени и прекопирани поради хардуерни съображения. Остава възможността картата да бъде откраднатa физически, но рискът от това е много по-малък в сравнение с риска от прекопирване на файл.

Стандарти за достъп до смарт карти

Смарт картите имат собствен криптопроцесор, собствена операционна система, защитена памет и собствена файлова система. Достъпът до тях става на различни нива с различни протоколи. Стандартът ISO 7816 (http://www.maxking.com/iso78161.htm) описва основните компоненти на смарт картите и работата с тях на ниско ниво.

На по-високо ниво за достъп до смарт карти се използва стандартът PKCS#11, който дефинира програмен интерфейс за взаимодействие с криптографски устройства (cryptographic tokens) – като смарт карти, хардуерни криптографски ускорители и други [PKCS#11].

Софтуерът, който се доставя заедно със смарт картите, обикновено съдържа имплементация на PKCS#11 за съответната карта и за съответния карточетец.

PKCS#11 стандартът не позволява физически да се извличат личните ключове от смарт карта, но дава възможност за използване на тези ключове с цел шифриране, дешифриране или подписване на данни. Разбира се, за да се извърши подобна операция, е необходимо преди това потребителят да въведе своя PIN код, който защитава достъпа до картата.

PKCS#11 стандартът предоставя интерфейс за достъп до защитени хранилища за ключове и сертификати, съхранявани върху смарт карта. По тази причина със смарт картите може да се работи по начин, много подобен на работата с PKCS#12 хранилища. Една PKCS#11 съвместима смарт карта, обаче, има много повече възможности от едно PKCS#12 хранилище.

1.2.4. Издаване и управление на цифрови сертификати

Когато един сертификационен орган издаде цифров сертификат на свой клиент, клиентът в крайна сметка получава едно защитено хранилище за ключове и сертификати, което съдържа издадения му сертификат, свързаният 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 върху смарт карта или се инсталира директно в неговия уеб браузър. След това то може да бъде експортирано в друг формат или да се използва директно за автентикация, за подписване на документи, за проверка на подпис, за шифриране и дешифриране на информация и т.н.

Издаване на сертификат през Интернет

Обикновено когато един сертификат се издава през Интернет, независимо по какъв начин потребителят потвърждава самоличността си, в процедурата по издаването важно участие взима потребителският уеб браузър.

При заявка за издаване на сертификат, отправена към даден сертификационен орган от неговия уеб сайт, уеб браузърът на потребителя генерира двойка публичен и личен ключ, след което изпраща публичния ключ към сървъра на сертификационния орган. Сертификационният орган, след като установи верността на данните за самоличността на своя клиент, му издава сертификат, в който записва получения от неговия уеб 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 съответния сертификационен орган.

След като сертификационният орган издаде сертификата на своя клиент, той го препраща към уеб страница, от която този сертификат може да бъде инсталиран в неговия уеб браузър. Реално потребителят получава по някакъв начин от сертификационния орган новоиздадения му сертификат заедно с пълната му сертификационна верига. Уеб браузърът междувременно е съхранил съответния на сертификата личен ключ и така в крайна сметка потребителят се сдобива със сертификат, съответна сертификационна верига и съответстващ личен ключ, инсталирани в неговия уеб 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 сертификационния орган и така този потребител може да бъде сигурен, че никой друг няма достъп до неговия личен ключ.

Сертификати за тестови цели

За тестови цели могат да бъдат използвани демонстрационни цифрови сертификати, които някои известни сертификационни орган 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/email/.

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.

Сертификатите в уеб браузърите

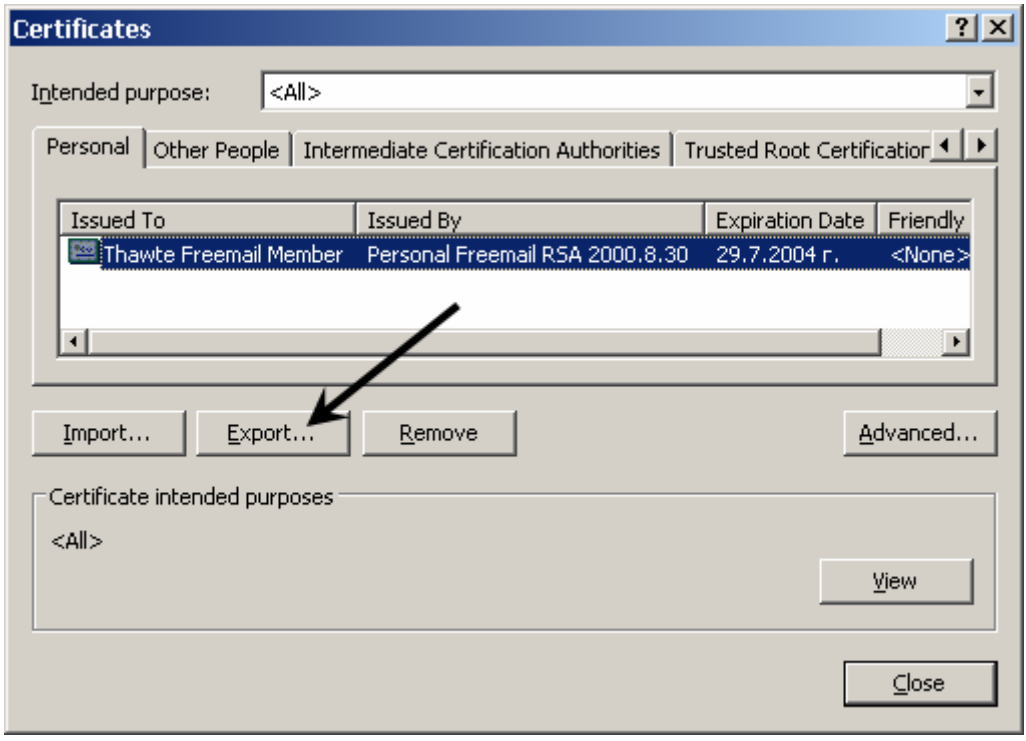
Повечето уеб браузъри могат да използват съхранените в тях сертификати и лични ключове за автентикация пред защитени SSL сървъри и за някои други цели. Много E-Mail клиенти също могат да използват сертификатите, съхранявани в уеб браузърите, при подписване, шифриране и дешифриране на електронна поща.

Има приложения, които не могат да използват директно сертификатите от потребителските уеб браузъри, но могат да работят с PFX хранилища за ключове и сертификати. В такива случаи потребителите могат да експортират от своите уеб браузъри сертификатите си заедно със съответните им лични ключове в PFX файлове и да ги използват от всякакви други програми.

Експортиране на сертификат от уеб браузър в PFX хранилище

След като имаме сертификат с личен ключ, инсталиран в нашия уеб браузър и искаме да го ползваме от външно приложение, трябва да го експортираме във формат PKCS#12 (в .PFX файл).

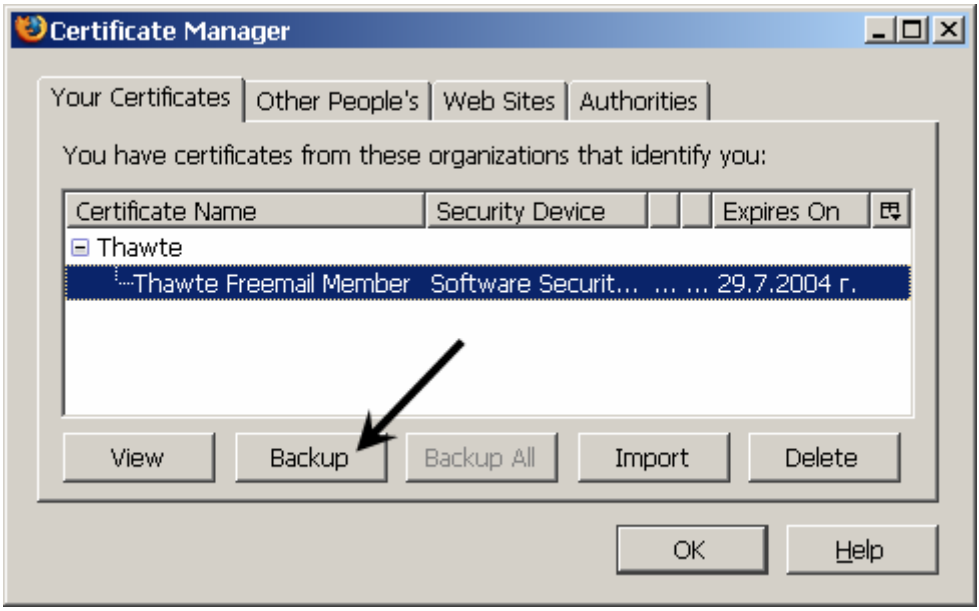
В Internet Explorer експортирането на сертификат и личен ключ става от главното меню чрез командите Tools | Internet Options | Contents | Certificates | Export (фигура 1-9):



Фигура 1-9. Експортиране на сертификат от Internet Explorer

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.

В Netscape и Mozilla можем да експортираме сертификати от хранилището на уеб браузъра чрез командите Edit | Preferences | Privacy & Security | Certificates | Manage Certificates | Backup (фигура 1-10):



Фигура 1-10. Експортиране на сертификат от Mozilla

Файлови формати за съхранение на сертификати

Форматът PKCS#12 (файлове от тип .PFX/.P12) се използва най-често за съхранение на цифров сертификат, придружен от сертификационната си верига и личен ключ, съответстващ на публичния ключ на сертификата.

При съхраняване на X.509 сертификати, за които нямаме съответен личен ключ, по-често се използват други файлови формати. 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 някой сертификационен орган, удостоверяващ, че този публичен ключ е наистина на това лице. CER файловете могат да съдържат пълната сертификационна верига за даден сертификат, но могат и да не я съдържат. Един CER файл може да бъде в двоичен вид или в текстов вид (с кодиране Base64).

Сертификационните органи разпространяват Root-сертификатите си във вид на CER файлове, обикновено от своя уеб сайт.

Смарт картите в уеб браузърите

Уеб браузърите могат да ползват и сертификати, които са съхранени върху смарт карта. За целта те трябва да ги импортират в своята система за управление на сигурността.

Internet Explorer използва вградената в Windows хранилище за сертификати (т. нар. Windows Certificate Store), в което могат да се импортират сертификати от смарт карти.

Mozilla използва свое собствено хранилище за сертификати и позволява в него да се импортират сертификати от смарт карта.

Импортирането става като се използва PKCS#11 имплементацията за съответната смарт карта. Това е библиотека (`.dll` файл под Windows и `.so` файл под UNIX/Linux), който осигурява достъп до информацията и функциите от смарт картата по стандарта PKCS#11. Тази библиотека обикновено се разпространява заедно с драйверите за карточетеца и съответната смарт карта.

1.2.5. Анулирани сертификати

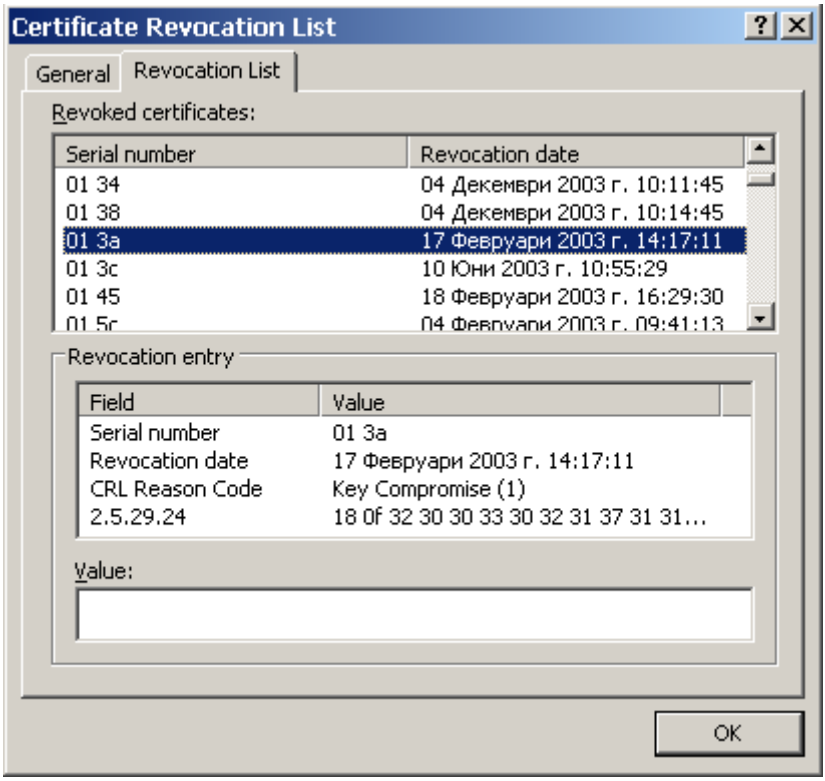
Понякога се случва някое лице или фирма да загуби контрол над собствените си сертификати и съответстващите им лични ключове и те да попаднат в трети лица, които могат евентуално да злоупотребяват с тях. В такива случаи е необходимо тези сертификати да бъдат обявени за невалидни (revoked certificates).

Списъци на анулираните сертификати (CRL)

Сертификационните органи 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 сертификационния орган, който ги издава, и се наричат **списъци на анулираните сертификати** (certificate revocation lists – CRL). В един такъв списък се посочват името на сертификационния орган, който го е издал, датата на издаване, датата на следващото издаване на такъв списък, серийните номера на анулираните сертификати и специфичните времена и причини за това анулиране.

Списъците с анулирани сертификати обикновено се записват във файлове с разширение .CRL и се разпространяват от уеб сайтовете на съответните сертификационни органи. Пример за списък с анулираните сертификати е този на StampIT: http://www.stampit.org/crl/stampit.crl (фигура 1-11):



Фигура 1-11. Списък с анулирани сертификати

Повече информация за CRL списъците може да се намери в тяхната официална спецификация – [RFC 3280, раздел 5].

Проверка на сертификати по OSCP

OCSP (Online Certificate Status Protocol) е протокол за проверка на валидността на сертификат в реално време. Чрез него може да се отправи заявка към сертификационния орган, издал даден сертификат, и да се провери дали в настоящия момент този сертификат е валиден.

Проверката на статуса се осъществява чрез използване на OCSP клиент (OCSP Requestor) и OCSP сървър (OCSP Responder). OCSP клиентът се обръща към OCSP сървъра на сертификационния орган, който дава отговор за статуса на сертификата. Този отговор е електронно подписан със сертификата на OCSP сървъра и съдържа както информация за проверявания сертификат, така и времето на обработка, което служи като доказателство за валидност. OCSP протоколът е описан в [RFC 2560].

1.3. Технологии за цифрово подписване в уеб среда

Да си представим, че разработваме някаква информационна система с уеб-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.

1.3.1. Общи съображения при подписването в уеб среда

За целите на цифровия подпис системата може 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 саморъчно-подписани (self-signed) или да бъдат издадени от някой сертификационен орган.

Кой издава сертификатите?

Ако потребителите саморъчно 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.

Използване на локален сертификационен орган

За да се избегне горният проблем, е възможен друг вариант – лицето, което поддържа системата (системният администратор) да издава цифрови сертификати на потребителите. Той може да осъществи това като създаде локален сертификационен орган или като директно генерира и раздава публичните и личните ключове на потребителите. На практика това е моделът на директно доверие, при който всички потребители се доверяват на една независима трета страна. Този модел работи добре в малки организации, но не е приложим в глобална среда, каквато е Интернет.

Използване на утвърден (глобален) сертификационен орган

Другият подход е да се използват услугите на утвърдените сертификационни органи. Всеки потребител на системата закупува сертификат от някой сертификационен орган и този орган гарантира, че издаденият сертификат е наистина на лицето, което е записано в сертификата. Личният ключ, съответстващ на този сертификат, остава достъпен единствено за неговия собственик и за никой друг (като за по-голяма сигурност може да се съхранява върху смарт карта). Благодарение на това когато даден потребител подпише даден документ със сертификата, издаден му от някой сертификационен орган, 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.

Закупуване на цифров сертификат

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 или смарт карта, съдържащи издадения му сертификат и личния му ключ, защитени с парола за достъп (или PIN код).

Ако сертификационният орган при издаване на сертификат на своя клиент го инсталира директно в неговия уеб 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 се извършва на машината на клиента

Подписването на документи използва 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 уеб приложенията подписването на документи на машината на клиента никак не е лесна задача, защото клиентът разполага единствено със стандартен уеб браузър, който няма вградени функции за подписване на документи при изпращането им (upload) към сървъра.

1.3.2. Цифров подпис в уеб браузъра на клиента

Стандартно със средствата на HTML и JavaScript не можем да подписваме файлове в клиентския уеб браузър. Това е проблем на уеб технологиите, който няма стандартизирано решение, което да се поддържа от всички уеб браузъри. JavaScript по спецификация не поддържа функционалност за работа с цифрови подписи и сертификати и не може да осъществява достъп нито до сертификатите, инсталирани в уеб браузъра на потребителя, нито до външни хранилища за ключове и сертификати.

Някои уеб браузъри (например Mozilla) имат специфични за тях разширения, които позволяват подписването на текст, но не позволяват подписването на файл при изпращането му към сървъра.

Трябва да търсим решение на проблема с подписването на файлове в уеб браузъра на клиента в стандартните и широкоразпространени разширения на уеб технологиите: Macromedia Flash, Java аплети, ActiveX контроли и др. Ще разгледаме предимствата и недостатъците на съществуващите технологии, с които можем да подписваме документи в уеб среда.

Външен, специализиран софтуер

Един възможен подход за подписване на документи на машината на потребителя е всеки потребител да си инсталира за целта някакъв специа-

лизиран софтуер. Това би могло да свърши работа, но има известни проблеми, които правят това решение неудобно.

Единият проблем е, че софтуерът за подписване трябва да има отделни версии за различните операционни системи, които потребителят би могъл да използва. Това не винаги е лесна задача, особено ако трябва да се поддържат голям брой различни платформи.

Има и проблем с поддръжката на различни типове хранилища за сертификати – PFX файлове, смарт карти и т.н. В различните операционни системи достъпът до такива хранилища става по различен начин.

Друг проблем с поддръжката е, че при всяка промяна в софтуера е необходимо да се заставят всички потребители да си изтеглят и инсталират променената версия. Ако потребителите са голям брой, това може да се окаже сериозен проблем.

Следващият немаловажен проблем е интеграцията на такъв софтуер с уеб интерфейса на системата. Това не е никак лесна задача, особено ако трябва да се поддържат различни уеб браузъри. Ако външният софтуер за подписване на документи не е добре интегриран с уеб системата, използването му ще е неудобно за потребителя.

Има и още един проблем: От съображения за сигурност потребителите може 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.

Специализирана ActiveX контрола в Internet Explorer

Да разгледаме един друг подход – ActiveX контролите. Те представляват Windows компоненти, базирани на COM технологията, които имплементират някаква функционалност, имат собствен графичен потребителски интерфейс и могат да се да се вграждат в уеб страници, подобно на обикновени картинки, и да се изпълняват след това вътре в страниците [MSDN ActiveX].

ActiveX контролите могат технически да решат проблема, но не са платформено независими – работят само под Windows. При тях не е проблем да осъществят достъп до хранилището за сертификати на Windows и Internet Explorer (т. нар. Windows Certificate Store), например посредством стандартната Windows библиотека CryptoAPI или чрез специализираната компонента CAPICOM. Предимство на този подход е, че се използва хранилището за сертификати на Windows, а то позволява използването както на PFX файлове, така и на смарт карти и други криптиращи устройства.

Основният проблем при този подход е, че върху друга операционна система, освен Windows, ActiveX контролите просто не могат да работят.

Друг проблем на ActiveX контролите е, че изискват от потребителя да се съгласи да ги инсталира преди да бъдат стартирани за пръв път, а това може да изисква промяна в настройките за сигурността.

.NET Windows Forms контрола в Internet Explorer

.NET Windows Forms контролите в Internet Exoplorer представляват графични компоненти, които могат да имплементират някаква специфична функционалност и да взаимодействат с потребителя чрез собствен графичен потребителски интерфейс [MSDN WinForms]. Те се изпълняват в клиентския уеб браузър, подобно на ActiveX контролите.

Технически .NET Windows Forms контролите могат да решат проблема с подписването, но при някои допълнителни условия. .NET Framework има силно развити средства за работа с цифрови подписи и сертификати и за достъп защитени хранилища, включително и смарт карти, така че това не е проблем. Проблем е невъзможността стандартно .NET контрола, изпълнявана в Internet Explorer, да чете от файловата система, а това е необходимо за да бъде прочетен файлът, който трябва да бъде подписан. Остава възможността потребителят да промени настройките на .NET сигурността (т. нар. .NET Security Policy) за да разреши на съответната контрола достъп до файловата система.

Най-големият проблем при подхода с Windows Forms контролите е, че те са силно платформено-зависими. Те работят само под Windows и то само с уеб браузър Internet Explorer 5.5 или по-висока версия и допълнително изискват на машината на клиента да има инсталиран Microsoft .NET Framework. Всички тези изисквания ги правят подходящи за използване в корпоративна среда, където администраторът може да осигури наличието на Internet Explorer, .NET Framework и настройките за сигурността, но в Интернет среда това решение няма да работи.

Контролата CAPICOM в Internet Explorer

В Windows среда, ако се използва Microsoft Internet Explorer, може да се инсталира ActiveX контролата CAPICOM. Тя представлява COM обвивка на Microsoft CryptoAPI и предоставя обектен модел за достъп криптографската функционалност на Windows [Lambert, 2001].

След като се инсталира, CAPICOM тя може да се използва през VBScript за подписване на текстови данни, например уеб форми. Ето прост пример как можем да подписваме текст чрез CAPICOM и VBScript код, вграден в уеб страница:

Sign-Data-with-CAPICOM-in-IE.html

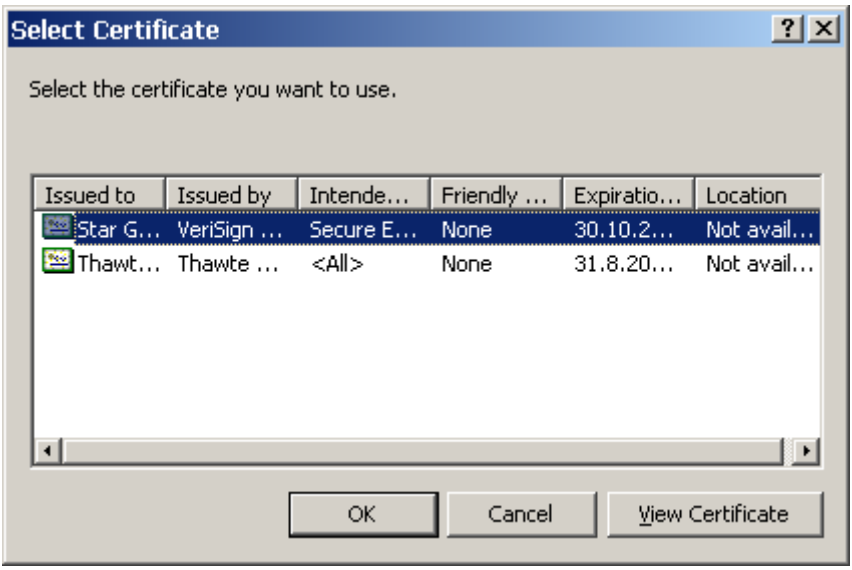
```
<script language="VBScript">
    TextToSign = "This text will be signed."
    document.write("<b>Text:</b> " + TextToSign + "<br>")

    On Error Resume Next
    Set SignedData = CreateObject("CAPICOM.SignedData")

    If Err.Number = 0 Then
        SignedData.Content = TextToSign
        Signature = SignedData.Sign(Nothing, True)
        document.write("<b>Signature:</b> " + Signature)
    Else
```

```
        document.write("CAPICOM not installed!")
    End If
</script>
```

При отваряне на примерния HTML документ с Internet Explorer, се изпълнява скриптът и ако ActiveX контролата CAPICOM е достъпна, се показва диалог за избор на сертификат (фигура 1-12):

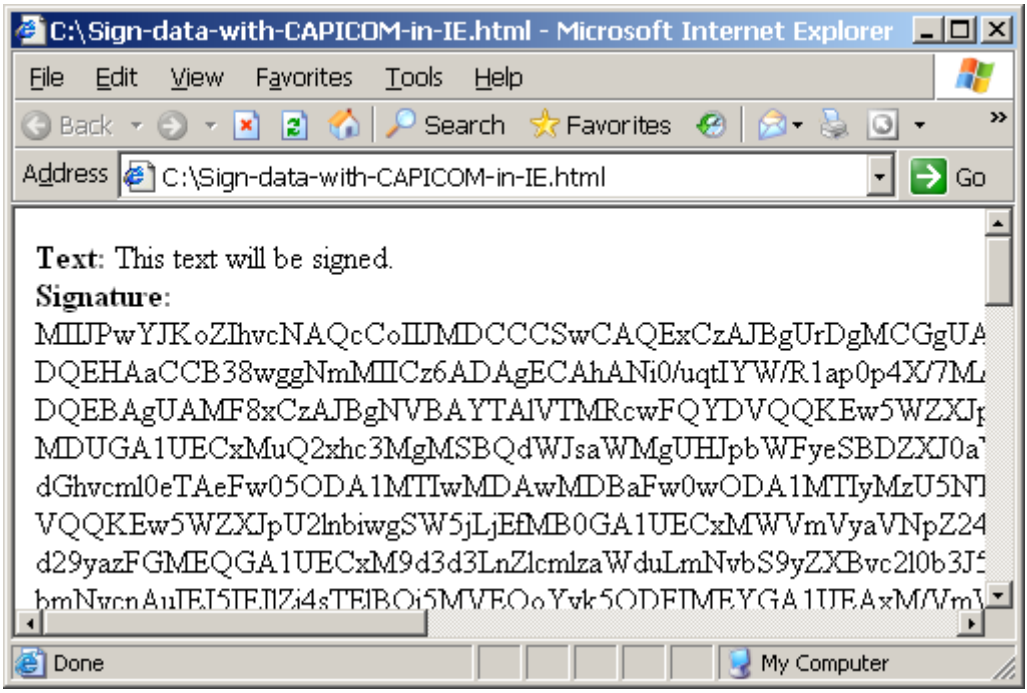


Фигура 1-12. Диалог за избор на сертификат в Internet Explorer

При успешен избор на сертификат и успешна автентикация за достъп до съответния личен ключ, подаденият текст се подписва. Полученият цифров подпис съдържа пълната сертификационна верига и цифровия подпис във формат PKCS#7, записани като текст с Base64 кодиране (фигура 1-13). Без проблеми се поддържа и подписване със смарт карта (понеже сертификатите от смарт картите могат да се импортират във Windows Certificate Store).

При подписването с CAPICOM има няколко проблема. Най-сериозният от тях е, че технологията работи само под Windows с уеб браузър Microsoft Internet Explorer. Под други операционни системи и уеб браузъри CAPICOM не е достъпна. Допълнително изискване е нуждата от еднократно инсталиране на CAPICOM ActiveX контролата върху клиентската машина, което може да създаде трудности.

Още един проблем е, че за да се подпише даден файл, той трябва да бъде прочетен, а VBScript не позволява достъп до файловата система.



Фигура 1-13. Подписани данни с CAPICOM в Internet Explorer

Методът crypto.signText() в Netscape и Mozilla

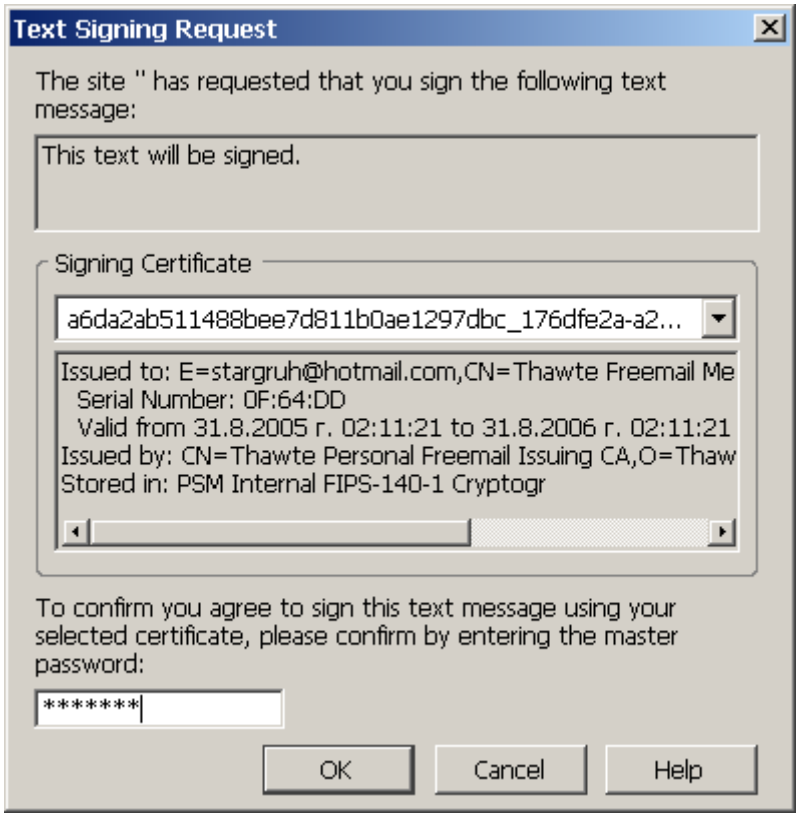
По-новите версии на уеб браузърите Mozilla и Netscape имат вградени функции за подписване на текст. Те поддържат JavaScript функцията **`crypto.signText(text, certificateSelectionMode)`**, която подписва цифрово даден символен низ [Netscape, 1997]. Ето примерен JavaScript фрагмент, който извършва такова подписване:

Sign-Data-in-Mozilla.html

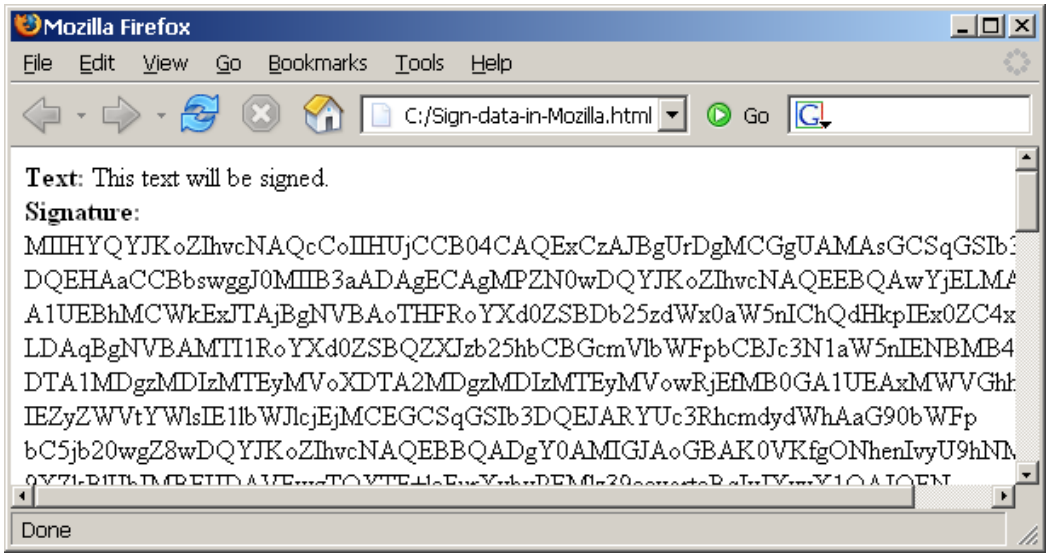
```
<script language="JavaScript">
    textToSign = 'This text will be signed.';
    document.write('<b>Text:</b> ' + textToSign + '\n<br>\n');
    signature = crypto.signText(textToSign, "ask");
    document.write('<b>Signature:</b> ' + signature);
</script>
```

При отваряне на примерния HTML документ с Mozilla, се изпълнява JavaScript кодът и потребителят се приканва да избере сертификат и парола за достъп до него (фигура 1-14).

След успешен избор на сертификат и въвеждане на паролата за достъп до него подаденият текст се подписва и се отпечатва резултатът. Получената сигнатура съдържа пълната сертификационна верига и цифровия подпис във формат PKCS#7, кодирани с Base64 (фигура 1-15).



Фигура 1-14. Диалог за избор на сертификат в Mozilla



Фигура 1-15. Подписани данни с crypto.Sign() в Internet Explorer

Предимство на тази технологията е, че не изисква инсталиране на никакъв допълнителен софтуер. Използва се хранилището за сертификати на уеб браузъра, а това дава възможност да се използват и смарт карти. Повече информация за функцията **`crypto.SignText()`** може да се намери в документацията на Netscape за JavaScript: http://devedge-temp.mozilla.org/library/manuals/2000/javascript/1.3/reference/window.html#1202035.

Основният проблем на тази технология е, че работи само с уеб браузърите Mozilla и Netscape (под всички платформи, за които се предлагат – Linux, Windows, Solaris и др.), но не се поддържа от Internet Explorer.

Другият проблем е, че за да се подпише файл, той трябва да се прочете, а това не може да стане с JavaScript. По тази причина с тези технология могат да се подписват само уеб форми или части от тях, но не и файлове.

Macromedia Flash приложение

Технологията Macromedia Flash представлява разширение на стандартните уеб браузъри, което позволява вмъкване на мултимедийно съдържание в уеб страниците. С Flash технологията могат да се създават не само анимации, но интерактивни мултимедийни приложения с графичен потребителски интерфейс, които се изпълняват в клиентския уеб браузър. Flash приложенията могат да съдържат графика, текст, форми и други елементи, които могат да се управляват с езика ActionScript.

За съжаление стандартните библиотеки на Flash не поддържат работа с цифрови подписи и сертификати, а освен това Flash няма достъп до локалната файлова система, а това е необходимо, защото за да бъде подписан един файл, съдържанието му трябва да бъде прочетено.

Комбинация между plug-in за Netscape/Mozilla и ActiveX контрола за Internet Explorer

За подписване на уеб форми може да се използва комбинация от технологии, които са достъпни за различните уеб браузъри, например подписване с **`crypto.SignText()`** в Mozilla и с CAPICOM в Internet Explorer. При този вариант може да бъде подписана уеб форма или част от уеб форма, но няма как да бъде подписан файл, защото JavaScript и VBScript нямат достъп до файловата система.

Подобен подходът е реализиран и в проекта „Secure Form Signing”, реализиран от специалисти от университета за наука и технологии в Хонг Конг (http://www.cyber.ust.hk/projects/eForm/). В проекта са комбинирани специално разработен plug-in в браузъра Netscape и ActiveX контрола в Internet Explorer.

При някои комерсиални решения са използвани и други комбинации от технологии, специфични за различни уеб браузъри и платформи, но при повечето реализации се подписват уеб форми, а не файлове. Нека си припомним, че нашата цел е да разработим технология за подписване на файл по време на изпращането му от уеб браузъра към уеб сървъра.

Специализиран Java аплет

След като разгледахме възможните алтернативи, остана една, последна, възможност – да се използват Java аплети.

Java аплетите са стандартни разширения на уеб технологиите и имат предимството, че могат да работят на всички по-известни уеб 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, които работят с по-високи права.

Java платформата поддържа стандартно работа с цифрови подписи и сертификати, а от версия 1.5 е имплементирана и стандартна поддръжка на смарт карти.

Предимствата на аплетите са, че работят върху всички по-известни уеб браузъри и операционни системи. Недостатъците им са, че изискват предварителна инсталация на Java Plug-In, който е обикновено се доставя отделно от браузъра. Ако се използват подписани аплети, е необходимо потребителят да потвърди доверието си към тях, за да им се отпуснат права за достъп до файловата система. Това може да е проблем за някои потребители.

Независимо от изброените недостатъци, Java аплетите са единствената технология, която може да реши по платформено-независим начин проблема с цифрово подписване на документи в уеб браузъра на потребителя. По тази причина ще изберем тази технология за реализацията на нашата система за цифрово подписване на документи в уеб среда.

Технологията на Java аплетите

Java аплетите представляват компилирани програми на Java, които се вграждат като обекти в HTML документи и се изпълняват от уеб браузъра, когато тези документи бъдат отворени. Вграждането на аплет в една уеб 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 [Sun, 1995].

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 уеб 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.

Java аплетите и сигурността

За да се осигури безопасността на потребителя, на аплетите не е позволено да извършват операции, които биха могли да осъществят достъп до потребителска информацията от машината, на която се изпълняват. Един аплет стандартно няма достъп до локалната файлова система, а това затруднява използването на тази технология за целите на цифрово подписване на документи.

Подписани Java аплети

Подписаните Java аплети предоставят механизъм за повишаване на правата, с които се изпълняват обикновените Java аплети. Те, също като обикновените аплети, представляват JAR архиви, които съдържат съвкупност от компилирани Java класове, но в допълнение към тях съдържат и цифрови подписи на всички тези класове, както и цифров сертификат (заедно с пълната му сертификационна верига), с който тези подписи могат да се верифицират.

При изпълнението на подписан аплет уеб 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. За улеснение на потребителя Java Plug-In автоматично валидира сертификатите на подписаните аплети и предупреждава за рисковете от изпълнението им. Доверяването на даден подписан аплет може да се извърши както еднократно, така и перманентно.

НАПС
НАЦИОНАЛНА АКАДЕМИЯ ПО РАЗРАБОТКА НА СОФТУЕР

НАЦИОНАЛНА АКАДЕМИЯ ПО РАЗРАБОТКА НА СОФТУЕР

Лекторите

» Светлин Наков е преподавател по съвременни софтуерни технологии в СУ "Св. Климент Охридски".

Той е автор на десетки научни и технически публикации и няколко книги, свързани с разработката на софтуер, заради което е търсен лектор и консултант.

През 2004 г. получава наградата "**Джон Атанасов**" от президента на България Георги Първанов за приноса му към развитието на информационните технологии и информационното общество.

» Мартин Кулов е изпълнителен директор във фирма "Код Атест", където разработва проекти за повишаване качеството на софтуерните продукти в България чрез автоматизация на процесите и внедряване на системи за управление на качеството.

Мартин е опитен лектор и сертифициран от Майкрософт разработчик по програмите MCSD и MCSD.NET.

» Други инструктори с опит като преподаватели и програмисти.

Академията

» **Национална академия по разработка на софтуер** (**НАПС**) е център за професионално обучение на софтуерни специалисти.

» **НАПС** провежда задълбочени курсове по разработка на софтуер и съвременни софтуерни технологии.

» Предлагани специалности:

.NET Enterprise Developer

Java Enterprise Developer

» **Качествено обучение** с много **практически упражнения**

» Завършвате само за 3 месеца.

» **Гарантирана работа** след успешно завършване!

» Професионална сертификация!

» **БЕЗПЛАТНО!**

Учите безплатно, плащате след като завършите и започнете работа.

Стипендии от софтуерни фирми.

http://academy.devbg.org

Глава 2. Цифрови подписи и сертификати в Java

При реализацията на Java приложения, които работят с цифрови подписи, сертификати и смарт карти, се използват специфични Java библиотеки. Тези библиотеки ще са ни необходими при реализацията на нашата система за подписване на документи в уеб среда и затова сега ще ги разгледаме в детайли.

2.1. Java Cryptography Architecture и Java Cryptography Extension

За целите на подписването на документи, проверката на цифрови подписи и управлението на цифрови сертификати в Java платформата се използва Java Cryptography Architecture (JCA). JCA представлява спецификация, която предоставя на програмистите достъп някои до криптографски услуги, работа с цифрови подписи и сертификати [JCA, 2004].

Наред с JCA за извършване на някои криптографски операции, като шифриране, дешифриране и генериране на ключове и др., се използва Java Cryptography Extension (JCE). JCE предоставя общ framework за имплементация и употреба на криптиращи алгоритми в Java [JCE, 2004].

От архитектурна гледна точка JCA и JCE са така проектирани, че да позволяват използването на различни имплементации на различни услуги от различни софтуерни доставчици. При работата с JCA и JCE 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 услуги и алгоритми, които са достъпни по име.

Спецификациите JCA и JCE установяват стандарти за различните типове криптографски услуги и специфицират начините на използване на криптографските алгоритми. Реализацията на самите алгоритми е оставена на софтуерните доставчици.

В Java 2 платформата стандартно са реализирани голям брой криптографски алгоритми и услуги, но JCA и JCE и позволяват добавяне и на допълнителни собствени имплементации.

2.1.1. Основни класове за работа с цифрови подписи и сертификати

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. Стандартната имплементация на JCA и JCE предоставя средства за работа с X.509 сертификати и сертификационни вериги, подписване с различни алгоритми (RSA, DSA, ECDSA и др.), достъп до PKCS#12

хранилища, алгоритми за проверка на цифрови подписи (напр. PKIX) и достъп до смарт карти (в Java 1.5 и по-високите версии).

Най-важните класове и интерфейси за работа с цифрови подписи и сертификати се намират в пакетите **java.security** и **java.security.cert**. Ще дадем кратко описание на най-важните от тях, които са свързани с имплементацията на нашата система за цифрово подписване на документи в уеб среда.

java.security.PublicKey

Интерфейсът **java.security.PublicKey** представлява публичен ключ. Той може да бъде извлечен от даден цифров сертификат и да се използва при проверката на цифрови подписи.

java.security.PrivateKey

Интерфейсът **java.security.PrivateKey** представлява личен ключ. Той може да бъде извлечен от защитено хранилище (**KeyStore**) и да се използва за създаване на цифров подпис.

PrivateKey инстанциите реално могат и да не съдържат ключа, а само да предоставят интерфейс за достъп до него. Например ако даден личен ключ е записан върху смарт карта, той не може да бъде извлечен от нея, но може да бъде използван за подписване или криптиране посредством този интерфейс.

java.security.cert.Certificate

Абстрактният клас **java.security.cert.Certificate** е базов за всички класове, които представляват цифрови сертификати. Съдържа в себе си публичен ключ и информация за лицето, което е негов собственик, както и информация за издателя на сертификата. 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

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

Стандартно е реализиран и метод **verify()**, който проверява дали сертификатът е подписан от даден публичен ключ. Тази възможност може да се използва за проверка дали даден сертификат е подписан от даден друг сертификат (т. е. дали вторият е издател на първия).

java.security.KeyStore

Класът **java.security.KeyStore** се използва за достъп до защитени хранилища за ключове и сертификати. Той предоставя методи за зареждане на хранилище от поток или от смарт карта, 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.

Поддържат се няколко формата на хранилища – PFX (съгласно PKCS#12 стандарта), JKS (Java Key Store) и смарт карта (по стандарта PKCS#11). При създаване на обект от класа **KeyStore** форматът на хранилището се задава от параметър. Възможни стойности в JDK 1.5 са "JKS", "PKCS12" и "PKCS11".

Обектите, записани в едно хранилище, са достъпни по име (alias), а личните ключове са достъпни по име и парола. Под едно и също име (alias) в дадено хранилище може да има записани едновременно както сертификат, така и сертификационна верига и личен ключ.

Ето един пример за зареждане на хранилище от файл и отпечатване на всички сертификати, записани в него:

```
KeyStore keyStore = KeyStore.getInstance("PKCS12");
FileInputStream stream = new FileInputStream("example.pfx");
try {
    String keyStorePassword = "**********";
    keyStore.load(stream, keyStorePassword.toCharArray());
} finally {
    stream.close();
}

Enumeration aliasesEnum = keyStore.aliases();
while (aliasesEnum.hasMoreElements()) {
    String alias = (String)aliasesEnum.nextElement();
    System.out.println("Alias: " + alias);
    X509Certificate cert = (X509Certificate) keyStore.getCertificate(alias);
    System.out.println("Certificate: " + cert);
}
```

java.security.Signature

Класът **java.security.Signature** предоставя функционалност за подписване на документи и проверка на цифрови подписи (сигнатури).

При създаването на инстанция на класа **Signature** се задава име на алгоритъм за цифрови подписи, който да бъде използван. Името на алгоритъма се образува от името на алгоритъма за хеширане и името на алгоритъма за подписване (криптиране) на хеш-стойността. Поддържат се различни алгоритми като **SHA1withRSA**, **SHA1withDSA**, **MD5withRSA** и др.

Полагане на цифров подпис с Java

За подписване на съобщения се използва класът **Signature**. Използват се методите му **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.

Ето примерен код за подписване на дадено съобщение с даден личен ключ:

```
public static byte[] signDocument(byte[] aDocument, PrivateKey aPrivateKey)
throws GeneralSecurityException {
    Signature signatureAlgorithm = Signature.getInstance("SHA1withRSA");
    signatureAlgorithm.initSign(aPrivateKey);
    signatureAlgorithm.update(aDocument);
    byte[] signature = signatureAlgorithm.sign();
    return signature;
}
```

Верификация на цифров подпис с Java

За верификация на цифров подпис се използва отново класът **Signature**. Използват се методите му **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.

Ето примерен код за верификация на сигнатурата на даден документ по даден публичен ключ:

```
public static boolean verifyDocumentSignature(byte[] aDocument,
    PublicKey aPublicKey, byte[] aSignature)
throws GeneralSecurityException {
    Signature signatureAlgorithm = Signature.getInstance("SHA1withRSA");
    signatureAlgorithm.initVerify(aPublicKey);
    signatureAlgorithm.update(aDocument);
    boolean valid = signatureAlgorithm.verify(aSignature);
    return valid;
}
```

java.security.cert.CertificateFactory

Класът **java.security.cert.CertificateFactory** предоставя функционалност за зареждане на сертификати, сертификационни вериги и CRL списъци от поток.

Зареждане на сертификат от поток

За прочитане на сертификат от поток или файл се използва методът **CertificateFactory.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).

Ето примерен код за прочитане на сертификат от стандартен .CER файл:

```
CertificateFactory certFactory = CertificateFactory.getInstance("X.509");
FileInputStream stream = new FileInputStream("VeriSign-Class-1-Root-CA.cer");
try {
    X509Certificate cert = (X509Certificate)
```

```
certFactory.generateCertificate(stream);
    // Use the X.509 certificate here ...
} finally {
    stream.close();
}
```

Зареждане на сертификационна верига

За прочитане на сертификационни вериги от поток се използва методът **`CertificateFactory.generateCertPath()`**. Той приема като параметър 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 от веригата не се запазва.

Ето примерен код за прочитане на сертификационна верига, съставена от X.509 сертификати, от бинарен файл с формат ASN.1 DER:

```
CertificateFactory certFactory = CertificateFactory.getInstance("X.509");
FileInputStream stream = new FileInputStream("chain-asn.1-der.bin");
try {
    CertPath certChain = certFactory.generateCertPath(stream, "PkiPath");
    // Use the certification chain here ...
} finally {
    stream.close();
}
```

Изключенията в JCA

При извършване на операции, свързани с криптография, класовете от JCA хвърлят изключения от тип **`java.security.GeneralSecurityException`** и **`java.security.cert.CertificateException`** (при работа със сертификати) за да съобщят за проблем, възникнал по време на работа.

2.1.2. Директна верификация на сертификати с Java

Както знаем, за установяване на доверие между непознати страни съществуват различни модели на доверие и по тази причина съществуват и различни начини за проверка на валидността на даден сертификат. Вече се запознахме с два от тях – директна верификация и верификация на сертификационна верига.

Директната верификация на сертификат в Java се извършва тривиално:

- проверява се срокът на валидност на сертификата;
- проверява се дали сертификатът е директно подписан от сертификат, на който имаме доверие.

Ето примерен сорс код, който верифицира даден сертификат по някакво предварително налично множество от доверени сертификати:

```
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
        }
    }

    // Cert. is not signed by any of the trusted certificates --> it is invalid
    throw new CertificateValidationException(
        "Can not find trusted parent certificate.");
}
```

Методът приключва нормално ако сертификатът е валиден или завършва с изключение, съдържащо причината за неуспех на верификацията.

2.1.3. Верификация на сертификационни вериги с Java

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 [Mullan, 2003].

Нека преди да разгледаме как се построяват и верифицират сертификационни вериги да дадем описание на най-важните класове от Java Certification Path API.

java.security.cert.CertPath

Класът **`java.security.cert.CertPath`** представлява сертификационна верига (наредена последователност от сертификати).

По конвенция една правилно построена сертификационна верига започва от някакъв клиентски сертификат, следван нула или повече сертификата на междинни сертификационни органи и завършва с Root-сертификат на някой сертификационен орган 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

Класът **java.security.cert.TrustAnchor** представлява крайна точка на доверие при верификацията на сертификационни вериги. Той се състои от публичен ключ, име на доверен сертификационен орган и съвкупност от ограничения, с които се задава множеството от пътища, което може да бъде проверявано. Обикновено тази съвкупност от ограничения се пропуска и по подразбиране няма такива ограничения.

За простота можем да разглеждаме един **TrustAnchor** обект като доверен Root-сертификат, който се използва при построяване и верификация на сертификационни вериги.

java.security.cert.PKIXParameters

java.security.cert.PKIXParameters е 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

Най-важният клас от Java Certification Path API е **java.security.cert.CertPathValidator**. Той предоставя функционалност за верификация на сертификационни вериги. Сертификационните вериги могат да бъдат съставени както от X.509 сертификати, така и от PGP или друг тип сертификати. За верификацията могат да се използват различни алгоритми в зависимост от модела на доверие, който се използва.

В JDK 1.5 е имплементиран стандартно единствено алгоритъмът **PKIX**, който верифицира сертификационна верига съгласно организацията на инфраструктурата на публичния ключ (PKI) е описан в [RFC 3280, раздел 6]. Този алгоритъм проверява валидността на отделните сертификати и връзката между тях, като поддържа и използване на CRL списъци.

Верификация на сертификационни вериги

За верификация на сертификационна верига със стандартните средства на Java платформата се използва класът **CertPathValidator**. При създаването на инстанция на този клас се задава алгоритъма за верификация, който да се използва. Стандартно се поддържа единствено алгоритъмът **PKIX**.

При проверка на сертификационни вериги алгоритъмът **PKIX** започва от първия сертификат във веригата (сертификата на потребителя), продължава със следващия, после със следващия и т.н. и накрая 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 някой сертификационен орган, 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 последният сертификат. В противен случай е възможно проверката да пропадне, независимо, че веригата е валидна.

Методът за верификация на сертификационна верига **`CertPathValidator.validate()`** изисква като входни параметри веригата, която ще се проверява (от която е премахнат последният сертификат) и параметрите на алгоритъма за проверка. За алгоритъма **`PKIX`** тези параметри представляват обект от класа **`PKIXParameters`**, който съдържа списъка от доверени Root-сертификати, в някои от които може да завършва веригата.

Ето пример за верификация на сертификационна верига с Java Certification Path API и алгоритъма **`PKIX`**:

```
public static void verifyCertificationChain(CertPath aCertChain,
    X509Certificate[] aTrustedCACertificates)
throws GeneralSecurityException {
    // 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 certification chain validator and a set of parameters for it
    PKIXParameters certPathValidatorParams = new PKIXParameters(trustAnchors);
    certPathValidatorParams.setRevocationEnabled(false);
    CertPathValidator chainValidator = CertPathValidator.getInstance("PKIX");

    // Remove the root CA certificate from the end of the chain
    CertPath certChainForValidation = removeLastCertFromCertChain(aCertChain);

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

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("X.509");
    CertPath certChainWithoutLastCert = cf.generateCertPath(certsWithoutLast);
    return certChainWithoutLastCert;
}
```

В примера се верифицира сертификационна верига по дадена съвкупност от доверени Root-сертификати без да се използват CRL списъци. Необходимо е сертификационната верига да е налична, т. е. тя не се построява от наличните сертификати, а трябва да е предварително построена.

Преди извикване на метода **`CertPathValidator.validate()`**, се построява списък от **`TrustAnchor`** обекти, съдържащи на доверените Root-сертификати задава се да не се ползват CRL списъци и се изтрива последния сертификат от веригата, която ще се проверява (вече обяснихме защо).

Ако верификацията е успешна, методът не връща нищо, а ако веригата е невалидна, се хвърля изключение **`CertPathValidatorException`**, в което се съдържа описание на проблема и номера на невалидния сертификат (ако има такъв).

2.2. Достъп до смарт карти от Java

От версия 1.5 на Java 2 платформата стандартно се поддържа достъп до смарт карти посредством интерфейса PKCS#11 (Cryptographic Token Interface Standard). Връзката с него се осъществява чрез доставчика на криптографски услуги "Sun PKCS#11 Provider".

Програмен интерфейс PKCS#11

Както знаем, PKCS#11 е стандарт, който предоставя програмен интерфейс за достъп до смарт карти и други криптоустройства (security tokens), и най-често е имплементиран в динамична библиотека (**`.dll`** или **`.so`** файл), която се доставя заедно с драйверите за смарт картата.

Например, ако използваме ISO 7816 съвместима Utimaco Safeware смарт карта, PKCS#11 имплементацията за тази карта се съдържа в софтуера "Utimaco SafeGuard Smartcard Provider", който идва заедно с картата. След инсталиране на този софтуер под Windows XP, библиотеката, която имплементира PKCS#11 е файлът **`C:\WINDOWS\system32\pkcs201n.dll`**. При други смарт карти и други операционни системи тази библиотека се намира съответно в друг файл, но би трябвало да е налична.

Sun PKCS#11 Provider

В Java 1.5 достъп за до смарт карти се използва доставчикът на криптографски услуги "Sun PKCS#11 Provider".

За разлика от повечето JCA доставчици, Sun PKCS#11 Provider не имплементира директно криптографска функционалност, а разчита на машинно-зависима (native) PKCS#11 имплементация, към която пренасочва всички операции. Тази имплементация трябва да е реализирана като **`.dll`** файл в Windows или **`.so`** файл в UNIX и Linux [Sun PKCS#11].

Например ако използваме Utimaco SafeGuard Smartcard Provider за Windows, PKCS#11 имплементацията е библиотеката **`pkcs201n.dll`**. Ако използваме друга смарт карта и други драйвери, файлът е друг.

Конфигуриране на Sun PKCS#11 Provider

За да бъде използван "Sun PKCS#11 Provider", той първо трябва да се регистрира като доставчик на криптографски услуги в JCA (Java

Cryptography Architecture). Регистрацията може да стане статично или динамично (по време на изпълнение).

Статична регистрация на Sun PKCS#11 Provider

Статичната регистрация изисква да се промени файлът **%JAVA_HOME%/lib/security/java.security** и в него да се добави още един доставчик на криптографски услуги, например по следния начин:

```
# Configuration for security providers 1..6 ommitted
security.provider.7=sun.security.pkcs11.SunPKCS11 C:\smartcards\config\pkcs11.cfg
```

Посоченият файл **pkcs11.cfg** трябва да съдържа настройките на Sun PKCS#11 Provider. Той представлява текстов файл, който описва някои параметри, като например пътя до PKCS#11 библиотеката.

Динамична регистрация на Sun PKCS#11 Provider

При динамична регистрация на "Sun PKCS#11 Provider" трябва да се инстанцира класът **sun.security.pkcs11.SunPKCS11** като му се подаде за параметър конфигурационният файл, от който да прочете настройките си, и след това да се регистрира в JCA. Ето пример как може да стане това:

```
String pkcs11ConfigFile = "c:\\smartcards\\config\\pkcs11.cfg";
Provider pkcs11Provider = new sun.security.pkcs11.SunPKCS11(pkcs11ConfigFile);
Security.addProvider(pkcs11Provider);
```

Конфигурационен файл на Sun PKCS#11 Provider

И при статичната и при динамичната регистрация е необходим конфигурационен файл, от който класът **SunPKCS11** да прочете пътя до библиотеката, която имплементира PKCS#11 стандарта. Ето пример за такъв конфигурационен файл:

pkcs11.cfg

```
name = SmartCard
library = c:\windows\system32\pkcs201n.dll
```

Както се вижда, файлът съдържа две настройки: **name** и **library**. Стойността на параметъра **name** се използва при образуване на името за инстанцията на PKCS#11 доставчика в JCA, а параметърът **library** задава пътя до библиотеката, която имплементира PKCS#11. Ако трябва да се работи с няколко смарт карти едновременно Sun PKCS#11 Provider, трябва да се регистрира няколко пъти с различни имена.

Конфигурационният файл може да задава различни параметри, които са описани в документацията, но задължителни са само **name** и **library**.

Използване на Sun PKCS#11 Provider без конфигурационен файл

Ако не искаме да използваме външен конфигурационен файл, можем да зададем настройките на Sun PKCS#11 Provider динамично чрез поток. Ето пример как може да се направи това:

```
String pkcs11config =
    "name = SmartCard\n" +
    "library = c:\\windows\\system32\\pkcs201n.dll";
byte[] pkcs11configBytes = pkcs11config.getBytes();
ByteArrayInputStream configStream = new ByteArrayInputStream(pkcs11configBytes);
Provider pkcs11Provider = new sun.security.pkcs11.SunPKCS11(configStream);
Security.addProvider(pkcs11Provider);
```

Извличане на KeyStore от смарт карта

След като сме регистрирали и конфигурирали успешно Sun PKCS#11 Provider, можем да го използваме за да извличаме сертификати и ключове от смарт карта. Това става посредством стандартния в Java клас за достъп до хранилища **`java.security.KeyStore`**.

Ето пример как можем да установим достъп защитено до хранилище за ключове и сертификати, съхранявано върху смарт карта:

```
char[] pin = {'1', '2', '3', '4'};
KeyStore smartCardKeyStore = KeyStore.getInstance("PKCS11");
smartCardKeyStore.load(null, pin);
```

Посоченият сорс код очаква, че Sun PKCS#11 Provider е бил успешно регистриран и конфигуриран. За прочитането на хранилище от смарт карта е необходимо да бъде посочен PIN кодът за достъп до картата.

Извличане на сертификат и личен ключ от смарт карта

След като сме установили достъп до хранилището на смарт картата, можем да извличаме от нея ключове и сертификати, както от обикновено хранилище. Всички ключове, сертификати и сертификационни вериги са записани под дадено име (alias) в хранилището. Имената могат да се извличат чрез итератор.

Ето един пример, в който се извличат и отпечатват всички сертификати от дадено хранилище заедно с информация за личните им ключове:

```
KeyStore keyStore = ...;

Enumeration aliasesEnum = keyStore.aliases();
while (aliasesEnum.hasMoreElements()) {
    String alias = (String)aliasesEnum.nextElement();
    System.out.println("Alias: " + alias);
    X509Certificate cert = (X509Certificate) keyStore.getCertificate(alias);
    System.out.println("Certificate: " + cert);
    PrivateKey privateKey = (PrivateKey) keyStore.getKey(alias, null);
    System.out.println("Private key: " + privateKey);
}
```

Примерът работи както за обикновени хранилища, така и за хранилища, разположени върху смарт карта. В случая за смарт карта за достъп до личните ключове не се изисква парола, защото PIN кодът се изпраща преди това при инстанцирането на **`KeyStore`** обекта. Затова в примера за парола се подава стойност **`null`**.

На пръв поглед изглежда, че личните ключове могат да се извличат от смарт картата, но на практика това не е така. Смарт картите не позволяват извличането на ключове, а само индиректен достъп до тях с цел подписване, верификация на подпис, шифриране и дешифриране. В горния пример не се извлича личният ключ, а само интерфейс за достъп до него.

Подписване на данни със смарт карта

След като е извлечен интерфейсът на даден личен ключ от смарт картата, той може да се използва за подписване на данни, както всеки друг личен ключ. Реално подписването става като се изчисли предварително хеш стойността на документа за подписване и се подаде тази хеш стойност на смарт картата за да го подпише тя със своя криптопроцесор. При успех картата връща изчислената сигнатура в резултат от подписването на хеш стойността. Така личният ключ не се излага на рискове, защото остава в тайна, скрит някъде в картата. Ето примерен код за подписване на данни по даден интерфейс към личен ключ, извлечен от смарт карта:

```
private static byte[] signDocument(byte[] aDocument, PrivateKey aPrivateKey)
throws GeneralSecurityException {
    Signature signatureAlgorithm = Signature.getInstance("SHA1withRSA");
    signatureAlgorithm.initSign(aPrivateKey);
    signatureAlgorithm.update(aDocument);
    byte[] digitalSignature = signatureAlgorithm.sign();
    return digitalSignature;
}
```

НАПС
НАЦИОНАЛНА АКАДЕМИЯ ПО РАЗРАБОТКА НА СОФТУЕР

НАЦИОНАЛНА АКАДЕМИЯ ПО РАЗРАБОТКА НА СОФТУЕР

Лекторите

» Светлин Наков е преподавател по съвременни софтуерни технологии в СУ "Св. Климент Охридски".

Той е автор на десетки научни и технически публикации и няколко книги, свързани с разработката на софтуер, заради което е търсен лектор и консултант.

През 2004 г. получава наградата "**Джон Атанасов**" от президента на България Георги Първанов за приноса му към развитието на информационните технологии и информационното общество.

» Мартин Кулов е изпълнителен директор във фирма "Код Атест", където разработва проекти за повишаване качеството на софтуерните продукти в България чрез автоматизация на процесите и внедряване на системи за управление на качеството.

Мартин е опитен лектор и сертифициран от Майкрософт разработчик по програмите MCSD и MCSD.NET.

» Други инструктори с опит като преподаватели и програмисти.

Академията

» **Национална академия по разработка на софтуер** (**НАПС**) е център за професионално обучение на софтуерни специалисти.

» **НАПС** провежда задълбочени курсове по разработка на софтуер и съвременни софтуерни технологии.

» Предлагани специалности:

.NET Enterprise Developer

Java Enterprise Developer

» **Качествено обучение** с много **практически упражнения**

» Завършвате само за 3 месеца.

» **Гарантирана работа** след успешно завършване!

» Професионална сертификация!

» **БЕЗПЛАТНО!**

Учите безплатно, плащате след като завършите и започнете работа.

Стипендии от софтуерни фирми.

http://academy.devbg.org

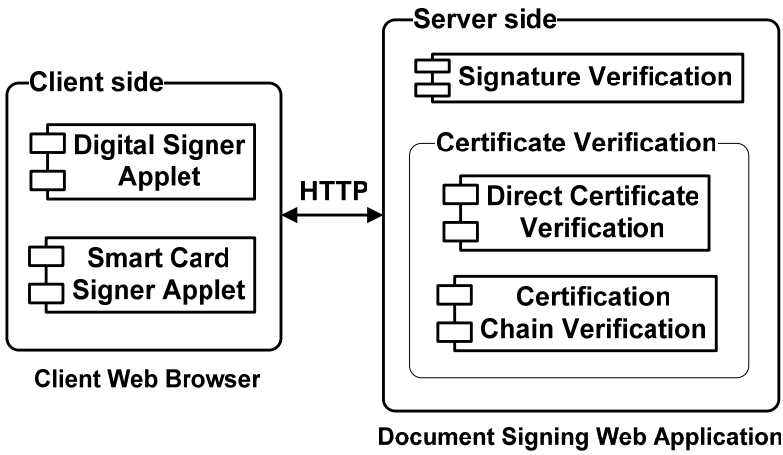
Глава 3. Проектиране на система за цифрово подписване в уеб среда

До момента разгледахме в технологията на цифровия подпис, инфраструктурата на публичния ключ и средствата на Java платформата за работа с цифрови подписи, сертификати и смарт карти. Направихме и анализ на съществуващите технологии за използване на цифров подпис в уеб и стигнахме до решението да използваме Java аплет за подписване на документи в уеб браузъра на потребителя преди изпращането им към сървъра. За да е напълно функционална нашата система за цифрово подписване на документи в уеб среда, ще добавим към нея и модул за верификация на цифровите подписи и сертификати, получени от клиентския уеб браузър.

Нека разгледаме в детайли проблемите, свързани с реализацията на такава система. Ще започнем от нейната архитектура и ще продължим по-нататък с конкретната имплементация.

3.1. Архитектура на системата

Системата за цифрово подписване на документи в уеб среда и верификация на цифрови подписи и сертификати, която ще разработим, е базирана на класическа клиент-сървър архитектура, реализирана чрез уеб браузър и уеб приложение (фигура 3-1):



Фигура 3-1. Архитектура на системата за подписване на документи в уеб среда

Тя се състои от следните компоненти:

- **DigitalSignerApplet** – Java аплет за подписване на документи в уеб браузъра на потребителя със сертификат от PKCS#12 хранилище.
- **SmartCardSignerApplet** – Java аплет за подписване на документи в уеб браузъра на потребителя със смарт карта.

- **DocumentSigningDemoWebApp** – Java-базирано уеб приложение за посрещане на подписан документ и верификация на неговия цифров подписи и сертификат. Приложението включва подсистема за верификация на цифрови подписи, подсистема за директна верификация на сертификат и подсистема за верификация на сертификационна верига.

От страна на клиента работи стандартен уеб браузър, в който се изпълняват Java аплети за подписване на документи. Подписаните документи се изпращат към уеб сървъра чрез стандартна уеб форма по протокол HTTP. Подписването чрез сертификат от PKCS#12 хранилище и подписването със смарт карта се реализират поотделно от два различни Java аплета.

От страна на сървъра работи Java-базирано уеб приложение, което посреща подписаните документи и проверява цифровия им подпис, както и сертификата, с който са подписани. Ако сертификатът пристига заедно със сертификационната си верига, тя също се верифицира. Реализирани са два метода за верификация – директна проверка на сертификат и проверка на сертификационна верига.

При директната проверка на сертификат се проверява дали сертификатът е валиден към датата на проверката и дали е подписан (издаден) от някой от сертификатите, на които сървърът има директно доверие.

При проверката на сертификационната верига даден сертификат се счита за валиден, ако е валидна веригата му и ако тя завършва с Root-сертификат на който сървърът има доверие.

Нека сега разгледаме в детайли отделните компоненти на системата.

3.2. Java аплет за подписване на документи

Сега ще разгледаме конкретните проблеми, които възникват при реализацията на Java аплет, който служи за подписване на документи в клиентския уеб браузър. Разликата между аплета, който подписва със сертификат, извлечен от PKCS#12 файл и аплета, който подписва със смарт карта, е минимална, така че ще разгледаме и двата едновременно.

3.2.1. Подписани Java аплети

Нека разгледаме в детайли технологията на подписаните аплети, защото ще трябва да я използваме за да повишим правата, с които се изпълнява нашият аплет.

Необходим е достъп до файловата система

Аплетът, който разработваме, трябва да може да подписва файлове от локалната машина и следователно трябва да може да ги чете. По принцип Java аплетите се изпълняват с намалени права и нямат достъп до файловата система на машината, на която работят. За да се преодолее това ограничение, е необходимо аплетът да работи с пълни права, а за тази цел той трябва да бъде подписан.

Подписване на Java аплети с инструмента keytool

За да повишим правата, с които се изпълнява даден аплет, е необходимо да го подпишем. Нека разгледаме как става това.

Да предположим, че сме написали кода на даден аплет 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. По подразбиране тя се инсталира в директория **%JAVA_HOME%\bin**. Ето един пример за нейното използване:

```
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 някой сертификационен орган (PFX файл, съдържащ сертификата и съответния му личен ключ) или трябва да си генерираме саморъчно-подписан сертификат. От JDK 1.5 **jarsigner** поддържа подписване и със смарт карта.

Генериране на саморъчно подписани сертификати с jarsigner

За да си генерираме саморъчно-подписан (self-signed) сертификат можем да използваме програмката **keytool**, която идва стандартно с JDK 1.4.

Ето примерна команда за генериране на саморъчно-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).

Подписване на аплет със саморъчно-подписан сертификат

Можем да използваме генерирания с предходната команда саморъчно-подписан 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 резултатът, както е в посочения пример, за изходен JAR файл се използва входният JAR файл.

Изпълнение на подписани аплети

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 уеб 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**.

Предупреждение за изпълнение на подписани аплети

Средата, която изпълнява аплети в уеб 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 (фигура 3-2).

Java Plug-In 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. Възможно е да се даде и перманентно доверие на аплета за да не се показва предупреждението при всяко негово зареждане.

Фигура 3-2. Диалог-предупреждение на Java Plug-In 1.5 за подписан аплет

3.2.2. Връзка между Java аплет и уеб браузър

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 някой сървърен URL или си комуникира с уеб 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.

Достъп до HTML форма от Java аплет

Да предположим, че имаме обикновена HTML форма, с която се изпращат файлове към дадено уеб 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. Уеб 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. Да разгледаме техническите средства за връзка между java аплет и уеб браузър.

Класът netscape.javascript.JSObject

Технически взаимодействие между аплет и уеб 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 обектния модел (всички методи и свойства) на текущия прозорец на уеб 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 [J2JS, 2004].

Класът **JSObject** поддържа статичен метод **getWindow()**, с който се извлича от уеб браузъра прозорецът, в който се намира аплетът. След това може да се осъществява достъп до обектния модел на този прозорец и документа, показан в него, чрез методите **getMember()**, **setMember()** и **eval()**.

Ето част от сорс кода на аплет, който извлича стойността на полето с име **FileName** от първата HTML форма на уеб страницата от която е зареден:

```
/** Initialize the applet */
public void init() {
    JSObject browserWindow = JSObject.getWindow(this);
    JSObject mainForm = (JSObject) browserWindow.eval("document.forms[0]");
    JSObject fileNameField = (JSObject) mainForm.getMember("FileName");
    String fileName = (String) fileNameField.getMember("value");

    // Continue the applet initialization here ...
}
```

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**".

Проблеми при достъп до аплет от JavaScript

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.

Описаният подход не работи винаги, защото в някои уеб 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 негов метод от JavaScript, този метод ще работи с намалени права. Това поведение може да е различно при различните уеб браузъри, но при всички случаи създава неприятности.

3.2.3. Проектиране на аплета за подписване

Нека сега проектираме конкретната функционалност на аплета за подписване на документи в уеб среда и опишем начинът му на работа.

Подписаният аплет трябва да се извиква директно от потребителя

Поради описаните преди малко проблеми вместо да извикваме Java функция за подписване на файл от JavaScript, е много по-добре да направим аплетът да има 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 може 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 става единствено от аплета и то само след успешно подписване. Така потребителят няма да има възможност да изпраща неподписани или грешно подписани файлове.

Сървърът получава документа, подпис и сертификат

При подписване на документ е необходимо на сървъра да се изпраща не само документът и изчисления от него цифров подпис, но също и сертификатът, 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 подписът. 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 на 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 сертификатът на потребителя заедно със сертификационната му верига, за да може тя да бъде верифицирана след като бъде получена.

Процесът на подписване на документ

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. Ако се използва смарт карта, се подканва потребителят да посочи библиотеката-имплементация на PKCS#11 и PIN кодът за достъп до смарт картата.
2. Зарежда се избраният PFX файл и от него се изваждат сертификата на потребителя, съответният му личен ключ и цялата сертификационна верига, свързана с този сертификат. При работа със смарт карта случаят е аналогичен – от картата се зареждат сертификатът, сертификационната му верига (ако е налична) и личният ключ.
3. Взема се името на файла за подписване от HTML формата, файлът се зарежда в паметта и се извършва подписването му с личния ключ на потребителя.
4. Резултатът от подписването на файла и сертификата на потребителя заедно с цялата му сертификационна верига се записват в определени полета от HTML формата в текстов вид за да бъдат изпратени към сървъра заедно в уеб формата. Може да се използва стандартният формат BASE64 за записване на бинарни данни в текстов вид.

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.

3.3. Уеб приложение за верификация на цифровия подпис и използвания сертификат

Освен аплетът, който подписва изпращаните от потребителя файлове, нашата уеб-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.

3.3.1. Система за верификация на цифровия подпис

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 публичният му ключ и се проверява дали сигнатурата на получения документ е направена със съответния му личен ключ.

3.3.2. Система за верификация на сертификати

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.

Верификация на цифрови сертификати

Има няколко механизма за верификация на цифрови сертификати. Ние ще разгледаме два от тях. При PKI инфраструктурата 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. Другият вариант е да се провери директно сертификатът дали е подписан от друг сертификат, на който имаме доверие.

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 саморъчно-подписан сертификат, този сертификат няма сертификационна верига. Следователно е възможно при получаването на подписан файл на сървъра да се получи сертификатът на потребителя заедно с пълната му сертификационна верига, но е възможно също да се получи само сертификата без никаква сертификационна верига.

Проверка на сертификационната 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, проверка на валидността на връзките между тях и проверка на Root-сертификата, от който започва веригата.

За целта може да се използват средствата на Java Certification Path API и алгоритъмът **`PKIX`**, който е реализиран стандартно в JDK 1.4. Необходимо е единствено приложението да поддържа множество от Root-сертификати на сертификационни органи от първо ниво, на които безусловно вярва (trusted CA root certificates).

Директна верификация на сертификат

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, може да се проверява само дали той е подписан директно от сертификат, на който имаме доверие. Системата може да поддържа съвкупност от сертификати, на които има доверие (trusted certificates) и при проверка на даден сертификат да 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 сертификационни органи, които потребителите биха могли да използват. Ако за даден потребителски сертификат системата не разполага със сертификата, който е негов директен издател, този потребителски сертификат няма да бъде успешно верифициран, дори ако е валиден. Тази схема за проверка може да бъде полезна, когато потребителите използват сертификати, които не са придружени от сертификационна верига.

3.3.3. Проектиране на уеб приложението

Уеб приложението трябва да е Java-базирано (за да можем да използваме стандартните криптографски възможности на Java платформата, които вече разгледахме) и трябва да се грижи за:

- Получаване на изпратения от уеб браузъра документ (файл) от получената уеб форма.
- Получаване и декодиране на цифровия подпис, получен с уеб формата.
- Получаване и декодиране на получения с уеб формата цифров сертификат, използван за подписването на получения документ.
- Верификация на цифровия подпис на получения документ по публичния ключ, извлечен от получения сертификат.
- Верификация на получения цифров сертификат. Трябва да се поддържа директна верификация и верификация на сертификационната верига (когато е налична). За целта трябва да се поддържат множество от доверени Root-сертификати и множество от сертификати за директна проверка.

Резултатите от всички описани проверки трябва да се визуализират по подходящ начин.

НАПС
НАЦИОНАЛНА АКАДЕМИЯ ПО РАЗРАБОТКА НА СОФТУЕР

НАЦИОНАЛНА АКАДЕМИЯ ПО РАЗРАБОТКА НА СОФТУЕР

Лекторите

» Светлин Наков е преподавател по съвременни софтуерни технологии в СУ "Св. Климент Охридски".

Той е автор на десетки научни и технически публикации и няколко книги, свързани с разработката на софтуер, заради което е търсен лектор и консултант.

През 2004 г. получава наградата "**Джон Атанасов**" от президента на България Георги Първанов за приноса му към развитието на информационните технологии и информационното общество.

» Мартин Кулов е изпълнителен директор във фирма "Код Атест", където разработва проекти за повишаване качеството на софтуерните продукти в България чрез автоматизация на процесите и внедряване на системи за управление на качеството.

Мартин е опитен лектор и сертифициран от Майкрософт разработчик по програмите MCSD и MCSD.NET.

» Други инструктори с опит като преподаватели и програмисти.

Академията

» **Национална академия по разработка на софтуер** (**НАПС**) е център за професионално обучение на софтуерни специалисти.

» **НАПС** провежда задълбочени курсове по разработка на софтуер и съвременни софтуерни технологии.

» Предлагани специалности:

.NET Enterprise Developer

Java Enterprise Developer

» **Качествено обучение** с много **практически упражнения**

» Завършвате само за 3 месеца.

» **Гарантирана работа** след успешно завършване!

» Професионална сертификация!

» **БЕЗПЛАТНО!**

Учите безплатно, плащате след като завършите и започнете работа.

Стипендии от софтуерни фирми.

http://academy.devbg.org

Глава 4. NakovDocumentSigner – система за подписване на документи в уеб среда

До момента разгледахме основните проблеми, които стоят пред цифровото подписване на документи в уеб 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 и направихме преглед на средствата на Java платформата за тяхното решаване. Нека сега разгледаме конкретната имплементация на системата за подписване на документи в уеб среда.

4.1. Рамкова система NakovDocumentSigner

NakovDocumentSigner представлява рамкова система (framework) за цифрово подписване на документи в Java-базирани уеб приложения [Nakov, 2004]. Системата е разработена в Софийски университет "Св. Климент Охридски" от Светлин Наков. Тя се състои от следните компоненти:

- Подписан Java аплет, който служи за цифрово подписване на файлове преди изпращането им. Поддържат се два варианта на аплета – DigitalSignerApplet за подписване със PKCS#12 хранилище (PFX файл) и SmartCardSignerApplet – за подписване със смарт карта.
- Примерно уеб приложение DocumentSigningDemoWebApp, 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 уеб приложение. Поддържа директна верификация на сертификат и верификация на сертификационна верига.

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

4.2. Java аплет за подписване с PKCS#12 хранилище

Да разгледаме имплементацията на Java аплетът **DigitalSignerApplet**, който подписва документи в клиентския уеб браузър.

Системни изисквания за Java аплета за подписване в уеб среда

Java аплетът за подписване с PKCS#12 хранилище 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 работи с пълни права и да

може да осъществява достъп до локалната файлова система на потребителя и работи нормално само ако потребителят му позволи да бъде изпълнен с пълни права.

Имплементация Java аплета

Аплетът следва твърдо описаната преди малко поредица от стъпки за подписване на документи в клиентския уеб браузър 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 файлът за подписване и имената на полетата, в които се записват изчислената сигнатура и използваният цифров сертификат, заедно с цялата му сертификационна верига.

Сорс кодът на аплета DigitalSignerApplet

Сорс кодът на аплета, който подписва документи в уеб браузъра на клиента със сертификат от PKCS#12 хранилище, се състои от няколко файла, достъпни от сайта на NakovDocumentSigner. Ще ги разгледаме един по един. Да започнем с основния клас на аплета – **DigitalSignerApplet**:

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.KeyStoreException;
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 certification chain and signature should be stored.
 *
 * If the signing process is sucecssfull, the signature and certification chain
 * fields in the HTML form are filled. Otherwise an error message explaining the
 * failure reason is shown to the user.
 *
```

```
 * 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 certification chain)
 * and its 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 considers taht the password for the keystore and the password for the
 * private key in it are the same (this is typical for the PFX files).
 *
 * In addition to the calculated signature the certification chain is extracted from
 * the PFX file and is placed in the HTML form too. The digital signature is stored
 * as Base64-encoded sequence of characters. The certification chain is stored as
 * ASN.1 DER-encoded sequence of bytes, additionally encoded in Base64.
 *
 * In case the PFX file contains only one certificate without its full certification
 * chain, a chain consisting of this single certificate is extracted and stored in
 * the HTML form instead of the full certification 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. 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 Java 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
 * National Academy for Software Development - http://academy.devbg.org
 * 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_PARAM = "fileNameField";
    private static final String CERT_CHAIN_FIELD_PARAM = "certificationChainField";
    private static final String SIGNATURE_FIELD_PARAM = "signatureField";
    private static final String SIGN_BUTTON_CAPTION_PARAM = "signButtonCaption";

    private static final String PKCS12_KEYSTORE_TYPE = "PKCS12";
    private static final String X509_CERTIFICATE_TYPE = "X.509";
    private static final String CERTIFICATION_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.
     * Actually the applet consists of a single button, that fills its surface. The
     * button's caption comes from the applet parameter SIGN_BUTTON_CAPTION_PARAM.
```

```
     */
    public void init() {
        String signButtonCaption = this.getParameter(SIGN_BUTTON_CAPTION_PARAM);
        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 comes from a field in the HTML
     * document. The result consists of the calculated digital signature and
     * certification 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_PARAM applet parameter. The names of the
     * output fields for the signature and the certification chain are obtained
     * from the parameters CERT_CHAIN_FIELD_PARAM and SIGNATURE_FIELD_PARAM.
     * The applet extracts the certificate, its chain and its private key from
     * a PFX file. The user is asked to select this PFX file and the password
     * for accessing it.
     */
    private void signSelectedFile() {
        try {
            // Get the file name to be signed from the form in the HTML document
            JSObject browserWindow = JSObject.getWindow(this);
            JSObject mainForm = (JSObject) browserWindow.eval("document.forms[0]");
            String fileNameFieldName = this.getParameter(FILE_NAME_FIELD_PARAM);
            JSObject fileNameField =
                (JSObject) mainForm.getMember(fileNameFieldName);
            String fileName = (String) fileNameField.getMember("value");

            // Perform file signing
            CertificationChainAndSignatureInBase64 signingResult=signFile(fileName);

            if (signingResult != null) {
                // Document signed. Fill the certificate and signature fields
                String certChainFieldName=this.getParameter(CERT_CHAIN_FIELD_PARAM);
                JSObject certChainField =
                    (JSObject) mainForm.getMember(certChainFieldName);
                certChainField.setMember("value", signingResult.mCertChain);
                String signatureFieldName = this.getParameter(SIGNATURE_FIELD_PARAM);
                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) {
            se.printStackTrace();
            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) {
            jse.printStackTrace();
            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 certification 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 certification 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 arise during the singing process
     * (e.g. invalid file format, invalid certificate, invalid password, etc.)
     */
    private CertificationChainAndSignatureInBase64 signFile(String aFileName)
    throws DocumentSignException {

        // Load the file for signing
        byte[] documentToSign = null;
        try {
            documentToSign = readFileInByteArray(aFileName);
        } catch (IOException ioex) {
            String errorMsg = "Can not read the file for signing " + aFileName + ".";
            throw new DocumentSignException(errorMsg, 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();
            if (keyStoreFileName.length() == 0) {
                String errorMessage = "It is mandatory to select a certificate " +
                    "keystore (.PFX or .P12 file)!";
                throw new DocumentSignException(errorMessage);
            }
            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  is invalid.";
                throw new DocumentSignException(errorMessage, ex);
            }

            // Get the private key and its certification chain from the keystore
            PrivateKeyAndCertChain privateKeyAndCertChain = null;
```

```
            try {
                privateKeyAndCertChain =
                    getPrivateKeyAndCertChain(userKeyStore, password);
            } catch (GeneralSecurityException gsex) {
                String errorMessage = "Can not extract certification chain and " +
                    "corresponding private key from the specified keystore file " +
                    "with given password. Probably the password is incorrect.";
                throw new DocumentSignException(errorMessage, gsex);
            }

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

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

            // Create the result object
            CertificationChainAndSignatureInBase64 signingResult =
                new CertificationChainAndSignatureInBase64();

            // Save X.509 certification chain in the result encoded in Base64
            try {
                signingResult.mCertChain = encodeX509CertChainToBase64(certChain);
            }
            catch (CertificateException cee) {
                String errorMessage = "Invalid certification 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[] signature = signDocument(documentToSign, privateKey);
                signingResult.mSignature = Base64Utils.base64Encode(signature);
            } 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 aKeyStorePass)
    throws GeneralSecurityException, IOException {
        KeyStore keyStore = KeyStore.getInstance(PKCS12_KEYSTORE_TYPE);
        FileInputStream keyStoreStream = new FileInputStream(aFileName);
        char[] password = aKeyStorePass.toCharArray();
        keyStore.load(keyStoreStream, password);
        return keyStore;
    }

    /**
     * @return private key and certification chain corresponding to it, 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 certification chain and the corresponding
     * private key.
     * If the certificate has no entries, an exception is trown. It the keystore has
     * several entries, the first is used.
     */
    private PrivateKeyAndCertChain getPrivateKeyAndCertChain(
        KeyStore aKeyStore, String aKeyPassword)
    throws GeneralSecurityException {
        char[] password = aKeyPassword.toCharArray();
        Enumeration aliasesEnum = aKeyStore.aliases();
        if (aliasesEnum.hasMoreElements()) {
            String alias = (String)aliasesEnum.nextElement();
            Certificate[] certificationChain = aKeyStore.getCertificateChain(alias);
            PrivateKey privateKey = (PrivateKey) aKeyStore.getKey(alias, password);
            PrivateKeyAndCertChain result = new PrivateKeyAndCertChain();
            result.mPrivateKey = privateKey;
            result.mCertificationChain = certificationChain;
            return result;
        } else {
            throw new KeyStoreException("The keystore is empty!");
        }
    }

    /**
     * @return Base64-encoded ASN.1 DER representation of given X.509 certification
     * chain.
     */
    private String encodeX509CertChainToBase64(Certificate[] aCertificationChain)
    throws CertificateException {
        List certList = Arrays.asList(aCertificationChain);
        CertificateFactory certFactory =
            CertificateFactory.getInstance(X509_CERTIFICATE_TYPE);
        CertPath certPath = certFactory.generateCertPath(certList);
        byte[] certPathEncoded = certPath.getEncoded(CERTIFICATION_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
     * certification chain corresponding to this private key.
     */
    static class PrivateKeyAndCertChain {
        public PrivateKey mPrivateKey;
        public Certificate[] mCertificationChain;
    }

    /**
     * Data structure that holds a pair of Base64-encoded
     * certification chain and digital signature.
     */
    static class CertificationChainAndSignatureInBase64 {
        public String mCertChain = 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);
        }
    }

}
```

Основният клас използва класа **CertificateFileAndPasswordDialog** за да предостави на потребителя възможност за избор на PFX файл и парола за достъп до него. Ето неговият сорс код:

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 choose 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's 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
 * National Academy for Software Development - http://academy.devbg.org
 * 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 PFX_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, 350, 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, 350, 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 selectedCertFile =
                fileChooser.getSelectedFile().getAbsolutePath();
            mCertFileNameTextField.setText(selectedCertFile);
        }
    }

    /**
     * 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(PFX_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(PFX_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 allows the user to choose a 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. Can not handle this. Do nothing
        }

        setModal(true);
        show();

        try {
            if (mResult)
                saveSettings();
        } catch (IOException ioex) {
            // Saving settings failed. Can not handle this. 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 (.PFX, .P12)";
        }
    }

}
```

Понеже в Java не предлага стандартно поддръжка на BASE64 кодиране, трябва да дефинираме собствена реализация. Ето нейният сорс код:

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;
  }

}
```

Как работи аплетът за подписване на файл?

Аплетът използва класа **netscape.javascript.JSObject** за достъп до полетата на HTML формата, взима от нея избрания от потребителя файл и го подписва. При подписването първо се прочита съдържанието на файла, след което се показва на потребителя диалогът 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 личният ключ и съответната му сертификационна верига. Тази верига винаги започва 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.

Следва извършване на 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.

За простота се очаква HTML документът, в който е разположен аплетът 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 файлът за подписване, при невъзможност да бъде прочетен PFX файлът, при невалиден формат на PFX файла, при липса на личен ключ, при липса на сертификат, при невалиден формат на сертификата, при невалидна парола за достъп до PFX файла, 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 във всички други необичайни ситуации.

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 личната директорията на потребителя (user home directory), 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 класът **java.security.KeyStore**. Аплетът очаква хранилището да бъде във формат PKCS#12 и да съдържа само един запис (alias), в който са записани личният ключ на потребителя и сертификационната верига на неговия сертификат. В частност тази сертификационна верига може да се състои и само от един сертификат. Аплетът очаква още паролата за достъп до хранилището да съвпада са паролата за достъп до личния ключ в него.

Компилиране и подписване на аплета

За да работи правилно аплетът, е необходимо той да бъде подписан. За подписването му може да се използва сертификат, издаден от някой сертификационен орган (PFX файл) или саморъчно-подписан сертификат. Можем да използваме следния скрипт за да си генерираме саморъчно-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 файлът **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 PKCS#12, 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\plugin.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 саморъчно-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 текущият път да е присъства **bin** директорията на JDK инсталацията.

Тестване на аплета с примерна HTML форма

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>
        Certification chain:
        <input type="text" name="certificationChain">
        <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="certificationChainField" value="certificationChain">
        <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"
            fileNameField="fileToBeSigned"
            certificationChainField="certificationChain"
            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>

    <!----------------------- old applet tag - don't use it! -----------------------
    <applet name="signApplet"
        archive="DigitalSignerApplet.jar"
        code="DigitalSignerApplet"
        width="130" height="25" mayscript
        fileNameField="fileToBeSigned"
        certificationChainField="certificationChain"
        signatureField="signature"
        signButtonCaption="Sign selected file">
    </applet>
    ------------------------------------------------------------------------------->

</body>

</html>
```

Важно е да не използваме остарелия таг **<applet>**, защото при него няма начин да укажем на уеб 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.

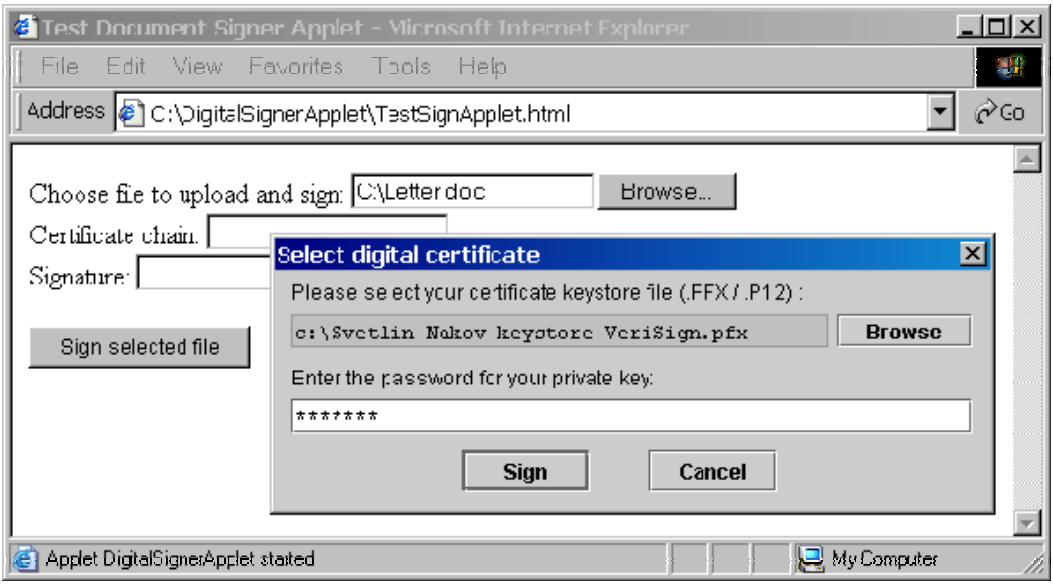
Аплетът за подписване в уеб среда в действие

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 автоматично към сайта, от който тя може да бъде изтеглена.

Ако на машината е инсталиран Java Plug-In 1.4 и уеб 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.

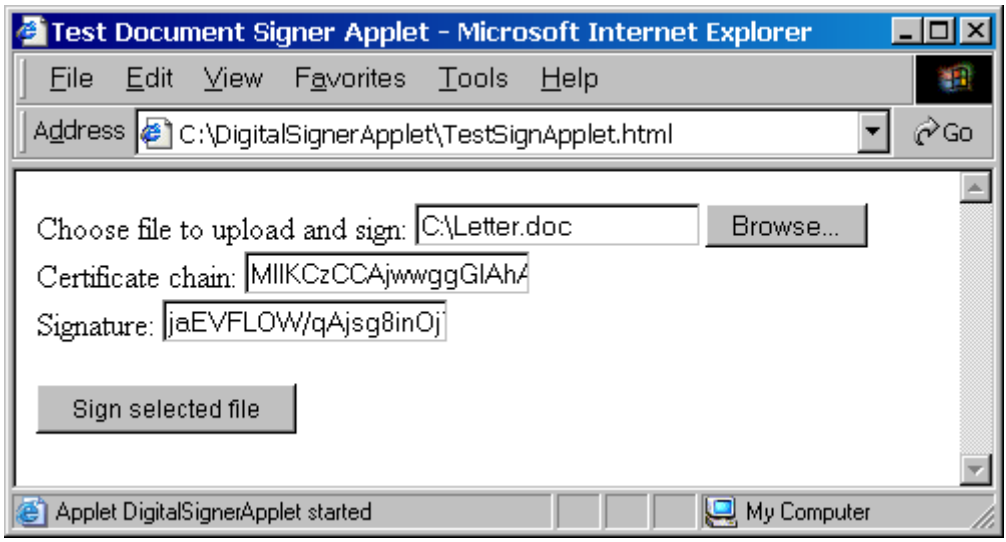
Целта на тестовата HTML страница от примера по-горе е да демонстрира подписването на файлове на машината на клиента. Примерът представлява HTML форма, съдържаща три полета, аплет за подписване и бутон за активиране на подписването, който всъщност се намира вътре в аплета. Трите полета във формата служат 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 със сертификат и личен ключ и парола за достъп до него (фигура 4-1):



Фигура 4-1. Подписване в уеб среда – диалог за избор на 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. След успешно подписване на избрания файл HTML формата изглежда по следния начин (фигура 4-2):



Фигура 4-2. HTML форма с подписан файл в нея

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

Сертификати за тестови цели

За да използва аплета, се очаква потребителят да притежава цифров сертификат и съответен на него личен ключ, записани в PFX файл (PKCS#12 хранилище), като паролата за достъп до този файл трябва да съвпада с паролата за достъп до личния ключ към сертификата. Такъв PFX файл може да се придобие при закупуването на сертификат от някой сертификационен орган или да се генерира с инструмента **`keytool`**.

За тестови цели могат да бъдат използвани демонстрационни цифрови сертификати, които могат да бъдат получени от уеб сайтовете на някои сертификационни органи като Thawte, VeriSign и GlobalSign. Тези сертификационни органи издават сертификатите си през Интернет и в резултат потребителите ги получават инсталирани директно в техните уеб браузъри. За да бъдат използвани такива сертификати за подписване на документи с **DigitalSignerApplet**, те първо трябва да бъдат експортирани заедно с личния им ключ в .PFX или .P12 файл.

4.3. Java аплет за подписване със смарт карта

В предната част видяхме как можем да реализираме аплет, който подписва документи със сертификат, намиращ се в PKCS#12 хранилище (PFX файл). Нека сега разгледаме имплементацията на аплета **SmartCardSignerApplet** за подписване на документи със смарт карта.

Системни изисквания за Java аплета за подписване със смарт карта

Java аплетът за подписване със смарт карта изисква инсталиран Java Plug-In версия 1.5 или по-нова на машината на клиента. Това е необходимо, защото аплетът използва Sun PKCS#11 Provider, който се появява стандартно в Java от версия 1.5 нататък.

Имплементация на аплета за подписване със смарт карта

Подписването със смарт карта не се различава много от подписването с PFX файл. Всъщност, разликата е само в начина на инстанциране на хранилището за ключове и сертификати. При работа с PKCS#12 хранилището се зарежда от PFX файл, а при работа със смарт карта, хранилището се зарежда от картата чрез интерфейса PKCS#11. Другата разлика е, че вместо парола за достъп се изисква PIN кода за картата. Всичко останало е еднакво – от зареждането на сертификата, до подписването на файла.

Да разгледаме сорс кода на аплета. Класът **`SmartCardSignerApplet`** реализира основната му функционалност:

SmartCardSignerApplet.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.io.ByteArrayInputStream;
import java.util.Arrays;
import java.util.Enumeration;
import java.util.List;
import java.security.*;
import java.security.cert.CertPath;
import java.security.cert.Certificate;
import java.security.cert.CertificateException;
import java.security.cert.CertificateFactory;
import java.lang.reflect.Constructor;

import netscape.javascript.JSException;
import netscape.javascript.JSObject;

/**
 * Applet for digital signing documents with a smart card. 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 certification chain and signature should be stored.
 *
 * If the signing process is sucecssfull, the signature and certification chain
 * fields in the HTML form are filled. Otherwise an error message explaining the
 * failure reason is shown.
 *
 * The applet asks the user to locate in his local file system the PKCS#11
 * implementation library that is part of software that come with the smart card
 * and the smart card reader. Usually this is a Windows .DLL file located in Windows
 * system32 directory or .so library (e.g. C:\windows\system32\pkcs201n.dll).
 *
 * The applet also asks the user to enter his PIN code for accessing the smart card.
 * If the smart card contains a certificate and a corresponding private key, the
 * signature of the file is calculated and is placed in the HTML form. In addition
 * to the calculated signature the certificate with its full certification chain is
 * extracted from the smart card and is placed in the HTML form too. The digital
 * signature is placed as Base64-encoded sequence of bytes. The certification chain
 * is placed as ASN.1 DER-encoded sequence of bytes, additionally encoded in Base64.
 * In case the smart card contains only one certificate without its full
 * certification chain, a chain consisting of this single certificate is extracted
 * and stored in the HTML form instead of a full certification chain.
 *
 * Digital singature algorithm used is SHA1withRSA. The length of the calculated
 * singature depends on the length of the private key on the smart card.
 *
 * 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. 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.
 *
 * Java Plug-In version 1.5 or hihger is required for accessing the PKCS#11 smart
 * card functionality, so the applet will not run in any other Java 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) 2005 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 SmartCardSignerApplet extends Applet {

    private static final String FILE_NAME_FIELD_PARAM = "fileNameField";
    private static final String CERT_CHAIN_FIELD_PARAM = "certificationChainField";
    private static final String SIGNATURE_FIELD_PARAM = "signatureField";
    private static final String SIGN_BUTTON_CAPTION_PARAM = "signButtonCaption";

    private static final String PKCS11_KEYSTORE_TYPE = "PKCS11";
    private static final String X509_CERTIFICATE_TYPE = "X.509";
    private static final String CERTIFICATION_CHAIN_ENCODING = "PkiPath";
    private static final String DIGITAL_SIGNATURE_ALGORITHM_NAME = "SHA1withRSA";
    private static final String SUN_PKCS11_PROVIDER_CLASS =
        "sun.security.pkcs11.SunPKCS11";

    private Button mSignButton;

    /**
     * Initializes the applet - creates and initializes its graphical user interface.
     * Actually the applet consists of a single button, that fills its all surface.
     * The button's caption is taken from the applet param SIGN_BUTTON_CAPTION_PARAM.
     */
    public void init() {
        String signButtonCaption = this.getParameter(SIGN_BUTTON_CAPTION_PARAM);
        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 comes from a field in the HTML
     * document. The result consists of the calculated digital signature and
     * certification 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_PARAM applet parameter. The names of the
     * output fields for the signature and the certification chain are obtained
     * from the parameters CERT_CHAIN_FIELD_PARAM and SIGNATURE_FIELD_PARAM. The
     * user is asket to choose a PKCS#11 implementation library and a PIN code
     * for accessing the smart card.
     */
    private void signSelectedFile() {
        try {
            // Get the file name to be signed from the form in the HTML document
            JSObject browserWindow = JSObject.getWindow(this);
            JSObject mainForm = (JSObject) browserWindow.eval("document.forms[0]");
            String fileNameFieldName = this.getParameter(FILE_NAME_FIELD_PARAM);
            JSObject fileNameField =
                (JSObject) mainForm.getMember(fileNameFieldName);
            String fileName = (String) fileNameField.getMember("value");

            // Perform the actual file signing
            CertificationChainAndSignatureBase64 signingResult = signFile(fileName);
```

```
            if (signingResult != null) {
                // Document signed. Fill the certificate and signature fields
                String certChainFieldName =
                    this.getParameter(CERT_CHAIN_FIELD_PARAM);
                JSObject certChainField =
                    (JSObject) mainForm.getMember(certChainFieldName);
                certChainField.setMember("value", signingResult.mCertificationChain);
                String signatureFieldName = this.getParameter(SIGNATURE_FIELD_PARAM);
                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) {
            se.printStackTrace();
            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) {
            jse.printStackTrace();
            JOptionPane.showMessageDialog(this,
                "Unable to access some of the fields of the\n" +
                "HTML form. Please check the applet parameters.");
        }
        catch (Exception e) {
            e.printStackTrace();
            JOptionPane.showMessageDialog(this, "Unexpected error: "+e.getMessage());
        }
    }

    /**
     * Signs given local file. The certificate and private key to be used for signing
     * come from the locally attached smart card. The user is requested to provide a
     * PKCS#11 implementation library and the PIN code for accessing the smart card.
     * @param aFileName the name of the file to be signed.
     * @return the digital signature of the given file and the certification chain of
     * the certificatie 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. smart card access problem, invalid certificate, invalid PIN code, etc.)
     */
    private CertificationChainAndSignatureBase64 signFile(String aFileName)
    throws DocumentSignException {

        // Load the file for signing
        byte[] documentToSign = null;
        try {
            documentToSign = readFileInByteArray(aFileName);
        } catch (IOException ioex) {
            String errorMsg = "Can not read the file for signing " + aFileName + ".";
            throw new DocumentSignException(errorMsg, ioex);
        }

        // Show a dialog for choosing PKCS#11 library and smart card PIN code
        PKCS11LibraryFileAndPINCodeDialog pkcs11Dialog =
```

```
            new PKCS11LibraryFileAndPINCodeDialog();
        boolean dialogConfirmed;
        try {
            dialogConfirmed = pkcs11Dialog.run();
        } finally {
            pkcs11Dialog.dispose();
        }

        if (dialogConfirmed) {
            String oldButtonLabel = mSignButton.getLabel();
            mSignButton.setLabel("Working...");
            mSignButton.setEnabled(false);
            try {
                String pkcs11LibraryFileName = pkcs11Dialog.getLibraryFileName();
                String pinCode = pkcs11Dialog.getSmartCardPINCode();

                // Do the actual signing of the document with the smart card
                CertificationChainAndSignatureBase64 signingResult =
                    signDocument(documentToSign, pkcs11LibraryFileName, pinCode);
                return signingResult;
            } finally {
                mSignButton.setLabel(oldButtonLabel);
                mSignButton.setEnabled(true);
            }
        }
        else {
            return null;
        }
    }

    private CertificationChainAndSignatureBase64 signDocument(
        byte[] aDocumentToSign, String aPkcs11LibraryFileName, String aPinCode)
    throws DocumentSignException {
        if (aPkcs11LibraryFileName.length() == 0) {
            String errorMessage = "It is mandatory to choose a PCKS#11 native " +
                "implementation library for for smart card (.dll or .so file)!";
            throw new DocumentSignException(errorMessage);
        }

        // Load the keystore from the smart card using the specified PIN code
        KeyStore userKeyStore = null;
        try {
            userKeyStore = loadKeyStoreFromSmartCard(
                aPkcs11LibraryFileName, aPinCode);
        } catch (Exception ex) {
            String errorMessage = "Can not read the keystore from the smart card." +
                "\nPossible reasons:\n" +
                " - The smart card reader in not connected.\n" +
                " - The smart card is not inserted.\n" +
                " - The PKCS#11 implementation library is invalid.\n" +
                " - The PIN for the smart card is incorrect.\n" +
                "Problem details: " + ex.getMessage();
            throw new DocumentSignException(errorMessage, ex);
        }

        // Get the private key and its certification chain from the keystore
        PrivateKeyAndCertChain privateKeyAndCertChain = null;
        try {
            privateKeyAndCertChain =
                getPrivateKeyAndCertChain(userKeyStore);
        } catch (GeneralSecurityException gsex) {
            String errorMessage = "Can not extract the private key and " +
                "certificate from the smart card. Reason: " + gsex.getMessage();
            throw new DocumentSignException(errorMessage, gsex);
```

```
        }

        // Check if the private key is available
        PrivateKey privateKey = privateKeyAndCertChain.mPrivateKey;
        if (privateKey == null) {
            String errorMessage = "Can not find the private key on the smart card.";
            throw new DocumentSignException(errorMessage);
        }

        // Check if X.509 certification chain is available
        Certificate[] certChain = privateKeyAndCertChain.mCertificationChain;
        if (certChain == null) {
            String errorMessage = "Can not find the certificate on the smart card.";
            throw new DocumentSignException(errorMessage);
        }

        // Create the result object
        CertificationChainAndSignatureBase64 signingResult =
            new CertificationChainAndSignatureBase64();

        // Save X.509 certification chain in the result encoded in Base64
        try {
            signingResult.mCertificationChain=encodeX509CertChainToBase64(certChain);
        }
        catch (CertificateException cee) {
            String errorMessage = "Invalid certificate on the smart card.";
            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(aDocumentToSign, privateKey);
            signingResult.mSignature = Base64Utils.base64Encode(digitalSignature);
        } catch (GeneralSecurityException gsex) {
            String errorMessage = "File signing failed.\n" +
                "Problem details: " + gsex.getMessage();
            throw new DocumentSignException(errorMessage, gsex);
        }

        return signingResult;
    }

    /**
     * Loads the keystore from the smart card using its PKCS#11 implementation
     * library and the Sun PKCS#11 security provider. The PIN code for accessing
     * the smart card is required.
     */
    private KeyStore loadKeyStoreFromSmartCard(String aPKCS11LibraryFileName,
        String aSmartCardPIN)
    throws GeneralSecurityException, IOException {
        // First configure the Sun PKCS#11 provider. It requires a stream (or file)
        // containing the configuration parameters - "name" and "library".
        String pkcs11ConfigSettings =
            "name = SmartCard\n" + "library = " + aPKCS11LibraryFileName;
        byte[] pkcs11ConfigBytes = pkcs11ConfigSettings.getBytes();
        ByteArrayInputStream confStream =
            new ByteArrayInputStream(pkcs11ConfigBytes);

        // Instantiate the provider dynamically with Java reflection
        try {
            Class sunPkcs11Class = Class.forName(SUN_PKCS11_PROVIDER_CLASS);
            Constructor pkcs11Con = sunPkcs11Class.getConstructor(
                java.io.InputStream.class);
```

```
            Provider pkcs11Provider = (Provider) pkcs11Con.newInstance(confStream);
            Security.addProvider(pkcs11Provider);
        } catch (Exception e) {
            throw new KeyStoreException("Can initialize Sun PKCS#11 security " +
                "provider. Reason: " + e.getCause().getMessage());
        }

        // Read the keystore form the smart card
        char[] pin = aSmartCardPIN.toCharArray();
        KeyStore keyStore = KeyStore.getInstance(PKCS11_KEYSTORE_TYPE);
        keyStore.load(null, pin);
        return keyStore;
    }

    /**
     * @return private key and certification chain corresponding to it, extracted
     * from given keystore. The keystore is considered to have only one entry that
     * contains both certification chain and its corresponding private key. If the
     * keystore has no entries, an exception is thrown.
     */
    private PrivateKeyAndCertChain getPrivateKeyAndCertChain(
        KeyStore aKeyStore)
    throws GeneralSecurityException {
        Enumeration aliasesEnum = aKeyStore.aliases();
        if (aliasesEnum.hasMoreElements()) {
            String alias = (String)aliasesEnum.nextElement();
            Certificate[] certificationChain = aKeyStore.getCertificateChain(alias);
            PrivateKey privateKey = (PrivateKey) aKeyStore.getKey(alias, null);
            PrivateKeyAndCertChain result = new PrivateKeyAndCertChain();
            result.mPrivateKey = privateKey;
            result.mCertificationChain = certificationChain;
            return result;
        } else {
            throw new KeyStoreException("The keystore is empty!");
        }
    }

    /**
     * @return Base64-encoded ASN.1 DER representation of given X.509 certification
     * chain.
     */
    private String encodeX509CertChainToBase64(Certificate[] aCertificationChain)
    throws CertificateException {
        List certList = Arrays.asList(aCertificationChain);
        CertificateFactory certFactory =
            CertificateFactory.getInstance(X509_CERTIFICATE_TYPE);
        CertPath certPath = certFactory.generateCertPath(certList);
        byte[] certPathEncoded = certPath.getEncoded(CERTIFICATION_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
     * certification chain corresponding to this private key.
     */
    static class PrivateKeyAndCertChain {
        public PrivateKey mPrivateKey;
        public Certificate[] mCertificationChain;
    }

    /**
     * Data structure that holds a pair of Base64-encoded
     * certification chain and digital signature.
     */
    static class CertificationChainAndSignatureBase64 {
        public String mCertificationChain = 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);
        }
    }

}
```

Основният клас на аплета използва още един допълнителен клас **PKCS11LibraryFileAndPINCodeDialog**, който дава възможност на потребителя да избере библиотека-имплементация на PKCS#11 и PIN код за достъп до смарт картата:

PKCS11LibraryFileAndPINCodeDialog.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 PKCS#11 implementation library file and PIN code for accessing
 * the smart card.  Allows the user to choose a PKCS#11 library file (.dll / .so) and
 * enter a PIN code for the smart card. The last used library file name is remembered
 * in the config file called ".smart_card_signer_applet.config" located in the user's
 * home directory in order to be automatically shown the next time when the same user
 * accesses 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) 2005 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 PKCS11LibraryFileAndPINCodeDialog extends JDialog {

    private static final String CONFIG_FILE_NAME=".smart_card_signer_applet.config";
    private static final String PKCS11_LIBRARY_FILE_NAME_KEY="last-PKCS11-file-name";

    private JButton mBrowseForLibraryFileButton = new JButton();
    private JTextField mLibraryFileNameTextField = new JTextField();
    private JLabel mChooseLibraryFileLabel = new JLabel();
    private JTextField mPINCodeTextField = new JPasswordField();
    private JLabel mEnterPINCodeLabel = 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 PKCS11LibraryFileAndPINCodeDialog() {
        // Initialize the dialog
        this.getContentPane().setLayout(null);
        this.setSize(new Dimension(426, 165));
        this.setBackground(SystemColor.control);
        this.setTitle("Select PKCS#11 library file and smart card PIN code");
        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
        mChooseLibraryFileLabel.setText(
            "Please select your PKCS#11 implementation library file (.dll / .so) :");
        mChooseLibraryFileLabel.setBounds(new Rectangle(10, 5, 400, 15));
        mChooseLibraryFileLabel.setFont(new Font("Dialog", 0, 12));

        // Initialize certificate keystore file name text field
```

```
        mLibraryFileNameTextField.setBounds(new Rectangle(10, 25, 315, 20));
        mLibraryFileNameTextField.setFont(new Font("DialogInput", 0, 12));
        mLibraryFileNameTextField.setEditable(false);
        mLibraryFileNameTextField.setBackground(SystemColor.control);

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

        // Initialize PIN code label
        mEnterPINCodeLabel.setText("Enter the PIN code to access your smart card:");
        mEnterPINCodeLabel.setBounds(new Rectangle(10, 55, 350, 15));
        mEnterPINCodeLabel.setFont(new Font("Dialog", 0, 12));

        // Initialize PIN code text field
        mPINCodeTextField.setBounds(new Rectangle(10, 75, 400, 20));
        mPINCodeTextField.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(mChooseLibraryFileLabel, null);
        this.getContentPane().add(mLibraryFileNameTextField, null);
        this.getContentPane().add(mBrowseForLibraryFileButton, null);
        this.getContentPane().add(mEnterPINCodeLabel, null);
        this.getContentPane().add(mPINCodeTextField, 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 libraryFileName = mLibraryFileNameTextField.getText();
                if (libraryFileName != null && libraryFileName.length() != 0)
                    mPINCodeTextField.requestFocus();
                else
                    mBrowseForLibraryFileButton.requestFocus();
            }
        });
    }
```

```
    /**
     * Called when the "Browse" button is pressed.
     * Shows file choose dialog and allows the user to locate a library file.
     */
    private void browseForLibraryButton_actionPerformed() {
        JFileChooser fileChooser = new JFileChooser();
        LibraryFileFilter libraryFileFilter = new LibraryFileFilter();
        fileChooser.addChoosableFileFilter(libraryFileFilter);
        String libraryFileName = mLibraryFileNameTextField.getText();
        File directory = new File(libraryFileName).getParentFile();
        fileChooser.setCurrentDirectory(directory);
        if (fileChooser.showOpenDialog(this) == JFileChooser.APPROVE_OPTION) {
            String selectedLibFile = fileChooser.getSelectedFile().getAbsolutePath();
            mLibraryFileNameTextField.setText(selectedLibFile);
        }
    }

    /**
     * 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;
        this.setVisible(false);
    }

    /**
     * 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;
        this.setVisible(false);
    }

    /**
     * @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 library file name with its path.
     */
    private void loadSettings()
    throws IOException {
        String configFileName = getConfigFileName();
        FileInputStream configFileStream = new FileInputStream(configFileName);
        try {
            Properties configProps = new Properties();
            configProps.load(configFileStream);

            // Apply setings from the config file
            String lastLibraryFileName =
                configProps.getProperty(PKCS11_LIBRARY_FILE_NAME_KEY);
            if (lastLibraryFileName != null)
                mLibraryFileNameTextField.setText(lastLibraryFileName);
            else
```

```
            mLibraryFileNameTextField.setText("");
        } finally {
            configFileStream.close();
        }
    }

    /**
     * Saves the dialog settings to the dialog configuration file. These settings
     * consist of a single value - the last used library file name with its path.
     */
    private void saveSettings()
    throws IOException {
        // Create a list of settings to store in the config file
        Properties configProps = new Properties();
        String currentLibraryFileName = mLibraryFileNameTextField.getText();
        configProps.setProperty(PKCS11_LIBRARY_FILE_NAME_KEY,currentLibraryFileName);

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

    /**
     * @return the library file selected by the user.
     */
    public String getLibraryFileName() {
        String libraryFileName = mLibraryFileNameTextField.getText();
        return libraryFileName;
    }

    /**
     * @return the PIN code entered by the user.
     */
    public String getSmartCardPINCode() {
        String pinCode = mPINCodeTextField.getText();
        return pinCode;
    }

    /**
     * Shows the dialog and allow the user to choose library file and enter a PIN.
     * @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. Default settings will be used.
        }

        setModal(true);
        this.setVisible(true);

        try {
            if (mResult) {
                saveSettings();
            }
        } catch (IOException ioex) {
            // Saving settings failed. Can not handle this problem.
```

```
        }

        return mResult;
    }

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

            String fileName = aFile.getName().toLowerCase();
            boolean accepted =
                (fileName.endsWith(".dll") || fileName.endsWith(".so"));
            return accepted;
        }

        public String getDescription() {
            return "PKCS#11 v2.0 ot later implementation library (.dll, .so)";
        }
    }

}
```

Аплетът използва и класа **`Base64Utils`**, който е същият като в аплета за подписване с PFX файл.

Как работи аплетът за подписване със смарт карта?

Аплетът за подписване на документи със смарт карта в уеб браузъра на потребителя работи по абсолютно същия начин като аплета за подписване с PFX файл, който вече разгледахме в детайли.

Първоначално чрез класа **`netscape.javascript.JSObject`** от HTML формата се извлича името на файла за подписване и този файл се зарежда в паметта. Това е възможно, защото аплетът е подписан и има достъп до локалната файлова система.

След това на потребителя се показва диалогът за избор на PKCS#11 библиотека и PIN код за достъп до смарт картата. След като той посочи библиотеката с PKCS#11 имплементацията и си въведе PIN кода, от смарт картата се извличат сертификатът заедно със сертификационната му верига (ако е начина) и интерфейс за достъп до личния ключ от картата.

Сертификационната верига се кодира в ASN.1 DER формат и се записва в текстов вид (с BASE64 кодиране) в съответното текстово поле на HTML формата. За достъп до уеб браузъра отново се използва класа **`JSObject`**.

Прочетеният в паметта файл се подписва с личния ключ от смарт картата и получената цифрова сигнатура се кодира в текстов вид с Base64 кодиране и се записва в поле от HTML формата.

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 файлът за подписване, при невалидна библиотека с PKCS#11 имплементация, при невалиден PIN код, при липса на сертификат или личен ключ в смарт картата, поради невъзможност за достъп до HTML формата и в още много други ситуации.

Графичният потребителски интерфейс на аплета е базиран на библиотеките AWT и JFC/Swing, които се доставят стандартно с JDK 1.5.

Диалогът за избор на PKCS#11 имплементация и PIN код дава възможност за избор само измежду **.dll** и **.so** файлове (Windows и Linux библиотеки). Използваната за последен път PKCS#11 библиотека се запомня заедно с пълния път до нея в конфигурационния файл на аплета, намиращ се личната директория на потребителя (user home directory), за да бъде използван при следващо показване на същия диалог.

За достъп до смарт картата се използва Sun PKCS#11 Provider, който се инстанцира с Java reflection и се конфигурира динамично. Класът **sun.security.pkcs11.SunPKCS11** се инстанцира с reflection (отражение на типовете), за да не се свързва статично с аплета. Ако класът е статично свързан с аплета и този клас липсва (например при версии на Java Plug-In по-малки от 1.5), аплетът изобщо няма да се зареди. Инстанцирането с reflection има предимството, че аплетът ще се зареди и ще хвърли изключение едва при опит за инстанциране на липсващия клас. Тогава на потребителя ще бъде показано подходящо съобщение за грешка.

След като е регистриран PKCS#11 доставчикът, достъпът до хранилището на смарт картата става чрез класа **java.security.KeyStore**. Аплетът очаква хранилището върху смарт картата да съдържа само един запис (alias), в който са записани сертификатът на потребителя (евентуално заедно със сертификационната му верига) и съответния му личен ключ.

Реализацията на всички използвани криптографски алгоритми, се осигурява от доставчика на криптографски услуги по подразбиране **SunJSSE**, който също е част от JDK 1.5.

За подписването на файлове се използва алгоритъмът за цифрови подписи **SHA1withRSA**, който е достъпен стандартно от Java посредством класа **java.security.Signature** и се поддържа от повечето смарт карти.

Компилиране и подписване на аплета

За да работи правилно аплетът, е необходимо той да бъде подписан. Можем да използваме следния скрипт за да си генерираме саморъчно-подписан сертификат, с който да подпишем след това аплета:

generate-certificate.bat

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

Резултатът от изпълнението на скрипта е хранилището за ключове и сертификати **SmartCardSignerApplet.jks**, съдържащо генерирания сертификат и съответния му личен ключ, записани под име "**signFiles**", достъпни с парола "**!secret**". Форматът на изходния файл е JKS (Java KeyStore), който се използва по подразбиране от инструмента **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

```
set JAVA5_HOME=C:\Progra~1\Java\jdk1.5.0_04

del *.class
%JAVA5_HOME%\bin\javac -classpath .;"%JAVA5_HOME%\jre\lib\plugin.jar" *.java

del *.jar
%JAVA5_HOME%\bin\jar -cvf SmartCardSignerApplet.jar *.class

%JAVA5_HOME%\bin\jarsigner -keystore SmartCardSignerApplet.jks -storepass !secret -
keypass !secret SmartCardSignerApplet.jar signFiles

pause
```

Посочената последователност от команди изтрива всички компилирани **.class** файлове, компилира всички **.java** файлове, които съставят аплета, пакетира получените **.class** файлове в архив **SmartCardSignerApplet.jar** и подписва този архив с генерирания преди това саморъчно-подписан сертификат (намиращ се в хранилището **SmartCardSignerApplet.jks**).

За компилацията е необходим JDK 1.5 или по-нова версия.

Тестване на аплета с примерна HTML форма

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

TestSmartCardSignerApplet.html

```
<html>

<head>
    <title>Test Smart Card Signer Applet</title>
</head>

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

```
    <object
        classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
        codebase="http://java.sun.com/update/1.5.0/jinstall-1_5-windows-
i586.cab#Version=5,0,0,5"
        width="130" height="25" name="SmartCardSignerApplet">
        <param name="code" value="SmartCardSignerApplet">
        <param name="archive" value="SmartCardSignerApplet.jar">
        <param name="mayscript" value="true">
        <param name="type" value="application/x-java-applet;version=1.5">
        <param name="scriptable" value="false">
        <param name="fileNameField" value="fileToBeSigned">
        <param name="certificationChainField" value="certificationChain">
        <param name="signatureField" value="signature">
        <param name="signButtonCaption" value="Sign selected file">

        <comment>
            <embed
                type="application/x-java-applet;version=1.5"
                code="SmartCardSignerApplet" archive="SmartCardSignerApplet.jar"
                width="130" height="25" scriptable="true"
                pluginspage="http://java.sun.com/products/plugin/index.html#download"
                fileNameField="fileToBeSigned"
                certificationChainField="certificationChain"
                signatureField="signature"
                signButtonCaption="Sign selected file">
            </embed>
            <noembed>
                Smart card signing applet can not be started because
                Java Plugin 1.5 or newer is not installed.
            </noembed>
        </comment>
    </object>
</body>

</html>
```

Важно е да използваме таговете **<object>** и **<embed>** вместо остарелия таг **<applet>**, защото при него няма начин да укажем на уеб браузъра, че аплетът изисква JDK версия 1.5 или по-висока.

Аплетът за подписване в уеб среда в действие

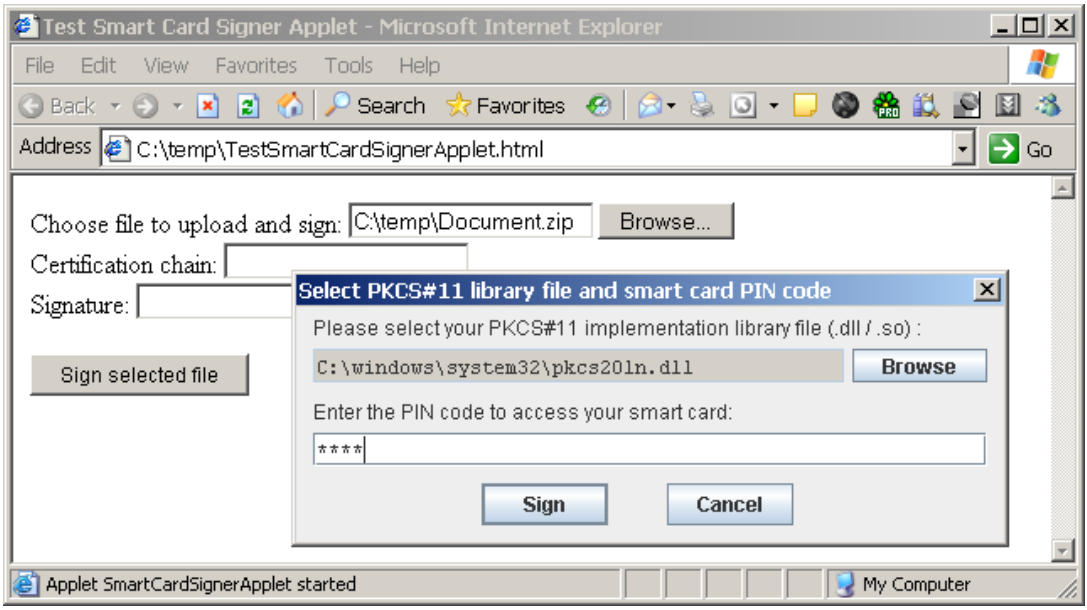
При отваряне на тестовият HTML документ първо се проверява дали на машината има инсталиран Java Plug-In 1.5 или по-висока версия и ако няма потребителят се препраща автоматично към сайта, от който тя може да бъде изтеглена.

Ако на машината е инсталиран Java Plug-In 1.5, при зареждане на тестовия HTML документ се появява диалог, който иска съгласие от потребителя за да бъде изпълнен аплетът с пълни права (фигура 4-3). Ако потребителят се съгласи, аплетът стартира нормално.

Фигура 4-3. Java Plug-In 1.5 иска съгласие за изпълнение на подписан аплет

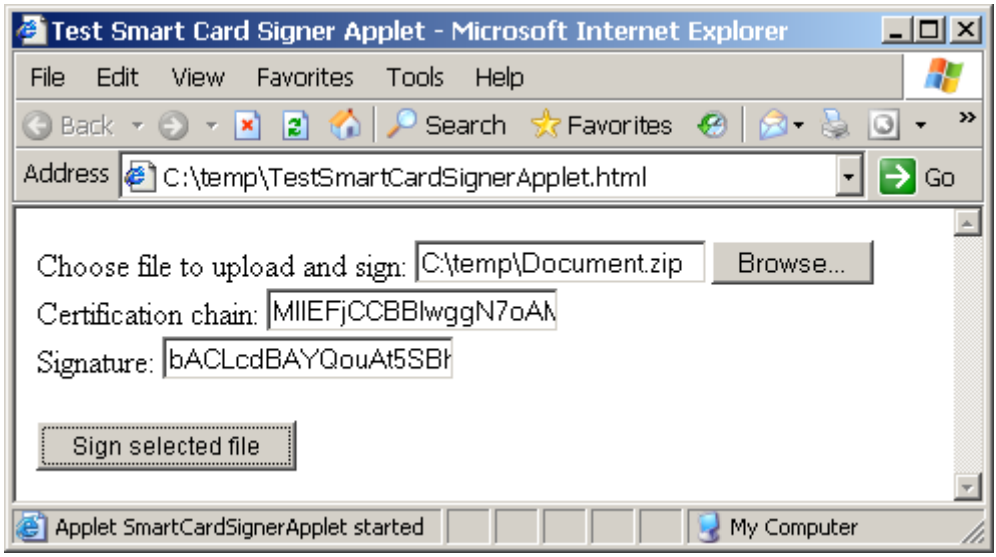
Примерната HTML страница съдържа HTML форма с три полета, аплет за подписване и бутон за активиране на подписването, който всъщност се намира вътре в аплета. Трите полета във формата служат съответно за избор на файл за изпращане, за записване на сертификационната верига, използвана при подписването и за записване на получената сигнатура.

При натискане на бутона за подписване на потребителя се дава възможност да избере библиотека-имплементация на PKCS#11 и PIN код за достъп до смарт картата (фигура 4-4):



Фигура 4-4. Подписване със смарт карта в уеб среда

При успешно подписване изчислената сигнатура и сертификатът, извлечени от смарт картата, се записват в HTML формата (фигура 4-5):



Фигура 4-5. HTML форма с подписан файл в нея

4.4. Уеб приложение за верификация на цифровия подпис и сертификата на изпращача

Като част от системата за подписване на файлове NakovDocumentSigner е разработено и примерно Java-базирано уеб приложение, което 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 директно доверие, без да се проверява сертификационната му верига.

Проверката на сертификационната верига на сертификата на изпращача (когато е налична) има за цел да провери валидността на сертификата и да потвърди самоличността на изпращача.

Как се извършват проверките на подписа и сертификата?

Проверката на сигнатурата се извършва по стандартния начин с използване на класа `java.security.Signature`. За целта първо се извлича публичният ключ на потребителя от получения сертификат и след това с него се проверява дали получената сигнатура съответства на получения файл.

За директната проверка на сертификат примерното уеб приложение поддържа множество от сертификати, на които има доверие и проверява дали сертификатът на потребителя е директно подписан от някой от тях. Проверява се и срокът на валидност на сертификата. Доверените сертификати за директна проверка се съхраняват като `.CER` файлове в специална директория на приложението, името на която се указва от константа.

За проверката на сертификационната верига на получения потребителски сертификат в демонстрационното уеб приложение се използва множество от доверени Root-сертификати на сертификационни органи от първо ниво. От това множество се построява съвкупността от крайни точки на доверие (trust anchors), която е необходима за проверката на сертификационната верига. Самата проверка на веригата сертификати се извършва по PKIX алгоритъма, който е реализиран в Java Certification Path API. Доверените Root-сертификати се съхраняват като `.CER` файлове в специална директория на приложението, името на която се указва от константа.

Уеб приложението е базирано на J2EE и Struts

Демонстрационното уеб приложение е базирано на Java технологиите и е реализирано със средствата на платформата J2EE (Java 2 Enterprise Edition) и широкоразпространения framework за Java уеб приложения Struts. В основната си част използва Servlet/JSP технологиите за реализация на потребителския си интерфейс и следва J2EE стандарта за Java уеб приложения. За изпълнението си изисква J2EE уеб контейнер, в който да работи.

Struts framework е използван, защото има вградена функционалност за посрещане на файлове, изпратени от HTML форма, докато такава възможност стандартно в JSP/Servlets технологията няма.

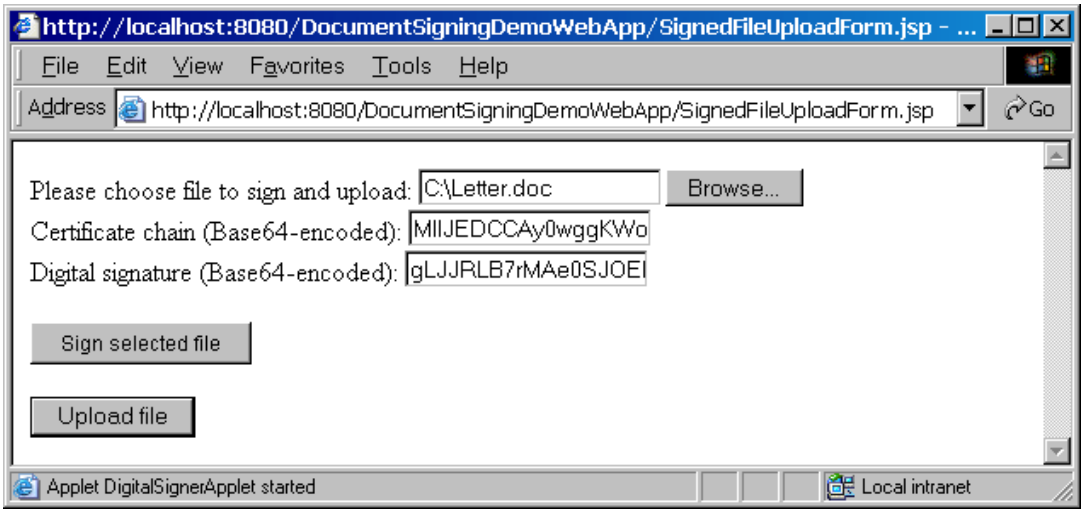
Приложението се състои две Struts-базирани HTML форми, Struts action клас за обработка на формите, Struts action form клас за съхранение на данните от формата, клас за проверка на цифрови подписи и сертификати, страница за обработка на изпратения подписан файл и няколко конфигурационни файла.

Заради програмния модел на Struts framework приложението стриктно следва архитектурата Model-View-Controller (MVC), макар и това да не е наложително за нашите цели.

Пълният сорс код на системата е достъпен за свободно изтегляне от сайта на NakovDocumentSigner – http://www.nakov.com/documents-signing/.

Уеб форма за подписване със сертификат от PFX файл

Формата за подписване на файл със сертификат от PKCS#12 хранилище (PFX файл) се състои от три полета, съответно за името на файла, сертификата и сигнатурата и съдържа още аплета за подписване и бутон за изпращане на формата:



Фигура 4-6. Уеб форма за подписване със сертификат от PKCS#12 хранилище

На картинката (фигура 4-6) аплетът за подписване изглежда като бутон със заглавие "Sign selected file". Тази форма се поражда от JSP страница, базирана на библиотеката от тагове "**struts-html**" и използва Struts тага за изпращане на файл **<html:file>**:

SignedFileUploadForm-PFX.jsp

```
<%
/**
 * A JSP that contains the form for signing and uploading a file. The form
 * contains 3 fields - the file to be uploaded, the certification chain and
 * the digital signature. It also contains the DigitalSignerApplet that signs
 * the selected 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
 * National Academy for Software Development - http://academy.devbg.org
 * 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>
    Certification chain (Base64-encoded): <html:text property="certChain"/> <br>
    Digital signature (Base64-encoded): <html:text property="signature"/> <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" name="DigitalSignerApplet">
        <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="certificationChainField" value="certChain">
        <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"
                fileNameField="uploadFile"
                certificationChainField="certChain"
                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 тагът **<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.

Уеб форма за подписване със смарт карта

Уеб формата за подписване със смарт карта (**SignedFileUploadForm-SmartCard.jsp**) е почти напълно еднаква с уеб формата за подписване с

PKCS#12 хранилище. Единствената разлика е в името на използвания аплет и във версията на JDK, която е указана за аплета:

SignedFileUploadForm-SmartCard.jsp

```
<%
/**
 * A JSP that contains the form for signing and uploading a file. The form
 * contains 3 fields - the file to be uploaded, the certification chain and
 * the digital signature. It also contains the SmartCardSignerApplet that
 * signs the selected 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
 * National Academy for Software Development - http://academy.devbg.org
 * 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>
    Certification chain (Base64-encoded): <html:text property="certChain"/> <br>
    Digital signature (Base64-encoded): <html:text property="signature"/> <br>

    <br>

    <object
        classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
        codebase="http://java.sun.com/update/1.5.0/jinstall-1_5-windows-
i586.cab#Version=5,0,0,5"
        width="130" height="25" name="SmartCardSignerApplet">
        <param name="type" value="application/x-java-applet;version=1.5">
        <param name="code" value="SmartCardSignerApplet">
        <param name="archive" value="SmartCardSignerApplet.jar">
        <param name="mayscript" value="true">
        <param name="scriptable" value="true">
        <param name="fileNameField" value="uploadFile">
        <param name="certificationChainField" value="certChain">
        <param name="signatureField" value="signature">
        <param name="signButtonCaption" value="Sign selected file">

        <comment>
            <embed
                type="application/x-java-applet;version=1.5"

pluginspage="http://java.sun.com/products/plugin/index.html#download"
                code="SmartCardSignerApplet"
                archive="SmartCardSignerApplet.jar"
                width="130"
                height="25"
                mayscript="true"
                scriptable="true"
```

```
                scriptable="true"
                fileNameField="uploadFile"
                certificationChainField="certChain"
                signatureField="signature"
                signButtonCaption="Sign selected file">
            </embed>
            <noembed>
                Smart card signing applet can not be started because
                Java Plugin 1.5 or newer is not installed.
            </noembed>
        </comment>
    </object>

    <br>
    <br>

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

</body>
</html>
```

В страницата е указано, че трябва да има инсталиран Java Plug-In версия 1.5 или по-нова. Ако това не е така, браузърът на потребителя автоматично се препраща към страницата, от която може да се изтегли последната версия на Java Plug-In.

Клас за съхранение на данните от двете уеб форми

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-PFX.jsp or SignedFileUploadForm-SmartCard.jsp). It is
 * actually a data structure that consist of the uploaded file, the sender's
 * certification 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
 * National Academy for Software Development - http://academy.devbg.org
 * 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 mCertChain;
    private String mSignature;
```

```
    public FormFile getUploadFile() {
        return mUploadFile;
    }

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

    public String getCertChain() {
        return mCertChain;
    }

    public void setCertChain(String aCertChain) {
        mCertChain = aCertChain;
    }

    public String getSignature() {
        return mSignature;
    }

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

}
```

Този клас не представлява нищо повече от обикновен Java bean, в който са дефинирани свойства, точно съответстващи на полетата от HTML 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.

Действието, което посреща изпратената форма

Получените данни, записани в action form обекта се обработват от действието **/SignedFileUpload**, което съответства на Struts action класа **SignedFileUploadAction**:

SignedFileUploadAction.java

```
package demo;

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

/**
 * Struts action class for handling the results of submitting the forms
 * SignedFileUploadForm-PFX.jsp and SignedFileUploadForm-SmartCard.jsp.
 *
 * 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 action redirects the user's Web browser to
 * ShowSignedFileUploadResults.jsp that is used to display the received file,
 * certificate, certification chain and digital signature and their validity.
 *
 * 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
 * National Academy for Software Development - http://academy.devbg.org
 * 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");
    }

}
```

Всичко, което това събитие прави, е да запише получения action form обект с данните от формата в потребителската сесия под ключ "**signedFileUploadActionForm**" и след това да пренасочи изпълнението на уеб приложението към страницата за анализ на получения подписан файл **ShowSignedFileUploadResults.jsp**, която е описана в конфигурационния файл на Struts.

Страница за анализ на получените данни

Страницата за анализ на получения подписан файл е малко по-сложна от останалите. Тя извлича action form обекта от сесията на потребителя и след това анализира получените данни и показва информация за тях. Извършва се верификация на цифровия подпис на получения файл и верификация на получения цифров сертификат, използван за подписването. Ако е приложена сертификационна верига, и тя се верифицира. Ето сорс кода на страницата **ShowSignedFileUploadResults.jsp**:

ShowSignedFileUploadResults.jsp

```
<%
/**
 * A JSP for verifying digital signature, certificate and certification chain of the
 * received signed file. It assumes that the data, received by submitting some of
 * the forms SignedFileUploadForm-PFX.jsp or SignedFileUploadForm-SmartCard.jsp
 * stays in the user's session in SignedFileUploadActionForm object stored with the
 * 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 certification 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
 * National Academy for Software Development - http://academy.devbg.org
 * 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 first.
    <%
        } else {
            try {
                // Analyse received signed file and display information about it
                processReceivedFile();
```

```
                displayFileInfo(mReceivedFile, mReceivedFileData);
                mOut.println("<hr>");

                // Analyse received certification chain
                processReceivedCertificationChain();

                // Analyse received digital signature
                processReceivedSignature();

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

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

                // Display certification chain, verify it and display the results
                displayCertificationChain(mCertChain);
                verifyReceivedCertificationChain();
            } 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/>
    Go to the <a href="index.html">start page</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 certification chain and extracts the certificates that it
```

```
     * consist of. The certification chain should be PkiPath-encoded (ASN.1 DER
     * formatted), stored as Base64-string. The extracted chain is stored in the
     * member variables mCertPath as a CertPath object 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 the received certification chain can not be decoded
     * (i.e. its encoding or internal format is invalid).
     */
    private void processReceivedCertificationChain()
    throws Exception {
        String certChainBase64Encoded = mActionForm.getCertChain();
        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 certification chain received.", e);
        }
    }

    /**
     * Displays given certification 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 displayCertificationChain(X509Certificate[] aCertChain)
    throws IOException {
        mOut.println("Certification 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 the 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 <br>");
            if (certKeyUsage[KEY_USAGE_KEY_ENCIPHERMENT])
                mOut.println("[keyEncipherment] - encipher keys for transport<br>");
            if (certKeyUsage[KEY_USAGE_DATA_ENCIPHERMENT])
                mOut.println("[dataEncipherment] - encipher user data <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 certs <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 during key agreement <br>");
            if (certKeyUsage[KEY_USAGE_DECIPHER_ONLY])
                mOut.println("[decipherOnly] - decipher during 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 cert.).
     */
```

```
    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 started)!");
        } catch (DigitalSignatureUtils.CertificateValidationException cve) {
            mOut.println("Certificate is INVALID! " + cve.getMessage());
        }
        mOut.println("</b>");
    }

    /**
     * Verifies received certification 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 verifyReceivedCertificationChain()
    throws IOException, GeneralSecurityException {
        // Create the most trusted CA set of trust anchors
        X509Certificate[] trustedCACerts =
            getCertificateList(mApplicationContext, TRUSTED_CA_ROOT_CERTS_DIR);

        // Verify the certification chain and display the verification results
        mOut.println("Certification chain verification: <b>");
        try {
            DigitalSignatureUtils.verifyCertificationChain(
                mCertPath, trustedCACerts);
            mOut.println("Certification chain verified to be VALID.");
        } catch (CertPathValidatorException cpve) {
            mOut.println("Certification chain is INVALID! Validation failed on " +
                "cert [" + 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 trustedCertsResNames =
            aServletContext.getResourcePaths(aCertificatesDirectory);

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

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

        return trustedCertificates;
    }

%>
```

Как се обработва полученият подписан файл?

Първоначално се проверява дали в сесията е записан action form обектът, който съдържа изпратените от потребителя данни. Ако такъв обект няма, това означава, че данни не са получени и потребителят се съобщава, че трябва да зареди страницата за подписване и изпращане на файл. Ако action form обектът 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 сертификатът на потребителя, използван при подписването. Този сертификат е винаги 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 извлича 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 сертифи-

кационната 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 уеб 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 уеб 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 (т. е. верига реално няма), тя се счита за невалидна. Ако веригата се състои от повече от един сертификат, от нея се премахва последният 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 проблемът.

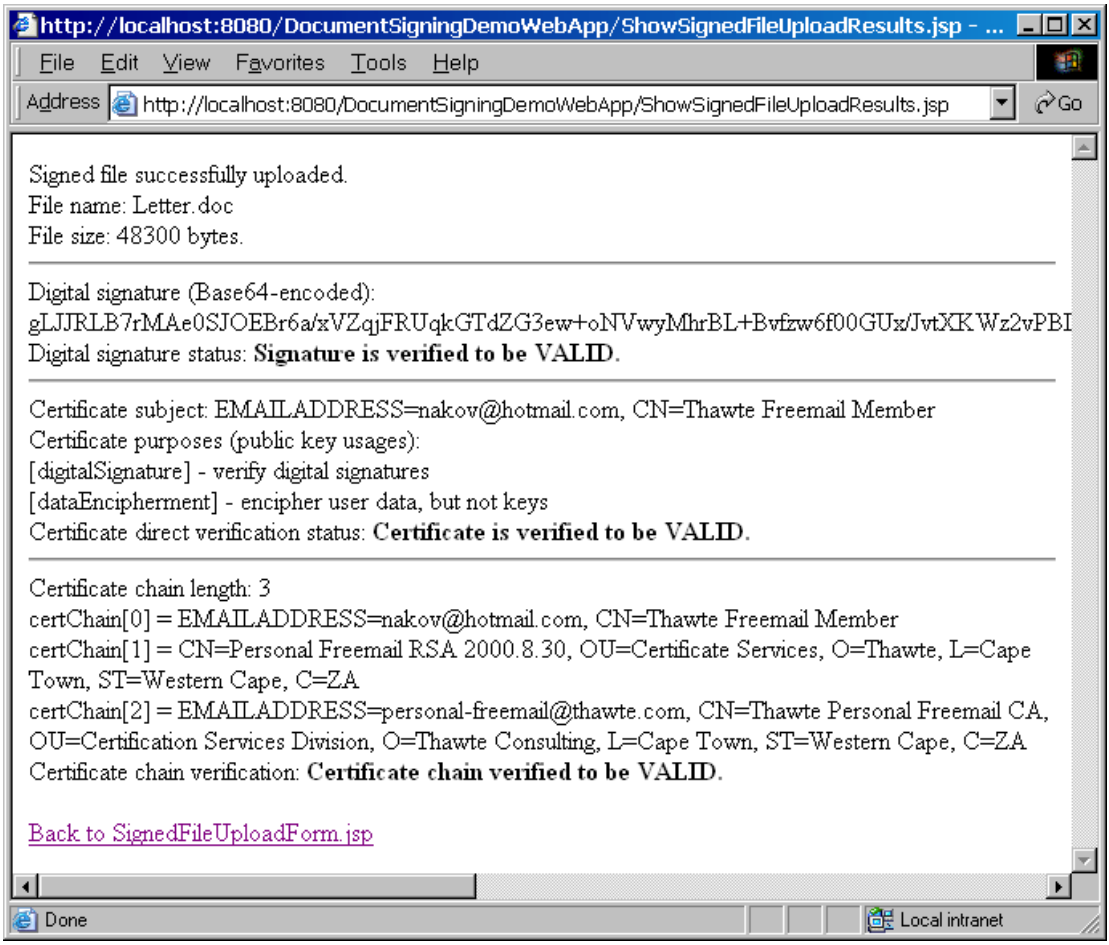
За четенето на всички файлове от дадена директория се използват методите **`getResourcePaths()`** и **`getResourceAsStream()`** на класа **`ServletContext`**.

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.

Верификацията на цифрови подписи и сертификати в действие

Резултатът от всички описани проверки, които се извършват при получаване на подписан файл, изглежда по следния начин (фигура 4-7):



Фигура 4-7. Резултатът от верификация на цифров подпис и сертификат

Основната криптографска функционалност на приложението

Основната функционалност по верификацията на сигнатури, сертификати и сертификационни вериги се намира в класа **`DigitalSignatureUtils`**. Класът е базиран на средствата от Java Cryptography Architecture (JCA), Java Cryptography Extension и Java Certification Path API. Както и при аплета, реализацията на всички криптографски алгоритми се осигурява от доставчика на криптографски услуги по подразбиране **`SunJSSE`**, който е част от JDK 1.4. Ето и сорс кода на 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 certification 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 certification chains aims to check if given certificate is
 * valid by analysing its certification chain. A certification 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
 * National Academy for Software Development - http://academy.devbg.org
 * 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 CERT_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 certification 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);
        InputStream certChainStream = new ByteArrayInputStream(certChainEncoded);
        CertPath certPath;
        try {
            certPath = cf.generateCertPath(certChainStream, CERT_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 certs
        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
            }
        }

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

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

    /**
     * Removes the last certificate from given certification 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 certChainWithoutLastCert = cf.generateCertPath(certsWithoutLast);
        return certChainWithoutLastCert;
    }

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

}
```

Как работи основната криптографска функционалност?

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 сертификат не може да бъде валидна, защото тя трябва да завършва с Root-сертификата на някой сертификационен орган 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 сертификационните органи, 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 някой сертификационен орган, 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, верификацията няма да е успешна.

Ако веригата не е валидна, се генерира изключение, в което се указва поредният номер на сертификата, в който е възникнал проблемът.

За работа с Base64-кодирана информация в уеб приложението се използва същият клас **Base64Utils**, който се използва и при аплета за подписване на файлове.

Конфигурационен файл на уеб приложението

Съгласно стандартите на платформата J2EE за работата на демонстрационното уеб 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>/index.html</welcome-file>
    </welcome-file-list>

</web-app>
```

Този файл съдържа настройките на приложението и указва, да бъде зареден и инициализиран Struts framework и неговият **ActionServlet**. Като стартова страница по подразбиране е указан файлът **index.html** от главната директория на приложението.

Стартова страница на приложението

Стартовата страница **index.html** е изключително проста. Тя съдържа просто две препратки – към страницата за подписване с PKCS#12 хранилище и към страницата за подписване със смарт карта. Ето нейният код:

index.html

```
<!--
 * 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
 * National Academy for Software Development - http://academy.devbg.org
 * 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.
 -->

<html>
<body>

<a href="SignedFileUploadForm-PFX.jsp">Digital document signing with .PFX file
    (PKCS#12 keystore) demo.</a>
```

```
<br>
<a href="SignedFileUploadForm-SmartCard.jsp">Digital document signing with
    smart card (PKCS#11 token) demo.</a>

</body>
</html>
```

Когато демонстрационното уеб приложение се стартира първоначално, то зарежда именно тази страница и дава възможност на потребителя да тества съответно двата аплета за подписване на документи.

Конфигурационен файл на Struts framework

За да се използва Struts framework е необходим и неговият конфигурационен файл **struts-config.xml**:

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"
            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 класовете, които се използват от уеб приложението.

Скрипт за компилиране и построяване на приложението

Съвкупността от всички описани файлове съставя демонстрационното уеб 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>
```

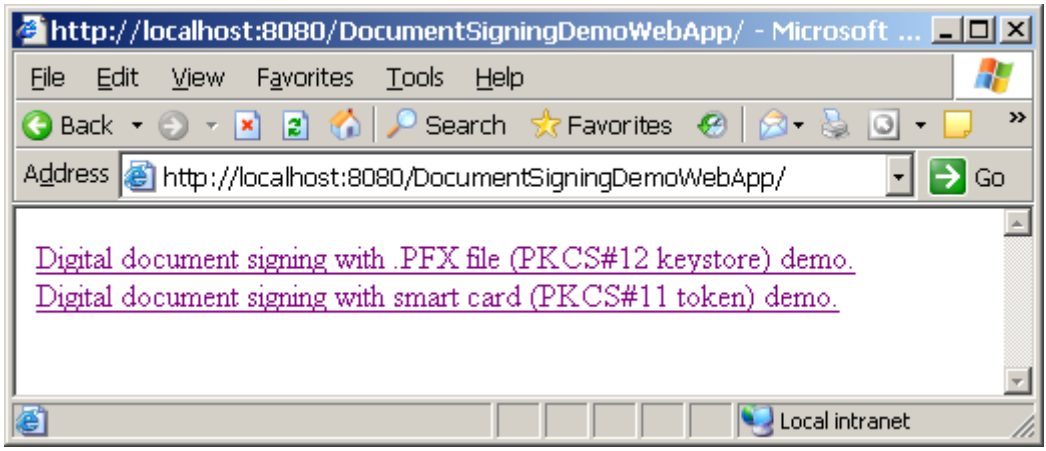
Скриптът компилира файловете на приложението и пакетира всички негови части (JSP файлове, ресурси, библиотеки, компилирани класове и др.) в архив като подрежда файловете и директориите по специален начин, съгласно J2EE спецификациите за уеб приложения.

Резултатът от изпълнението с инструмента **ant** на този скрипт е файлът **DocumentSigningDemoWebApp.war**, който съдържа компилираното уеб приложение във вид готов за изпълнение върху стандартен уеб контейнер.

Инсталиране (deployment) на приложението

За да изпълним приложението **`DocumentSigningDemoWebApp.war`** е необходимо да го инсталираме (deployment) на някой J2EE сървър или сървър за Java уеб приложения (Servlet container).

Ако използваме сървъра за J2EE уеб приложения Apache Tomcat, е достатъчно до копираме файла **`DocumentSigningDemoWebApp.war`** в директория **`%TOMCAT_HOME%/webapps`** и да стартираме Tomcat. След това (при стандартна инсталация и конфигурация на Tomcat) приложението е достъпно от адрес: http://localhost:8080/DocumentSigningDemoWebApp/ (фигура 4-8):



Фигура 4-8. Уеб приложението DocumentSigningDemoWebApp

НАПС
НАЦИОНАЛНА АКАДЕМИЯ ПО РАЗРАБОТКА НА СОФТУЕР

НАЦИОНАЛНА АКАДЕМИЯ ПО РАЗРАБОТКА НА СОФТУЕР

Лекторите

» Светлин Наков е преподавател по съвременни софтуерни технологии в СУ "Св. Климент Охридски".

Той е автор на десетки научни и технически публикации и няколко книги, свързани с разработката на софтуер, заради което е търсен лектор и консултант.

През 2004 г. получава наградата "**Джон Атанасов**" от президента на България Георги Първанов за приноса му към развитието на информационните технологии и информационното общество.

» Мартин Кулов е изпълнителен директор във фирма "Код Атест", където разработва проекти за повишаване качеството на софтуерните продукти в България чрез автоматизация на процесите и внедряване на системи за управление на качеството.

Мартин е опитен лектор и сертифициран от Майкрософт разработчик по програмите MCSD и MCSD.NET.

» Други инструктори с опит като преподаватели и програмисти.

Академията

» **Национална академия по разработка на софтуер** (**НАПС**) е център за професионално обучение на софтуерни специалисти.

» **НАПС** провежда задълбочени курсове по разработка на софтуер и съвременни софтуерни технологии.

» Предлагани специалности:

.NET Enterprise Developer

Java Enterprise Developer

» **Качествено обучение** с много **практически упражнения**

» Завършвате само за 3 месеца.

» **Гарантирана работа** след успешно завършване!

» Професионална сертификация!

» **БЕЗПЛАТНО!**

Учите безплатно, плащате след като завършите и започнете работа.

Стипендии от софтуерни фирми.

http://academy.devbg.org

Глава 5. Тестване, оценка и усъвършенстване

Една от основните идеи, залегнали при проектирането и имплементирането на системата за цифрово подписване на документи в уеб среда, е тя да бъде платформено независима и да работи под всички масово използвани уеб браузъри и операционни системи.

5.1. Поддържани платформи

Сървърната част на системата е стандартно J2EE уеб приложение и може да работи безпроблемно върху различни хардуерни платформи, операционни системи и J2EE сървъри за приложения.

Клиентската част на приложението е проектирана да използва Java Plug-In, който се поддържа във всички масово използвани уеб браузъри и платформи – Internet Explorer, Mozilla, Firefox, Netscape, Opera и др. под Windows, Linux, Solaris и др.

5.2. Експериментално тестване на системата

По време на тестовете на системата NakovDocumentSigner използвахме следния софтуер от страна на сървъра:

- Windows XP SP2
- Java Runtime Environment версия 1.4.2
- сървър за J2EE уеб приложения Apache Tomcat версия 5.0.19
- Struts framework версия 1.2

От страна на клиента системата успешно тествахме с:

- Java Plug-In 1.4 (за аплета с PKCS#12 хранилище) и Java Plug-In 1.5 (за аплета със смарт карта)
- уеб браузъри Internet Explorer 5.0, Internet Explorer 6.0, Mozilla 1.3, Mozilla 1.7, Firefox 1.0, Netscape Communicator 4.5 и Netscape 6.1 върху Windows 98, Windows 2000, Windows XP и Windows 2003 Server
- Mozilla 1.3 върху Red Hat Linux 8.0 (с графична среда KDE 3.0)
- смарт карта Utimaco Safeware с PKCS#11 драйвери Utimaco SafeGuard Smartcard Provider и карточетец ACR38
- цифрови сертификати от VeriSign, Thawte, GlobalSign и StampIT

Върху всички посочени платформи и уеб браузъри системата работи нормално и без проблеми.

Системата беше разгледана и тествана и от специалисти от Асоциация за информационна сигурност (ISECA), както и от експерти, работещи в областта

на удостоверителните услуги от „Информационно обслужване” АД (StampIT). Изказаното мнение беше като цяло позитивно. Критиките бяха свързани основно с необходимостта от доверие към подписания Java аплет, което може да притесни потребителя, както и да наруши сигурността на локалната му машина.

Части от системата бяха успешно използвани и внедрени в комерсиален проект, свързан с управлението на документи в държавната администрация, изпълнен от екип специалисти от фирма Технологика.

5.3. Недостатъци и проблеми

Ще разгледаме най-важните проблеми и недостатъци на системата NakovDocumentSigner:

- Аплетите за подписване изискват наличие на Java Plug-In, който не е стандартна част от уеб браузърите, а се инсталира към тях допълнително. Изтеглянето и инсталирането на Java Plug-In може да създаде трудности, например ако потребителят няма права да инсталира софтуер върху работната станция.
- Аплетите за подписване не използват стандартните хранилища за сертификати на уеб браузърите. Това създава неудобство на потребителя, като го отделя от унифицирания достъп до сертификатите, с който вероятно е свикнал. Допълнително неизползването на вградените в браузърите хранилища за сертификати може да създаде рискове за сигурността – например PIN кодът на смарт картата или паролата за достъп до PKCS#12 хранилището биха могли да бъдат подслушани полесно, а файловете със сертификати могат да бъдат копирани с помалко усилия.
- При изпълнение под Java Plug-In 1.4 потребителят не може да укаже, че вярва перманентно на даден подписан аплет, а трябва да потвърждава доверието си в него при всяко негово стартиране. Този проблем е решен при Java Plug-In 1.5.
- Системата подписва само файлове, но не може да подписва уеб форми. При някои сценарии това може да се окаже неудобно.
- Подписването на файл се осъществява отделно от изпращането на HTML формата. Това позволява формата да бъде изпратена без подпис или подписът да бъде променен преди изпращането на формата. И в двата случая сървърната част ще регистрира проблем със сигнатурата, но това може да създаде неудобства.
- Подписаният файл се доставя отделно от неговия цифров подпис и от сертификата. Не е използван стандартът PKCS#7 (ftp://ftp.rsasecurity.com/pub/pkcs/doc/pkcs-7.doc), който позволява да се запишат в структура SignedData като едно цяло подписаният файл, неговата сигнатура и използваният при подписването сертификат (заедно с пълната му сертификационна верига).

- Системата за верификация на сертификати не поддържа CRL списъци от анулирани сертификати и не ползва OSCP протокола за проверка на сертификати в реално време.
- Системата позволява работа само с X.509 сертификати. Не се поддържат широкоразпространените PGP сертификати.

5.4. Бъдещо развитие и усъвършенстване

Системата NakovDocumentSigner има добри перспективи да се разшири и подобри в бъдещите си версии. Ето някои от посоките за подобрение:

- Може да се имплементира възможност за подписване на уеб форми – да се подписва не само даден файл, а всички полета от дадена HTML форма, заедно с прикачените към нея файлове. Това ще разшири приложението на системата и ще позволи тя да се ползва в много повече уеб базирани системи.
- Може да се имплементира възможност за подписване на уеб форма и съхранение на резултата като „подписан XML" по стандарта XMLDSIG (http://www.w3.org/TR/xmldsig-core/). Това ще отвори системата за полесно използване в хетерогенна среда.
- Може да се реализира съхраняване като едно цяло в PKCS#7 SignedData обект на подписания файл, неговата сигнатура и използвания сертификат (заедно с пълната му сертификационна верига). Това ще улесни сървърните приложения, които проверяват цифровия подпис и използвания сертификат.
- Системата за верификация на сертификати може да се разшири да поддържа CRL списъци от анулирани сертификати и OSCP протокола за проверка на сертификат в реално време. Това е все по-често изискване при изграждане на критични за сигурността информационни системи.
- Би било добре системата да не използва Java Plug-In, защото това създава нужда от инсталиране на допълнителен софтуер и не позволява да бъде използвано хранилището за сертификати на уеб браузъра. Този въпрос може да се реши в бъдеще, когато напреднат уеб стандартите за работа с цифрови подписи и сертификати.
- Може да се реализира поддръжка и на PGP сертификати, макар и те да нямат толкова голямо приложение при извършването на електронни транзакции.

НАПС
НАЦИОНАЛНА АКАДЕМИЯ ПО РАЗРАБОТКА НА СОФТУЕР

НАЦИОНАЛНА АКАДЕМИЯ ПО РАЗРАБОТКА НА СОФТУЕР

Лекторите

» Светлин Наков е преподавател по съвременни софтуерни технологии в СУ "Св. Климент Охридски".

Той е автор на десетки научни и технически публикации и няколко книги, свързани с разработката на софтуер, заради което е търсен лектор и консултант.

През 2004 г. получава наградата "**Джон Атанасов**" от президента на България Георги Първанов за приноса му към развитието на информационните технологии и информационното общество.

» Мартин Кулов е изпълнителен директор във фирма "Код Атест", където разработва проекти за повишаване качеството на софтуерните продукти в България чрез автоматизация на процесите и внедряване на системи за управление на качеството.

Мартин е опитен лектор и сертифициран от Майкрософт разработчик по програмите MCSD и MCSD.NET.

» Други инструктори с опит като преподаватели и програмисти.

Академията

» **Национална академия по разработка на софтуер** (**НАПС**) е център за професионално обучение на софтуерни специалисти.

» **НАПС** провежда задълбочени курсове по разработка на софтуер и съвременни софтуерни технологии.

» Предлагани специалности:

.NET Enterprise Developer

Java Enterprise Developer

» **Качествено обучение** с много **практически упражнения**

» Завършвате само за 3 месеца.

» **Гарантирана работа** след успешно завършване!

» Професионална сертификация!

» **БЕЗПЛАТНО!**

Учите безплатно, плащате след като завършите и започнете работа.

Стипендии от софтуерни фирми.

http://academy.devbg.org

Заключение

Технологията на цифровия подпис все повече навлиза в ежедневието ни. С въвеждането на закона за електронния документ и електронния подпис и напредъка на електронното правителство нуждата от програмни средства за приложение на цифровия подпис в различни информационни системи също нараства. Липсата на единен технологичен стандарт, който позволява подписване на уеб форми и файлове във всички уеб браузъри, става все по-важен проблем.

В нашата работа именно този проблем беше анализиран в дълбочина и бяха разгледани подходите за неговото решаване. Направен бе преглед на технологиите, свързани с цифровия подпис, цифровите сертификати и инфраструктурата на публичния ключ, анализирани бяха съществуващите технологии за подписване в уеб среда и бяха дискутирани техните предимства и недостатъци.

В резултат на анализите се стигна се до решението да се реализира система за цифрово подписване на документи в уеб среда чрез Java аплет и Java базирано уеб приложение. Бяха разгледани проблемите при подписването на документи с PKCS#12 хранилище за сертификати и със смарт карта. Бяха проучени стандартните средства на Java платформата за работа с цифрови подписи и сертификати, както и подходите за верификация на сертификати и сертификационни вериги.

След всички направени проучвания бе реализирана системата за цифрово подписване на документи в уеб среда NakovDocumentSigner. Тя бе изградена като типично клиент-сървър приложение. От страна на клиента работи стандартен уеб браузър с HTML форма, в която са вградени Java аплети, които подписват документи с PKCS#12 хранилище и със смарт карта. От страна на сървъра работи Struts базирано J2EE уеб приложение, което посреща изпратените данни и проверява цифровия подпис, сертификат и сертификационна верига.

Системата NakovDocumentSigner е работещ пример, който илюстрира един платформено независим подход за използване на цифрови подписи от Java-базирани уеб приложения. Тя решава проблемите, които възникват при подписване на документи и файлове на машината на клиента, в неговия уеб браузър, и демонстрира как със средствата на Java платформата могат да се верифицират цифрови подписи, сертификати и сертификационни вериги.

NakovDocumentSigner се разпространява напълно свободно и може да бъде използван в чист или променен вид за всякакви цели, включително и в комерсиални продукти.

Нуждата от информационна сигурност при уеб приложенията непрекъснато нараства и това неизбежно води до развитие и усъвършенстване на свързаните с нея на технологии. Много е вероятно в някои бъдещи версии на най-разпространените уеб браузъри да се появят вградени средства за подписване на документи и HTML форми, използващи за целта сертифика-

тите, инсталирани в тези браузъри, но докато тези средства се появят и се утвърдят, NakovDocumentSigner вероятно ще си остане едно от малкото свободни, платформено независими решения в това направление.

Използвана литература

1. [Johner, 2000] Heinz Johner, Seiei Fujiwara, Amelia Sm Yeung, Anthony Stephanou, Jim Whitmore, Deploying a Public Key Infrastructure, IBM Redbooks, 2000, ISBN 0738415731
2. [Wikipedia-PKC, 2005] Wikipedia, Public-key cryptography, 2005 – http://en.wikipedia.org/wiki/Public-key_cryptography
3. [Steffen, 1998] Daniel Steffen, PowerCrypt - a free cryptography toolkit for the Macintosh, Macquarie University, Sydney, Australia, 1998 – http://www.maths.mq. edu.au/~steffen/old/PCry/report/node9. html
4. [PGP, 1999] PGP 6.5.1 User's Guide in English, An Introduction to Cryptography, Network Associates, Inc., 1999 – ftp://ftp.pgpi.org/pub/pgp/6.5/docs/english/IntroToCrypto.pdf
5. [NZSSC, 2005] New Zealand State Services Commission, PKI Architectures and Trust Models, 2005 – http://e.govt.nz/docs/see-pki-paper-4/chapter3.html
6. [GlobalSign, 1999] Общи условия на публични сертификационни услуги на GlobalSign (Certification Practice Statement – CPS), версия 3.0, раздел 13.1 (дефиниции), 1999 – http://www.bia-bg.com/globalsign.bg/cps_chapter13.htm
7. [Raina, 2003] Kapil Raina, PKI Security Solutions for the Enterprise, John Wiley & Sons Inc., 2003, ISBN 047131529X
8. [Wikipedia, SSC, 2005] Wikipedia, Self-signed certificate, 2005 – http://en.wikipedia.org/wiki/Self-signed_certificate
9. [RFC 3280] R. Housley, W. Polk, W. Ford, D. Solo, RFC 3280 - Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile, 2002 – http://www.rfc.net/rfc3280.html
10. [PKCS#12] PKCS#12: Personal Information Exchange Syntax, version 1.0, RSA Laboratories, 1999 – ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-12/pkcs-12v1.doc
11. [PKCS#11] PKCS #11: Cryptographic Token Interface Standard, version 2.20, RSA Laboratories, 2004 – ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-11/v2-20/pkcs-11v2-20.doc
12. [RFC 2560] M. Myers, R. Ankney, A. Malpani, S. Galperin, C. Adams, RFC 2560 - Internet X.509 Public Key Infrastructure Online Certificate Status Protocol – OCSP, 1999 – http://www.rfc.net/rfc2560.html
13. [MSDN ActiveX] MSDN Library: Introduction to ActiveX Controls, Microsoft, 1996 – http://msdn.microsoft.com/workshop/components/activex/intro.asp

14. [MSDN WinForms] MSDN Library: Developing Windows Forms Controls, Microsoft, 1999 – http://msdn.microsoft.com/library/en-us/cpguide/html/cpconcreatingwinformscontrols.asp

15. [Lambert, 2001] John Lambert, MSDN Library: Introducing CAPICOM, Microsoft, 2001 – http://msdn.microsoft.com/library/en-us/dnsecure/html/intcapicom.asp

16. [Netscape, 1997] Signing Text from JavaScript, Netscape Communications, 1997 – http://docs.sun.com/source/816-6152-10/

17. [Sun, 1995] The Java Tutorial: Writing Java Applets, Sun Microsystems, 1995 – http://java.sun.com/docs/books/tutorial/applet/

18. [JCA, 2004] Java Cryptography Architecture API Specification & Reference, Sun Microsystems, 2004 – http://java.sun.com/j2se/1.5/docs/guide/security/CryptoSpec.html

19. [JCE, 2004] Java Cryptography Extension (JCE) Reference Guide, Sun Microsystems, 2004 – http://java.sun.com/j2se/1.5/docs/guide/security/jce/JCERefGuide.html

20. [Mullan, 2003] Sean Mullan, Java Certification Path API Programmer's Guide, Sun Microsystems, 2003 – http://java.sun.com/j2se/1.5/docs/guide/security/certpath/CertPathProgGuide.html

21. [Sun PKCS#11] Java PKCS#11 Reference Guide, Sun Microsystems, 2004 – http://java.sun.com/j2se/1.5/docs/guide/security/p11guide.html

22. [J2JS, 2004] Java Plug-in Developer Guide for J2SE 5.0: Java-to-Javascript Communication, Sun Microsystems, 2004 – http://java.sun.com/j2se/1.5/docs/guide/plugin/developer_guide/java_js.html

23. [Nakov, 2004] Svetlin Nakov, NakovDocumentSigner: A System for Digitally Signing Documents in Web Applications, Developer.com, 2004 – http://www.developer.com/java/web/article.php/3298051