Univerza v Ljubljani

Fakulteta za računalništvo in informatiko

Seminarska naloga:

Java Crypto

JAVANSKA KRIPTOGRAFSKA KNJIŽNICA

iz predmeta

Kriptografija in računalniška varnost

pri

Prof. Dr. A. Jurišiču

Roman Verhovšek

December, 2000

1. KAZALO

2. POVZETEK

Naloga te seminarske naloge je:

1. Pregledati javansko tehnologijo z vidika varnosti (Java Cryptography API oz. JCA na kratko), ki pokriva varovanje podatkov, programov in samega računalnika, ter spremembe, ki jih prinašajo nove verzije.

2. Preveriti varnostne luknje JCA in odpravo le-teh.

3. Preveriti kriptografski del JCA, ki se nahaja v Java Cryptography Extension knjižnici oz. JCE na kratko.

4. Preveriti eno izmed implementacij JCE avstralskega podjetja eSec Limited (prej znanega kot Australian Business Access). Njihova implementacija je brezplačna in ponuja zraven tudi izvorno kodo.

5. Preizkusiti hitrost posameznih kriptografskih algoritmov avstralske implementacije JCE.

6. Izdelava preproste uporabniške kriptografske knjižnice, ki jo uporabljamo v podjetju Cocoasoft d.o.o.

3. UVOD

Po informacijah iz ZDA je povpraševanje po javanskih programerjih lani preseglo povpraševanju po C-jaših, kar samo potrjuje dejstvu, da je Java programski jezik številka 1. Odgovor na to, zakaj je postala Java tako popularna, lahko najdemo v njeni preprostosti, obširni knjižnici in njeni orientiranosti na omrežne sisteme, ki postajajo zaradi spleta vedno bolj pomembni v našem življenju.

Ker je varnost ena od pomembnih točk omrežnih sistemov, Sun ponuja v Javi svojim uporabnikov knjižnico JCA (Java Cryptography API), ki zaradi prepovedi izvoza močne kriptografije iz ZDA predstavlja le širok spekter vmesnikov (abstraktnih metod) in legalnih algoritmov, ne pa tudi njihovih implementacij. Te je potrebno zaradi zakonskih ovir posameznih držav razviti v svoji regiji uporabe. Zato so se različne ne-ameriške programske hiše odzvale s svojimi implementacijami, med katerimi izstopa brezplačna knjižnica podjetja eSec Limited, ki zraven ponujajo tudi kompletno izvorno kodo.

Ker je kriptografija le podmnožica varnosti v Javi, bodo naslednja poglavja obdelala zaščito uporabnika javanskih programov, podpisovanje distribuirane izvršilne kode, novosti v Javi 2, varnostne luknje skozi zgodovino Jave in na koncu še kriptografijo samo.

4. VARNOST V JAVI

Beseda Java ima v računalniškem svetu dva pomena:

1. Java je programski jezik (razlog za malo začetnico)
2. Java je tudi navidezni računalnik, ki teče znotraj našega računalnika (Java Virtual Machine oz. JVM)

Torej ima beseda drugačen pomeni pri razvijalcih kot pri uporabnikih.

Slabost Jave kot jezika se kaže v zmožnosti neke tretje osebe, da lahko s preprostimi orodji (imenovanimi decompilerji) javansko zbirniško kodo (imenovano tudi byte kodo) prevede nazaj v izvorno kodo, ne da bi pri tem izgubil katerikoli del kode (celo imena spremenljivk ostanejo ista!!!). Sun je celo razglašal, da ima JVM preverjevalec zbirne kode (verifier), ki zazna spremembo določenega razreda, pa se je potem izkazalo, da stvar ne deluje, kot bi morala. Rešitev je prišla v obliki zavajevalcev (obfuscators), ki prevedeno kodo preimenujejo, zamešajo, stisnejo in kodirajo, tako da je otežen 'reverse engineering'. Prav tako je možno v Javi digitalno podpisati razrede, tako da lahko ugotovimo integriteto naloženih razredov.

Z vidika navideznega stroja pa nastopi problem v tem, da Java originalno ne vsebuje implementacij kriptografije. To pomeni, da uporabnik spletne javanske aplikacije (imenovane applet) vsakič preko spleta naloži kriptografski modul, tega pa lahko neka tretja oseba prestreže in tako uporabniku ponudi svojo verzijo, ne da bi uporabnik ti tudi opazil. Da se izognemo taki situaciji, mora vsak uporabnik sam nastaviti varnostno knjižnico na lokalnem disku, tako da se le-ta ne naloži preko spleta.

Seveda pa ni vse tako črno. Java sama po sebi omogoča veliko varnostnih mehanizmov, ki jih drugi (npr. ActiveX, JavaScript, itd.) ne omogočajo. Te so sledeče:

- Programček (applet), ki se naloži s strežnika, nima dostopa do uporabnikovih računalniških sredstev (npr.: ne more formatirati diska, brati podatkov, itd.). S tem je dosežena zaščita na strani odjemalca. Java 2 omogoča nastavljanje dostopnosti, če naredimo razrede zaupanja vredne (trusted). Iz te točke lahko vidimo, zakaj pri Sunu pravijo, da deluje JVM kot peskovnik (sandbox). Programčki so kot otroci, ki pacajo po pesku znotraj samega peskovnika (JVM), izven pa ne, ker jih omejuje ograja.
- Programček, ki se naloži s strežnika, se ne more povezati z na noben drug strežnik kot na tistega, s katerega se je naložil. S tem smo dosegli zaščito z vidika strežnika. Če bi radi dosegli povezavo na kak drugi strežnik, mora naš strežnik delovati kot proxy.

Vsi programčki so zaupanje nevredni, razen če niso drugače definirani. To pomeni, da ne morejo:

- Brati datotek z diska,
- Pisati po datotekah z diska,
- Brisati datoteke,
- Preimenovati datoteke,
- Kreirati imenike,
- Izlistati vsebino imenikov,

- Preveriti obstoj datoteke,
- Dobiti informacije o datoteki,
- Narediti omrežno povezavo z računalnikov, s katerega se niso naložili,
- Sprejemati povezav,
- Kreirati GUI okno brez opozorilne vrstice,
- Prebrati sistemske lastnosti, uporabniškega imena in gesla,
- Definirati sistemske lastnosti,
- Zagnati domorodne (native oz. ne-javanske) aplikacije,
- Klicati metodo System.exit(),
- Nalagati dinamične (DLL, SO ali podobne) knjižnice,
- Manipulirati z nitmi (threads), ki ne spadajo v isto nitno skupino,
- Kreirati nalagalnik razredov,
- Kreirati nadzornika varnosti,
- Specificirati omrežne nadzorne funkcije in
- Definirati razrede, ki so v istem paketu (package) kot lokalni razredi.

4.1. PESKOVNIK

Peskovnik je osnova javanskega navideznega stroja z vidika varnosti. Predstavlja del delovanja Jave in se deli na tri dele, ki so med seboj povezani:

- Preverjevalnik (verifier),
- Nalagalnik razredov (class loader) in
- Varnostni nadzornik (security manager).

4.1.1. PREVERJEVALNIK

Preverjevalnik preverja byte kodo, če ustreza standardom in če se notri ne nahaja nezaupljiv del kode. Je del JVM in zaradi tega nedostopen programerjem. Poleg tega preverja, če so razredi med seboj binarno povezljivi (npr.: ni problema kasneje dodati novo metodo, je pa problem kakšno metodo kasneje zbrisati). Dostikrat uporabniki narobe mislijo, ko rabijo nekaj časa za zagon aplikacije. Preverjevalnik dostikrat vzame več časa kot pa samo nalaganje kode.

Proces preverjanja je razdeljen v dva dela:

- Interno preverjanje, ki preveri kompleten razred, in
- Tekoče (runtime) preverjanje, ki potrdi obstoj in prenosljivost referenc na razrede, polja (fields – globalne spremenljivke) in metod.

Ko preverjevalnik opravi delo s pozitivnim rezultatom, lahko o razredu vemo sledeče lastnosti:

- Razred je pravilnega formata,
- Sklad (stack) ne bo prekoračen (overflowed in underflowed),
- Byte kodni ukazi so pravilni s pravilnimi parametri,
- Ni ilegalnih podatkovnih pretvarjanj (type casting),
- Dostopnosti (private, protected, public in default) so legalne in
- Vsi dostopi in shranjevanja na registre so legalni.

Različne organizacije so dokazale, da lahko kreiramo škodljivo byte kodo, ki jo preverjevalnik sprejme za neškodljivo.

4.1.2. NALAGALNIK RAZREDOV

Ena od prednosti Jave je v mobilnosti kode. To pomeni, da je aplikacija razbita na koščke (razrede), ki se naložijo dinamično z diska ali omrežja, ko jih potrebujemo. Za to poskrbijo nalagalniki razredov. Ti skrbijo za dve stvari:

- Preko ustreznih metod poiščejo in naložijo razrede, in
- Definirajo prostore poimenovanj (namespaces), v katerem hranijo reference na posamezne razrede in njihove odnose med seboj.

Nalaganje razredov poteka v sledečih korakih:

1. Preveri, če je bil razred že naložen. Če je bil, potem ga vrni kot rezultat.
2. Uporabi primarni nalagalnik razredov, ki poskuša naložiti razred iz imenikov, na katere kaže sistemska spremenljivka CLASSPATH. Ti razredi so zaupanja vredni.
3. Drugače naloži razrede preko omrežja. Ti razredi so nezaupanja vredni.
4. Razred je prebran kot niz (array) 0-bitnih celoštevilskih vrednosti.
5. Konstrira se objekt tipa Class.
6. Kličejo se statični bloki.
7. Preveri razred s preverjevalnikom.

4.1.3. VARNOSTNI NADZORNIK

Naloga varnostnega nadzornika je v preverjanju, kdo lahko dostopa do določenih sredstev in metod. To delo opravlja v tekočem času (runtime).

Koraki, ki jih opravi nadzornik, so sledeči:

1. Javanski program kliče potencialno nevarno operacijo.
2. Nadzornik preveri, če se to lahko zgodi.
3. Če se ne more, potem nadzornik vrže SecurityException izjemo.
4. V drugačnem primeru nadzornik ne naredi ničesar.

Naloge nadzornika so sledeče:

- Preprečuje nastavitev novih nalagalnikov razredov,
- Ščiti niti in skupine niti,
- Nadzira izvrševanje programa,
- Nadzira končanje programa,
- Nadzira dostop do računalnikovih sredstev,
- Nadzira operacije z diski,
- Nadzira operacije na omrežju in
- Nadzira dostop do javanskih paketov.

4.1.4. VARNOST V RAZLIČNIH VERZIJAH JAVE

V Javi 1.0 se je le koda, naložena z omrežja preverila s preverjevalnikom, medtem pa je koda naložena z lokalnega diska obšla preverjanje.

V Javi 1.1 je veljalo isto, le da je sedaj koda, naložena z omrežja, lahko tudi digitalno podpisana in s tem tudi zaupanja vredna. Tudi tu je sistem črno/bel – torej imamo za le zaupanja vredne in zaupanja nevredne programčke.

V Javi 2 (Java 1.2 in Java 1.3) pa se stvar zaplete. Digitalno podpisani programčki so sedaj delno zaupanja vredni glede na zaupanje in dostopnost do JVM.

4.2. PODPISANA KODA

Z digitalnim podpisovanjem kode dobimo boljši nadzor v varnosti na mobilni kodi. S tem pridobimo na:

- Možnosti dajanja privilegijev,
- Možnosti, da koda operira s čim manj privilegiji, in
- Možnosti, da bližje nadziramo sistemsko varnostno konfiguracijo.

Podpisovanje je možno preko Java Cripto API, ki ponuja prstne odtise imenovane zgoščevalne funkcije. Te bazirajo na MD5 in SHA algoritmih, lahko pa tudi uporabimo DES enkripcijo, ki pa je del JCE.

Druga možnost je delo z X.509v3 cetifikati, ki jih Java 1.1 podpira. Izmenjava je možna preko Java SSL, ki je podoben Netscapovem, in je podprt v Javi 2.

4.3. VARNOST V JAVI 2

Nova varnostna arhitektura v Javi 2 prinaša štiri lastnosti:

- Določanje posameznih dostopnosti,
- Konfiguracijo varnostnih načel,
- Razširljivo strukturo dostopnosti in
- Varnostna preverjanja v vsej javanskih programih.

Glavno vodilo nove varnosti je v dejstvu, da se sedaj vse preverja – tudi zaupanja vredne aplikacije. Prav tako so uvedeni novi pojmi, kot so:

- Identiteta (identity),
- Dovoljenja (permission),
- Implikacije (implies),
- Načela (policy),
- Domene zaščite (protection domains),
- Dostopnost (access control) in
- Privilegiji (privilege).

4.3.1. IDENTITETA

Vsak del kode potrebuje svojo identiteto, ki je sestavljena iz svoje izvora (origin) in podpisa (signature). Mnogi mislijo, da nam identiteta pove, od kod prihaja koda in kdo jo je napisal. To seveda ni res. V resnici le vemo, kdo jo je distibuiral.

4.3.2. DOVOLJENJA

Zahteve po izvedbi določene operacije lahko spravimo v dovoljenje. Načela nam lahko povejo, katera dovoljenja so odobrena in katere ne.

4.3.3. IMPLIKACIJE

Vsako dovoljenje mora implementirati metodo implies(), ki kaže na implikacije med različnimi dovoljenji (npr. dovoljenje X avtomatično omogoča tudi dovoljenje Y).

4.3.4. NAČELA

Načela so mapiranja identitete in jih običajno nastavlja sistemski administrator. Lahko jih tudi uporabnik, kar pa mogoče ni dobro iz varnostnih razlogov. Običajno so zapisane v tekstovni datoteki na sistemu in vsebujejo spisek dovoljenj, ki jih omogočimo.

4.3.5. DOMENE ZAŠČITE

Domene zaščite niso nič drugega kot skupina razredov, ki jih obravnavamo z enakimi pogoji. Te razrede grupiramo s paketi.

4.3.6. DOSTOPNOST

Dostopnost je realizirana v razredu AccessController, ki preverja sklade. Metoda ob napaki vrne izjemo, drugače pa se ne zgodi nič.

4.3.7. PRIVILEGIJI

S privilegiji lahko omejimo delovanje posameznim delom kode. Na začetku je moral programer sam poskrbeti v try/finally bloku za začetek in konec priviligiranega dela, kasneje pa so to implementirali z abstraktnim razredom PrivilegedAction. Tega moramo implementirati kot notranji (inner) razred, kar pa je seveda zelo ironično, saj nam luknje v varnosti pravijo, da se notranjim razredom rajši izognimo.

5. VARNOSTNE LUKNJE

Skozi kratko zgodovino razvoja Jave se je do izida knjige o varnosti Jave [3] pojavilo 15 resnih varnostnih lukenj, ki pa so jih pri Sunu hitro odpravili z novimi podverzijami. Luknje so odkrile različne organizacije ali posamezniki, ki so na srečo rajši stopili v kontakt s Sunovimi strokovnjaki kot pa novo-odkrite luknje izkoristili za zlorabo. Javnost ni nikoli do sedaj predčasno izvedela za te luknje, preden bi prišli ven popravki.

5.1. SKAKANJE ČEZ POŽARNI ZID

Datum:
Februar 1996

Organizacija:
Princeton University (Dean, Felten in Wallach)

Opis:
Pravilo vsakega programčka je, da se lahko poveže le s tistim strežnikom, s katerega je prišel. Java je to preverila na tak način, da je pač primerjala IP številko računalnika, na katerega se je programček hotel povezati, s tisto, ki jo ima web strežnik (npr. 172.16.16.16), s katerega je prišel. Ker ima lahko vsak računalnik več IP naslovov, je Java IP naslov dobila kar od DNS strežnika, v kateri domeni se je ta-isti web strežnik nahajal. Zato je lahko napadalec v DNS strežnik vnesel še IP naslov računalnika, katerega je pač hotel napasti (v našem primeru torej 172.16.16.16 in 10.10.10.2). Od obeh IP številk je ena pravilna, zato se je lahko programček povezal na žrtev in poskušal z raznimi vdori.

Popravek:
Sun je takoj to napako odpravil na preprost način – na začetku se shrani IP naslov web strežnika.

5.2. SLASH AND BURN

Datum:
Marec 1996

Organizacija:
Oxford Univeristy (Hopwood)

Opis:
V prvi verziji Jave so vsi programčki, naloženi z lokalnega diska, bili zaupanja vredni (trusted), in vsi, naloženi z omrežja, pa ne (untrusted). Ker Java zna dinamično nalagati razrede, bi lahko nekdo preko ftp ali javnih imenikov naložil zlonamerni razred, ki bi ga potem lahko zagnali iz HTML strani. Brskalniki namreč vedno najprej preverijo, če se razred nahaja na disku, potem pa še na omrežju. Kako ve Java, kateri razred naložiti? Preprosto: iz imena (npr. com.cocoasoft.A je razred A iz imenika com/cocoasoft/). Java je znala . spremeniti v \ oz. / (odvisno od operacijskega sistema), tako da se je lahko

prebila do potrebnih razredov in jih zagnala. Vse kar je bilo potrebno, je le shraniti zlonamerni razred na potrebni lokaciji in s potrebnim tekstom zagnati razred (npr. /home.romanv.ByeBye).

Popravek:
Java ne podpira več znakov \ in /, ampak samo še ., tako da ni več možno skakati po absolutnih imenikih.

5.3. PROBLEM S TIPI

Datum:
Junij 1996

Organizacija:
Princeton University (Balfanz, Dean in Felten)

Opis:
Odkrito je bilo, da bytecode datoteka, ki jo naloži Java, lahko vsebuje niz novega tipa, ki vrne napako, a se vseeno zapiše v interno tabelo. To lahko omogoči penetracijo v sistem.

Popravek:
Napaka je bila popravljena v Netscape Communicatorju 3.0. Ekipa iz Princetona pa je razvila Type Confusion Toolkit, ki zaradi nevarnosti uporabe ni dostopen javnosti. Je pa bil uporabljen pri razvoju Jave 2.

5.4. PODIVJANI PROGRAMČKI

Datum:
Marec 1996

Organizacija:
Princeton University (Dean, Felten in Wallach)

Opis:
Nalogo nalaganja razredov je v Javi prevzel tako imenovani nalagalnik. Nalagalnik skrbi za to, da so razredi pravilno naloženi in to le enkrat. To doseže s sledečim zaporedjem nalaganja:

1. Kliče se metoda loadClass(), ki sprejme ime razreda.
2. Nalagalnik preveri svoje slovarje (vsak razred ima svojega) in če najde razred v njem, le tega vrne.
3. V drugačnem primeru se razred naloži z lokalnega disk ali omrežja.
4. Nalagalnik kliče metodo defineClass(), ki iz byte kode kreira razred.
5. Nato vrne kreiran razred.

Ker je nalagalnik razred, bi lahko vsakdo razširil ClassLoader in na novo implementiral nalaganje razredov. Tako bi se lahko namesto enega razreda naložila dva (eden bi bil zlonameren). Problem nastopi v tem, da Java ne bi smela pustiti razširjati nalagalnika. Vendar se je to v stari verziji Jave dalo. In ne samo to. Vsak konstruktor avtomatično

kliče konstruktor svojega starševskega razreda in ta bi moral vreči varnostno izjemo (exception), vendar pa se da temu izogniti.

Popravek:
V ClassLoader razredu so uvedli privatno spremenljivko, ki se postavi na true le, če se kliče starševski konstruktor.

5.5. PRETVARJANJE TIPOV

Datum:
Maj 1996

Odkril:
Tom Cargill s pomočjo Princeton University

Opis:
Java omogoča tako imenovani omejevanje dostopnosti metod pri dedovanju. To pomeni, da če je metoda v enem razredu javna, jo lahko v svojem nasledniku omejimo na privatno uporabo. Vse metode v vmesnikih so vedno javne. Toda ugotovili so, da ne glede na to, da ima razred privatno metodo, ki je bila v vmesniku javna, se do te metode da dostopati.

Popravek:
Netscape je to napako rajši odpravil na bolj zapleten način – popravil je javanski navidezni stroj. To se je pozneje obrestovalo, kajti po odpravi napake se je pozneje ugotovilo, da če sta dva razreda implementirala isti vmesnik in le eden od teh je dostopnost metode spremenil v privatno, in če smo najprej kreirali objekt razreda z javno metodo, nalagalnik sploh ni pri kreiranju objekta drugega razreda več preveril, ali je metoda privatna. Prav tako pozneje luknja pri predpomnilniku ni delovala na Netscapeu.

5.6. ZAZNAMKI

Datum:
Junij 1996

Organizacija:
Oxford University (Hopwood)

Opis:
Vsaka web stran ima lahko na sebi več programčkov, ki med seboj komunicirajo preko javnih objektov ali posebne niti. V starih verzijah Jave se je lahko zgodilo, da so si programčki različno predstavljali določen razred. Npr. razred Security je lahko pri enem programčku imel privatno metodo, pri drugem pa javno. V tem primeru, bi se dostopnost spremenila že pri prenosu objekta med programčki. Nalagalnik na srečo poskrbi za verodostojnost na tak način, da poleg imena razreda primerja tudi ime nalagalnika, ki je razred naložil. Problem je nastopil pri prvih verzijah Jave, ker druge primerjave ni bilo.

Popravek:
Popravek je prišel z Javo 1.1.

5.7. PAKETI

Datum:
Avgust 1996

Organizacija:
Princeton University (Balfanz in Felten)

Opis:
Vsi razredi so shranjeni v paketih, katere naloga je sledeča:

1. Poskušajo se izogniti problemu poimenovanj razredov (npr. oseba X lahko razvija razred Calender, ki pa že obstaja v Javi). Ker je pravo ime razreda sestavljeno iz imena paketa in razreda samega (npr. java.lang.String), do problemov podvojitve ne more priti – seveda ob dejstvu, da programerji uporabljajo Sunovo specifikacijo poimenovanja (predpona paketov naj bo kar obratno ime domene – npr.: com.cocoasoft).
2. Poskušajo omejiti dostopnost:
    - Private – dostopnost znotraj razreda
    - Protected – dostopnost znotraj razreda in v vseh razširjenih razredih
    - Public – dostopnost povsod
    - PRAZNO – dostopnost znotraj razreda in v samem paketu

Problem je nastal pri Internet Explorerju 3.0 Beta pri zaupanja nevrednih programčkih, ki so se postavili v paket com.ms, v katerem so se nahajali vsi Microsoftovi zaupanja vredni razredi. Ti programčki so zato dobili dostopnost do vseh računalnikovih virov in so lahko sistem spravile na kolena.

Popravek:
Kljub že najavljenim hitro prihajajočem roku izdaje brskalnika so programerji Microsofta v zadnjem hipu odpravili napako.

5.8. KRAJA IP

Datum:
Februar 1997

Odkrili:
Hekerji

Opis:
Angleška hekerja sta odkrila pri Netscape Communicatorju, da lahko programček, zagnan na odjemalcu, z metodo InetAddress.getLocalHost() dobi odjemalčevo IP številko, ne glede na to, da se nahaja za požarnim zidom.

Popravek:
Sun je to napako odpravil že 10 mesecev pred odkritjem. Rešitev je bila preprosta: metoda vrne »localhost/127.0.0.1«.

5.9. NATRPANI PREDPOMNILNIK

Datum:
Februar 1997

Odkrili:
Hekerji

Opis:
Ta napaka se je pojavila le pri Internet Explorerju, ker jo je Netscape že odpravil pri napadu Slash and burn. Problem je nastopil, ker:

1. Explorer shranjuje HTML in Java razrede v istem cache imeniku pod nespremenjenimi imeni.
2. S predpono file:// lahko zaženemo programčke tudi iz predpomnilnika in taki programčki imajo popolni dostop do sistema (so zaupanja vredni).

Obiskovalec web strani je moral le dvakrat priti na isto stran. Prvič, da se je programček naložil v predpomnilnik. In drugič, da ga je ponovno zagnal iz predpomnilnika.

Popravek:
Napaka je bila odpravljena na preprost način – file:// ne more dostopati do predpomnilnika razen, če ga ne damo v CLASSPATH.

5.10. NAVIDEZNI VOODOO

Datum:
Marec 1997

Podjetje:
JavaSoft

Opis:
Sun je odkril napako v zvezi s preverjevalnikom, kar je tudi javno priznal v času popravka. Na žalost se zelo malo ve o tej napaki.

Popravek:
Predvideva se, da je nekdo od zunaj odkril to napako, ker drugače Sun ne bi dal javne objave. Napaka je bila odpravljena.

5.11. ČAROBNI PLAŠČ

Datum:
April 1997

Organizacija:
Princeton University

Opis:
Princeton team je odkril, da metoda Class.getSigners() vrne spisek vseh uporabnikov, ki jih javanski navidezni stroj pozna. Vsak uporabnik ima namreč svoje ključe, certifikate in privilegije. Ta spisek je lahko programček popravil, tako da je lahko določena oseba dobila privilegije neke druge osebe, ne da bi sistem za to vedel.

Popravek:
Metoda po novem vrne kopijo spiska in ne spiska samega. Tako se spremembe na spisku ne odražajo na sistemu. Na srečo je bila ta napaka odkrita v času, ko niti Netscape niti Microsoft nista podpirala Jave 1.1.

5.12. PREVERJEVALNIK

Čas:
Maj 1997

Organizacija:
University of Washington (Bershad and Kilmera Group)

Opis:
Kimera Group je pod vodstvom profesorja Brian Bershada hotela razviti centraliziran preverjevalnik, ki bi zreduciral kodo na odjemalčevi strani in ta bi bil majhen, preprost in neodvisen. Ker so ga morali preveriti, so avtomatizirali proces testiranja z milijoni različnih vzorcev byte kode. Da pa bi lahko odkrili možne napake, so se odločili, da bodo iste teste izvedli tudi na drugih komercialnih preverjevalnikih. Slaba stran takega preverjanja je v tem, da napačen odziv na vseh preverjevalnikih bi pomenil pozitiven rezultat testa, a so na tak način vseeno odkrili 24 napak na Sunovi in 17 na Microsoftovi verziji Jave. Med njimi je bila največja napaka v dejstvu, da se je dalo primitivno spremenljivko tipa long ali double spremeniti v kazalec na objekt in s tem dostopati do različnih podatkov.

Popravek:
Sunu je uspelo še pred objavo Kimere Group izdelati popravek, pa še to le za največjo napako. Ostali popravki so prišli pozneje.

5.13. VAKUUMSKI HROŠČ

Datum:
Julij 1997

Organizacija:
Kimera Group

Opis:

Ker je Sun najavil, so vsi hrošči najdeni pri Kimeri Group nepomembni razen enega, so se pri Kimeri Group odločili podrobneje preveriti enega od nepomembnih. Ugotovili so, da lahko določena byte koda pride čez preverjevalnik, pa vseeno sesuje sistem zaradi dostopa do nedovoljenega dela pomnilnika. Poleg tega, se je dalo z ukazi move na Stringih dostopati do informacij v brskalniku, kot na primer: e-mail naslov, cache, zgodovina brskanja,...

Popravek:
Ni znano.

5.14. LOOK OVER THERE

Datum:
Avgust 1997

Odkril:
Ben Mesander

Opis:
Ben Mesander iz Colorada je odkril, da lahko preko CGI skripte preusmeri povezavo brskalnika (Internet Explorerja in Netscape Communicatorja) na nek tretji strežnik. Napisal je primer, ki naloži sliko iz Microsoftove strani:

```
Image img = getImage(new URL(getDocumentBase(),
  "cgi-bin/redirect?where=" + URLencode(
  "www.microsoft.com/library/" +
  "images/gifts/homepage/tagline.gif")));
```

Popravek:
Ni znano.

5.15. BEAT THE SYSTEM

Datum:
Julij 1998

Organizacija:
Princeton University (Balfanz, Dean, Felten in Wallach)

Opis:
Princeton team je leta 1998 odkril napako v javanskem navideznem stroju (ki se jo je dalo raziskovati le na Netscape Communicatorje 4.0), ki je lahko izklopila varnostne mehanizme v brskalniku. Razlogi so bili sledeči:

1. Netscape Communicator nezaupnim programčkom ne onemogoči kreiranja svojega lastnega nalagalnika razredov.
2. Java 1.2 Beta 3 in Netscape Communicator 4.0 (ter verjetno še Internet Explorer 4.01) včasih omogočijo zlonamernemu nalagalniku razredov, da povozi definicijo vgrajenih razredov.

3. Obstaja hrošč v preverjevalniku, ki pa zaradi varnostnih razlogov ni bil razkrit.

Kombinacija vseh treh točk lahko omogoči napadalcu, da izzove zmedo nad tipi, kar lahko onemogoči varnost.

Popravek:
Šele Netscape Communicator 4.5 je odpravil te napake.

6. KRIPTOGRAFIJA V JAVI

Kriptografija je orodje, ki nam omogoča:

- Zaupnost (nepooblaščeni uporabniki ne morejo videti naše vsebine),
- Integriteto (podatki se niso spremenili brez naše vednosti) in
- Avtentikacijo (nam zagotavlja, da komuniciramo s pravo osebo).

Zaupnost nam omogočajo:

- Simetrični ključi,
- Asimetrični ključi,
- Hibridni sistemi (kombinacija simetričnih in asimetričnih ključev),
- Distribucija ključev in
- Protokoli za izmenjavo ključev.

Integriteto nam omogočajo zgoščevalne funkcije.

Avtentikacijo nam omogočajo certifikati in agencije, ki nam le-te dajo.

Kriptografija je le del dveh javanskih knjižnic: Java Cryptography API (JCA), ki je del Jave že od verzije 1.1 naprej, in Java Cryptography Extension (JCE), ki jo dobimo pri različnih ponudnikih.

6.1. PONUDNIKI

Java pozna poseben razred z imenom Provider, ki omogoča dostop do različnih kripto knjižnic in je osnova tem knjižnicam. Te so lahko naložene statično (ročna nastavitev konfiguracijske datoteke) ali dinamično (v programu sami določimo našega ponudnika).

Primer dinamičnega dodajanja ponudnika:

```
Security.addProvider(new au.net.aba.crypto.provider.ABAProvider());
```

Ker je možnih več ponudnikov, nam Java sama omogoča uporabo kombinacije le-teh (npr.: naša bančna aplikacija uporablja eno vrsto kriptografije za komunikacijo z banko in drugo vrsto kriptografije za komunikacijo z Agencijo za plačilni promet).

Različni ponudniki ponujajo različne algoritme in uporabnik se mora sam odločiti glede na ceno, hitrost in vpogled v izvorno kodo (ameriške firme se temu izogibajo, ne-ameriške pa ne).

Razred Security drži spisek vseh ponudnikov. Ko se potrebuje določen algoritem, Security objekt vpraša ponudnika, če le ta obstaja.

6.2. ŠIFRIRANJE

Za šifriranje in dešifriranje Java uporablja simetrične (kot sta npr. DES in RC4) in asimetrične (kot sta npr. RSA in ElGamal) algoritme. Algoritmi so lahko tokovni ali bločni, za vsak algoritem pa lahko podamo različne načine delovanja in nastavljamo posamezne parametre.

Glavni razred za (de)šifriranje je razred Cipher, ki ponuja potrebne mehanizme za (de)šifriranje podatkov z uporabo ustreznih algoritmov podanih od ponudnikov. Objekt tipa Cipher kreiramo s statično metodo Cipher.getInstance(). Kot parameter sprejme opisno besedilo, ki je lahko v sledeči obliki:

- Algoritem (npr. RC4) ali
- Algoritem/Način/Podlaga (npr. RSA/ECB/PKCS1Padding).

Po kreaciji je potrebno objekt razreda Cipher inicializirati z metodo Cipher.init(), ki ji podamo način delovanja (šifriranje ali dešifriranje) in potrebni ključ. Inicializacija lahko poteka tudi s pomočjo nastavljanj posameznih parametrov.

Ko je objekt inicializiran, lahko začnemo procesirati podatke z metodo Cipher.update(), nato pa končamo s klicanjem metode Cipher.doFinal().

Za lažje delo nam Java omogoča dva enosmerna tokova CipherInputStream in CipherOutputStream, preko katerih lahko pošiljamo ali sprejemamo niz celoštevilskih 8-bitnih podatkov.

Java pa gre seveda še naprej pri poenostavljanju (de)šifriranja podatkov. Omogoča nam razred SealedObject, ki lahko zašifrira in odšifrira katerikoli serijski objekt (razred mora implementirati vmesnik Serializable).

Primer šifriranja:

```
symCipher.init(Cipher.DECRYPT_MODE, key);
ByteArrayInputStream in = new ByteArrayInputStream(value);
CipherInputStream cin = new CipherInputStream(in, symCipher);
cin.read(value);
cin.close();
in.close();
return nValue;
```

Primer dešifriranja:

```
symCipher.init(Cipher.DECRYPT_MODE, key);
ByteArrayInputStream in = new ByteArrayInputStream(value);
CipherInputStream cin = new CipherInputStream(in, symCipher);
cin.read(value);
cin.close();
in.close();
return value;
```

Pri dešifriranju je potrebno potem še dobljeno vrednost (niz 8-bitnih celih števil) pretvoriti v objekt.

6.3. ZGOŠČEVALNE FUNKCIJE

JCA pozna zgoščevalne funkcije (message digest), ki jih običajno uporabljamo pri avtentikacijah.

Primer uporabe:

```
MessageDigest digest = MessageDigest.getInstance("SHA");
byte msg[] = "Poljubni tekst".getBytes();
digest.update(msg);
byte result[] = digest.digest();
```

6.4. AVTENTIKACIJA SPOROČIL

Za avtentikacijo sporočil (Message Authentication Code - MAC) Java uporablja razred Mac. V resnici je to le abstrakten razred, iz katerega so razširjeni razredi z različnimi algoritmi. Obstajata dve vrsti algoritmov:

- Algoritmi, ki bazirajo na zgoščevalnih funkcijah, in
- Algoritmi, ki bazirajo na šifrirnih algoritmih.

Razred Mac se uporablja na enak način kot razred Cipher – s funkcijami:

- Mac.init(),
- Mac.update() in
- Mac.doFinal().

Za preverjanje sporočila je potrebno ponoviti proceduro in primerjati poslan MAC s tistim, ki ga sami izračunamo.

6.5. DIGITALNI PODPISI

Digitalni podpisi so digitalni ekvivalent klasičnim podpisom s svinčnikom na papir. Uporabljajo se za avtentikacijo avtorja dokumenta kot tudi za dokaz, da je določena oseba tudi res podpisala ta dokument. Bazirajo na metodah šifriranja z javnim ključem, kar pomeni, da lahko kdorkoli, ki poseduje javni ključ, preveri, če je lastnik dokumenta pravi.

Primer podpisovanja dokumenta:

```
Signature sig = Signature.getInstance("MD5withRSA");
sig.initSign(privKey);
byte document[] = "Besedilo".getBytes();
sig.update(document);
byte signature[] = sig.sign();
```

Primer preverjanja podpisa:

```
Signature sig = Signature.getInstance("MD5withRSA");
```

```
sig.initVerify(pubKey);
byte document[] = "Besedilo".getBytes();
sig.update(document);
if (sig.verify(signature))
{
      // OK
}
else
{
      // NI OK
}
```

Poleg podpisovanja dokumentov je možno tudi podpisovati javanske objekte, ki pa morajo biti serijski (razredi morajo implementirati vmesnik Serializable). Podobno kot pri (de)šifriranju uporabimo tu razred SignedObject, ki nam omogoča mehanizme za avtentikacijo objektov in dokazuje, da objekt ni bil spremenjen ali zamenjan brez vednosti avtorja.

6.6. DELO S KLJUČI

JCA/JCE omogoča delo s ključi na dva načina:

- Na način, ki je odvisen od ponudnika, in
- Na način, ki je neodvisen od ponudnika.

Vsak ponudnik omogoča številne mehanizme za generacijo od ponudnika odvisnih ključev ali pretvarjanje od ponudnika neodvisne ključe v od ponudnika odvisne ključe.

Za generacijo simetričnih ključev se uporablja razred KeyGenerator, za generacijo asimetričnih ključev pa razred KeyPairGenerator.

Primer generacije simetričnega ključa:

```
keyGen = KeyGenerator.getInstance("DES");
keyGen.init(64);
secretKey = keyGen.generateKey();
```

Primer generacije asimetričnih ključev:

```
keyPairGen = KeyPairGenerator.getInstance("RSA");
keyPairGen.initialize(1024);
KeyPair keyPair = keyPairGen.generateKeyPair();
publicKey = keyPair.getPublic();
privateKey = keyPair.getPrivate();
```

Z uporabo razreda KeyFactory lahko ponovno kreiramo ključe.

Ključe lahko generiramo tudi z uporabo razreda KeyAgreement, s katerim lahko dostopamo do različnih mehanizmov, s katerimi se lahko dve ali več strank dogovorijo za skupni simetrični ključ (npr. po Diffie-Hellmanu). Vsaka stranka mora klicati metodo KeyAgreement.doPhase(), pri tem da zadnja postavi zastavico lastPhase na true. Ko so vse faze mimo, zgeneriramo naš simetrični ključ:

Primer:

```
KeyAgreement keyAg = KeyAgreement.getInstance("DH");
keyAg.init(ourKey);
...
keyAg.doPhase(remoteKey, true);
...
SecretKey key = keyAg.generateSecret("DES");
```

Dostikrat se zgodi, da bi radi zgenerirani ključ shranili. To običajno naredimo tako, da ključ shranimo na disk ali kakšno drugo spominsko enoto. Java ponuja razred KeyStore, ki omogoča shranjevanje ključev in certifikatov. Ker so ključi občutljiv podatek, jih shranimo pod določenim geslom. Za nalaganje ključev uporabljamo metodo KeyStore.load(), za shranjevanje pa KeyStore.store().

Primer uporabe certikata:

```
CertificateFactory cf = CertificateFactory.getInstance("X.509");
X509Certificate cert = (X509Certificate)
      cf.generateCertificate(inStream);
```

Vsak certifikat ima dve uporabni metodi:

- getPublicKey() za pridobivanje javnega ključa in
- verify() za preverjanje veljavnosti certifikata.

6.7. TESTI POSAMEZNIH ALGORITMOV

Za Javo obstaja več možnih implementacij JCE, ki krijejo JCE specifikacijo v celoti ali le delno. Sledeča podjetja in organizacije so ponudniki javanske kriptografije:

- RSA Data Security, Inc. (http://www.rsa.com/rsa/products/jsafe)
- eSec Limited (http://www.openjce.org/)
- Forge Research (http://www.forge.com.au)
- DSTC (http://security.dstc.edu.au/projects/java/release3.html)
- IAIK (http://jcewww.iaik.tu-graz.ac.at/jce/jce.htm)
- Entrust(R) Technologies (http://www.entrust.com/toolkit/java/index.htm)
- Cryptix (http://www.nl.cryptix.org/products/jce/index.html)

Med vsemi izstopa kriptografska knjižnica podjetja eSec Limited, ki je brezplačna za uporabo, vsebuje kompletno izvorno kodo knjižnice in ni omejena z izvoznimi omejitvami.

Knjižnica podpira sledeče dele kriptografije:

- Zgoščevalne funkcije (SHA0, SHA1 in MD5),
- Podpise (MD5withRSA),
- MAC (DESMac),
- Šifriranje (Blowfish, DES, DESede, IDEA, RC4, RSA in Twofish) in
- Generacijo ključev (kot pri šifriranju).

Za hitrostno primerjavo posameznih algoritmov sem opravil test, ki meri čas kreiranja ključa in šifriranja podatkov. Pri generaciji ključev sem opazil časovna odstopanja, zato sta vpisana minimalni in maksimalni čas generacije.

NAČIN ŠIFRIRANJA	ČAS GEN. KLJUČA [s]	ČAS ŠIFRIRANJA [s]
DES	5,5 – 6,5	0,28
DESede	5,5 – 6,5	0,47
RSA 512	7,5 – 9,5	3,60
RSA 1024	7,5 – 9,5	5,30
Blowfish	5,5 – 7,5	0,28
Twofish	5,5 – 6,5	0,95
IDEA	6,5	0,28

Tabela 1 – Primerjava algoritmov

Za test sem uporabil PC računalnik z Windows NT 4.0 okoljem, procesorjem Pentium PII (400 MHz) in 256 MB delovnega pomnilnika. Test je šifriral objekt tipa Vector velikosti 100000 celoštevilskih elementov (tipa Integer).

7. UPORABNIŠKA KNJIŽNICA FORTIFY

Ideja za pisanje nove visokonivojske (high level) knjižnice, ki kliče funkcije JCA in JCE, leži v dejstvu, da je kriptografija za razvijalca grafičnih okolij, ki je običajno prešel z Visual Basica ali Delphija na Javo in je vajen dela z RAD (Rapid Application Development) orodji, še vedno preveč kompliciran. Taki razvijalci si želijo knjižnico, ki v nekaj ukazih generira ključe, jih izmenja, zašifrira podatke in jih na koncu odšifrira.

Da bi bila knjižnica čim bolj preprosta (in s tem tudi ne-univerzalna), sem upošteval sledeče točke, ki bazirajo na naš projekt (vseevropski študentski portal www.e-loft.com):

- Vsi strežniki, med katerimi tečejo zaupni podatki, se nahajajo znotraj relativno varovane stavbe, zato lahko le administratorji dostopajo do njih.
- Certifikati niso potrebni, ker so sistemi znani.
- Strežniki hranijo svoje asimetrične ključe.
- Strežniki ob vsaki povezavi generirajo svoj simetrični ključ, ki ga asimetrično šifriranega izmenjajo med seboj. Ker ima portal več aplikacijskih strežnikov (enega na posamezno državo) in le enega za Autonomy sistem (search engine), slednji skrbi za generacijo simetričnega ključa.
- Za simetrično (de)šifriranje se uporabi DES.
- Za asimetrično (de)šifriranje se uporabi RSA.

7.1. IZVORNA KODA

Knjižnica se nahaja v podimeniku si/cocoasoft/fortify:

Za delo so potrebni sledeči paketi:

- java.io.*, za delo z datotekami in tokovi,
- java.security.*, za delo z varnostjo,
- java.security.spec.*, za delo z varnostnimi specifikacijami,
- javax.crypto.*, za delo s kriptografijo, in
- javax.crypto.spec.*, za delo s kriptografskimi specifikacijami.

Ime našega razreda, ki predstavlja našo kriptografsko knjižnico, je Fortify.

Razred ima sledeče globalne spremenljivke:

- m_asymCipher, ki prestavlja objekt za delo z asimetričnimi algoritmi,
- m_symCipher, ki prestavlja objekt za delo s simetričnimi algoritmi,
- m_secretKey, ki predstavlja simetrični ključ,
- m_privateKey, ki predstavlja asimetrični zasebni ključ,
- m_publicKey, ki predstavlja asimetrični javni ključ,
- m_remotePublicKey, ki predstavlja asimetrični javni ključ drugega sistema,
- m_keyPairGen, ki predstavlja generator asimetričnih ključev, in
- m_keyGen, ki predstavlja generator simetričnega ključa.

Konstruktor razreda Fortify je privaten, kar pomeni, da ga v resnici kreiramo s statično metodo Fortify.getInstance().

Metoda initGenerators() inicializira vse objekte za začetek (de)šifriranja:

```
public void initGenerators() throws NoSuchAlgorithmException,
    NoSuchPaddingException
{
  // Inicializira asim. kljuc - RSA 1024 bitov
  m_keyPairGen = KeyPairGenerator.getInstance("RSA");
  m_keyPairGen.initialize(1024);

  // Inicializira sim. kljuc - DES 64 bitov
  m_keyGen = KeyGenerator.getInstance("DES");
  m_keyGen.init(64);

  // Inicializacija sifrirnih objektov
  m_asymCipher = Cipher.getInstance("RSA");
  m_symCipher = Cipher.getInstance("DES");
}
```

Metoda generateAsymKeys() zgenerira javni in zasebni asimetrični ključ:

```
public void generateAsymKeys()
{
  KeyPair keyPair = m_keyPairGen.generateKeyPair();
  m_publicKey = keyPair.getPublic();
  m_privateKey = keyPair.getPrivate();
}
```

Metoda generateSymKey() na podoben način zgenerira simetrični ključ:

```
public void generateSymKey()
{
  m_secretKey = m_keyGen.generateKey();
}
```

Metoda loadSymKey() naloži simetrični ključ z diska:

```
public void loadSymKey(String strPassword)
    throws IOException, InvalidKeySpecException,
    NoSuchAlgorithmException, InvalidKeyException
{
  try
  {
    KeyStore store = KeyStore.getInstance("JKS");
    char cPwd[] = strPassword.toCharArray();
    store.setKeyEntry("secret_key", m_secretKey, cPwd, null);
  }
  catch (Exception ex)
  {
    ex.printStackTrace();
  }
}
```

Metoda saveSymKey() shrani simetrični ključ na disk:

```
public void saveSymKey(String strPassword)
    throws FileNotFoundException, IOException,
    InvalidKeySpecException, NoSuchAlgorithmException,
    InvalidKeyException
{
  try
  {
    KeyStore store = KeyStore.getInstance("JKS");
    char cPwd[] = strPassword.toCharArray();
    m_secretKey = (SecretKey)store.getKey("secret_key", cPwd);
  }
  catch (Exception ex)
  {
    ex.printStackTrace();
  }
}
```

Metoda loadAsymKeys() naloži asimetrična ključa z diska:

```
public void loadAsymKeys(String strPassword)
    throws IOException, InvalidKeySpecException,
    NoSuchAlgorithmException, InvalidKeyException
{
  try
  {
    KeyStore store = KeyStore.getInstance("JKS");
    char cPwd[] = strPassword.toCharArray();
    store.setKeyEntry("private_key", m_privateKey, cPwd, null);
    store.setKeyEntry("public_key", m_publicKey, cPwd, null);
  }
  catch (Exception ex)
  {
    ex.printStackTrace();
  }
}
```

Metoda saveAsymKeys() shrani asimetrična ključa na disk:

```
public void saveAsymKeys(String strPassword)
    throws FileNotFoundException, IOException,
    InvalidKeySpecException, NoSuchAlgorithmException,
    InvalidKeyException
{
  try
  {
    KeyStore store = KeyStore.getInstance("JKS");
    char cPwd[] = strPassword.toCharArray();
    m_privateKey = (PrivateKey)store.getKey("private_key", cPwd);
    m_publicKey = (PublicKey)store.getKey("public_key", cPwd);
  }
  catch (Exception ex)
  {
    ex.printStackTrace();
  }
}
```

Metoda secretKeyToArray() pretvori objekt, ki predstavlja simetrični ključ, v niz 8-bitnih celoštevilskih vrednosti, ki se lahko nato prenesejo preko omrežja:

```
public byte[] secretKeyToArray() throws NoSuchAlgorithmException,
    InvalidKeySpecException, IOException, InvalidKeyException
{
  SecretKeyFactory factory = SecretKeyFactory.getInstance("DES");
  KeySpec spec = factory.getKeySpec(m_secretKey, DESKeySpec.class);
  DESKeySpec keySpec = (DESKeySpec)spec;
  byte nValue[] = keySpec.getKey();
  return encodeAsymetriclly(nValue);
}
```

Metoda arrayToSecretKey() pretvori niz 8-bitnih celih števil v objekt, ki prestavlja simetrični ključ:

```
public void arrayToSecretKey(byte nKey[]) throws InvalidKeyException,
    NoSuchAlgorithmException, InvalidKeySpecException, IOException
{
  DESKeySpec keySpec = new DESKeySpec(decodeAsymetriclly(nKey));
  SecretKeyFactory factory = SecretKeyFactory.getInstance("DES");
  m_secretKey = factory.generateSecret(keySpec);
}
```

Metoda encodeAsimetriclly() asimetrično zašifrira podatke:

```
public byte[] encodeAsymetriclly(byte nValue[]) throws InvalidKeyException,
    IOException
{
  if ((m_asymCipher == null) || (m_remotePublicKey == null))
  {
    throw new
      InvalidKeyException("Ni asim. sifr. objekta ali tujega jav. kljuca.");
  }
```

```
    m_asymCipher.init(Cipher.ENCRYPT_MODE, m_remotePublicKey);
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    CipherOutputStream cout = new CipherOutputStream(out, m_asymCipher);
    cout.write(nValue);
    cout.close();
    nValue = out.toByteArray();
    return nValue;
  }
```

Metoda decodeAsimetriclly() asimetrično odšifrira podatke:

```
  public byte[] decodeAsymetriclly(byte nValue[]) throws InvalidKeyException,
      IOException
  {
    if ((m_asymCipher == null) || (m_privateKey == null))
    {
      throw new
        InvalidKeyException("Ni asim. sifr. objekta ali zas. kljuca.");
    }
    m_asymCipher.init(Cipher.DECRYPT_MODE, m_privateKey);
    ByteArrayInputStream in = new ByteArrayInputStream(nValue);
    CipherInputStream cin = new CipherInputStream(in, m_asymCipher);
    cin.read(nValue);
    cin.close();
    in.close();
    return nValue;
  }
```

Metoda encodeObject() simetrično zašifrira objekt:

```
  public byte[] encodeObject(Object obj) throws InvalidKeyException,
      IOException
  {
    if ((m_symCipher == null) || (m_secretKey == null))
    {
      throw new InvalidKeyException("Ni sim. sifr. objekta ali sim. kljuca.");
    }
    m_symCipher.init(Cipher.ENCRYPT_MODE, m_secretKey);

    ByteArrayOutputStream out = new ByteArrayOutputStream();
    ObjectOutputStream oout = new ObjectOutputStream(out);
    oout.writeObject(obj);
    int nLength = (out.size() / 8 + 1) * 8;
    byte nValue[] = new byte[nLength];
    System.arraycopy(out.toByteArray(), 0, nValue, 0, out.size());
    oout.close();
    out.close();

    out = new ByteArrayOutputStream();
    CipherOutputStream cout = new CipherOutputStream(out, m_symCipher);
    cout.write(nValue);
    cout.close();
    out.close();
    return out.toByteArray();
  }
```

Metoda decodeObject() simetrično odšifrira objekt:

```
  public Object decodeObject(byte nValue[]) throws InvalidKeyException,
      IOException, ClassNotFoundException
  {
    if ((m_symCipher == null) || (m_secretKey == null))
    {
      throw new InvalidKeyException("Ni sim. sifr. objekta ali sim. kljuca.");
    }
    m_symCipher.init(Cipher.DECRYPT_MODE, m_secretKey);

    ByteArrayInputStream in = new ByteArrayInputStream(nValue);
    CipherInputStream cin = new CipherInputStream(in, m_symCipher);
    cin.read(nValue);
    cin.close();
    in.close();

    in = new ByteArrayInputStream(nValue);
    ObjectInputStream oin = new ObjectInputStream(in);
```

```
    Object obj = oin.readObject();
    oin.close();
    in.close();
    return obj;
  }
}
```

7.2. UPORABA

Računalnik, ki sprejme skupni simetrični ključ, kliče sledeče metode:

```
Fortify fortify = Fortify.getInstance();
fortify.initGenerators();
fortify.generateAsymKeys();
fortify.setRemotePublicKey(…); // Prebere tuj javni ključ
byte encodedKey[] = …; // Prebere niz, ki predstavlja sim. Ključ
byte key[] = fortify.decodeAsymetriclly(encodedKey);
fortify.arrayToSecretKey(key); // Nastavi tuj javni ključ
```

Računalnik, ki zgenerira simetrični ključ, pa kliče sledeče metode:

```
Fortify fortify = Fortify.getInstance();
fortify.initGenerators();
fortify.generateAsymKeys();
fortify.generateSymKey();
…(fortify.getPublicKey()); // Pošlje svoj javni ključ
byte key[] = fortify.secretKeyToArray();
byte encodedKey[] = fortify.encodeAsymetriclly(key);
… (encodedKey); // Pošlje simetrični ključ
```

Na koncu z metoda encodeObject() in decodeObject() prenašamo podatke varno po omrežju:

```
byte value[] = fortify.encodeObject("Blabla");
String text = (String)fortify.decodeObject(value);
```

8. ZAKLJUČEK

Java kot trenutno najbolj uporabljen omrežni programski jezik poskuša doseči zaupanje vseh spletnih uporabnikov. To omogoča na več nivojih zaščite:

- deluje kot peskovnik,
- omogoča uporabo kriptografski knjižnic, ki ustrezajo Sunovemu javanskem standardu JCA in
- z razširljivostjo omogoča poznejšim verzijam Jave implementacije dodatne varnosti.

Praksa je tudi pokazala, da več sto milijonov spletnih uporabnikov lažje in hitreje odkrije varnostne hibe tega programskega jezika, kar tudi omogoča hitrejši razvoj in odpravljanje napak.

Omejitev izvoza kriptografskih knjižnic iz ZDA je omogočila ne-monopol na ponudbi javanskih kriptografskih knjižnic, kar se kaže na širokem spektru le-teh. Med vsemi izstopa knjižnica podjetja eSec Ltd.

9. LITERATURA

[1] J. Knudsen, **JAVA Cryptography**, maj 1998, O'Reilly

[2] S. Oaks, **JAVA Security**, maj 1998, O'Reilly

[3] G. McGraw, E. W. Felten, **Securing JAVA – Getting Down to Business with Mobile Code**, 1999, Wiley

[4] **JCA/JCE API Overview**, februar 1999, http://www.aba.net.au/solutions/crypto/jce_api_overview.html

ABA's JCE - Specifications

Version : $Id: jcespecsheet.html,v 1.1 2000/03/27 06:34:18 allison Exp $

Send Comments to ABA

1.0 Introduction

From http://java.sun.com;

> *The Java Cryptography Extension (JCE) 1.2 provides a framework and implementations for encryption, key generation and key agreement, and Message Authentication Code (MAC) algorithms. Support for encryption includes symmetric, asymmetric, block, and stream ciphers. The software also supports secure streams and sealed objects.*

The ABA JCE consists of;

- a clean room implementation of the Java Cryptography Extension (JCE) API as defined by Sun Microsystems;
- a provider of underlying cryptographic algorithms.

Readers looking for further introductory information should examine The ABA JCE - Executive Overview.

Readers looking for more technical information should examine the ABA JCE Specification document.

2.0 Specifications

- clean room implementation of the JCE API
- Java 1.02 and 1.1 compatible with the available ABA compatibility classes.
- Java 1.2 compatible

2.1 Linking in the Provider

The ABA provider can be linked into a Java program in one of two ways:

- by explicitly adding the provider in the source for the program (using Security.addProvider());
- or by implicitly including the proviver by adding the ABA provider class to the java.security file in /usr/java/lib/security/java.security.

The class representing the ABA provider is called:

```
au.net.aba.crypto.provider.ABAProvider
```

2.2 Algorithm Support

The following message digest algorithms are supported by the ABA provider:

Algorithm
SHA-0
SHA, (SHA-1)
MD5

The following block ciphers are supported by the ABA provider:

Algorithm	Modes	Key Lengths	Padding
DES	ECB, CBC	64 bit	PKCS5Padding, PKCS7Padding, NoPadding
DESede	ECB, CBC	192 bit	PKCS5Padding, PKCS7Padding, NoPadding
IDEA™	ECB, CBC	128 bit	PKCS5Padding, PKCS7Padding, NoPadding
Blowfish	ECB, CBC	128 bit to 448 bit	PKCS7Padding, NoPadding
Twofish	ECB, CBC	128, 192 or 256 bit	PKCS7Padding, NoPadding
RSA	ECB, CBC	512 bit up	PKCS1Padding, NoPadding

The following stream ciphers are supported by the ABA provider:

Algorithm	Key Lengths
RC4®	40 - 128 bit

Key generation is currently available for all algorithms. Key factory classes are provided for all algorithms.

3.0 Dependencies

The JCE was designed by Sun to be an "add-on" to JDK 1.2, however the ABA implementation also supports JDK 1.0 and JDK 1.1. Developers wishing to use the ABA JCE and cryptographic provider will need to have a copy of the ABA compatibility classes.

4.0 Restrictions

This software may be subject to the import/export controls, regulation or legislation of your country.

JCA/JCE API Overview

$Date: 2000/03/27 06:34:16 $

$Revision: 1.1 $

1.0 Scope

The purpose of this document is to provide an introduction to the Java APIs that provide security and cryptographic services. These APIs are known as Java Cryptography Architecture (JCA) and Java Cryptography Extension (JCE).

Whilst reading this document it is suggested to have both the JCA and JCE API documentation at hand. The JCA documentation can be found in the Java2 release or online at http://java.sun.com/products/jdk/1.2/docs/api/index.html, and the JCE documentation is currently available at http://java.sun.com/security/JCE1.2/earlyaccess/apidoc/index.html. Finally, your vendor's JCE provider should include reference documentation detailing the specific algorithms they have implemented and any extensions to the API.

This document assumes the reader is familiar with the Java programming language.

2.0 Glossary

API
: Application Programming Interface, an interface used by an application developer to interface to a set of functionality provided by a third party.

Asymmetric Cipher
: An encryption algorithm that uses different keys for encryption and decryption. These ciphers are usually also known as public-key ciphers as one of the keys is generally public and the other is private. RSA and ElGamal are two asymmetric algorithms.

Block Cipher
: A cipher that processes input in a fixed block size greater than 8 bits. A common block size is 64 bits.

Certificate
: A binding of an identity (individual, group, etc.) to a public key which is generally signed by another identity. A certificate chain is a list of certificates that indicates a chain of trust, i.e. the second certificate has signed the first, the third has signed the second and so on.

Decryption
: The process of recovering the plaintext from the ciphertext.

DES
: Data Encryption Standard as defined in FIPS PUB 46-2 which may be found at http://www.itl.nist.gov/div897/pubs/fip46-2.htm.

Digital Signature
: A mechanism that allows a recipient or third party to verify the originator of a document and to ensure that the document has not be altered in transit.

DSA
: Digital Signature Algorithm as defined in FIPS PUB 186 which may be found at http://www.itl.nist.gov/div897/pubs/fip186.htm.

Encryption
: The process of converting the plaintext data into the ciphertext so that the content of the data is no longer obvious. Some algorithms perform this function in such a way that there is no known mechanism, other than decryption with the appropriate key, to recover the plaintext. With other algorithms there are known flaws which reduce the difficulty in recovering the plaintext.

JCA
: Java Cryptography Architecture.

JCE
: Java Cryptography Extension.

MAC
: Message authentication code. A mechanism that allows a recipient of a message to determine if a message has been tampered with. Broadly there are two types of MAC algorithms, one is based on symmetric encryption algorithms and the second is based on Message Digest algorithms. This second class of MAC algorithms are known as HMAC algorithms. A DES based MAC is defined in FIPS PUB 113, see http://www.itl.nist.gov/div897/pubs/fip113.htm. For information on HMAC algorithms see RFC-2104 at http://www.ietf.org/rfc/rfc2104.txt.

Message Digest
: A condensed representation of a data stream. A message digest will convert an arbitrary data stream into a fixed size output. This output will always be the same for the same input stream however the input cannot be reconstructed from the digest.

Padding
: A mechanism for extending the input data so that it is of the required size for a block cipher. The PKCS documents contain details on the most common padding mechanisms of PKCS#1 and PKCS#5.

PEM
: Privacy Enchanced Mail, includes standards for certificates, see RFC1422 http://www.ietf.org/rfc/rfc1422.txt.

PKCS
: Public Key Cryptography Standards. A set of standards (currently PKCS#1 to PKCS#15) developed by RSA Laboratories, see http://www.rsa.com/rsalabs/pubs/PKCS/.

RFC
: Request for Comments, proposed specifications for various protocols and algorithms archived by the Internet Engineering Task Force (IETF), see http://www.ieft.org.

RSA
: A public-key encryption algorithm, see http://www.rsa.com.

Symmetric Cipher
: An encryption algorithm that uses the same key for encryption and decryption. DES, RC4 and IDEA are all symmetric algorithms.

X.509 Certificate
: Section 3.3.3 of X.509v3 defines a certificate as: "user certificate; public key certificate; certificate: The public keys of a user, together with some other information, rendered unforgeable by encipherment with the private key of the certification authority which issued it.".

3.0 Introduction

The Java platform provides two APIs for dealing with security and cryptographic services. The first is known as the Java Cryptography Architecture (JCA) and provides a framework for basic security functions such as certificates, digital signatures and message digests. The JCA and a default provider (the Sun provider) are included with Java2. The Java Cryptography Extension (JCE) extends the JCA to provide encryption, key exchange, key generation and message authentication services. The JCE is released as a standard extension to the Java2 platform.

The JCA/JCE do not directly provide specific implementations of the various algorithms. Rather they are an interface between the application and a number of specific implementations of the algorithms. Generally a vendor will group the algorithms they have developed into a `Provider` which may then be installed into the Java runtime environment. Once installed, an application may specifically request a particular provider's implementation or, if not the framework will choose an implementation from the highest priority `Provider` that implements the requested algorithm.

his architecture is achieved by providing "factory classes" that are used to create object instances that implement a pecific algorithm. Each of the factory classes has a `private` constructor, instances can only be constructed by calling a `ublic static` method which returns an instance of the desired type. When an algorithm is requested, the factory class vill iterate through the installed providers and return the first implementation it finds (unless the application has equested a specific `Provider`).

he `java.security.Security` class is responsible for maintaining a list of available providers. When this class is nitialised it will read the `java.security` properties file (which is located at `JAVA_HOME/lib/security/java.security`). This properties file has a list of the installed providers, ordered by preference. For example the Sun and Acme providers could be listed as;

```
ecurity.provider.1=sun.security.provider.Sun
ecurity.provider.2=org.acme.crypto.provider.Acme
```

A `Provider` may also be installed dynamically by an application at runtime. This is achieved by using the `ecurity.addProvider()` method, passing the `Provider` instance of the vendor to be installed. For example:

```
Security.addProvider(new org.acme.crypto.provider.Acme());
```

he following packages are provided as part of the JCA:

`ava.security`
: Provides the interfaces for the security framework. Generally, these classes do not have `public` constructors, rather they consist of factory methods which will create `Provider` based implementations of the requested algorithms. Here you will find the `KeyFactory`, `KeyPairGenerator`, `KeyStore`, `MessageDigest` and `Signature` classes.

`ava.security.cert`
: The interfaces for parsing and managing certificates, in particular X.509 v3 certificates.

`ava.security.interfaces`
: The provider-independent interfaces for dealing with RSA and DSA public/private keys.

`ava.security.spec`
: The provider-independent interfaces for key and algorithm specifications.

The following packages are provided as part of the JCE:

`avax.crypto`
: The core services provided by the JCE. Here you will find the `Cipher`, `KeyGenerator` and `Mac` classes.

`avax.crypto.interfaces`
: Provider-independent interfaces for Diffie-Hellman keys.

`avax.crypto.spec`
: Provider independent specifications for DES, DESede, Diffie-Hellman and various other keys and algorithm parameters.

4.0 Encryption/Decryption

The JCE supports encryption and decryption using symmetric algorithms (such as DES and RC4) and asymmetric algorithms (such as RSA and ElGamal). The algorithms may be stream or block ciphers, with each algorithm supporting different modes, padding or even algorithm-specific parameters.

4.1 The Cipher Class

The basic interface used to encipher or decipher data is the `javax.crypto.Cipher` class. The class provides the necessary mechanism for encrypting and decrypting data using arbitrary algorithms from any of the installed providers.

To create a `Cipher` instance, use one of the `Cipher.getInstance()` methods. This method will accept a transformation string and an optional provider name. The transformation string is used to specify the encryption algorithm as well as the cipher mode and padding. The transformation is specified in the form;

- "algorithm"
- "algorithm/mode/padding"

In the first instance, we are requesting the algorithm with its default mode and padding mechanism. The second instance fully qualifies all options. For a list of support algorithms consult the provider's documentation. Some common transformations are;

- "RC4"
- "DES/CBC/PKCS5Padding"
- "RSA/ECB/PKCS1Padding"

The following code will create a cipher for performing RC4 encryption or decryption, a cipher for doing RSA in ECB mode with PKCS#1 padding provided by the ABA provider and a cipher for performing DESede encryption/decryption in CBC mode with PKCS#5 padding:

```
Cipher rc4Cipher = Cipher.getInstance("RC4");
Cipher rsaCipher = Cipher.getInstance("RSA/ECB/PKCS1Padding",
   "ABA");
Cipher desEdeCipher = Cipher.getInstance("DESede/CBC/PKCS5Padding");
```

Once we have a `Cipher` instance, we will need to initialise the `Cipher` for encryption or decryption. We will also need to provide a `Key`, see section 8.0 for a discussion of key management.

```
Key desKey, rsaKey;

desCipher.init(Cipher.ENCRYPT_MODE, desKey);
rsaCipher.init(Cipher.DECRYPT_MODE, rsaKey);
```

As you can see, the first value passed to the `Cipher.init()` method indicates whether we are initialising for encryption or decryption. The second argument provides the key to use during encryption or decryption.

There are a number of other initialisation methods for providing algorithm specific parameters (such as Initialisation Vectors, the number of rounds to use etc.). See section 4.4 for a discussion on algorithm parameters.

Now that our `Cipher` is initialised, we can start processing data. To do so we use the `Cipher.update()` and `Cipher.doFinal()` methods. The `Cipher.update()` methods may be used to incrementally process data. Once all the data is processed, one of the `Cipher.doFinal()` methods must be called.

In the simplest usage, a single `Cipher.doFinal()` call may be passed all the data:

```
byte[] plainText = "hello world".getBytes();
byte[] cipherText = desCipher.doFinal(plainText);
```

Once the `Cipher.doFinal()` method has been called, the `Cipher` instance will be reset to the state it was in after the last call to the `Cipher.init()` method. That means the `Cipher` may be reused to encipher or decipher more data using the same `Key` and parameters that were specified in the initialisation.

4.2 Cipher Input and Output Streams

Rather than deal with the complications of buffering enciphered or deciphered data produced by the `Cipher.update()` methods, it may be desirable to use a Java Input/Output Stream type interface. Fortunately, the JCE provides us with such a mechanism.

The `javax.crypto.CipherInputStream` and `javax.crypto.CipherOutputStream` are based on the Java IO filter streams. This allows them to process data and pass on that data to an underlying stream.

To create a cipher stream, firstly create and initialise a `javax.crypto.Cipher` instance and the underlying stream and then instantiate the required stream type with these two arguments.

For example, the following code fragment will create a `CipherOutputStream` that will encipher its data (using DES) and pass the result to a `ByteArrayOutputStream`. We can access the ciphertext by calling `ByteArrayOutputStream.toByteArray()`.

```
Key desKey;
Cipher cipher = Cipher.getInstance("DES");
cipher.init(Cipher.ENCRYPT_MODE, desKey);

ByteArrayOutputStream bout = new ByteArrayOutputStream();
CipherOutputStream cout = new CipherOutputStream(bout, cipher);
cout.write("hello world".getBytes());
cout.close();

byte[] cipherText = bout.toByteArray();
```

Once we can encipher and decipher data using a simple stream, interface, we can create much more complicated scenarios. For example the `OutputStream` could just as easily be a `SocketOutputStream` or we could construct an `ObjectOutputStream` on top of our cipher stream and encipher Java objects directly.

4.3 SealedObject

The `javax.crypto.SealedObject` class provides the mechanism to encipher a `Serializable` object. This class allows the application to encipher a Java object and then recover the object all through a simple interface. The `SealedObject` is also `Serializable` to simplify the transport and storage of the enciphered objects.

A `SealedObject` can be constructed through either serialisation or by its constructor. The constructor is used to create a new enciphered object. The constructor's arguments are the object to encipher and the `Cipher` to use. The provided `Cipher` instance must be initialised for encryption before the `SealedObject` is created. This means calling a `Cipher.init()` method with `Cipher.ENCRYPT_MODE` as the mode, the required encryption `Key` and any algorithm parameters.

The following fragment will create a new `SealedObject` containing the enciphered `String` "hello world":

```
Key desKey = ...
Cipher cipher = Cipher.getInstance("DES");
cipher.init(Cipher.ENCRYPT_MODE, deskey);

SealedObject so = new SealedObject("hello world", cipher);
```

To recover the original object, the `SealedObject.getObject()` methods may be used. These methods take either a `Cipher` or `Key` object. When providing the `Cipher` parameter, the instance must be initialised in the `Cipher.DECRYPT_MODE` mode, with the appropriate decryption key and the same algorithm parameters as the original

Cipher. When providing a Key parameter, the encryption algorithm and algorithm parameters are extracted from the SealedObject.

The following fragment will extract a SealedObject object from an ObjectInputStream and then recover the protected object:

```
ObjectInputStream oin ...
Key desKey = ...

SealedObject so = (SealedObject)oin.readObject();
String plainText = (String)so.getObject(deskey);
```

One important security aspect to note with this class is that it does not use a digital signature to ensure the object is not tampered with in its serialised form. It is therefore possible that the object could be altered in storage or transport without detection. Fortunately, the JCA provides the java.security.SignedObject mechanism which can be used in conjunction with the SealedObject class to avoid this problem. (See section 7.2 for a discussion on the SignedObject class).

1.4 Algorithm Parameters

Some cipher algorithms support parameterisation, for example the DES cipher in CBC mode can have an initialisation vector as an algorithm parameter and other ciphers may have a selectable block size or round count. The JCE provides support for algorithm-independent initialisation via the java.security.spec.AlgorithmParameterSpec and java.security.AlgorithmParameters classes.

The java.security.spec.AlgorithmParameterSpec derived classes can be constructed programatically by an application. The following classes are provided by the JCA/JCE:

java.security.spec	
DSAParameterSpec	Used to specify the parameters used with the DSA algorithm. The parameters consist of the base g, prime p and sub-prime q.
javax.crypto.spec	
DHGenParameterSpec	The set of parameters used for generating Diffie-Hellman parameters for use in Diffie-Hellman key agreement.
DHParameterSpec	The set of parameters used with Diffie-Hellman as specified in PKCS#3.
IvParameterSpec	An initialisation vector for use with a feedback cipher. That is an array of bytes of length equal to the block size of the cipher.
RC2ParameterSpec	Parameters for the RC2 algorithm. The parameters are the effective key size and an optional 8-byte initialisation vector (only in feedback mode).
RC5ParameterSpec	Parameters for the RC5 algorithm. The parameters are a version number, number of rounds, a word size and an optional initialisation vector (only in feedback mode).

Your provider may also include further classes for passing parameters to the algorithms it implements.

The JCA also has mechanisms for dealing with the provider-dependent AlgorithmParameters. This class is used as an opaque representation of the parameters for a given algorithm and allows an application to store persistently the parameters used by a Cipher.

There are three situations where an application may encounter an AlgorithmParameters instance:

1. `Cipher.getParameters()`
   After a `Cipher` has been initialised, it may have generated a set of parameters (based on supplied and/or default values). The value returned by the `getParameters()` method allows the `Cipher` to be re-initialised to exactly the same state.
2. `AlgorithmParameters.getInstance()`
   Rather than generating the parameters via the `Cipher` class, it is possible to generate them either based on an encoded format or an `AlgorithmParameterSpec` instance. To do so create an uninitialised instance using the `getInstance` method and then initialise it by calling the appropriate `init()` method.
3. `AlgorithmParameterGenerator.getParameters()`
   Finally, a set of parameters can be generated using the `AlgorithmParameterGenerator`. Firstly, a generator is created for the required algorithm using the `getInstance()` method. Then the generator is initialised by calling one of the `init()` methods, finally to create the instance use the `getParameters` method.

   This class provides the concept of algorithm-independent parameter generation, in that the initialisation can be based on a "size" and a source of randomness. In this case the "size" value is interpreted differently for each algorithm.

5.0 Message Digests

The JCA provides support for the generation of message digests via the `java.security.MessageDigest` class. This class uses the standard factory class design, so to create a `MessageDigest` instance use the `getInstance()` method with the desired algorithm name and optional provider as parameters.

Once created use the various `update()` methods to process the message data and then finally call the `digest()` method to calculate the final digest. At this point the instance may be re-used to calculate a digest for a new message.

```
MessageDigest digest = MessageDigest.getInstance("SHA");

byte[] msg = "The message".getBytes();
digest.update(msg);

byte[] result = digest.digest();
```

6.0 Message Authentication Code (MAC)

The `javax.crypto.Mac` API is used to access a "Message Authentication Code" (MAC) algorithm. These algorithms are used to check the integrity of messages upon receipt. There are two classes of MAC algorithms in general, those that are based on message digests (known as HMAC algorithms) and those on encryption algorithms. In both cases a shared secret is required.

`Mac` is used in the same fashion as a `Cipher`. First, use the factory method `Mac.getInstance()` to get the provider implementation of the required algorithm, then initialise the algorithm with the appropriate key via the `Mac.init()` method. Then, use the `Mac.update()` method to process the message and finally use the `Mac.doFinal()` method to calculate the MAC for the message.

To verify the message, follow the same procedure and compare the supplied MAC with the calculated MAC.

Note that it is not necessary to use the `Mac.init()` method to check multiple messages if the shared secret has not changed. The `Mac` will be reset after the call to `Mac.doFinal()` (or a call to `Mac.reset()`).

```
/*
 * on the sender
```

```
 */
Mac senderMac = Mac.getInstance("HMAC-SHA1");
senderMac.init(shaMacKey);
byte[] mac = senderMac.doFinal(data);

/*
 * now transmit message and mac to receiver
 */
Mac recMac = Mac.getInstance("HMAC-SHA1");
recMac.init(shaMacKey);
byte[] calcMac = recMac.doFinal(data);

for (int i = 0; i < calcMac.length; i ++)
{
   if (calcMac[i] != mac[i])
   {
      /* bogus MAC! */
      return false;
   }
}

/* all okay */
return true;
```

7.0 Authentication

7.1 Digital Signatures

The `java.security.Signature` class provides the functionality of a digital signature algorithm. Digital signatures are the digital equivalent of the traditional pen and paper signature. They can be used to authenticate the originator of a document, as well as to prove that a person signed the document. Generally, digital signatures are based on public-key encryption which means that, unlike a MAC, anyone that has access to the public key (and the document) can check the validity of the document.

The `Signature` interface supports generation and verification of signatures. Once a signature instance has been created using the `Signature.getInstance()` method, it needs to be initialised with the `Signature.initSign()` method for creation of a signature, or `Signature.initVerify()` method for verification of a signature.

Once initialised, the document to be processed should be passed to the signature via the `Signature.update()` methods. Once the entire document has been processed, the `Signature.sign()` method may be called to generate the signature, or the `Signature.verify()` method to verify a supplied signature against a previously generated signature.

After a signature has been generated or verified, the `Signature` instance is reset to the state it was in after it was last initialised, allowing another signature to be generated or verified using the same key.

One such signature algorithm is "MD5 with RSA" and is defined in PKCS#1. This algorithm specifies that the document to be signed is passed through the MD5 digest algorithm and then an ASN.1 block containing the digest, along with a digest algorithm identifier, is enciphered using RSA.

To create such a signature;

```
/*
 * Assume this private key is initialised
 */
PrivateKey rsaPrivKey;

/*
```

```
    * Create the Signature instance and initialise
    * it for signing with our private key
    */
   Signature rsaSig = Signature.getInstance("MD5withRSA");
   rsaSig.initSign(rsaPrivKey);

   /*
    * Pass in the document data via the update() methods
    */
   byte[] document = "The document".getBytes();
   rsaSig.update(document);

   /*
    * Generate the signature
    */
   byte[] signature = rsaSig.sign();
```

To verify the generated signature:

```
   /*
    * Assume this public key is initialised
    */
   PublicKey rsaPubKey;

   /*
    * Create the Signature instance and initialise
    * it for signature verification with the public key
    */
   Signature rsaSig = Signature.getInstance("MD5withRSA");
   rsaSig.initVerify(rsaPubKey);

   /*
    * Pass in the document data via the update() methods
    */
   byte[] document = "The document".getBytes();
   rsaSig.update(document);

   /*
    * Check the generated signature against the supplied
    * signature
    */
   if (rsaSig.verify(signature))
   {
      // signature okay
   }
   else
   {
      // signature fails
   }
```

.2 Object Signing

The `java.security.SignedObject` provides a mechanism for ensuring that a Java object can be authenticated and annot be tampered with without detection. The mechanism used is similar to the `SealedObject` in that the object to be rotected is serialised and then a signature is attached. The `SealedObject` is `Serializable` so it may be stored or ansmitted via the object streams.

o create a `SignedObject`, firstly create an instance of the signature algorithm to use via the `ignature.getInstance()` method, then create the new `SignedObject` instance by providing the object to be signed, ıe signing key and the `Signature` instance. Note that there is no need to initialise the `Signature` instance; the `ignedObject` constructor will perform that function.

```
Signature signingEngine = Signature.getInstance(
   "MD5withRSA");
SignedObject so = new SignedObject("hello world",
   privateKey, signingEngine);
```

To verify a `SignedObject`, simply create the `Signature` instance for the required algorithm and then use the `SignedObject.verify()` method with the appropriate `PublicKey`. Again, there is no need to initialise the `Signature` instance.

```
Signature verifyEngine = Signature.getInstance(
   "MD5withRSA");
if (so.verify(publicKey, verifyEngine))
{
   // object okay, extract it
   Object obj = so.getObject();
}
else
{
   // object not authenticated
}
```

Note that this class only provides a mechanism for authentication and verification, it does not provide confidentiality (i.e. encryption). The `SealedObject` may be used for this purpose (see section 4.3). The following example combines these two classes to provide a confidential, authenticated, tamper-proof object:

```
/*
 * sealedObj will contain the signed, enciphered data
 */
SignedObject signedObj = new SignedObject(
   "hello world", privateKey, signingEngine);
SealedObject sealedObj = new SealedObject(
   signedObj, cipher);

/*
 * to verify and recover the original object
 */
SignedObject newObj = sealedObject.getObject(cipher);
if (newObj.verify(publicKey, verificationEngine))
{
   // object verified tampered
   String str = (String)newObj.getObject();
}
else
{
   // object tampered with!
}
```

3.0 Key Management

The JCA/JCE framework manages keys in two forms, a provider-dependent format and a provider-independent format.

The provider-dependent keys will implement either the `java.security.Key` interface (or one of its sub classes) for public-key algorithms or the `javax.crypto.SecretKey` interface for secret-key algorithms. Provider keys can be generated randomly, via a key agreement algorithm or from their associated provider-independent format.

The provider-independent formats will implement the `java.security.spec.KeySpec` interface. Subclasses of this type exist for both specific key types and for different encoding types. For example, the `java.security.spec.RSAPublicKeySpec` can be used to construct an RSA public key from its modulus and exponent

and a `java.security.spec.PKCS8EncodedKeySpec` can be used to construct a private key encoded using PKCS#8.

Each `Provider` will supply a number of mechanisms that will create the provider-dependent keys or convert the provider-independent keys into provider based keys.

8.1 Generating Random Keys

The simplest mechanism to create keys for a given provider is to use their random key generators. Random keys are most often generated for use as "session-keys" which will be used for a given dialogue or session and are then no longer required. In the case of public-key algorithms, however, they may be generated once and then stored for later use. The JCE framework provides key generation via the following classes:

`javax.crypto.KeyGenerator`
: Generation of symmetric keys (ie DES, IDEA, RC4)

`java.security.KeyPairGenerator`
: Generation of public/private key pairs (ie RSA)

For instance, to create a random 128-bit key for RC4 and initialise a `Cipher` for encryption with this key;

```
/*
 * Create the key generator for the desired algorithm,
 * and then initialise it for the required key size.
 */
KeyGenerator rc4KeyGen = KeyGenerator.getInstance("RC4");
rc4KenGen.init(128);

/*
 * Generate the key and then initialise the Cipher
 */
SecretKey rc4Key = rc4KeyGen.generateKey();
Cipher rc4Cipher = Cipher.getInstance("RC4");
rc4Cipher.init(Cipher.ENCRYPT_MODE, rc4Key);
```

Here, the `SecretKey` returned by the `KeyGenerator.generateKey()` method is a provider-dependent key. The returned key can then be used with that provider's algorithms.

Some algorithms have keys that are considered *weak*, for example with a weak DES key the ciphertext may be the same as the plaintext! Generally the `KeyGenerator` will not generate those keys, however it is best to check the provider documentation for details on the specific algorithm.

The code to generate a public/private key pair is quite similar;

```
KeyPairGenerator rsaKeyGen = KeyPairGenerator.getInstance("RSA");
rsaKeyGen.initialize(1024);

KeyPair rsaKeyPair = rsaKeyGen.generateKeyPair();
Cipher rsaCipher = Cipher.getInstance("RSA");
rsaCipher.init(Cipher.ENCRYPT_MODE, rsaKeyPair.getPrivate());
```

.2 Key Conversion

Two interfaces are provided to convert between a provider-dependent `Key` and the provider-independent `KeySpec`; `java.security.KeyFactory` and `javax.crypto.SecretKeyFactory`. The `KeyFactory` class is used for public-key algorithms and the `SecretKeyFactory` class for secret-key algorithms.

An application may choose to store its keys in some way and then re-create the key using a `KeySpec`. For example, the application may contain an embedded RSA public key as two integers; the `RSAPublicKeySpec` along with a `KeyFactory` that can process `RSAPublicKeySpec` instances could then be used to create the provider-dependent key.

Each provider will generally supply a number of `KeyFactory`/`SecretKeyFactory` classes that will accept the various `KeySpec` classes and produce `Key` instances that may be used with the provider algorithms. These factories are not likely to support all `KeySpec` types, so the provider documentation should provide the details as to what conversions will be accepted.

There are a number of `KeySpec` classes provided by the JCA/JCE;

java.security.spec	
`PKCS8EncodedKeySpec`	A DER encoding of a private key according to the format specified in the PKCS#8 standard.
`X509EncodedKeySpec`	A DER encoding of a public or private key, according to the format specified in the X.509 standard.
`RSAPublicKeySpec`	An RSA public key
`RSAPrivateKeySpec`	An RSA private key
`RSAPrivateCrtKeySpec`	An RSA private key, with the Chinese Remainder Theorem (CRT) values
`DSAPublicKeySpec`	A DSA public key
`DSAPrivateKeySpec`	A DSA private key

javax.crypto.spec	
`DESKeySpec`	A DES secret key
`DESedeKeySpec`	A DESede secret key
`PBEKeySpec`	A user-chosen password that can be used with password base encryption (PBE)
`SecretKeySpec`	A key that can be represented as a byte array and have no associated parameters. The encoding type is know as RAW.

To convert a `KeySpec` instance into a provider based `Key`, firstly create a `KeyFactory` or `SecretKeyFactory` of the appropriate type using the `getInstance()` method. Once the instance has been created, use the `KeyFactory.generatePrivate()`, `KeyFactory.generatePublic()` or `SecretKeyFactory.generateSecret()` method.

In the following example we will create a `Key` from a `KeySpec` and then recover the `KeySpec` from the `Key`.

```
/*
 * This is the raw key
 */
byte[] keyBytes = { (byte)0x1, (byte)0x02, (byte)0x03,
   (byte)0x04, (byte)0x05, (byte)0x06, (byte)0x07, (byte)0x08 };

/*
 * Create the provider independent KeySpec
 */
DESKeySpec desKeySpec = new DESKeySpec(keyBytes);

/*
 * Create the KeyFactory to do the Key<->KeySpec translation
 */
SecretKeyFactory keyFact = KeyFactory.getInstance("DES");
```

```
/*
 * Create the provider based SecretKey
 */
SecretKey desKey = keyFact.generateSecret(desKeySpec);

/*
 * Convert the provider Key into a generic KeySpec
 */
DESKeySpec desKeySpec2 = keyFact.getKeySpec(desKey,
   DESKeySpec.class);
```

3.3 Key Agreement Protocols

Keys may also be generated using the `javax.crypto.KeyAgreement` API. This interface provides the functionality of a key agreement (or key exchange) protocol. For example, a Diffie-Hellman `KeyAgreement` instance would allow two or more parties to generate a shared Diffie-Hellman Key.

To generate the key, it is necessary to call `KeyAgreement.doPhase()` for each party in the exchange with a `Key` object that represents the current state of the key agreement. The last call to `KeyAgreement.doPhase()` should have the `lastPhase` set to `true`.

Once all the key agreement phases have been processed, the shared `SecretKey` may be generated by calling the `KeyAgreement.generateSecret()` method.

The `KeyAgreement` API does not define how each of the parties communicates the necessary information for each exchange in the protocol. The required information is passed to the `KeyAgreement.doPhase()` method as a `Key`. This key will generally be generated using either a `KeyGenerator` or a `KeyFactory`. The provider documentation will detail the specific steps required for a given protocol.

```
/*
 * Create the KeyAgreement instance for the required
 * protocol and initialise it with our key.  In the
 * case of Diffie-Hellman this would be our private
 * key.
 */
KeyAgreement keyAg = KeyAgreement.getInstance("DH");
keyAg.init(ourKey);

/*
 * Exchange information as per the key exchange
 * protocol.  For DH we would exchange public keys.
 * Note since there is only two parties in this
 * case the return value is not relevant.
 */
keyAg.doPhase(remotePubKey, true);

/*
 * Create the shared secret-key
 */
SecretKey key = keyAg.generateSecret("DES");
```

3.4 Key Storage

Once a `Key` has been generated you may wish to store it for future use. Generally, you'll be saving public/private keys so that you can reuse them at a later date in a key exchange.

The `java.security.KeyStore` API provides one mechanism for management of a number of keys and certificates. There are two types of entries in a `KeyStore`; `Key` entries and `Certificate` entries. Key entries are sensitive

information whereas certificates are not.

As `Key` entries are sensitive, they are therefore are protected by the `KeyStore`. The API allows for a password, or pass phrase, to be attached to each key entry. What the actual implementation does with the password is not defined, although it may be used to encipher the entry. A key entry may either be a `SecretKey`, or a `PrivateKey`. In the case of a `PrivateKey`, the entry is saved along with a `Certificate` chain which is the chain of trust. The chain of trust starts with the `Certificate` containing the corresponding `PublicKey` and ends with a self-signed certificate.

A certificate entry represents a "trusted certificate entry", that is a `Certificate` whose identity we trust. This type of entry can be used to authenticate other parties.

To create a `KeyStore` instance, use the `KeyStore.getInstance()` method. This will return an empty key store which may then be populated by calling the `KeyStore.load()` method. This method accepts an `InputStream` instance and an optional password. Each individual `KeyStore` will treat these parameters differently, so check the provider documentation for details.

The Sun provider supplies a `KeyStore` known as "JKS". This `KeyStore` is used by the `keytool` and `jarsigner` applications.

```
/*
 * Create an instance of the Java Key Store (defined by Sun)
 */
KeyStore keyStore = KeyStore.getInstance("JKS");
```

To add a new entry into the `KeyStore`, use either `setCertificateEntry()` or one of the `setKeyEntry()` methods. This will add the new entry with the associated alias.

```
char[] myPass;
SecretKey secretKey;

/*
 * Store a SecretKey in the KeyStore, with "mypass"
 * as the password.
 */
keyStore.setKeyEntry("mysecretkey", secretKey, myPass, null);

/*
 * assume that privateKey contains my PrivateKey
 * and myCert contains a Certificate with the
 * corresponding PublicKey
 */
PrivateKey privateKey;
Certificate myCert;

keyStore.setKeyEntry("myprivatekey", privateKey, myPass, myCert);
```

To extract an entry, use the `getKey()` method to extract a `Key` or `getCertificate()` for a `Certificate`.

```
/*
 * recover the SecretKey
 */
SecretKey key = (SecretKey)keyStore.getKey("mysecretkey", myPass);

/*
 * recover the PrivateKey
 */
PrivateKey privKey = (PrivateKey)keyStore.getKey("myprivatekey",
   myPass);
```

```
/*
 * recover the Certificate (containing the PublicKey) corresponding
 * to our PrivateKey
 */
Certificate cert = keyStore.getCertificate("myprivatekey");
```

If the `KeyStore` supports persistence via the `store()` and `load()` methods, the provider documentation will explain what types of `Key` types may be stored.

3.5 Certificates

The JCA framework provides support for generic certificates, as well as X.509v3 certificates. Certificates may be stored using the `KeyStore` API, or they may be generated from their encoded format (either the PEM or PKCS#7 encoding).

To create a `java.security.cert.Certificate` instance from its encoded format, firstly create a `java.security.cert.CertificateFactory` instance of the required type (eg X.509). Then use the `generateCertificate()` or `generateCertificates()` methods to convert your `InputStream` into `Certificate` instances.

```
CertificateFactory cf = CertificateFactory.getInstance("X.509");
X509Certificate cert = (X509Certificate)cf.generateCertificate(
   inputStream);
```

Two useful methods of the `Certificate` class are `getPublicKey()` and `verify()`. The first of these allows access to the `PublicKey` of the certificate's owner and the second allows an application to verify that the certificate was signed using the private key that corresponds to the provided public key.

The `java.security.cert.X509Certificate` class, which extends the `Certificate` class, provides methods to access the other attributes of a X.509 certificate such as the Issuer's distinguished name or its validity period.

The `keytool` application provided with JDK1.2 can be used to generate certificates and store them in a `KeyStore`. Check the JDK documentation for information on how to use this application.

4.0 Error handling and Exceptions

The JCA/JCE framework includes a number of specialised exception classes:

java.security	
DigestException	Thrown if an error occurs during the final computation of the digest. Generally this indicates that the output buffer is of insufficient size.
InvalidAlgorithmParameterException	Thrown by classes that use AlgorithmParameters or AlgorithmParameterSpec instances where the supplied instance is not compatible with the algorithm or the supplied parameter was null and the algorithm requires a non-null parameter.
InvalidKeyException	Thrown by the various classes that use Key objects, such as Signature, Mac and Cipher when the provided Key is not compatible with the given instance.
InvalidParameterException	Only used in the deprecated interfaces in the Signature class and the deprecated class Signer.
KeyStoreException	Thrown by the KeyStore class when the object has not been initialised properly.
NoSuchAlgorithmException	Thrown by the getInstance() methods when the requested algorithm is not available.
NoSuchProviderException	Thrown by the getInstance() methods when the requested provider is not available.
SignatureException	Thrown by the Signature class during signature generation or validation if the object has not been initialised correctly or an error occurs in the underlying ciphers.
javax.crypto	
BadPaddingException	Thrown by the Cipher class (or classes which use a Cipher class to process data) if this cipher is in decryption mode, (un)padding has been requested, and the deciphered data is not bounded by the appropriate padding bytes.
IllegalBlockSizeException	Thrown by the Cipher class (or classes which use a Cipher class to process data) if this cipher is a block cipher, no padding has been requested (only in encryption mode), and the total input length of the data processed by this cipher is not a multiple of block size
NoSuchPaddingException	Thrown by the Cipher class by the getInstance() method when a transformation is requested that contains a padding scheme that is not available.
ShortBufferException	Thrown by the Cipher class when an output buffer is supplied that is too small to hold the result.