Polsko-Japońska Wyższa Szkoła Technik Komputerowych

Poufność, prywatność, spójność oraz niezaprzeczalność

Kazimierz Subieta, Edgar Głowacki
edgar.glowacki@pjwstk.edu.pl

Definicje

Jak pamiętamy bezpieczne oprogramowanie powinno zapewnić:

- **uwierzytelnienie (*authentication*)**, czyli weryfikację tożsamości podmiotu żądającego dostępu do usługi;
- **autoryzacji** (***authorization***) – weryfikacja upoważnienia żądającego dostępu do usługi – autoryzacja pełni rolę służebną wobec poufności;
- **poufność (*confidentiality*)** – dostęp do informacji jest ograniczony wyłącznie do grona podmiotów do tego upoważnionych;
- **spójność/integralność** (***integrity***) – zagwarantowanie, że informacja nie zostanie zmodyfikowana w sposób nieuprawniony, a wszelkie modyfikacje zostaną wykryte;
- **dostępność** (***availability***) – do informacji można uzyskać dostęp w każdych okolicznościach, które są dopuszczone przez politykę bezpieczeństwa informacji;
- **niezaprzeczalność (*non-repudiation*)** – strona podejmująca jakieś działania przy użyciu oprogramowania nie może wyprzeć się, że wykonywała określone operacje z wykorzystaniem oprogramowania.

Poza wymienionymi cechami od bezpiecznego oprogramowania wymaga się czasami również zapewnienia **prywatności** (***privacy***), czyli możliwość sprawowania przez podmiot kontroli nad informacją dotyczącą jego samego.

Podstawy kryptografii (1)

Poufność, spójność, prywatność i niezaprzeczalność są w oprogramowaniu realizowana za pomocą mechanizmów kryptograficznych.

Podstawowym pojęciem występującym w kryptografii jest tzw. prymityw kryptograficzny.

Wyróżniamy trzy podstawowe rodzaje prymitywów kryptograficznych:
- **algorytmy szyfrowania,**
- **jednokierunkowe funkcje skrótu,**
- **schematy podpisów cyfrowych**.

Algorytmy szyfrowania dzielimy na:
- **kryptosystemy symetryczne (*ciphers*)** korzystające z tzw. **klucza tajnego**, oraz
- **kryptosystemy asymetryczne (*encryption algorithms*)** wykorzystujące **parę kluczy: prywatny i publiczny**.

3

Podstawy kryptografii (2)

O **algorytmie symetrycznym** możemy w ogólności mówić, jeśli znając jeden z pary kluczy (zasadniczo chodzi tutaj o klucz szyfrujący) można w sposób „łatwy” (*in a computionally feasible manner*) ustalić odpowiadający mu klucz wykorzystywany w trakcie operacji odwrotnej.

Powyższy warunek implikuje konieczność utrzymywania obydwu kluczy w tajemnicy – stąd nazwa **klucz tajny (*secret key*)**.

W wykorzystywanych praktycznie algorytmach klucz szyfrujący jest tożsamy z kluczem deszyfrującym.

Istnieją dwa rodzaje algorytmów symetrycznych:
- **szyfry blokowe (*block ciphers*)**,
- **szyfry strumieniowe (*stream ciphers*)**.

Zasada działania **szyfrów blokowych** polega na podziale wiadomości na bloki o odpowiedniej długości, po czym wykonaniu operacji kryptograficznych na każdym takiego bloku z osobna.
Jeśli ostatni fragment wiadomości jest zbyt krótki, by wypełnić cały blok jest odpowiednio uzupełniany (*padded*).

4

Podstawy kryptografii (3)

Do podstawowych (atomowych) operacji wykonywanych na bloku zaliczamy:

- **podstawienie (*substitution*)**, czyli zastąpienie jednej grupy znaków (również o długości 1) inną grupą znaków.
  Podstawienie jest operacją podatną na ataki statystyczne wykorzystujące fakt, iż częstość pojawiania się grup znaków w szyfrogramie (*ciphertext*) odpowiada częstości, z jaką pojawiają się one w tekście otwartym.
- **przestawianie (*transposition*)**, czyli wykonywanie permutacji grup znaków wewnątrz bloku.

Żaden z wymienionych typów operacji nie ma najmniejszych szans na efektywne praktyczne zastosowanie.

Powszechnie znane algorytmy blokowe stanowią kompozycję atomowych operacji zapewniając w ten sposób wysoki poziom bezpieczeństwa – są to tzw. **szyfry złożone**.

W szyfrach złożonych bardzo często operacje atomowe grupuje się w pary: podstawienie + przestawianie tworząc tzw. **rundę (*round*)** – wykorzystywane współcześnie szyfry blokowe wykorzystują wiele rund w celu zaszyfrowania bloku.

5

Podstawy kryptografii (4)

Zadaniem **podstawienia** w każdej rundzie jest **utrudnienie wychwycenia związku między kluczem a wynikowym szyfrogramem**, natomiast **przestawianie** powoduje **rozproszenie informacji statystycznej**.

Do szyfrów blokowych zaliczają się najpowszechniej stosowane algorytmy szyfrowania symetrycznego: (1) Data Encryption Standard (DES), (2) Triple DES, (3) Rijandel – Advanced Encryption Standard (AES) – następca DES, (4) IDEA, (5) Blowfish, (6) RC2, i inne.

Szyfry strumieniowe to tak naprawdę szyfry blokowe o długości bloku równej jeden znak.
Zasadnicza różnica, a jednocześnie siła szyfrów strumieniowych polega na tym, że **każdy kolejny znak może być poddany innemu przekształceniu przy użyciu innego klucza**.

Wykorzystywane praktycznie szyfry strumieniowe stosują bardzo proste przekształcenia w oparciu o ciąg jednoznakowych kluczy – **strumień kluczy (*keystream*)**.

6

Podstawy kryptografii (5)

Strumień kluczy może być generowany przez generator liczb losowych tworząc tym samym jedyny znany szyfr, o którym wiadomo na pewno, że jest niemożliwy do złamania – tzw. **szyfr jednokrotny (*one-time pad*)**.

Prosta suma modulo 2 (XOR) znaku wiadomości i klucza, czyli tzw. ***Vernam Cipher*** był z powodzeniem wykorzystywany w komunikacji radzieckiej siatki szpiegowskiej działającej w Stanach Zjednoczonych w latach 40-tych ubiegłego wieku.

Koncepcja **kryptosystemów asymetrycznych** jest oparta na założeniu, że pełne wykorzystanie wszystkich mechanizmów kryptograficznych jest możliwe przy **utajnieniu klucza deszyfrującego** oraz wszelkich informacji, które mogłyby „ułatwić" jego ustalenie.

W stosunku do **wszystkich pozostałych elementów** – zwłaszcza algorytmów – wykorzystywanych w trakcie operacji kryptograficznych powinno być przyjęte założenie, że są one **powszechnie dostępne**.

7

Podstawy kryptografii (6)

Praktyka dowiodła, że utrzymanie informacji w tajemnicy jest zadaniem wyjątkowo trudnym.

Aby w stopniu akceptowalnym być pewnym, że informacja pozostaje niedostępna dla osób do tego niepowołanych należy:

1. Dokonywać okresowej zmiany zabezpieczenia zmniejszając w ten sposób prawdopodobieństwo ewentualnego jego złamania (*compromising*);
2. Bezzwłocznie przeprowadzać zmianę zabezpieczenia w momencie wykrycia jego naruszenia;
3. Ograniczyć informację decydującą o zapewnieniu poufności danych do **niezbędnego minimum** oraz przechowywać ją w możliwie najmniejszej liczbie miejsc.

W ogólności o **algorytmie asymetrycznym** możemy mówić, jeśli dla każdej pary odpowiadających sobie kluczy szyfrującego i deszyfrującego jest **obliczeniowo „niewykonalne" (*computionally unfeasible*) ustalenie klucza deszyfrującego w oparciu o klucz szyfrujący**.

8

Podstawy kryptografii (7)

Funkcja szyfrująca w kryptosystemie asymetrycznym to w języku kryptografii **„funkcja jednostronna z oknem wyłazowym” (*trapdoor one-way function*)**, w której wspomniane „okno wyłazowe” stanowi klucz deszyfrujący.

Najpowszechniej wykorzystywanym w praktyce algorytmem asymetrycznym jest RSA, który wziął swoją nazwę od pierwszych liter twórców: Ronalda Rivesta, Adiego Shamira oraz Leonarda Adlemana.
Bezpieczeństwo algorytmu RSA jest oparte o problem faktoryzacji (rozkładu na czynniki pierwsze).

Jednokierunkowe funkcje skrótu (*one-way hash functions*) to funkcje, które w wyniku otrzymania na wejściu „długiego” ciągu znaków o dowolnej długości generują tzw. **skrót (*digest*)**, zwany też **cyfrowym podpisem palca (*digital fingerprint*)** – ciąg znaków o stałej długości.

O bezpieczeństwie funkcji skrótu decyduje stopień „niewykonalności” (*unfeasibility*) znalezienia wejściowego (oryginalnego, bądź dowolnego innego) ciągu znaków dla danego skrótu wiadomości (*message digest*).

9

Podstawy kryptografii (8)

Najpowszechniej stosowanymi algorytmami skrótu są: Message Digest 5 (MD5), oraz Secure Hash Algorithm 1 (SHA-1).
W obu wymienionych algorytmach znaleziono błędy, które powodują, że stopniowo odchodzi się od nich na rzecz SHA-2 – następcy SHA-1.

Schematy podpisów cyfrowych (*digital signature schemes*) są sposobem na weryfikację tożsamości (*authentication*) informacji cyfrowej i pełnią rolę podpisu tradycyjnego.

Działanie schematu podpisu elektronicznego polega na wyliczeniu skrótu dla wiadomości, po czym zaszyfrowanie go kluczem prywatnym podmiotu podpisującego.

Najpowszechniej stosowanymi algorytmami szyfrowania asymetrycznego w podpisach cyfrowych są: RSA, oraz Digital Signature Algorithm (DSA) – zazwyczaj w połączeniu z SHA-1.

10

Podstawy kryptografii (9)

Komunikacja dwustronna z wymianą kluczy przez kanał bezpieczny – **jedyna możliwość** jeśli korzystamy z algorytmu, w którym **klucz deszyfrujący może być „łatwo” uzyskany z klucza szyfrującego**.

Złośnik podsłuchuje kanał niezabezpieczony, ale nie jest w stanie przechwycić ani zmodyfikować wiadomości

(3)

Bob przesyła zaszyfrowaną wiadomość kanałem niezabezpieczonym

Alicja przesyła Bobowi swój klucz szyfrujący kanałem bezpiecznym

Bob szyfruje wiadomość kluczem publicznym Alicji

(2) **(1)**

Podstawy kryptografii (10)

Teoretycznie korzystanie z kryptografii klucza publicznego umożliwia transmisję klucza szyfrującego (publicznego) kanałem niezabezpieczonym.

Złośnik jedynie przysłuchujący się komunikacji

Złośnik tylko podsłuchuje kanał niezabezpieczony. Nic nie pomoże mu jednak przechwycenie klucza szyfrującego Alicji. Zanim zdoła go złamać, Alicja już dawno zdąży go parokrotnie zmienić. Złośnik nie jest w stanie odczytać treści wiadomości.

Bob

(3)

Bob przesyła zaszyfrowaną wiadomość kanałem niezabezpieczonym

Alicja

Alicja przesyła Bobowi swój klucz szyfrujący kanałem niezabezpieczonym

Bob szyfruje wiadomość kluczem publicznym Alicji

(2) **(1)**

Podstawy kryptografii (11)

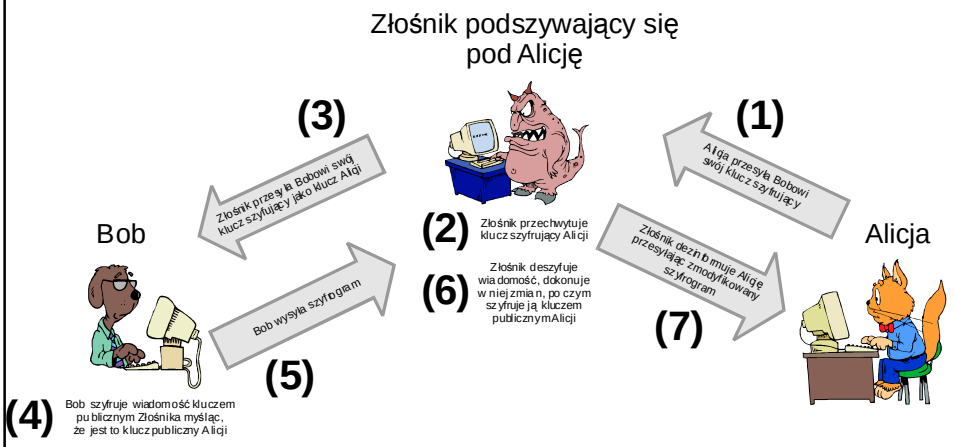
Sam fakt korzystania z kryptografii klucza publicznego w trakcie komunikacji **nie daje nam jednak stuprocentowej gwarancji, że wymiana danych między stronami będzie odbywała się w sposób bezpieczny**.
Klucz publiczny jest ciągiem bajtów (o określonych właściwościach), który **nie dostarcza nam jednak żadnych informacji nt. tożsamości właściciela**.

Tzw. **aktywny przeciwnik (*active adversary*)** może deszyfrować wiadomości podszywając się pod odbiorcę – jest to tzw. ***impersonation attack***, lub ***man in-the-middle attack***.

Atak przebiegałby zgodnie z następującym scenariuszem:

1. Złośnik przechwytuje klucz publiczny Alicji, który miał dotrzeć do Boba.
2. Zamiast klucza publicznego Alicji Złośnik przekazuje Bobowi klucz publiczny wygenerowany przez siebie.
3. Bob szyfruje wiadomość kluczem publicznym otrzymanym od Złośnika będąc przekonanym, że jest to klucz publiczny Alicji.
4. Złośnik przechwytuje wiadomość zaszyfrowaną przez Boba, dokonuje w niej zmian po czym szyfruje zmodyfikowaną wiadomość kluczem publicznym Alicji przyczyniając się tym samym do jej dezinformacji.

Podstawy kryptografii (12)



Atak na opisany protokół dystrybucji kluczy **nie wynika ze słabości kryptografii klucza publicznego**, a faktu, że projektanci protokołu zapomnieli zastosować w nim mechanizmów chroniących **integralność klucza publicznego (szyfrującego)**.

Podstawy kryptografii (13)

W celu uchronienia się przed *man-in-the-middle attack* strona wysyłająca wiadomość musi przeprowadzić **uwierzytelnienie klucza publicznego** odbiorcy tej wiadomości.

Zanim jednak nadawca będzie mógł uwierzytelnić klucz odbiorcy wiadomości, sam odbiorca wiadomości musi certyfikować swój klucz publiczny u tzw. zaufanej trzeciej strony (*trusted third party*), która cieszy się zaufaniem obu stron komunikacji (przede wszystkim nadawcy).

Proces certyfikacji w tym przypadku polega na:
- sprawdzeniu tożsamości odbiorcy;
- weryfikacji, czy właścicielem klucza publicznego, który ma podlegać certyfikacji jest rzeczywiście podmiot, który go nadesłał.

Jeśli weryfikacja wypadnie pomyślnie *trusted third party* tworzy tzw. dane certyfikatu – **informację jednoznacznie identyfikującą podmiot** (np. imię, nazwisko, data urodzenia, itp.) **nierozerwalnie powiązaną z kluczem publicznym**. Proces certyfikacji zamyka złożenie przez zaufaną trzecią stronę podpisu na danych certyfikatu.

15

Podstawy kryptografii (14)

Z kolei podstawą do zaufania do strony wystawiającej certyfikat jest fakt **posiadania przez nadawcę kopii ważnego klucza publicznego *trusted third party***, który przykładowo otrzymał bezpieczną drogą (np. w pakiecie dystrybucyjnym oprogramowania).

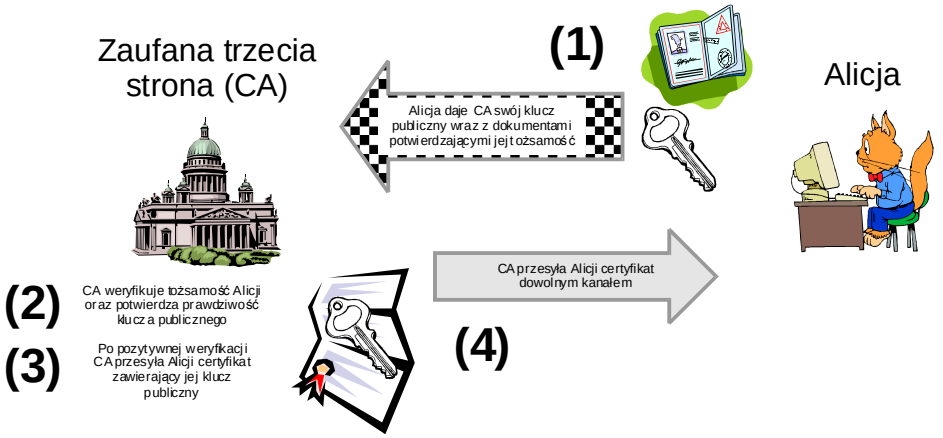
W oparciu o klucz publiczny *trusted third party* **strona wysyłająca może przeprowadzić weryfikację**, czy otrzymany klucz publiczny rzeczywiście pochodzi od odbiorcy wiadomości.
Sam certyfikat może być przesłany dowolnym kanałem: bezpiecznym, bądź niezabezpieczonym.

Po otrzymaniu certyfikatu nadawca wiadomości sprawdza jego autentyczność, następnie akceptuje klucz publiczny znajdujący się w certyfikacie.
Od tego momentu nadawca **może szyfrować korespondencję do okaziciela certyfikatu mając pewność**, że nie zostanie ona odczytana, ani zmodyfikowana z pewnością taką, z jaką ufa *trusted third party*.

16

Podstawy kryptografii (15)

Proces certyfikacji klucza publicznego za pośrednictwem zaufanej trzeciej strony.



17

Podstawy kryptografii (16)

W ogólności pod pojęciem **Public Key Infrastructure (PKI)** kryje się instytucjonalne rozwiązanie, którego celem jest wkomponowanie kryptografii klucza publicznego w system informacyjny zgodnie z wymogami stawianymi przez politykę bezpieczeństwa.

Na PKI składają się:

- **zasady postępowania i procedury** wspomagające wykorzystanie kryptografii klucza publicznego w organizacji;
- wykorzystywane **zaplecze techniczne** (sprzęt i oprogramowanie);
- oraz **ludzie (pełnione przez nich role i związany z tym zakres odpowiedzialności)** biorący udział w wystawianiu, dystrybucji, oraz zarządzaniu elementami kryptografii publicznej.

PKI pełni w kontekście kryptografii klucza publicznego rolę analogiczną do roli jaką odgrywa polityka bezpieczeństwa informacji w ramach instytucji.

Sprawując kontrolę nad wystawianymi certyfikatami PKI określa poziom zaufania pomiędzy (1) z jednej strony okazicielami certyfikatów, (2) z drugiej strony podmiotami akceptującymi certyfikat.

18

Podstawy kryptografii (17)

Proces tworzenia PKI – podobnie jak polityki bezpieczeństwa informacji – zawsze powinno rozpoczynać szczegółowe określenie wymagań.

Punktem wyjściowym powinna być **specyfika działalności organizacji oraz operacje biznesowe, które mają być wspierane przez kryptografię publiczną**. Dopiero na dalszym etapie powinna być rozpatrywana wykorzystywana technologia.

Na świecie wypracowano dwie koncepcje rozwiązania problemów związanych z uwierzytelnianiem klucza szyfrującego w oparciu o kryptografię klucza publicznego:

- **wzajemna certyfikacja podmiotów** biorących udział w komunikacji – rolę zaufanej trzeciej strony może pełnić każdy z uczestników komunikacji;
- **wyznaczenie specjalnego podmiotu**, który dla wszystkich uczestników komunikacji będzie odgrywał rolę *trusted third party* – na tym wyodrębnionym obiekcie będzie spoczywała cała odpowiedzialność za wiarygodność certyfikatów.

By uniknąć konieczności wzajemnej certyfikacji każdego obiektu przez każdy inny, która uczyniłaby pierwsze podejście zupełnie niepraktycznym, przyjęto założenie „propagacji zaufania" – w skrócie jeśli A ufa B, a B ufa C to A jednocześnie ufa C.

19

Podstawy kryptografii (18)

Pierwsza z wymienionych metod zakłada **solidarną odpowiedzialność użytkowników za wiarygodność certyfikatów**.

To właśnie przesądziło, że pierwsze podejście nie znalazło uznania wśród organizacji, natomiast znakomicie sprawdziło się w zastosowaniach prywatnych, na takim bowiem rozproszonym modelu PKI jest oparte **Pretty Good Privacy (PGP)**.

Philip Zimmermann – twórca PGP – wyszedł z założenia, że każdy powinien świadomie chronić swoją prywatność – nie zdając się w tym względzie na żadną instytucję.
W rzeczywistości – mimo deklaracji – **większość ludzi nie dba o swoją prywatność, co wynika głównie z braku świadomości i lenistwa**.
Z tego pewnie względu PGP lub GNU Privacy Guard (GPG) nie ma aż tak wielu użytkowników jak można by oczekiwać.

W zastosowaniach komercyjnych oraz w administracji publicznej znakomicie przyjął się z kolei scentralizowany model PKI.

Najpowszechniej stosowanym standardem zapisu certyfikatu w oparciu o centralny model PKI jest X.509.

20

Podstawy kryptografii (19)

PKI jest odpowiedzialne za: (1) **wystawianie certyfikatów**, (2) **unieważnianie certyfikatów**, (3) **uaktualnianie** oraz **odnawianie certyfikatów**, a także (4) **rejestrację wszystkich powyższych zdarzeń**.
Z tego względu w praktyce trudno raczej mówić o PKI innym niż zarządzane centralnie przez specjalnie wydesygnowaną instytucję *trusted third party*, czyli tzw. ***Certificate Authority* (CA)**.

W niektórych implementacjach PKI obowiązki związane z:
- weryfikacją tożsamości podmiotów ubiegających się o certyfikat,
- generacją pary kluczy

spoczywają na barkach specjalnie w tym celu wyznaczonego ciała zwanego ***Registration Authority*** (RA).

Zazwyczaj jedno CA może wystawiać certyfikaty podmiotom zweryfikowanym przez wiele zaufanych RA.
W takim modelu RA przejmuje odpowiedzialność za **wnioski certyfikacji (*certificate signing requests*)** wysyłane do CA.

21

Podstawy kryptografii (20)

Certyfikat składa się z trzech zasadniczych elementów:
- **nazwy jednostki** dla której został on wystawiony – jest to tzw. **podmiot (*subject*)** certyfikatu;
- **klucza publicznego** skojarzonego z podmiotem certyfikatu;
- **podpisu cyfrowego złożonego przez CA**, który stanowi podstawę dla weryfikacji informacji zawartych w certyfikacie.

Istnieje wiele formatów zapisu certyfikatów klucza publicznego, jednak najpowszechniej stosowanym formatem zapisu certyfikatu jest X.509v3.
Chociaż istnieją inne standardy certyfikatów (np. PGP/GPG) – standard X.509 w wersji 3 jest dominujący.

Elementy składowe certyfikatu zapisanego w formacie X.509 version 3:
- ***Distinguished Name* podmiotu**, dla którego wystawiono certyfikat;
- ***Distinguished Name* urzędu certyfikacji**;
- **data wystawienia** oraz **data ważności** certyfikatu – certyfikat jest ważny wyłącznie w przedziale czasu ograniczonym tymi datami;
- **wersja** – istnieją różne wersje standardu X.509 – ostatnią i jednocześnie najpowszechniej używaną jest wersja 3.

22

Podstawy kryptografii (21)

Elementy składowe certyfikatu zapisanego w formacie X.509 version 3:

- **numer seryjny (*serial number*)** – każdy certyfikat wydawany przez urząd certyfikacji posiada niepowtarzalny numer seryjny, który jest unikalny dla danego CA – zatem kombinacja tego numeru i oraz DN urzędu certyfikacji gwarantuje globalną niepowtarzalność;
- **informację o algorytmach** przy użyciu których został złożony podpis przez jednostkę certyfikującą (np. `sha1RSA`);
- **długość oraz rodzaj klucza publicznego** zawartego w danych certyfikatu;
- informację o tym, jakie może być **przeznaczenie certyfikatu**: podpisywanie innych certyfikatów – dla CA, podpisywanie ***Certificate Revocation List***, podpisywanie i szyfrowanie poczty elektronicznej, uwierzytelnienie serwera – witryny internetowej, bądź serwera poczty elektronicznej, itp.
- informacja, skąd może być pobierany ***certificate revocation list*** – tzw. **punkt pobierania CRL (*CRL distribution point*)**, np. `http://www.sun.com/pki/pkismica.crl`

23

Podstawy kryptografii (22)

***Distinguished Name* (DN)** stanowi pełną ścieżkę w drzewie X.500, jest to stosunkowo długi łańcuch znaków w którym poszczególne węzły w ścieżce są oddzielone przecinkami.

W skład Distinguished Name zazwyczaj wchodzą następujące pola:

- ***Common Name* (CN)** – pełna nazwa podmiotu dla którego wystawiono certyfikat: jednostki organizacyjnej, osoby, bądź nazwa DNS hosta, na którym umieszczono witrynę, bądź działa serwer poczty;
- ***Organization Unit* (OU)** – jednostka organizacyjna, do której należy podmiot certyfikatu;
- ***Organization* (O)** – organizacja, do której należy podmiot
- ***State* (S)** – stan, bądź prowincja, z którą związany jest podmiot – ma zastosowanie głównie w przypadku Stanów Zjednoczonych;
- ***Country* (C)** – państwo, z którym związany jest podmiot.

Format DN sprawia, że podmiot identyfikowany za pomocą danego CN (zazwyczaj osoba) może mieć wystawionych wiele różnych certyfikatów – np. jeśli jest związany z wieloma organizacjami jednocześnie.

24

Podstawy kryptografii (23)

```
# exemplary DN --- from the keystore of Internet Explorer
# DN may not contain EOLs
CN = CC Signet – RootCA, OU = Centrum Certyfikacji Signet,
O = TP Internet Sp. z o.o., C = PL
```

Certyfikaty X.509 **dzieli się na klasy w zależności od procesu weryfikacji**, dokumentów wymaganych od aplikanta, oraz oczywiście uiszczanej opłaty.

Przyznanie certyfikatu wysokiego poziomu zaufania zazwyczaj wiąże się z bardzo rygorystyczną (i kosztowną) weryfikacją tożsamości przez urząd certyfikacji.
Z kolei utworzenie wystawieniu certyfikatu osobistego towarzyszy co najwyżej sprawdzenie poprawności adresu poczty elektronicznej.

Zatem w podczas **decydowania o tym, czy zaufać** jednostce przedstawiającej certyfikat bardzo istotnym elementem jest **sprawdzenie klasy certyfikatu**.

Klasa certyfikatu określa obszary zastosowania certyfikatu: popisywanie korespondencji, szyfrowanie korespondencji, uwierzytelnienie, itp.

25

Podstawy kryptografii (24)

PKI sprawuje pełną kontrolę nad wystawionymi przez siebie certyfikatami.
Gdy istnieje niebezpieczeństwo wykorzystania certyfikatu w celach innych, niż został on wystawiony **tylko PKI jest w stanie unieważnić certyfikat**.

Powiadomienie o unieważnieniu certyfikatu jest realizowane poprzez publikację ***Certificate Revocation List*** **(CRL)**.

W najprostszym przypadku CRL stanowi lista numerów seryjnych certyfikatów podpisanych przez dane CA, które zostały unieważnione.
Tego rodzaju listę umieszcza się w publicznie dostępnym repozytorium skąd może być ona skopiowana przez obiekty ufające danemu CA.

Obiekt, któremu przedstawiono certyfikat wystawiony przez dane CA powinien sprawdzić odpowiednie CRL – na podobnej zasadzie jak sprzedawca weryfikuje kartę płatniczą u jej wystawcy.

Dla zminimalizowania czasu dzielącego unieważnienie certyfikatu od chwili, kiedy o fakcie tym zostaje poinformowany obiekt ufający danemu CA czasami poza standardową metodą ***pull*** stosuje się również metodę ***push***.

26

Podstawy kryptografii (25)

Przy zastosowaniu tzw. metody *push* CA rozsyła CRL do wszystkich obiektów, które mu ufają – oczywiście tego rodzaju rozwiązanie ma ograniczone zastosowanie (np. w ramach jednej korporacji, czy departamentu).

Dla potrzeb systemów zabezpieczających wyjątkowo „wrażliwe" dane, w których wady CRL wykluczają jego efektywne zastosowanie opracowano **Online Certificate Status Protocol (OCSP)** oraz **Data Validation and Certification Server Protocol (DVCS)** umożliwiające weryfikację statusu certyfikatu w czasie rzeczywistym.

Istota działania wspomnianych protokołów jest wyjątkowo nieskomplikowana – obiekt ufający danemu CA zamiast pobierać całą CRL wysyła zapytanie o status przedstawianego mu w danym momencie certyfikatu.

Do dystrybucji certyfikatów przechowywanych w repozytorium certyfikatów można posłużyć się dowolnymi protokołami komunikacji – w najprostszym przypadku można posłużyć się w tym celu zwykłą pocztą elektroniczną (SMTP).
Zostały opracowane standardy dostępu do repozytorium certyfikatów przez FTP oraz HTTP. Samo PKIX Working Group rekomenduje do tego celu usługi katalogowe z dostępem przez LDAP.

27

Podstawy kryptografii (26)

Duże implementacje PKI obsługujące wiele tysięcy użytkowników posiadają więcej niż jedno CA – poszczególne certyfikaty są natomiast wystawiane tylko przez jedno z takich CA.

By certyfikaty wystawione przez różne CA mogły być wykorzystywane wśród użytkowników danej implementacji PKI istnieje konieczność stworzenia tzw. **architektury zaufania PKI**.

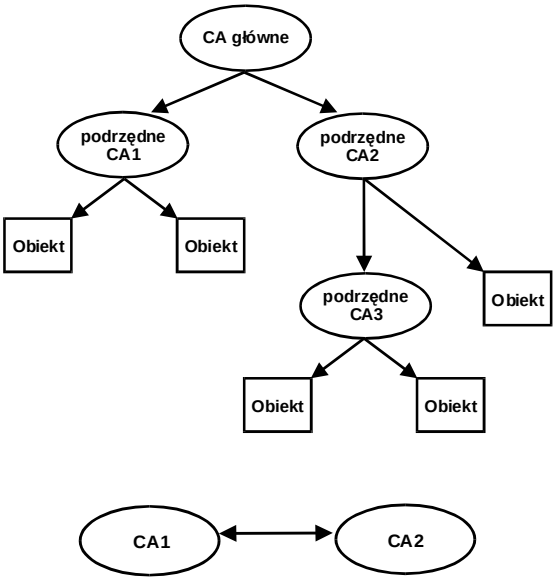
Chyba najbardziej oczywistą architekturą zaufania PKI jest **architektura hierarchiczna** – na samym szczycie hierarchii znajduje się **główne ciało certyfikujące (*root certificate authentication authority – root CA*)**.

Wszystkie certyfikaty zarządzane przez dane PKI – w tym certyfikaty podległych CA – są podpisane (bezpośrednio, bądź pośrednio) certyfikatem wystawionym przez CA główne.

Jeśli obiekt ufa certyfikatowi *root CA* ufa również wszystkim certyfikatom wystawionym przez dowolne z podległych CA.
Ujawnienie klucza prywatnego *root CA* powoduje załamanie całej architektury – czyli mamy w tym przypadku do czynienia z tzw. ***single point-of-failure vulnerability***.

28

Podstawy kryptografii (27)

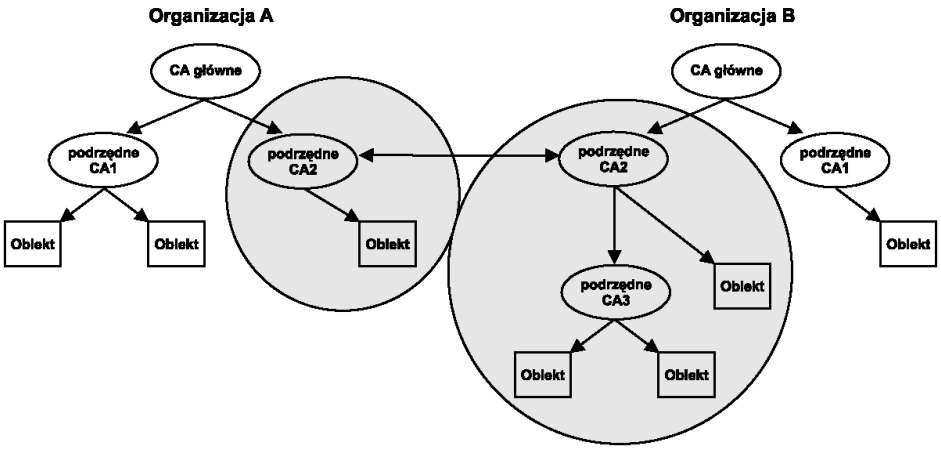


Drugim niespecjalnie skomplikowanym rodzajem architektury zaufania PKI jest tzw. **architektura wzajemnego zaufania** jest opierająca się wzajemnym podpisywaniu certyfikatów przez CA.
Tego rodzaju architektura znajduje zastosowanie w momencie łączenia dwóch (lub więcej) dotychczas odrębnych PKI.

29

Podstawy kryptografii (28)

Ostatnim schematem łączenia drzew zaufania do certyfikatów CA jest tzw. **hybrydowa architektura zaufania** posiadająca cechy dwóch wcześniej wymienionych metod i polegająca na stworzeniu wzajemnego zaufania na dowolnym poziomie hierarchii.



30

W następnej części wykładu ...

Korzystanie z mechanizmów kryptograficznych w aplikacjach tworzonych w oparciu o platformę Java™

Security providers (1)

Mechanizmy kryptograficzne w Java™ wykorzystywane w trakcie szyfrowania, obliczania skrótów wiadomości, czy składania podpisów cyfrowych są dostarczane jako zestaw klas abstrakcyjnych umieszczonych w pakietach: **`java.security*`** oraz **`java.crypto*`**.

Konkretne implementacje tych klas dostarczane są razem z SDK – przykładowo Java 2 Standard Environment w wersji 1.2 dostarcza dwie różne implementacje podpisów elektronicznych – jedną opartą o RSA, drugą o DSA.

Oczywiście istnieje również możliwość korzystania z implementacji powyższych mechanizmów pochodzących od niezależnych dostawców oprogramowania (*independent software vendors - ISV*).

Mechanizmy kryptograficzne w Java są zrealizowane w oparciu o architekturę tzw. **dostawców bezpieczeństwa (*security providers*)**, która umożliwia dowolne włączanie, bądź usuwanie konkretnych implementacji par: mechanizm kryptograficzny-algorytm w Java.

Security providers (2)

Infrastruktura *security providers* umożliwia wyszukanie w trakcie działania aplikacji konkretnych implementacji poszczególnych mechanizmów bez konieczności wprowadzania zmian w kodzie aplikacji.

Korzystanie z infrastruktury *security providers* polega na **korzystaniu ze spójnego API, które uniezależnia aplikację od dostawcy implementacji** poszczególnych mechanizmów kryptograficznych.

Koncepcja *security providers* opiera się na dwóch zasadniczych elementach: (1) **mechanizmie (*engine* lub *service*)** oraz (2) **algorytmie (*algorithm*)**.

Mechanizm to jeden z rodzajów operacji dostarczanych przez *security provider*, którym może być prymityw kryptograficzny, schemat podpisu cyfrowego, magazyn kluczy (*keystore*), generator liczb losowych, itp.

Przykładem *engine* jest skrót wiadomości (*message digest*).
Sama koncepcja skrótu jest niezależna od konkretnego sposobu wyliczania skrótu wiadomości – skróty wiadomości posiadają pewne wspólne właściwości.

Klasa hermetyzująca ogólną koncepcję skrótu w oparciu o ściśle określony interfejs nazywa się mechanizmem.
Mechanizmy stanowią abstrakcję niezależną od konkretnego algorytmu.

33

Security providers (3)

Security providers dostarczane przez Sun Microsystems® razem z Java 2 Standard Edition SDK wspierają 17 różnych mechanizmów kryptograficznych obejmujących m.in. reprezentacje:

- **prymitywów kryptograficznych**: `Cipher`, `MessageDigest`;
- **podpisu cyfrowego**: `Signature`;
- **kodu potwierdzającego autentyczność otrzymanej wiadomości (*message authentication code*)**: `Mac` – MAC wyliczony jako skrót wiadomości sparametryzowany kluczem tajnym określany jest mianem HMAC (RFC 2104);
- **generatorów i fabryk**: `KeyGenerator`, `KeyFactory`, `KeyPairGenerator`, `SecretKeyFactory`, `CertificateFactory`, `SecureRandom`;
- **magazynu kluczy**: `KeyStore`;
- **parametry przekazywane algorytmom oraz generatory takich parametrów**: `AlgorithmParameters`, `AlgorithmParameterGenerator`;
- **protokół negocjacji kluczy**: `KeyAgreement` (Diffie-Hellman);
- **kontekst SSL/TLS**: `SSLContext`.

Mechanizm jest zawsze implementowany przez konkretny algorytm.

Przykładowo skrót wiadomości może być implementowany przez konkretny algorytm, jak np. MD5 lub SHA-1.

34

Security providers (4)

Implementacja algorytmu jest dostarczana jako konkretna klasa rozszerzająca abstrakcyjną klasę danego mechanizmu (dokładnie – implementacja SPI) – nic nie stoi na przeszkodzie, by dany algorytm nie mógł być dostarczany przez wiele klas.

Przykładowo można korzystać z implementacji SHA-1 dostarczonej razem z platformą Java™, jak również otrzymaną od niezależnego dostawcy (np. BC). Oczywiście w opisanej sytuacji na wyjściu obydwu implementacji tego samego algorytmu muszą pojawiać się jednakowe wyniki przy takich samych danych wejściowych.

Infrastruktura *security providers* umożliwia kojarzenie mechanizmów z algorytmami oraz odwzorowanie pary: mechanizm/algorytm na konkretną implementację.

Celem infrastruktury dostawców bezpieczeństwa jest:

- **od strony administratora aplikacji**: umożliwienie podmiany jednej implementacji danego algorytmu (np. SHA-1) – inną – bez konieczności wprowadzania modyfikacji w kodzie aplikacji;
- **od strony programisty**: sprowadzenie do minimum operacji koniecznych do zastąpienia jednych algorytmów (realizujących dany mechanizm) innymi.

35

Security providers (5)

Za zarządzanie oraz wyszukiwanie konkretnych dostawców bezpieczeństwa zawierających implementację żądanej pary: algorytmy/mechanizm jest w *Java™ Cryptography Architecture* odpowiedzialna klasa **`java.security.Security`**.

Nierozszerzalna (**`final`**) klasa **`Security`** jest typową implementacją wzorca projektowego *singleton* z konstruktorem prywatnym – podobnie jak klasy **`System`**, czy **`Math`** z pakietu **`java.lang`** programista nie może utworzyć instancji tej klasy.

Istnieją dwie metody na dodawanie/usuwanie dostawców bezpieczeństwa:

- poprzez modyfikację listy dostępnych *security providers* zamieszczonej w konfiguracji tzw. pakietu bezpieczeństwa Java™ (pakietów: **`java.security*`** oraz **`java.crypto*`**) – czyli JCA – konfiguracja jest zapisana w pliku: **`$JREHOME/lib/security/java.security`**.
- bezpośrednio w kodzie aplikacji za pośrednictwem API dostarczanego przez klasę **`Security`**.

W momencie wywołania metody z interfejsu udostępnianego przez pakiet bezpieczeństwa maszyna wirtualna Java™ jest odpowiedzialna za ustalenie – w oparciu o konfigurację, który z zainstalowanych *security providers* powinien być użyty.

36

Security providers (6)

```
# extract from the configuration of Java 2 SDK 1.5
# (java.security) --- list of security providers and their
# preference orders
security.provider.1=sun.security.provider.Sun
security.provider.2=sun.security.rsa.SunRsaSign
security.provider.3=com.sun.net.ssl.internal.ssl.Provider
security.provider.4=com.sun.crypto.provider.SunJCE
security.provider.5=sun.security.jgss.SunProvider
security.provider.6=com.sun.security.sasl.Provider
```

Identyfikator znajdujący się po nazwie właściwości **`security.provider`** określa priorytet przyznany danemu dostawcy bezpieczeństwa w trakcie wyszukiwania implementacji dla pary: mechanizm/algorytm.

Każdy dostawca bezpieczeństwa udostępniany przez dane JRE musi być opatrzony unikalnym identyfikatorem począwszy od 1.
Umieszczenie identyfikatora o wartości 1 oznacza wskazanie domyślnego dostawcy bezpieczeństwa dla danej maszyny wirtualnej.

37

Security providers (7)

Oczywiście w przypadku, gdy ten sam algorytm jest realizowany w ramach dwóch dostawców wykorzystana zostanie implementacja dostarczana przez *security provider* o wyższym priorytecie – czyli opatrzonego niższym identyfikatorem.

Zarządzanie *security providers* bezpośrednio w kodzie aplikacji, jest możliwe za pośrednictwem następujących metod udostępnianych przez klasę **`Security`**:

- **`addProvider()`**, która dodaje nowego dostawcę na koniec listy;
- **`insertProvider()`** – wstawia dostawcę na określoną pozycję;
- **`removeProvider()`** – usuwa dostawcę.

Ze względu na zachowanie spójności polityki bezpieczeństwa JRE wymienione metody klasy **`Security`** wywołują metodę **`checkSecurityAccess()`** klasy **`SecurityManager`**, która – jak wiemy – sama korzysta z **`AccessController`**.

Instalacja dostawcy bezpieczeństwa o określonej nazwie wymaga:

```
java.security.SecurityPermisssion
   insertProvider.<provider-name>
```

Z kolei usunięcie dostawcy bezpieczeństwa wymaga:

```
java.security.SecurityPermission removeProvider.<provider-name>
```

38

Security providers (8)

Klasy reprezentujące mechanizmy (*engines*) w ramach pakietu bezpieczeństwa są klasami abstrakcyjnymi realizującymi wzorzec projektowy *factory*.

Wywołanie przez programistę w kodzie aplikacji metody **`getInstance()`** klasy reprezentującej dany *engine* (np. **`java.security.MessageDigest`**) oznacza żądanie utworzenia instancji klasy konkretnej stanowiącej właściwą implementację pary: mechanizm/algorytm.

Jako argument wywołania programista podaje klucz, pod którym zarejestrowany został żądany algorytm – opcjonalnie programista może również określić dostawcę bezpieczeństwa, z którego chciałby skorzystać.

Klasa reprezentująca dany mechanizm (**`MessageDigest`**, **`KeyPairGenerator`**, itd.) przekazuje żądanie programisty do klasy **`Security`**, której zadaniem jest znalezienie odpowiedniej implementacji w ramach konfiguracji JRE.

Zapytanie przekazywane do **`Security`** jest łańcuchem znaków zgodnym z formatem: **`"<engine-name>.<algorithm-name>"`**, np. **`MessageDigest.SHA1`**, który stanowi jednocześnie klucz pod którym zarejestrowano właściwą implementację w ramach danego dostawcy.

39

Security providers (9)

Infrastruktura dostawców bezpieczeństwa umożliwia również stosowanie synonimów (*aliases*) w miejscu nazwy algorytmu, np. **`MessageDigest.SHA`**, może stanowić alias do implementacji **`MessageDigest.SHA1`**, bądź następcy SHA-1 – **`MessageDigest.SHA2`**.

O ile programista nie podał jawnie w wywołaniu, z usług którego *security provider* chciałby skorzystać klasa **`Security`** lokalizuje pierwszego dostawcę (w kolejności zapisanej w konfiguracji) zawierającego implementację żądanej pary: mechanizm/algorytm, np. **`KeyPairGenerator.DSA`**.

Na żądanie klasy **`Security`** klasa dostawcy przekazuje instancję klasy, o której sam dostawca wie – na podstawie wewnętrznego rejestru, że stanowi ona implementację żądanej pary.

Zazwyczaj sama klasa reprezentująca *security provider* – czyli rozszerzająca abstrakcyjną klasę: **`java.security.Provider`**, nie stanowi właściwej implementacji żądanej pary.

Podstawowym bowiem zadaniem klasy dostawcy jest ustalenie – na podstawie swojego wewnętrznego rejestru, która klasa (w ramach reprezentowanego przez nią *security provider*) implementuje żądaną parę.

40

Security providers (10)

Jeżeli klasa **`Security`** odnajdzie implementację pary: mechanizm/algorytm w ramach dostawcy bezpieczeństwa: (1) określonego przez programistę aplikacji, bądź (2) w zestawie dostępnych *security providers*, tworzona jest instancja znalezionej klasy, która następnie jest przekazywana do metody **`getInstance()`** danego mechanizmu.

Jeżeli klasa **`Security`** nie jest w stanie odnaleźć implementacji żądanej pary: mechanizm/algorytm – np. implementacji **`MessageDigestSpi`** zarejestrowanej pod kluczem **`"MessageDigest.3DES"`** – zgłaszany jest wyjątek: **`NoSuchAlgorithmException`**.

```
/** calculating message digest for the specific string of
    characters */
MessageDigest md = MessageDigest.getInstance("SHA");
String data = "String I wish to calculate digest for";
byte[] dataDigest = md.digest(data.getBytes());
```

41

Security providers (11)

```
/** direct calculation of HMAC */
MessageDigest md = MessageDigest.getInstance("SHA");
String data = "String I wish to calculate HMAC for";
String secretKey = "Mein streng geheimes ganz langes passwort";
byte[] skBytes = secretKey.getBytes();
md.update(data.getBytes());
md.update(skBytes);
bytes[] digest = md.digest();
md.update(skBytes);
md.update(digest);
byte[] hmac = md.digest();
```

```
/** calculation of HMAC using dedicated API */
Mac mac = Mac.getInstance("HmacSHA1");
KeyStore keyStore = KeyStore.getInstance(
   KeyStore.getDefaultType());
String data = "My input data";
String password = "My top secret quite lenghty password";
mac.init(keyStore.getKey("username", password.toCharArray()));
byte[] hmac = mac.doFinal(data.getBytes());
```

42

Security providers (12)

```
/** signing a message */
try {
   String message = "A message I wish to sign",
          username = "username", password = "password";
   Signature signAlgorithm = Signature.getInstance(
      "SHA1withDSA");
   KeyStore keyStore =
      KeyStore.getInstance( KeyStore.getDefaultType() );
   PrivateKey privKey = (PrivateKey) keyStore.getKey(
      username, password.toCharArray() );
   SignedObject signedObj = new SignedObject(
      message, privKey, signAlgorithm );
   byte[] signature = signedObj.getSignature();
   /** serialize the signed object */
   FileOutputStream out = new FileOutputStream("signed.dat");
   ObjectOutputStream oout = new ObjectOutputStream(out);
   oout.writeObject(signedObj);
} catch (Exception ex) {
   ex.printStackTrace();
}
```

43

Security providers (13)

Chcąc utworzyć własny zestaw klas implementujących wybrane operacje kryptograficzne – czyli własnego dostawcę bezpieczeństwa – w oparciu o określone algorytmy, należy rozszerzyć klasę **`java.security.Provider`**.

Mimo, iż sama klasa **`Provider`** jest abstrakcyjna – żadna ze zdefiniowanych w jej ciele metod nie jest abstrakcyjna, co sprawia, że w zazwyczaj stworzenie własnego dostawcy sprowadza się do rejestracji wybranych implementacji mechanizmów kryptograficznych.

Własny (*proprietary*) *security provider* musi posiadać publiczny konstruktor domyślny (bezparametrowy) – taki bowiem konstruktor jest wywoływany przez klasę **`Security`**.

Oczywiście tworzenie własnego *security provider* ma sens, o ile dostarcza on rzeczywistej funkcjonalności (implementacji par: mechanizm/algorytm), co wymaga powiązania dostawcy z właściwymi implementacjami.

Tylko rejestracja poszczególnych implementacji umożliwia dostawcy bezpieczeństwa dokonać odwzorowania mechanizmu i algorytmu, bądź synonimu (*alias*) na konkretną klasę.

44

Security providers (14)

Wszystkie metody odpowiedzialne za rejestrację implementacji w ramach **`Provider`** wywołują **`SecurityManager.checkSecurityAccess()`** zapewniając tym samym spójność polityki bezpieczeństwa JRE.

Wywoływany przez **`SecurityManager`** **`AccessController`** sprawdza, czy klasa dostawcy posiada **`SecurityPermission`** o nazwie umożliwiającej wykonanie żądanych operacji.

Wywoływanie metody **`put()`** dodającej właściwość – rejestrującej implementację mechanizm/algorytm (alias) w ramach danego *security provider* – wymaga:

```
java.security.SecurityPermission
"putProviderProperty.<provider-name>"
```

Wywoływanie metody **`remove()`** usuwającej konkretną właściwość dostawcy wymaga:

```
java.security.SecurityPermission
   "removeProviderProperty.<provider-name>"
```

45

Security providers (15)

Wywołanie metody **`clear()`** usuwającej wszystkie właściwości danego dostawcy – czyszczącej rejestr implementacji dostarczanych przez *security provider* wymaga:

```
java.security.SecurityPermission
   "clearProviderProperty.<provider-name>"
```

Oczywiście **`<provider-name>`** może być zastąpiony przez **`"*"`**.

Zatem *codebase*, w którym umieszczono klasę reprezentującą własnego dostawcę musi mieć przyznane uprawnienia umożliwiające modyfikację wewnętrznych rejestrów implementacji.

Klasa **`Provider`** jest sama podklasą **`java.util.Properties`**.
Z tego względu w stosunku do zarejestrowanych implementacji algorytmów używa się również terminu: **właściwości (*properties*)**.

Korzystając z faktu, że klasa **`Provider`** rozszerza **`Properties`** dostawca może również ustawić dowolne inne (w tym własne) właściwości, jak np. informację o tym, że dostarcza natywnej implementacji algorytmów kryptograficznych.

```
put("NativeImplementation","false");
```

46

Security providers (16)

Bardzo wątpliwe, by własne (*proprietary*) właściwości zdefiniowane w ramach *security provider* zostały kiedykolwiek użyte przez zewnętrznych użytkowników, ale ich stosowanie zwiększa elastyczność wewnętrznej implementacji dostawcy.

Przykład „wewnętrznej" właściwości informującej, że *security provider* nie dostarcza natywnej implementacji określonego algorytmu.

```
put("Alg.NativeImplementation.MyAlgorithm","false");
```

```
package com.mycompany.security;
import java.security.*;
public class MyCompanyProvider extends java.security.Provider {
   public MyProvider () {
      super("My Company", "MyCompany Security Provider v.1");
      put("KeyGenerator.XOR",
         "com.mycompany.security.XORKeyGenerator");
      put("KeyPairGenerator.MyCompany",
         "com.mycompany.security.MyCompanyKeyPairGenerator");
      put("Alg.Alias.MessageDigest.SHA-1", "SHA");
   }
}
```

47

Klucze (1)

Klucze są elementem nieodłącznie związanym z wykorzystaniem prymitywów kryptograficznych – w szczególności są niezbędne w czasie składania i weryfikacji podpisów cyfrowych, bądź szyfrowania.

W ogólności klucze są wykorzystywane przez wszystkie stosowane praktycznie algorytmy szyfrowania.

W Java™ istnieją dwa rodzaje mechanizmów kryptograficznych (*engines*) operujących bezpośrednio na kluczach: (1) generatory kluczy (*key generators*), oraz (2) fabryki kluczy (*key factories*).

Zadaniem klas reprezentujących **generatory kluczy jest tworzenie kluczy od podstaw** – wygenerowane klucze mogą być tworzone bez przekazywania dodatkowych danych wejściowych, bądź z danymi inicjalizującymi stan generatora przed utworzeniem klucza.

W infrastrukturze bezpieczeństwa Java™ odpowiedzialność za generowanie kluczy tajnych spoczywa na barkach klas rozszerzających: **`javax.crypto.KeyGenerator`**, pary kluczy: publiczny i prywatny są z kolei tworzone przez klasy rozszerzające: **`java.security.KeyPairGenerator`**.

48

Klucze (2)

Zadaniem klas reprezentujących fabryki kluczy jest dwukierunkowa konwersja obiektów reprezentujących klucze w ramach architektury bezpieczeństwa Java™ z i do zewnętrznej reprezentacji kluczy (np. tablicę bajtów, czy specyfikację klucza).

Poza samymi mechanizmami kryptograficznymi wykorzystywanymi w trakcie wykonywania operacji na kluczach architektura kryptografii Java™ (*Java™ Cryptography Architecture* – JCA) obejmuje również szereg klas reprezentujących same klucze używane w trakcie operacji kryptograficznych.

Pojęcie klucza kryptograficznego jest reprezentowane przez interfejs **`java.security.Key`**.

Ze względu na konieczność przekazywania klucza między stronami komunikacji interfejs **`Key`** rozszerza **`Serializable`**, tak aby wszystkie klucze kryptograficzne mogły być przesyłane za pośrednictwem strumieni.

Gdy klucz jest przekazywany między stronami komunikacji zwykle jest kodowany jako ciąg bajtów zgodnie ze ściśle określonym formatem kodowania właściwym dla danego rodzaju klucza.

49

Klucze (3)

Chcąc dowiedzieć się, jaki format kodowania jest właściwy dla konkretnego rodzaju należy wywołać metodę **`getFormat()`**.

Samą reprezentację klucza zgodną z formatem kodowania uzyskamy natomiast po wywołaniu metody **`getEncoded()`**.

Szyfrowanie asymetryczne wymaga pary kluczy: prywatnego i publicznego, co znalazło swoje odzwierciedlenie w JCA, w której zdefiniowano dwa dodatkowe interfejsy – odpowiednio: **`PrivateKey`** oraz **`PublicKey`**.

W JCA – specjalnie z myślą o kluczach wykorzystywanych przez algorytm DSA – wprowadzono również interfejs **`DSAKey`**, który umożliwia odczyt parametrów specyficznych dla obu rodzajów kluczy DSA: G, P, Q.

Jak wiemy DSA jest algorytmem niesymetrycznym, zatem w JCA wprowadzono dodatkowo: **`DSAPrivateKey`** oraz **`DSAPublicKey`**, które dodatkowo pozwalają na odczyt parametrów specyficznych dla odpowiednio klucza prywatnego i publicznego DSA.

Poza interfejsami kluczy DSA w JCA wprowadzono również interfejsy posiadające cechy charakterystyczne kluczy RSA: **`RSAPrivateKey`**, **`RSAPublicKey`**.

50

Klucze (4)

Obiekt nierozszerzalnej (`final`) klasy **`java.security.KeyPair`** przechowuje parę kluczy: prywatny i publiczny – instancja tej klasy powinna być zawsze inicjalizowana w konstruktorze obiektami reprezentującymi obydwa klucze.

Uwaga: wywołanie metody **`getPrivate()`** na obiekcie klasy **`KeyPair`** nie jest weryfikowane przez **`SecurityManager`**.

***Java™ Cryptography Extension* (JCE)** to zestaw pakietów: **`javax.crypto*`** rozszerzających funkcjonalność JCA o implementację prymitywów kryptograficznych podlegających ograniczeniom eksportowym obowiązującym w Stanach Zjednoczonych.

JCE zawiera głównie implementacje:

- algorytmów symetrycznych (DES, czy AES, TripleDES – po zainstalowaniu polityki umożliwiającej nieograniczone wykorzystanie funkcjonalności JCE);
- algorytmów negocjacji kluczy (*key agreement*) – Diffie-Hellman;
- algorytmów wyliczających kod uwierzytelniający wiadomości (HmacMD5, HmacSHA1);
- schematów szyfrowania wykorzystujących hasło: ***Password Based Encryption* (PBE)** (PWEWithMD5AndDES, PWEWithMD5AndTripleDES).

51

Klucze (5)

Dopiero od wersji Java 2 Standard Edition w wersji 1.4 JCE zostało włączone do standardowych dystrybucji: JRE i SDK – wcześniej było dystrybuowane jako zupełnie odrębny pakiet.

Do tej pory jednak wymagane jest pobranie ze strony Sun Microsystems® *Java™ archives* zawierających implementację polityki użycia kryptografii umożliwiającą korzystanie z rozszerzonej wersji prymitywów kryptograficznych zawartych w JCE, np. AES, czy TripleDES.

To właśnie JCE zawiera definicję interfejsu reprezentującego klucz tajny: **`javax.crypto.SecretKey`**.

Wewnątrz samego JCE znajdują się implementacje wielu rodzajów kluczy tajnych (dla różnych algorytmów symetrycznych, algorytmów wyliczających kod uwierzytelniający wiadomości, schematów szyfrowania opartych o hasło): **`Blowfish`**, **`DES`**, **`DESede`**, **`AESHmacMD5`**, **`HmacSHA1`**, **`PBEWithMD5AndDES`**, **`PBEWithMD5AndTripleDES`**.

W odróżnieniu do kluczy asymetrycznych, poszczególne rodzaje kluczy tajnych nie posiadają specyficznych interfejsów i są hermetyzowane przez jeden wspólny interfejs: **`SecretKey`**.

52

Klucze (6)

Za generowanie kluczy asymetrycznych w ramach JCA są odpowiedzialne konkretne klasy rozszerzające **`java.security.KeyPairGenerator`**.
Sam **`KeyPairGenerator`** – jak wszystkie klasy reprezentujące mechanizmy – jest klasą abstrakcyjną skonstruowaną w oparciu o wzorzec projektowy *factory*.

Generowanie pary kluczy jest procesem dość czasochłonnym – zazwyczaj wymagającym generowania przerwań sprzętowych (np. poprzez poruszanie myszką), co gwarantuje rzeczywistą przypadkowość – a co za tym idzie siłę – klucza.

Po uzyskaniu instancji generatora kluczy (za pomocą **`getInstance()`**) możemy przeprowadzić jej inicjalizację, na którą mogą składać się:

- określenie tzw. „siły" (*strength*) generowanego klucza – zazwyczaj tożsamej z długością klucza;
- ustalenie generatora liczb losowych (bądź pseudolosowych), który ma być wykorzystywany w trakcie generowania klucza.

53

Klucze (7)

Poszczególne implementacje określonych algorytmów asymetrycznych powinny określać dopuszczalne wartości dla *strength*.

Przykładowo w przypadku implementacji klucza dla algorytmów DSA i Diffie-Hellman dostarczanych z J2SE dopuszczalne są wielokrotności 64 z przedziału: 512-1024.
Z kolei „siła" klucza RSA może być ustalona na dowolną liczbę z przedziału: 512-2048.

```
try {
   KeyPairGenerator kpg = KeyPairGenerator.getInstance("DSA");
   kpg.initialize(512, new SecureRandom());
   KeyPair pair = kpg.generateKeyPair();
} catch (NoSuchAlgorithmException ex) { ex.printStackTrace(); }
```

Jeśli zostanie przekazana wartość spoza wymaganego zakresu zostanie zgłoszony wyjątek **`InvalidParameterException`**.

Jeśli podczas inicjalizacji generatora kluczy nie zostanie jawnie określona instancja klasy generatora liczb losowych (rozszerzającej **`SecureRandomSpi`**), generator będzie korzystał z domyślnej implementacji (**`SecureRandom`**).

54

Klucze (8)

W zależności od zawartości konfiguracji zapisanej w pliku: **`java.security`** klasa **`SecureRandom`** może być generatorem liczb losowych, bądź pseudolosowych.

Oczywiście przełączenie do trybu pseudolosowego zmniejszy czas generowania kluczy, ale jednocześnie sprawi, że klucze staną się przewidywalne, co oczywiście będzie jednoznaczne ze znaczącym osłabieniem ich siły.

Zarówno na platformach Un*x (Sun Solaris, Linux), jak i na platformie Microsoft Windows ziarno (*seed*) dla **`SecureRandom`** może być pobierane ze źródła zapisanego we właściwości: **`securerandom.source`**.

Wspomniana właściwość zazwyczaj przyjmuje dwie możliwe wartości: **`file:/dev/urandom`** (dla ciągów losowych), bądź **`file:/dev/random`** (dla ciągów pseudolosowych).

Na platformach Un*x **`/dev/urandom`**, oraz **`/dev/random`** są plikami reprezentującymi urządzenia znakowe, których odczyt pozwala uzyskać odpowiednio losowy, bądź pseudolosowy ciąg znaków.

55

Klucze (9)

Na platformie Microsoft Windows, gdzie oba wspomniane pliki nie występują fizycznie w systemie plików (Windows w ogólności nie wspiera dostępu do urządzeń poprzez pliki), umieszczenie **`file:/dev/random`**, bądź **`file:/dev/urandom`** oznacza wykorzystanie funkcji ***Windows CryptoAPI***.

Inicjalizacja generatora pary kluczy nie jest obowiązkowa – teoretycznie, jeśli użytkownik jawnie nie użyje metody **`initialize()`** powinny być użyte wartości domyślne dla danej implementacji.

Niestety w praktyce bywa z tym różnie, bowiem z powodu błędów w konkretnej implementacji może zostać zgłoszony wyjątek **`NullPointerException`** – zatem dobrym zwyczajem jest przeprowadzanie jawnej inicjalizacji generatora.

Abstrakcyjny interfejs dostarczany przez **`KeyPairGenerator`** jest całkowicie wystarczający dla większości przypadków, kiedy zachodzi konieczność wygenerowania pary kluczy.

Czasami jednak programista może chcieć przekazać generatorowi inne niż domyślne wartości specyficznych parametrów klucza.

56

Klucze (10)

Przykładowo w implementacji dostarczanej z J2SE generator pary kluczy dla DSA dla kluczy o długości 512 oraz 1024 używa wstępnie wyliczonych wartości parametrów specyficznych dla klucza DSA (wspomnianych już: P, Q, oraz G).

Korzystanie z wyliczonych wcześniej wartości P, Q oraz G skraca czas niezbędny do wygenerowania klucza DSA.

O ile w konkretnej sytuacji wskazane jest generowanie pary kluczy z innymi – niż domyślne – wartościami parametrów specyficznych dla kluczy, programista powinien starać się używać bardziej specjalizowanych interfejsów generatorów kluczy – o ile oczywiście używana implementacja dopuszcza taką możliwość.

```
/** use DSAKeyPairGenerator if possible */
KeyPairGenerator gen = KeyPairGenerator.getInstance("DSA");
if (gen instanceof DSAKeyPairGenerator) {
   DSAKeyPairGenerator dsaGen = (DSAKeyPairGenerator)gen;
   dsaGen.initialize(512, true, new SecureRandom());
} else {
   gen.initialize(512);
}
```

57

Klucze (11)

Generator kluczy tajnych reprezentuje klasa **`javax.crypto.KeyGenerator`** znajdująca się w JCE rozszerzającym standardową JCA.

Od strony użytkowej funkcjonalność **`KeyGenerator`** jest zbliżona do **`KeyPairGenerator`**, z tą różnicą, że zamiast pary kluczy jest generowany pojedynczy klucz tajny.

Podobnie jak klasy reprezentujące wszystkie dostępne w JCA i JCE mechanizmy **`KeyGenerator`** nie zawiera konstruktorów publicznych – instancja konkretnej klasy jest tworzona za pomocą metody klasowej (**`static`**) **`getInstance()`**.

Dołączane do J2SE 1.4 JCE dostarcza generatory kluczy dla:

- algorytmów szyfrowania symetrycznego: Blowfish, DES, DESede, AES;
- algorytmów wyliczających kod uwierzytelniający wiadomości (MAC): HmacMD5, HmacSHA1.

Instancja podklasy **`KeyGenerator`** – podobnie jak **`KeyPairGenerator`** – powinna być zawsze inicjalizowana po utworzeniu.
Zalecane jest ograniczone zaufanie wobec wartości domyślnie przypisanych w konstruktorach konkretnych klas rozszerzających **`KeyGenerator`**.

58

Klucze (12)

```
int KEY_SIZE = 256; /** size of key (in bits) */
try {
   KeyGenerator keyGen = KeyGenerator.getInstance("AES");
   keyGen.init(KEY_SIZE, new SecureRandom());
   SecretKey key = keyGen.generateKey();
} catch (NoSuchAlgorithmException ex) {
   ex.printStackTrace();
}
```

Podobnie jak w przypadku **`KeyPairGenerator`** przekazanie długości klucza spoza wymaganego zakresu spowoduje zgłoszenie **`InvalidParameterException`**.

```
int KEY_SIZE = 256; /** will cast InvalidParameterException ---
                        key size for DES must be 56 */
try {
   KeyGenerator keyGen = KeyGenerator.getInstance("DES");
   keyGen.init(KEY_SIZE, new SecureRandom());
   SecretKey key = keyGen.generateKey();
} catch (NoSuchAlgorithmException ex) {
   ex.printStackTrace();
}
```

59

Klucze (13)

Korzystanie z **fabryk kluczy (*key factories*)** stanowi alternatywny sposób uzyskiwania kluczy na podstawie tzw. **specyfikacji klucza (*key specifications*)** – w odróżnieniu do poznanych generatorów, które tworzyły klucze zupełnie od zera.

Specyfikacja klucza to reprezentacja w zewnętrznym formacie – zazwyczaj wykorzystywanym do transportu klucza, zatem w rzeczywistości fabryki kluczy w ogóle nie generują klucza, a przekształcają na wewnętrzną reprezentację JCA.

Key factories umożliwiają dwustronną konwersję klucza (czyli import i eksport) z i do zewnętrznej reprezentacji.

Specyfikacje kluczy mogą hermetyzować – w zależności od rodzaju specyfikacji:

- ciąg (tablicę) bajtów zgodny z wymaganym formatem zapisu – gdy mamy do czynienia z klasą rozszerzającą **`EncodedKeySpec`**, np. **`X509EncodedKeySpec`**, **`PKCS8EncodedKeySpec`**;
- parametrów, które zostały użyte do wygenerowania danego klucza – np. **`DSAPublicKeySpec`**, **`DHPrivateKeySpec`**, **`RSAPrivateKeySpec`**.

W celu przekształcenia specyfikacji w wewnętrzną reprezentację klucza można posłużyć się klasą **`KeyFactory`** wspierającą konwersję kluczy asymetrycznych, bądź **`SecretKeyFactory`** konwertującą wyłącznie klucze tajne.

60

Klucze (14)

```
try {
   File f = new File("x509-encoded-public-key.der");
   FileInputStream is = new FileInputStream(f);
   byte[] arr = new byte[f.length()]; is.read(arr);
   KeySpec pubKeySpec = new X509EncodedKeySpec( arr );
   KeyFactory keyFactory = KeyFactory.getInstance("DSA");
   PublicKey pubKey = keyFactory.generatePublic(pubKeySpec);
} catch (Exception ex) { ex.printStackTrace(); }
```

Korzystanie z fabryk kluczy jest de facto jedynym sposobem na wygenerowanie klucza tajnego, dla którego zewnętrzną reprezentacją jest hasło (*password based encryption* – PBE).

```
try {
   String passwd = "ein sehr geheimes passwort";
   KeySpec pbeKeySpec = new PBEKeySpec( passwd.toCharArray() );
   SecretKeyFactory kf
      = SecretKeyFactory.getInstance("PBEWithMD5AndTripleDES");
   SecretKey key = kf.generateSecret(pbeKeySpec);
} catch (Exception ex) { ex.printStackTrace(); }
```

61

Certyfikaty (1)

W ramach J2SE reprezentację ogólnych cech certyfikatu stanowi abstrakcyjna klasa **`java.security.cert.Certificate`**.

Chociaż w ogólności istnieją różne standardy certyfikatów (np. PGP), Java™ standardowo wspiera wyłącznie dominujący format zapisu certyfikatów: X.509v3.

Z myślą o certyfikatach X.509 v.3 zostało opracowane osobne API w postaci abstrakcyjnej klasy **`java.security.cert.X509Certificate`** obejmujące wsparcie dla elementów specyficznych dla X.509, jak np. DN, czy numer seryjny.

Chcąc korzystać z innego formatu zapisu certyfikatu należy we własnym zakresie zaimplementować podklasę **`Certificate`**, bądź skorzystać z implementacji dostawców niezależnych (*independent software vendors* – ISV)

Oparta o wzorzec projektowy *factory* – jak większość klas w szkielecie JCA – klasa **`java.security.cert.CertificateFactory`** hermetyzuje implementacje tzw. **fabryk certyfikatów (*certificate factories*)**.

Oczywiście tworząc konkretną instancję fabryki certyfikatów za pomocą **`getInstance()`** należy przekazać przyjętą w ramach JCA nazwę formatu certyfikatu, bądź synonim zarejestrowany w ramach *security provider*, np. **`"X509"`**.

62

Certyfikaty (2)

Implementacje **`java.security.cert.CertificateFactorySpi`** są odpowiedzialne za tworzenie, a właściwie import certyfikatów – na tej samej zasadzie co klasy implementujące: **`KeyFactorySpi`**, czy **`SecretKeyFactorySpi`**.

Dostarczana standardowo implementacja fabryki certyfikatów umożliwia import certyfikatu ze strumienia wejściowego.

W celu importu certyfikatów ze strumienia programista może użyć jednej dwóch metod udostępnianych w tym zakresie przez klasę **`CertificateFactory`**: **`generateCertificate()`** oraz **`generateCertificates()`**.

Nazwy metod mogą być mylące, ponieważ API dostarczane przez **JCA pozwala jedynie na import (konwersję) – nie generowanie certyfikatów**.

Dostępna standardowo implementacja **`generateCertificate()`** oczekuje na wejściu pojedynczego certyfikatu X.509 zapisanego w kodowaniu DER (RFC 1421).

Z kolei **`generateCertificates()`** poza importem pojedynczego certyfikatu zapisanego w DER umożliwia odczyt łańcucha certyfikatów zapisanego w formacie PKCS#7 (również zakodowanego w DER).

63

Certyfikaty (3)

```
...
try {
   FileInputStream fileStream
      = new FileInputStream("certificate.cer");
   CertificateFactory cf
      = CertificateFactory.getInstance("X509");
   X509Certificate cert
      = (X509Certificate)cf.generateCertificate(fileStream);
   System.out.println("DN: " + cert.getSubjectDN() );
   System.out.println("Issuer DN: " + cert.getIssuerDN() );
   System.out.println("Valid from: " + cert.getNotBefore()
      + " to " + cert.getNotAfter() );
   System.out.println("Serial number: "
      + cert.getSerialNumber() );
   System.out.println("Signature algorithm: "
      + cert.getSigAlgName() );
} catch (Exception ex) {
   e.printStackTrace();
}
...
```

64

Certyfikaty (4)

```
/** exemplary output generated by the code snippet presented
    on the previous slide */
DN: CN = CC Signet - PCA Klasa 2, OU = Centrum Certyfikacji
Signet, O = TP Internet Sp. z o.o., C = PL
Issuer DN: CN = CC Signet – RootCA, OU = Centrum Certyfikacji
Signet, O = TP Internet Sp. z o.o., C = PL
Valid from: 18 kwietnia 2002 16:54:08 to 21 września 2026
17:42:19
Serial number: 3cbede10
Signature algorithm: SHA1WithRSA
```

Jak pamiętamy CA może w dowolnej chwili unieważnić wystawiony przez siebie certyfikat – np. w momencie ujawnienia klucza prywatnego certyfikatu, lub wykorzystania certyfikatu dla innych celów niż został on wystawiony.

Data ważności stanowiąca wymagany element każdego certyfikatu X.509 nie stanowi wystarczającego zabezpieczenia przed niewłaściwym wykorzystaniem wycofanego certyfikatu, który musi zostać umieszczony na *certificate revocation list*.

65

Certyfikaty (5)

Bezpieczna aplikacja powinna sprawdzać, czy okazywany jej certyfikat nie figuruje na CRL wystawiającego urzędu certyfikacji – tego rodzaju **weryfikacja powinna odbywać się przed każdą akceptacją certyfikatu**.

W ramach JCA **zostało opracowane spójne semantycznie API do przeprowadzania tego rodzaju weryfikacji**.
Standardowo w J2SE jest dostępne wsparcie CRL wyłącznie dla certyfikatów X.509 – choć oczywiście pojęcie CRL nie dotyczy wyłącznie certyfikatów zapisanych w tym formacie.

Niestety również w samym obszarze certyfikatów X.509 nie wypracowano jednego standardu publikowania, a istniejące sposoby publikacji podlegają ciągłym zmianom.

Z wymienionych powodów wsparcie dla CRL dostępne w chwili obecnej w Java™ nie umożliwia pełnej współpracy z większością urzędów certyfikacji.

Najprawdopodobniej z tego powodu dostarczane standardowo z J2SE implementacje metody **`Certificate.validate()` odpowiedzialnej za weryfikację certyfikatu domyślnie nie komunikują się z żadnym CRL**.

66

Certyfikaty (6)

Z tego względu weryfikacja, czy certyfikat znajduje się na CRL musi być jawnie wykonywana przez programistę aplikacji.

Unieważnione certyfikaty X.509 są w ramach JCA reprezentowane jako wystąpienia **`java.security.cert.X509CRLEntry`**, natomiast sama CRL grupująca wycofane certyfikaty jako instancja klasy **`java.security.cert.X509CRL`**.

`X509CRL` jest w ramach JCA specyficznym rodzajem **`CertificateFactory`**, które dla **`X509CRLEntry`** pełni tę samą rolę, co **`CertificateFactory`** dla **`Certificate`**.

W ogólności mechanizmy korzystania z fabryk certyfikatów oraz fabryk CRL są oparte na tej samej koncepcji.

W celu importu CRL należy w pierwszej kolejności uzyskać dostęp do strumienia zawierającego CRL zapisanego w formacie wspieranym przez zainstalowane *security providers*.

W następnym kroku należy wyszukać odpowiednią implementację **`CertificateFactory`** reprezentującą CRL (np. **`"X509CRL"`**), by na samym końcu móc dokonać właściwego importu CRL za pomocą metody **`generateCRL()`**.

67

Certyfikaty (7)

```
InputStream getCRLStream (Principal p) { ... }
PublicKey getIssuerPublicKey (Principal p) { ... }
...
try {
   InputStream certStream = new FileInputStream("cert.cer");
   CertificateFactory cf
      = CertificateFactory.getInstance("X509");
   X509Certificate cert
      = (X509Certificate)cf.generateCertificate(stream);
   Principal p = cert.getIssuserDN();
   PublicKey pubKey = getIssuerPublicKey(p);
   cert.verify(pubKey); /** does not verify CRL */
   InputStream crlStream = getCRLStream(p);
   cf = CertificateFactory.getInstance("X509CRL");
   X509CRL crl = (X509CRL) cf.generateCRL(crlStream);
   if (crl.isRevoked(cert))
      throw new CertificateException( "Certificate revoked" );
} catch (Exception ex) { e.printStackTrace(); }
...
```

68

Keystores (1)

Zarządzanie kluczami i certyfikatami w ogólności polega na ich przechowywaniu, a także udostępnieniu możliwości ich pobierania za pomocą własnych aplikacji, jak również narzędzi stworzonych przez niezależnych dostawców.

Poza opisanym zasadniczym celem rozwiązanie zarządzające kluczami i certyfikatami może oczywiście dostarczać użytkownikowi dodatkowej wiedzy jak np. informację o stopniu zaufania dla certyfikatu wystawionego przez dane CA.

W ramach JCA zarządzanie kluczami i certyfikatami zostało oparte o tzw. **magazyny kluczy (*keystores*)**.

Dostarczana z J2SE implementacja magazynu kluczy jest oparta o plik zapisany w ściśle określonym formacie (JKS, PKCS#12, bądź JCEKS).
JCA nie zakłada istnienia domyślnej lokalizacji *keystore*, choć udostępniane razem z J2SE narzędzia, takie jak `keytool`, czy `jarsigner` przyjmują domyślnie, że miejscem przechowywania magazynu kluczy jest plik `$HOME/.keystore`.

Zapis magazynu kluczy w postaci pojedynczego pliku sprawia, że standardowa implementacja udostępniana przez Sun Microsystems® razem z J2SE raczej **nie nadaje się do innych zastosowań niż jednoużytkownikowe**.

Keystores (2)

Każdy klucz, bądź certyfikat przechowywany wewnątrz danego *keystore* należy do jakiegoś podmiotu.

Tzw. **alias (synonim)** jest skróconą nazwą danego podmiotu obowiązującą w ramach danego *keystore*.

Alias jest w rzeczywistości kluczem (w rozumieniu tablicy haszującej) pod którym umieszczono klucz, bądź certyfikat umożliwiającym jednoznaczną identyfikację wpisu (klucza, bądź certyfikatu) w ramach danego *keystore*.

W odróżnieniu do zapisanego w certyfikacie DN jednoznacznie identyfikującego podmiot globalnie, **alias jest bytem lokalnym dla konkretnego *keystore***.

Magazyn kluczy może zawierać dwa rodzaje pozycji:

Pierwszym rodzajem jest tzw. **pozycja klucza (*key entry*)** umożliwiająca zapis pary: klucz prywatny i certyfikat klucza publicznego lub pojedynczego klucza tajnego.

W miejscu certyfikatu może być przechowywany tzw. łańcuch certyfikatów – wówczas pierwszy w łańcuchu certyfikatów zawiera klucz publiczny podmiotu. Pozostałe certyfikaty zapisane w łańcuchu określają tzw. **ścieżkę zaufania (*cert path*)** – aż do certyfikatu *root CA*.

Keystores (3)

Pozycja certyfikatu (*certificate entry*) stanowiąca drugi rodzaj pozycji w ramach *keystore* może zawierać co najwyżej łańcuch certyfikatów klucza publicznego – bez możliwości związania z nim klucza prywatnego.

W pliku `$JREHOME/lib/cacerts` znajduje magazyn kluczy zawierający zestaw zaufanych certyfikatów *root CA* standardowo dostarczanych razem z J2SE. Początkowym hasłem dostępu do tego magazynu kluczy jest: `"changeit"`.

Dostawcy implementacji J2SE rozwiązali zatem problem inicjalizacji początkowej ścieżki zaufania na podobnej zasadzie jak dzieje się to w przypadku przeglądarek internetowych – dołączając certyfikaty *root CA* do produktu.

Należy zauważyć, że choć jest to rozwiązanie sprawdzające się w praktyce, nie jest ono do końca szczelne – istnieje możliwość zmiany zawartości pakietu dystrybucyjnego jeśli jest on przesyłany kanałem nie zapewniającym integralności.

Wszystkie certyfikaty *root CA* przechowywane w pliku `cacerts` są zapisane jako pozycje certyfikatów w ramach magazynu kluczy i jest to jednocześnie najczęstsze zastosowanie pozycji certyfikatów.

71

Keystores (4)

Standardowo z J2SE dostarczają trzy różne formaty magazynowania kluczy:

Pierwszym i zarazem domyślnym formatem zapisu magazynu kluczy dostarczanym standardowo z J2SE jest ***Java Key Store* – JKS**.

JKS umożliwia przechowywanie zarówno pozycji kluczy, jak i pozycji certyfikatów – jednak w pozycjach kluczy można przechowywać co najwyżej klucze asymetryczne (pary: klucz prywatny, certyfikat klucza publicznego).

Chcąc przechowywać w *keystore* klucze tajne należy skorzystać z drugiego standardowo dostępnego formatu zapisu – **JCEKS**, którego implementacja stanowi część JCE.

Zmiana domyślnego algorytmu zapisu *keystore* wymaga zmiany odpowiedniej właściwości w pliku `java.security`.

```
keystore.type=JCEKS
```

Ostatnim dostępnym formatem *keystore* wspieranym standardowo przez J2SE jest **PKCS#12**, który w przeciwieństwie do dwóch poprzednich nie daje możliwości pełnego zarządzania kluczami.

72

Keystores (5)

Głównym przeznaczeniem PKCS#12 jest możliwość wymiany magazynów kluczy z przeglądarkami internetowymi Netscape, czy Mozilla.

Ze standardową dystrybucją J2SE jest dostarczany również **`keytool`** – narzędzie udostępniające interfejs z linii poleceń pozwalające na wykonywanie typowych operacji administracyjnych: tworzenie kluczy, import i eksport certyfikatów, itp.

Ważniejsze opcje globalne `keytool` – dostępne we wszystkich trybach pracy:

`-alias <alias>` – określa alias, którego będzie dotyczyć operacja;
`-dname <DN>` – określa DN, np. **`-dname "CN = Certum CA, O = Unizeto Sp. z o.o., ..."`**;
`-storepass <keystore-password>` – hasło do *keystore*;
`-storetype <storetype>` – przesłania domyśny rodzaj magazynu kluczy z konfiguracji **`java.security`**, np. **`-storetype jceks`**
`-keystore <filename>` – plik, w którym zapisano magazyn kluczy;
`-keypass <key-password>` – hasło wykorzystywane do zabezpieczenia pozycji (klucza prywatnego, bądź tajnego) – oczywiście hasła zabezpieczające różne klucze powinny również być różne.

Keystores (6)

Uruchomienie **`keytool`** z opcją **`-genkey`** spowoduje utworzenie nowej pozycji klucza zawierającej parę: klucz prywatny oraz podpisany tym kluczem prywatnym (*self-signed*) certyfikat zawierający odpowiadający klucz publiczny.

Ważniejsze przełączniki dostępne w trybie `-genkey`:

`-keyalg <algorithm-name>` – ustala algorytm dla którego ma być wygenerowania pary kluczy – domyślnie DSA – oczywiście należy użyć w tym miejscu nazwę algorytmu używaną przez zainstalowany *security provider*;
`-keysize <size>` – określa długości klucza (standardowo 1024);
`-sigalg <signature-algorithm>` – określa algorytm, który powinien być wykorzystany do podpisania *self-signed certificate* (domyślnie **`SHA1WithDSA`**) – oczywiście to również musi być nazwa wspierana przez *security provider*;
`-validity` – określa ważność certyfikatu (domyślnie 90 dni).

Certyfikat wygenerowany w trybie **`-genkey`** jest oczywiście w pełni funkcjonalnym certyfikatem X.509 – niestety o znikomej użyteczności, ze względu na fakt, że **nie został podpisany przez powszechnie akceptowane CA**.

Chcąc, by wygenerowany klucz publiczny zapisany w certyfikacie był powszechnie akceptowany musimy uzyskać podpis znanego CA – np. jednego z tych, których certyfikaty umieszczono w **`cacerts`**.

Keystores (7)

Pierwszym krokiem jaki musimy podjąć chcąc zdobyć podpis znanego CA jest wygenerowanie **żądania podpisu certyfikatu (*certificate signing request* – CSR)**.

Certificate request to nic innego jak **DN oraz klucz publiczny certyfikatu zapisanego pod danym aliasem podpisane odpowiadającym kluczem prywatnym**.

`keytool` umożliwia nam wygenerowanie CSR za pomocą opcji **`-certreq`** – oczywiście w oparciu o wcześniej utworzoną pozycją klucza.

W wyniku uruchomienia keytool z opcją **`-certreq`** do pliku o nazwie podanej po opcji **`-file`** zostanie zapisany CSR w formacie PKCS#10 – domyślnie wynik działania **`-certreq`** jest wyprowadzany na konsolę.

W następnej kolejności należy przekazać CSR do CA, które zweryfikuje:
- podpis złożony na CSR, oraz
- tożsamość podmiotu występującego o certyfikat.

Jeśli obie te czynności zakończą się pozytywnie dla ubiegającego się o certyfikat, biuro certyfikacji wystawi certyfikat potwierdzający ważność klucza publicznego.

75

Keystores (8)

Import certyfikatu podpisanego przez znane CA możemy wykonać za pomocą **`keytool`** uruchomionego w trybie **`-import`**.

Jako plik wsadowy (opcja **`-file`** – domyślnie standardowe wejście) powinniśmy przekazać do **`keytool`** łańcuch certyfikatów zapisany zgodnie ze standardem PKCS#7 zakodowany w DER (RFC 1421).

Pierwszy w „łańcuchu" certyfikatów będzie ten, który zostanie skojarzony z danym aliasem, następne w kolejności będą: certyfikat CA, które bezpośrednio złożyło swój podpis na CSR, oraz certyfikaty kolejnych CA w łańcuchu zaufania – aż do *root CA*.

Importowany certyfikat może być podpisany przez CA nieznane JRE – wówczas **`keytool`** poinformuje o tym fakcie użytkownika oraz poprosi o potwierdzenie chęci importu łańcucha certyfikatów.

Opcja **`-trustcacerts`** sprawi, że **`keytool`** zweryfikuje, czy na samym końcu łańcucha znajduje się jeden z certyfikatów z pliku **`cacerts`** – jeśli tak – użytkownik nie będzie musiał potwierdzać importu.
Bez podania **`-truscacerts`** żaden certyfikat CA nie będzie traktowany jako zaufany – również taki, który na końcu łańcucha ma certyfikat z **`cacerts`**.

76

Keystores (9)

Certyfikat zostanie zaimportowany o ile jest ważny – czyli (1) posiada ważny podpis oraz (2) klucz publiczny w certyfikacie odpowiada kluczowi prywatnemu zapisanemu pod danym aliasem.

W efekcie opisanych czynności w pozycji klucza będzie umieszczona będzie para: klucz prywatny oraz podpisany przez zaufane CA certyfikat zawierający klucz publiczny odpowiadający kluczowi prywatnemu.

Tworzenie pozycji certyfikatu jest realizowane za pomocą importu istniejących certyfikatów – bez wcześniejszego generowania pary kluczy, certyfikatu opakowującego klucz publiczny i CSR.

Certyfikat importowany do pozycji certyfikatu może być certyfikatem *root CA*, lub dowolnym certyfikatem jednostki, z którą wymieniamy informację.

Z oczywistych względów nie ma żadnej możliwości modyfikacji pozycji certyfikatu w ramach *keystore* – dany magazyn certyfikatów na żadnym możliwym etapie nie uczestniczył w procesie wystawiania certyfikatu.

Wszelkie modyfikacje pozycji certyfikatu wymagają usunięcia certyfikatu skojarzonego z danym aliasem i wstawienie w tym miejscu innego certyfikatu.

77

Keystores (10)

Tryb **`-selcert`** udostępniany przez **`keytool`** dopuszcza pewne ograniczone modyfikacje pozycji klucza – bez możliwości zmiany samej pary kluczy.

Dopuszczalne modyfikacje mogą polegać na podstawieniu samopodpisanego (*self-signed*) certyfikatu stanowiącego podstawę dla CSR w miejsce łańcucha certyfikatów (zakończonego certyfikatem *root CA*).
Tego rodzaju ingerencji może towarzyszyć również zmiana DN.

Do ważniejszych opcji `keytool` możemy zaliczyć jeszcze:

`-list` – umożliwia wyświetlenie jednej, bądź więcej pozycji *keystore*;
`-export` – eksportuje wybrany certyfikat zapisany pod danym aliasem do formatu PKCS#7 w kodowaniu RFC 1421;
`-printcert` – wyświetla zawartość certyfikatu zapisanego w RFC 1421 – opcja ta nie korzysta w ogóle z *keystore* – jedynie wyświetla zawartość podanego certyfikatu.

Tryby **`-list`** oraz **`-export`** są zasadniczo jedynymi udostępnianymi przez **`keytool`** dla *keystores* zapisanych w formacie PKCS#12.

78

Keystores (11)

Dostęp do *keystores* jest możliwy nie tylko za pośrednictwem narzędzia **`keytool`**, ale także z poziomu API udostępnianego przez JCA pozwalającego zarówno na odczyt, jak i modyfikację zawartości magazynów kluczy.

Sam **`keytool`** posiada pewne ograniczenia polegające m.in. na braku możliwości dodawania do *keystore* kluczy tajnych.

Funkcjonalność magazynu kluczy – czyli zbiór aliasów oraz przypisanych im kluczy prywatnych i certyfikatów – jest w ramach JCA reprezentowana przez klasę **`java.security.KeyStore`**.

Instancja **`KeyStore`** jest zapisaną w pamięci operacyjnej reprezentacją informacji zapisanej w magazynie kluczy utrwalonym w na dysku.

Podobnie jak wszystkie klasy w ramach szkieletu JCA **`KeyStore`** stanowi realizację wzorca projektowego *factory*.

Do metody **`getInstance()`** zwracającej wystąpienie **`KeyStore`**, które w przyszłości chcielibyśmy skojarzyć z konkretnym magazynem kluczy jako argument należy przekazać format zapisu tego wybranego magazynu, np. **`"JCEKS"`**.

79

Keystores (12)

Samo wywołanie metody **`getInstance()`** tworzy pusty magazyn, który za pomocą metody **`load()`** należy wypełnić danymi zapisanymi w pliku przechowującym *keystore*.

W przypadku formatów zapisu *keystore* wspieranych standardowo przez J2SE integralność danych zapisanych w magazynie jest oparta o MAC – do właściwych danych jest dodawane hasło, a następnie dla całości jest wyliczany skrót dopisywany na koniec pliku *keystore*.

Jeśli w trakcie ładowania przy użyciu **`load()`** w miejsce hasła zostanie przekazana wartość **`null`**, dane zapisane w *keystore* zostaną umieszczone w pamięci operacyjnej bez uprzedniej weryfikacji integralności.

Oczywiście załadowanie magazynu kluczy do pamięci nie oznacza wcale, że użytkownik będzie w stanie odczytać chronione hasłami klucze prywatne znajdujące się w tym *keystore*.

Do utrwalania magazynów kluczy w pliku (w ogólności serializacji *keystore* do strumienia wejściowego) służy metoda **`store()`**, z kolei metoda **`getDefaultType()`** pozwala ustalić domyślny format zapisu *keystore* na podstawie wpisu w pliku konfiguracyjnym **`java.security`**.

80

Keystores (13)

```
KeyStore loadKeyStore (String ksFile, String pwd) {
   try {
      KeyStore ks = KeyStore.getInstance(
         KeyStore.getDefaultType());
      InputStream stream = new FileInputStream( ksFile );
      /** verify the integrity of a keystore */
      ks.load( stream, pwd.toCharArray() );
   } catch (Exception ex) {
      ex.printStackTrace();
   }
}
void storeKeyStore (KeyStore ks, String ksFile, String pwd) {
   try {
      OutputStream stream = new FileOutputStream(ksFile);
      ks.store( stream, pwd.toCharArray() );
   } catch (Exception ex) {
       ex.printStackTrace();
   }
}
/** to be continued on the next slide */
```

81

Keystores (14)

```
void printKeyEntry (KeyStore ks, String alias, String pwd) {
   System.out.println( "Private key: "
      + ks.getKey(alias,pwd.toCharArray() );
   Certificate certs[] = ks.getCertificateChain(alias);
   for (int i = 0; i < certs.length; i++) {
      if (c intanceof X509Certificate) {
         X509Certificate c = (X509Certificate) certs[i];
         System.out.println( "Subject DN: " + c.getSubjectDN()
            + ", Issuer DN: " + c.getIssuerDN() );
      }
   }
}
void printCertificateEntry (KeyStore ks, String alias) {
   Certificate c = ks.getCertificate(alias);
   if (c instanceof X509Certificate) {
      X509Certificate x509 = (X509Certificate) c;
      System.out.println( "Subject DN: " + x509.getSubjectDN()
         "Issuer DN: " + x509.getIssuerDN() );
   }
} /** to be continued on the next slide */
```

82

Keystores (15)

```
/** the continuation from the previous slides */

public static void main (String[] args) {
   String ksFile = System.getProperty("user.home")
      + File.separator + ".keystore";
   KeyStore ks = loadKeyStore( ksFile, "secret password" );
   String alias = "edek";
   if ( ks.isKeyEntry(alias) ) {
      printKeyEntry( ks,alias,
         "my very secret password protecting private key" );
   } else if ( ks.isCertificateEntry(alias) ) {
      printCertificateEntry( ks,alias );
   } else {
      System.out.println(
         "There is no entry for the alias: \"" + alias
         + "\" in the keystore" );
   }
}
```


Skróty wiadomości (1)

Reprezentowany przez klasę **`java.security.MessageDigest`** skrót wiadomości jest najprostszym – zarówno pod względem koncepcji, jak i użycia - spośród standardowo dostępnych mechanizmów kryptograficznych (*engines*).

Jak wiemy, wygenerowanie skrótu stanowi pierwszy etap zarówno tworzenia, jak również weryfikacji podpisów cyfrowych.

Oczywiście sam skrót wiadomości również umożliwia weryfikację, czy wiadomość nie została zmodyfikowana w czasie transportu – jednakże **pod warunkiem, że zapewniona jest integralność skrótu**.

Chcąc wyliczyć skrót wiadomości musimy w pierwszej kolejności uzyskać implementację **`MessageDigestSpi`** za pomocą metody **`getInstance()`**.

Następnie przez implementację pary: mechanizm/algorytm skrótu musimy przepuścić całą treść wiadomości korzystając z metody **`update()`**, której wywołanie odpowiednio **aktualizuje stan wewnętrznego akumulatora**.

Przekazywany w postaci tablicy bajtów skrót jest wyliczany przez metodę **`digest()` w oparciu o bieżący stan akumulatora**.

Skróty wiadomości (2)

```
try {
   FileOutputStream dgstFile =
      new FileOutputStream("digest");
   MessageDigest md = MessageDigest.getInstance("SHA");
   ObjectOutputStream output
      = new ObjectOutputStream(dgstFile);
   FileInputStream inputFile = new FileInputStream("message”);
   /** The size of the buffer and number of update() calls
     * do not influence the final result, of course */
   byte[] buffer = byte[1024];
   while (inputFile.read(buffer) > -1) {
      md.update(buffer);
   }
   output.writeObject(md.digest());
} catch (Exception ex) {
   ex.printStackTrace();
}
```

Wyliczenie skrótu powoduje każdorazowo wyzerowanie stanu akumulatora, dzięki czemu instancja klasy implementującej skrót wiadomości może być ponownie wykorzystywany w kolejnych obliczeniach.

85

Skróty wiadomości (3)

```
try {
   FileInputStream dgstFile = new FileInputStream("digest");
   ObjectInputStream input = new ObjectInputStream(inputFile);
   byte[] digest = (byte[])input.readObject();
   MessageDigest md = MessageDigest.getInstance("SHA");
   FileInputStream msgFile = new FileInputStream("message");
   byte[] buffer = new bytes[1024];
   int result;
   while ((result = msgFile.read(buffer)) > -1) {
      md.update(buffer,0,result);
   }
   if ( MessageDigest.isEqual(md.digest(),digest) ) {
      System.out.println(
         "Message has not been modified during transfer" );
   }
} catch (Exception ex) {
   ex.printStackTrace();
}
```

W celu porównywania skrótów programista może posłużyć się klasową metodą **`isEqual()`** udostępnianą przez **`MessageDigest`**.

86

Skróty wiadomości (4)

Jak wiemy, sam skrót nie zapewnia nam integralności wiadomości – nic bowiem nie zabezpiecza nas przed sytuacją, w której osoba postronna zmodyfikuje skrót dostosowując go w ten sposób do zmienionej treści wiadomości.

Kod uwierzytelnienia wiadomości (*message authentication code* – MAC) nie jest generowany wyłącznie w oparciu o samą treść wiadomości, ale także klucz, który strony komunikacji muszą utrzymać w tajemnicy.

Należąca do JCE klasa **`javax.crypto.Mac`** stanowi reprezentację mechanizmu kodu uwierzytelnienia wiadomości.

W ramach samego JCE są dostępne dwa algorytmy wyliczania MAC: **`HmacSHA1`**, **`HmacMD5`**.

Oczywiście korzystanie w MAC oznacza, że strony komunikacji muszą w pierwszej kolejności ustalić klucz tajny – np. stosując wymianę w oparciu o algorytm Diffie-Hellman.

W interfejsie **`Mac`** rolę **`digest()`** znanej z **`MessageDigest`** pełni metoda **`doFinal()`** – przed rozpoczęciem aktualizacji wewnętrznego akumulatora implementacja **`Mac`** musi być za pomocą metody **`init()`** zainicjalizowana kluczem tajnym.

87

Skróty wiadomości (5)

```
try {
   KeyStore ks = KeyStore.getInstance(
      KeyStore.getDefaultType());
   InputStream stream = new FileInputStream( ksFile );
   ks.load( stream, "keystore password".toCharArray() );
   Mac mac = Mac.getInstance("HmacSHA1");
   mac.init( (SecretKey)ks.getKey(
      "alias", "password".toCharArray() );
   FileInputStream msgFile = new FileInputStream("message");
   byte[] buffer = new byte[1024];
   int result;
   while ((result = msgFile.read(buffer)) > -1) {
      mac.update(buffer,0,result);
   }
   FileOutputStream macFile = new FileInputStream("mac");
   ObjectOutputStream output = new ObjectOutputFile(macFile);
   output.writeObject(mac.doFinal());
} catch (Exception ex) {
   ex.printStackTrace();
}
```

88

Skróty wiadomości (6)

MAC może być również wyliczony bezpośrednio – poniżej wyliczanie MAC w oparciu o hasło współdzielone między nadawcą i odbiorcą – prostsze we wdrożeniu od rozwiązania wykorzystującego klucz tajny.

```
try {
   MessageDigest md = MessageDigest.getInstance("SHA");
   String msg = "Message body";
   String passphrase = "Secret passphrase";
   byte[] msgBytes = msg.getBytes(),
          passBytes = passphrase.getBytes();
   md.update(passBytes);
   md.update(dataBytes);
   /** digest for message supplemented with passphrase */
   byte[] digest = md.digest();
   md.update(passBytes);
   md.update(digest);
   /** Message Authentication Code */
   byte[] mac = md.digest();
} catch (Exception ex) {
   ex.printStackTrace();
}
```


Skróty wiadomości (7)

Oczywiście bezpieczeństwo integralności wiadomości jest w przedstawionym przykładzie zależne od siły hasła.

Trudne do złamania hasło – czyli poddające się jedynie *brute force attack* – to zupełnie przypadkowy długi ciąg bajtów wykorzystujący możliwie duży zbiór wartości – choć raczej nie stosuje się znaków innych niż drukowalne.

Interfejs **`MessageDigestSpi`** zarówno do reprezentacji danych, jak i samego skrótu wiadomości wykorzystuje tablicę bajtów, co nie jest oczywiście najwygodniejszą formatem zapisu jeśli korzystamy ze strumieni danych.

W tego rodzaju przypadkach (czyli de facto najczęstszych) należy raczej stosować strumienie skrótów wiadomości (*message digest streams*) – rodzaje filtrów strumieni danych.

Klasy **`DigestInputStream`** oraz **`DigestOutputStream`** reprezentujące odpowiednio: wejściowy i wyjściowy strumień skrótu wiadomości aktualizują stan akumulatora (wywołując **`update()`** wykorzystywanej implementacji skrótu) w miarę jak dane ze strumienia przechodzą przez ich wewnętrzny bufor.

Skróty wiadomości (8)

Ponieważ klasa **`Mac`** nie rozszerza **`MessageDigest`**, nie jesteśmy za pomocą standardowo dostępnych mechanizmów kryptograficznych wyliczyć MAC dla strumienia danych.

Oczywiście stosunkowo niewielkim wysiłkiem możemy stworzyć własną implementacje **`MacInputStream`** oraz **`MacOutputStream`**.

```
try { /** example of usage of DigestOutputStream */
   FileOutputStream outFile
      = new FileOutputStream("message-with-digest");
   MessageDigest md = MessageDigest.getInstance("SHA");
   DigestOutputStream digestOutput
      = new DigestOutputStream(outputFile,md);
   ObjectOutputStream output
      = new ObjectOutputStream(digestOutput);
   String msg = "Message body";
   output.writeObject(msg);
   digestOutput.on(false); /** switch off the digest update */
   output.writeObject(md.digest());
} catch (Exception ex) { ex.printStackTrace(); }
```

91

Skróty wiadomości (9)

Metoda **`on()`** służy do włączania bądź wyłączania aktualizacji wewnętrznego akumulatora (domyślnie aktualizacja jest włączona).

W przedstawionym poniżej przykładzie nie jest konieczne wyłączenie aktualizacji przed wyliczeniem skrótu – jest dobrą praktyką programistyczną, by była on włączona jedynie w czasie odczytu, bądź zapisu do strumienia.

```
try { /** example of DigestInputStream usage */
   FileInputStream f = new FileIntputStream("msg-with-digest");
   MessageDigest md = MessageDigest.getInstance("SHA");
   DigestInputStream dgstInput = new DigestInputStream(f,md);
   ObjectInputStream input = new ObjectInputStream(dgstInput);
   String msg = (String) input.readObject();
   digestInput.on(false);
   byte[] digest = (byte[]) input.readObject();
   if (MessageDigest.isEqual(md.digest(), digest)) {
      System.out.println( "Has not been modified" );
   }
} catch (Exception ex) {
   ex.printStackTrace();
}
```

92

Podpisy cyfrowe (1)

Wszystkie operacje wykorzystywane do składania oraz weryfikacji podpisów cyfrowych są zostały uwzględnione w interfejsie klasy **`java.security.Signature`** stanowiącej implementację **`SignatureSpi`**.

W ramach JCA standardowo są dostarczane implementacje **`SignatureSpi`** w oparciu o RSA oraz DSA.

Chcąc złożyć podpis cyfrowy należy w kodzie aplikacji wykonać następujące czynności:

1. Uzyskać klucz prywatny, który będzie wykorzystywany do podpisu.
2. Uzyskać za pomocą **`getInstance()`** konkretną implementację podpisu cyfrowego oraz zainicjalizować ją przekazując do **`initSign()`** klucz prywatny podmiotu składającego podpis.
3. Wyliczyć za pomocą metody **`update()`** skrót podpisywanej wiadomości poprzez aktualizację wewnętrznego akumulatora implementacji podpisu pomocą metody – w razie potrzeby **`update()`** należy wywoływać wielokrotnie.
4. Wygenerować właściwy podpis za pomocą metody **`sign()`** – odpowiednik **`digest()`** z **`MessageDigest`** i **`doFinal()`** z **`Mac`**.

93

Podpisy cyfrowe (2)

Generowanie podpisu cyfrowego jest bardzo zbliżone do generowania skrótu wiadomości lub MAC – jedyna różnica polega na konieczności dostarczenia klucza prywatnego podmiotu składającego swój podpis.

Wykorzystanie kryptografii klucza publicznego do zapewnienia integralności podpisu sprawia, że podpis cyfrowy jest rozwiązaniem o wiele bardziej atrakcyjnym (czyt. lepszym niż MAC).

Weryfikacja podpisu również składa się z czterech kroków – symetrycznych do tych wykonywanych podczas składania podpisu.

Zasadnicza różnica polega na tym, że tak jak metoda **`initSign()`** oczekiwała klucza prywatnego, tak metoda **`initVerify()`** wymaga przekazania klucza publicznego lub certyfikatu.
Z kolei sam proces weryfikacji jest realizowany przez metodę **`verify()`**.

94

Podpisy cyfrowe (3)

```
/** class KeyStoreSupport which will be further used for
  * getting the keystore from file */
class KeyStoreSupport {
   /** In order to load a keystore from file a user must pass
     * the name of the file which contains the keystore
     * and password which allows to verify keystore
     * consistency */
   static KeyStore loadKeyStore (String ksFile, String passwd)
   {
      try {
         KeyStore ks = KeyStore.getInstance(
            KeyStore.getDefaultType());
         InputStream stream = new FileInputStream( ksFile );
         ks.load( stream, password.toCharArray() );
      } catch (Exception ex) {
         ex.printStackTrace();
      }
      return ks;
   }
}
```

95

Podpisy cyfrowe (4)

```
/** generating a digital signature */
try {
   KeyStore ks = KeyStoreSupport.loadKeyStore(ksFile,"passwd");
   PrivateKey privKey = (PrivateKey) ks.getKey(
      "alias", "password".toCharArray() );
   FileOutputStream sFile = new FileOutputStream("signature");
   ObjectOutputStream out = new ObjectOutputStream(sFile);
   FileInputStream mgsFile = new FileInputStream("message");
   /** signature algorithm must correspond to the key obtained
     * from the keystore */
   Signature s = Signature.getInstance("SHA1WithDSA");
   s.initSign(privKey);
   byte[] buffer = new byte[1024]; int result;
   while ((result = msgFile.read(buffer)) > -1) {
      s.update(buffer,0,result);
   }
   out.writeObject(s.sign());
} catch (Exception ex) {
   ex.printStackTrace();
}
```

96

Podpisy cyfrowe (5)

```
/** verifying a digital signature */
try {
   KeyStore ks = KeyStoreSupport.loadKeyStore(ksFile,"passwd");
   Certificate cert = ks.getCertificate("alias");
   PublicKey pubKey = cert.getPublicKey();
   /** signature algorithm must correspond to the key obtained
     * from the keystore */
   Signature s = Signature.getInstance("SHA1WithDSA");
   s.initVerify(pubKey);
   FileInputStream sFile = new FileInputStream("signature");
   ObjectInputStream input = new ObjectInputStream(sFile);
   byte[] signature = (byte[]) input.readObject();
   FileInputStream mgsFile = new FileInputStream("message");
   byte[] buffer = new byte[1024]; int result;
   while ((result = msgFile.read(buffer)) > -1)
      s.update(buffer,0,result);
   if (s.verify(signature))
      System.out.println("Message has not been modified");
} catch (Exception ex) { ex.printStackTrace(); }
```


Podpisy cyfrowe (6)

W dotychczas prezentowanych przykładach konieczne było określenie w kodzie podmiotu, który złożył podpis – plik podpisu zawierał surowe dane, zatem nie było możliwości określenia podmiotu podpisującego.

W zastosowaniach praktycznych raczej nie zaszywa się w kodzie nazwy podmiotu, który złożył podpis na danych, bo to stanowczo zmniejszyłoby elastyczność kodu.

Można co prawda zrzucić odpowiedzialność za wyszukiwanie kluczy na użytkownika umożliwiając przekazywanie parametrów do aplikacji, jednakże wiązałoby się to z dużą niewygodą w zakresie użytkowania – użytkownik:

- musiałby za każdym razem wiedzieć, kto jest nadawcą wiadomości,
- a także pamiętać czyje certyfikaty są zapisane w lokalnym magazynie kluczy.

W ramach JCA wprowadzono specjalną nierozszerzalną (**`final`**) klasę **`java.security.SignedObject`**, która hermetyzuje dane wraz z towarzyszącym im podpisem cyfrowym.

Oczywiście **`SignedObject`** jest klasą szeregowalną (**`Serializable`**), zatem istnieje możliwość przekazywania instancji tej klasy za pośrednictwem strumieni.

Podpisy cyfrowe (7)

Instancja **`SignedObject`** zawiera tzw. głęboką kopię (*deep copy*) podpisywanego obiektu – dzięki czemu zmiany w obiekcie oryginalnym nie wpływają na podpis.

Przykładowo: jeśli **`SignedObject`** zostanie utworzony w oparciu o bufor bajtów, kopia tego bufora zostanie zapisana w **`SignedObject`** – zmiana zawartości bufora pierwotnego nie wpłynie na zawartość obiektu podpisanego.

Udostępniana przez **`SignedObject`** metoda **`verify()`** umożliwia programiście weryfikację podpisu zapisanego w podpisanym obiekcie w oparciu o przekazane do tej metody: klucz publiczny oraz określoną parę mechanizmu/algorytm podpisu.

Wadą klasy **`SignedObject`** jest fakt, że nie umożliwia ona dołączenia certyfikatu. Ze względu na zachowanie przenośności między implementacjami, jest **`SignedObject`** jest również klasą nierozszerzalną, co uniemożliwia utworzenie jej podklasy zawierającej certyfikat.

Załączanie samego klucza publicznego nie zapewnia integralności zarówno samego klucza, jak i podpisywanej wiadomości.
W stosowanych w praktyce rozwiązaniach podpis cyfrowy obejmuje również certyfikat zawierający klucz publiczny odpowiadający kluczowi prywatnemu, za pomocą którego został złożony podpis.

99

Podpisy cyfrowe (10)

```
class SignedMessageSupport {
   static void verifySigner (Certificate c, String issuerDN) {
      ...
      KeyStore ks = KeyStoreSupport.loadKeyStore("keys","pwd");
      if (ks.getCertificateAlias(c) != null) {
         System.out.println("Contains signer's cert"); return;
      }
      X509Certificate issuerCert = null; /** issuer cert */
      for (String s : ks.aliases()) {
         issuerCert = (X509Certificate) ks.getCertificate(s);
         String certDN = issuerCert.getSubjectDN().getName();
         if (issuerDN.equals(certDN))
            break;
      }
      if (issuerCert == null)
         System.out.println("Issuer not known");
      c.verify(issuer.getPublicKey());
      ...
   }
}  /** to be continued on the next slide */
```

100

Podpisy cyfrowe (11)

```
try { /** verify a SignedMessage obtained from a stream */
   FileInputStream sFile = new FileInputStream("signature");
   ObjectInputStream in = new ObjectInputStream(sFile);
   SignedMessage m = (SignedMessage)in.readObject();
   X509Certificate cert = (X509Certificate) m.cert;
   /** verify if the certificate contained in the signature
     * is still valid */
   cert.checkValidity();
   SignerMessageSupport.verifySigner(
      cert,cert.getIssuerDN().getName());
   PublicKey pk = cert.getPublicKey();
   /** verify the message consistency against signature */
   if (m.obj.verify(pk,Signature.getInstance("SHA1WithRSA")) {
      System.out.println("Message has not been modified");
   } else {
      Sytem.out.println("Signature is invalid");
   }
} catch (Exception ex) {
   ex.printStackTrace();
}
```


Podpisy cyfrowe (12)

Jednym z podstawowych zastosowań podpisów cyfrowych w Java™ jest podpisywanie klas umożliwiające rozszerzenie zestawu uruchamianych klas bez naruszania bezpieczeństwa piaskownicy Java™.

Jak pamiętamy w pliku polityki administrator może zezwolić na uruchamianie klas pochodzących z określonego źródła – podpisanych przez określonego dostawcę (*vendor*).

```
grant signedBy "pjiit", codeBase "http://www.pjwstk.edu.pl/" {
   java.io.FilePermission "-", "read,write";
}
```

Oczywiście, by cały mechanizm mógł zafunkcjonować, musi być dostępna również ładowarka klas, która potrafi zweryfikować podpis (złożony na klasach w *Java™ archive*) oraz przekazać informację o aliasie do **`SecurityManager`**.

Wspomnianą funkcjonalność posiada np. klasa **`URLClassLoader`**.

Podpisy cyfrowe (13)

Do podpisywania klas (zapisanych w archiwach Java™) oraz weryfikacji podpisów złożonych na klasach razem z J2SE jest dostarczane specjalne narzędzie: **`jarsigner`**.

Do składania podpisów oraz ich weryfikacji **`jarsigner`** wykorzystuje dane (klucze i certyfikaty) zapisane w magazynach kluczy.

Oczywiście **`jarsigner`** może wykorzystywać wszystkie implementacje **`KeyStoreSpi`** dostępne w ramach konkretnej instalacji JRE.
Jeśli użytkownik posiada własną implementację *keystore*, może bez przeszkód korzystać z niej za pomocą **`jarsigner`** dostarczanego standardowo z J2SE.

Co prawda w ramach J2SE nie istnieje pojęcie „domyślnego magazynu kluczy”, ale **`jarsigner`** – podobnie jak **`keytool`** – przyjmuje, że domyślną lokalizacją **`keystore`** jest zapisany w pliku: **`$HOME/.keystore`**.

Podpisane *Java™ archive* – w porównaniu do nie podpisanego – zawiera nieco zmodyfikowany pod względem zawartości plik manifestu: **`MANIFEST.MF`**, oraz dodatkowo w podkatalogu **`META-INF`** znajdują co najmniej dwa dodatkowe pliki: **`<signer>.SF`** i **`<signer>.RSA`**, bądź **`<signer>.DSA`**.

103

Podpisy cyfrowe (14)

Plik z rozszerzeniem **`.SF`** zawiera skróty SHA1 oraz MD5 każdej podpisanej klasy zapisanej w archiwum Java™.

Część podstawowa (*base*) nazwy pliku – czyli bez rozszerzenia (**`.SF`**) – jest nazwą aliasu w magazynie kluczy, pod którym umieszczono klucz prywatny, przy pomocy którego został złożony podpis.

Pliki **`.DSA`** lub **`.RSA`** zawierają podpis cyfrowy pliku **`.SF`** o tej samej części podstawowej nazwy (*base*).

Rozszerzenie pliku odpowiada nazwie algorytmu wykorzystywanego do podpisywania, który z kolei jest zależny od rodzaju klucza prywatnego znajdującego się pod danym aliasem.

Podpis zawarty w pliku **`<signer>.DSA`**/**`<signer>.RSA`** jest zapisany zgodnie ze standardem PKCS#7 – powszechnie wykorzystywanym formatem zapisu podpisu cyfrowego obejmującym certyfikat podmiotu składającego podpisującego.

Sam plik **`MANIFEST.MF`** zawiera listę nazw plików klas – przy każdym pliku znajduje informacja o skrótach zawartości plików: MD5 i SHA1.

104

Podpisy cyfrowe (15)

```
$ jarsigner archive.jar edek
```

Uruchomienie **`jarsigner`** z powyższymi opcjami spowoduje złożenie podpisu na wszystkich klasach znajdujących się w pliku **`archive.jar`** kluczem prywatnym umieszczonym w lokalnym domyślnym *keystore* pod aliasem **`"edek"`**.

Lista skrótów SHA1 oraz MD5 plików klas oraz plik zawierający podpis pliku **`.SF`** będą nazywały się odpowiednio: **`EDEK.SF`** oraz **`EDEK.DSA`** lub **`EDEK.RSA`**.

Ważniejsze opcje `jarsigner`:

`-keystore` – umożliwia wybór magazynu kluczy;
`-storepass` – pozwala przekazać *passphrase* jeśli integralność magazynu kluczy jest chroniona za pomocą hasła;
`-sigfile` – nazwa bazowa dla plików: **`.SF`**, **`.DSA`**/**`.RSA`**;
`-signedjar` – cała zawartość podpisywanego archiwum zostanie przekopiowana do wskazanego pliku, po czym jej zawartość zostanie podpisana.
`-sigfile` – do weryfikacji podpisu zostanie użyta określona nazwa bazowa – opcja przydatna w sytuacji, gdy zawartość *Java™ archive* została podpisana przez wiele podmiotów.

Podpisy cyfrowe (16)

```
$ jarsigner –verify archive.jar
```

Powyższe wywołanie przeprowadzi weryfikację integralności wszystkich plików (w większości **`.class`**) ujętych w liście zapisanej w pliku **`.SF`**.

W procesie sprawdzania spójności zostanie użyty klucz publiczny z certyfikatu towarzyszącego podpisowi zapisanego w pliku **`.RSA`**/**`.DSA`**.

```
$ jarsigner –verify -verbose archive.jar
sm      6401 Wed Feb 12 11:19:12 CET 2003 Class1.class
sm      1823 Wed Feb 12 11:19:12 CET 2003 Class2.class
sm     12052 Wed Feb 12 11:19:12 CET 2003 Class3.class
  s = signature was verified
  m = entry is listed in manifest
  k = at least one certificate was found in keystore
  i = at least one certificate was found in identity scope
jar verified.
```

Zazwyczaj uzyskamy informację, że wszystkie klasy są podpisane, ale oczywiście istnieje możliwość, że część klas będzie dodana do archiwum po złożeniu podpisu przez dany podmiot – wówczas nie będą one w ogóle podpisane, lub będą podpisane innym kluczem prywatnym (innego podmiotu).

Szyfrowanie (1)

W ramach API udostępnianego przez Java™ 2 Standard Edition Klasa reprezentację mechanizmu szyfrowania obu rodzajów (symetrycznego i asymetrycznego) stanowi klasa **`javax.crypto.Cipher`**.

Analogicznie do **`MessageDigest`**, czy **`Mac`** klasa **`Cipher`** udostępnia interfejs do szyfrowania i deszyfrowania danych w tablicach bajtów.

Podobnie jak działo się w przypadku pozostałych składowych JCA i JCE mechanizm szyfrowania reprezentowany przez **`Cipher`** jest realizowany przez konkretne implementacje poszczególnych algorytmów.

Cechą specyficzną mechanizmu **`Cipher`** jest fakt istnienia specjalnej konwencji nazewniczej przy określaniu drugiego członu pary: mechanizm/algorytm.

Zgodnie ze wspomnianą konwencją nazwa konkretnej realizacji mechanizmu szyfrowania (czyli algorytmu) musi zawierać trzy elementy: nazwę **właściwego algorytmu**, **schemat dopełnienia (*padding scheme*)** oraz **tryb szyfrowania (*mode*)**.

Szyfrowanie (2)

Wszystkie implementacje **`CipherSpi`** standardowo dołączane do J2SE stanowią realizacje szyfrów blokowych – J2SE nie dostarcza żadnej implementacji szyfrów strumieniowych.

Niektóre spośród algorytmów, których implementacje są udostępniane w ramach SunJCE:

- **`DES`** (*Data Encryption Standard*)
- **`DESede`** (lub potrójny DES) wykorzystujący DES w celu następujących po sobie wielokrotnie (trzykrotnie) szyfrowania i deszyfrowania (*encryption-decryption-encryption*).
  `DESede` wymaga trzech kluczy, które w praktyce są one zapisywane jako jeden klucz o długości 168 bitów.
- **`PBEWithMD5AndDES`** – szyfrowanie oparte o hasło – zgodnie z opracowanym przez firmę RSA Data Security Inc. algorytmem PKCS#5 hasło łączone z tablicą losowych bajtów, tzw. solą (*salt*) oraz skrótem MD5 wiadomości stanowią razem podstawę do wyliczenia klucza tajnego.
- **`Blowfish`** – algorytm opracowany przez Bruce'a Schneiera, którego opis – w odróżnieniu od np. DES – był od samego początku dostępny publicznie.

Szyfrowanie (3)

Zadaniem **trybów szyfrowania (*modes*)** jest utrudnienie złamania szyfrogramu.

Przykładowo, powtarzające się wielokrotnie w tekście otwartym wzorce (nagłówki, stopki, itp.) mogą stanowić potencjalne ułatwienie złamania szyfru (klucza). Właściwie dobrany tryb szyfrowania pozwala ukryć tego rodzaju informację przed napastnikiem.

Zastosowanie niektórych trybów może zmusić szyfr blokowy do przetwarzania mniejszych porcji danych, niż całe bloki, co oznacza, że zacznie się on zachowywać podobnie do szyfru strumieniowego.

Jest to bardzo istotna cecha, która najczęściej znajduje zastosowanie przy zabezpieczaniu interaktywnych połączeń sieciowych, kiedy to istnieje konieczność szyfrowania transmisji jednego, dwóch znaków (bajtów) w danym momencie.

Najprostszym możliwym trybem szyfrowania jest **`ECB` (*electronic cookbook mode*)**, który pobiera blok danych i szyfruje go w całości nie utrudniając w żaden sposób wyszukiwania wzorców – przetwarzanie bloku nie ma wpływu na szyfrowanie bloków kolejnych.

109

Szyfrowanie (4)

Ze względu na oczywiste ułomności tryb **`ECB`** jest całkowicie nieprzydatny w przypadku zabezpieczania tekstu lub innego rodzaju danych zawierających wzorce.

`ECB` jest zalecany co najwyżej do szyfrowania danych losowych.
Algorytm pracujący w trybie **`ECB`** może przetwarzać wyłącznie całe bloki, z tego względu **`ECB`** wymusza stosowanie dopełnienia (*padding*), ale nie wymaga wektora inicjalizującego.

W przypadku trybu **`CBC` (*cipher block chaining mode*)** dane z bloku poprzedzającego mają – w odróżnieniu do **`ECB`** – wpływ na szyfrowanie bloku następnego.

`CBC` umożliwia zatem zakamuflowanie wzorca i jest jedynym trybem, który może być wykorzystywany razem z **`PBEWithMD5andDES`** – bardzo dogodny m.in. dla danych tekstowych.

`CBC` może operować wyłącznie na całych blokach – zatem podobnie jak **`EBC`** wymaga dopełniania.
`CBC` wymaga również wektora inicjalizującego (*initialization vector*) przed rozpoczęciem deszyfrowania.

110

Szyfrowanie (5)

CFB (*cipher feedback mode*) – czyli tryb **tzw. sprzężenia zwrotnego szyfru** jest podobny w zakresie funkcjonalności do **CBC**, choć w odróżnieniu do tego drugiego nie wymaga całkowicie wypełnionego bloku w celu rozpoczęcia szyfrowania.

CFB jest korzystnym rozwiązaniem w zastosowaniach, gdzie konieczne jest przetwarzanie danych w mniejszych fragmentach niż pełne (zazwyczaj 64-bitowe) bloki – np. przy interaktywnych połączeniach sieciowych.

Tryb **CFB** – podobnie jak **CBC** – wymaga wektora inicjalizującego.

OFB (*output feedback mode*) – tryb **tzw. sprzężenia zwrotnego wyjścia** znajduje zastosowanie w rozwiązaniach, w których istnieje możliwość wystąpienia przekłamań podczas transmisji danych.
W innych trybach utrata 1 bitu wiąże się z utratą całego bloku, w przypadku **OFB** tracony jest tylko 1 bit danych.

OFB przetwarza wyłącznie całe bloki (stosuje dopełnienie), wymaga również wektora inicjalizującego.

111

Szyfrowanie (6)

PCBC (*propagating cipher block chaining*) – czyli wiązanie bloków z propagowaniem szyfru było w przeszłości wykorzystywane w Kerberos v4, ze względu na znane sposoby ataku został zarzucony w wersji 5.

Algorytm szyfrowania działający w trybie **PCBC** może przetwarzać wyłącznie pełne bloki, do rozpoczęcia pracy jest również wymagany wektor inicjalizujący.

Udostępniane w SunJCE schematy dopełnienia:
- **PKCS5Padding** – dane wejściowe są dopełniane do wielokrotności 8 bajtów.
- **NoPadding** – dopełnienie nie jest w ogóle wykonywane, a zatem liczba bajtów przekazywanych do szyfru musi być wielokrotnością bloku szyfru – inaczej zostanie zgłoszony wyjątek.

Instancję konkretnej implementacji **CipherSpi** uzyskujemy za pomocą metody **getInstance()** na zasadach znanych z omówionych wcześniej mechanizmów kryptograficznych.

Przekazywana do **getInstance()** nazwa algorytmu może obejmować jedynie nazwę algorytmu właściwego (np. **DES**, **DESede**, **AES**) jak również może być podana w formacie **<algorytm>/<tryb>/<dopełnienie>**.

```
DES/ECB/PKCS5Padding
```

112

Szyfrowanie (7)

Tradycyjnie jeśli nie zostały wyspecyfikowane ani tryb, ani schemat dopełnienia do `getInstance()` są przekazywane wartości domyślne specyficzne dla wykorzystywanej implementacji – zależne od konkretnego *security provider*.

Przykładowo w SunJCE domyślnym trybem jest **`ECB`**, lub **`CBC`** jeśli algorytmem jest **`PBEWithMD5AndDES`**, domyślnym schematem dopełnienia zaś jest **`PKCS5Padding`**.

Uzyskaną za pomocą `getInstance()` implementację mechanizmu szyfrowania należy w zależności od jej przyszłego wykorzystania odpowiednio zainicjalizować.

Do inicjalizacji służy metoda `init()`, która rozróżnia następujące sposoby wykorzystania mechanizmu szyfrowania: (1) szyfrowanie, (2) deszyfrowanie, (3) opakowywanie oraz (4) odpakowywanie klucza.

Ponadto w wywołaniu `init()` należy przekazać odpowiedni klucz – w zależności od rodzaju wykorzystywanego algorytmu właściwego (symetrycznego, bądź asymetrycznego).

Ponowne wywołanie `init()` przywraca mechanizm do stanu początkowego, zatem ten sam obiekt może być z powodzeniem wykorzystywany najpierw do szyfrowania, a później do deszyfrowania.

113

Szyfrowanie (8)

Po inicjalizacji do instancji szyfru należy dostarczyć dane za pomocą metody `update()`, która w zależności od konfiguracji obiektu szyfru wykonuje szyfrowanie, bądź deszyfrowanie danych przekazanych w buforze wejściowym.

Metoda `doFinal()` robi dokładnie to samo, co `update()` z tą różnicą, że jej użycie sygnalizuje, że do mechanizmu jest przekazywany ostatni blok danych.

Jeżeli długość danych przekazanych do metody `update()` nie jest wielokrotnością długości używanego bloku, częściowo wypełniony ostatni blok danych jest buforowany w instancji mechanizmu do następnego wywołania `update()`, bądź `doFinal()`.

Wywołanie `doFinal()` powoduje automatyczne dopełnienie danych znajdujących się w buforze – o ile oczywiście aktualnie używany tryb daje taką możliwość.

Jeśli deszyfrowanie wymaga użycia bufora inicjującego, do instancji mechanizmu musi zostać przekazany ten sam wektor, który był wykorzystywany w trakcie szyfrowania.

W celu uzyskania wektora inicjującego (np. w celu późniejszego wysłania odbiorcy wiadomości) należy posłużyć się metodą `getIV()`.

114

Szyfrowanie (9)

```
...
try {
   KeyGenerator kg = KeyGenerator.getInstance("DES");
   Cipher c = Cipher.getInstance("DES/CBC/PKCS5Padding");
   Key k = kg.generateKey();
   c.init(Cipher.ENCRYPT_MODE,k);
   byte[] plainText = "Plain text".getBytes();
   byte[] cipherText = c.doFinal(input);
   IvParameterSpec ivSpec = new IvParameterSpec(c.getIV());
   c.init(Cipher.DECRYPT_MODE,k,ivSpec);
   byte[] decrypted = c.doFinal(cipherText);
   System.out.println("Before encryption: " + plainText);
   System.out.println("After decryption: " + decrypted);
} catch (Exception ex) {
   ex.printStackTrace();
}
...
```


Szyfrowanie (10)

Kroki wykonywane w trakcie szyfrowania:

1. Inicjalizacja instancji mechanizmu
2. Właściwe szyfrowanie – do instancji mechanizmu szyfrowania przekazywany jest ciąg bajtów poprzez wielokrotne wywołanie **`update()`** – ostatni blok powinien być przekazany do metody **`doFinal()`**
3. Odczyt wektora inicjującego za pomocą metody **`getIV()`** i przekazanie odbiorcy – oczywiście ten krok jest zbędny w przypadku trybu **`ECB`**

Kroki wykonywane w trakcie deszyfrowania:

1. Inicjalizacja instancji mechanizmu (w tym przekazanie wektora inicjalizującego)
2. Właściwe deszyfrowanie – analogicznie do szyfrowania

Wektor inicjalizujący, który musi być dostarczony odbiorcy przez nadawcę, może być przesłany kanałem niezabezpieczonym – sam *initialization vector* nie wystarcza bowiem do odszyfrowania wiadomości.

Dość specyficzną implementacją **`CipherSpi`** udostępnianą w ramach JCE jest nierealizująca żadnego szyfrowania klasa **`NullCipher`**.

Podstawowym zastosowaniem **`NullCipher`** jest stanowienie atrapy mechanizmu szyfrowania w trakcie testowania funkcjonalności rozwijanej aplikacji.

Szyfrowanie (11)

Realizacja mechanizmu szyfrowania oparta o PBE (*password based encryption*) wymaga – poza przekazaniem samego hasła – inicjalizacji specjalnymi danymi: tzw. solą (*salt*) oraz licznikiem iteracji (*iteration count*).

Celem wprowadzenia obu wymienionych elementów jest podniesienie bezpieczeństwa zabezpieczenia opartego na PBE – „sól" jest losową liczbą, z kolei *iteration count* określa liczbę iteracji wykonywanych w trakcie generowania klucza.

Im dłuższy ciągiem bajtów jest sól, oraz im więcej iteracji jest wykonywanych podczas generowania klucza, tym lepsze jest zabezpieczenie – oczywiście kosztem czasu potrzebnego na utworzenia klucza.

Niezależnie od stosowania dodatkowych zabezpieczeń w postaci soli i liczby iteracji hasło stanowiące trzeci parametr wejściowy dla PBE powinno być trudne do odgadnięcia – czyli nie powinno poddawać się metodzie słownikowej.

Szyfrowanie (12)

```
/**
  * code snippet that presents generation of PBE key based on
  * salt, and password
  */
try {
   String password = "Secret password";
   byte[] salt = byte[16];
   SecureRandom rand = new SecureRandom();
   rand.nextBytes(salt);
   PBEParameterSpec spec = new PBEParameterSpec(salt,30);
   PBEKeySpec keySpec = new PBEKeySpec(password.toCharArray());
   SecretKeyFactory f =
      SecretKeyFactory.getInstance("PBEWithMD5AndDES");
   SecretKey k = kf.generateSecret(keySpec);
   Cipher c = Cipher.getInstance("PBEWithMD5AndDES");
   c.init(Cipher.ENCRYPT_MODE,k,spec);
} catch (Exception ex) {
   ex.printStackTrace();
}
```

Szyfrowanie (13)

Algorytmy symetryczne są w ogólności o wiele bardziej wydajne niż algorytmy asymetryczne, z drugiej jednak strony korzystanie z kryptografii klucza tajnego jest organizacyjnie o wiele trudniejsze, niż korzystanie z kryptografii klucza publicznego.

Sposobem na połączenie zalet obu rodzajów szyfrowania jest wykorzystanie kryptografii klucza publicznego do wymiany kluczy tajnych.

Ponieważ potrzeba zaszyfrowania klucza tajnego w celu przekazania go drugiej stronie komunikacji jest dość powszechna, klasa **`Cipher`** dostarcza metody, które znacząco ułatwiają realizację tego zadania.

Opakowywanie (szyfrowanie) oraz odpakowywanie (deszyfrowanie) klucza wymaga w pierwszej kolejności przełączenie instancji mechanizmu szyfrowania do odpowiedniego trybu za pomocą metody **`init()`**.

Oczywiście w trakcie inicjalizacji należy przekazać również klucz, który będzie użyty do pakowania, bądź odpakowywania.

Do właściwego pakowania i odpakowywania kluczy służą odpowiednio metody: **`wrap()`** oraz **`unwrap()`**.

119

Szyfrowanie (14)

```
try {
   KeyGenerator kg = KeyGenerator.getInstance("AES");
   Key sharedKey = kg.generateKey();
   String password = "Secret password"; byte[] salt = byte[16];
   SecureRandom rand = new SecureRandom();
   PBEParameterSpec spec = new PBEParameterSpec(salt,30);
   PBEKeySpec keySpec = new PBEKeySpec(password.toCharArray());
   SecretKeyFactory f =
      SecretKeyFactory.getInstance("PBEWithMD5AndDES");
   SecretKey passKey = kf.generateSecret(keySpec);
   Cipher c = Cipher.getInstance("PBEWithMD5AndDES");
   /** wrap key */
   c.init(Cipher.WRAP_MODE,passKey,spec);
   byte[] wrappedKey = c.wrap(sharedKey);
   /** unwrap key */
   c.init(Cipher.UNWRAP_MODE,passKey,spec);
   Key unwrappedKey =
      c.unwrap(wrappedKey,"AES",Cipher.SECRET_KEY);
} catch (Exception ex) { ex.printStackTrace(); }
```

120

W następnej części wykładu ...

Korzystanie z mechanizmów kryptograficznych w aplikacjach tworzonych w oparciu o platformę Microsoft .NET

Krótkie wprowadzenie (1)

Interfejs programistyczny umożliwiający wykorzystanie kryptografii w aplikacjach tworzonych w oparciu o platformę .NET został umieszczony w przestrzeni nazw `System.Security.Cryptography`.

Klasy udostępniające mechanizmy kryptograficzne tworzą 3-poziomową strukturę:

1. na szczycie hierarchii znajdują się klasy abstrakcyjne reprezentujące podstawowe mechanizmy: (1) algorytmy symetryczne, (2) algorytmy asymetryczne, oraz (3) funkcje skrótu.
   Wymienione typy abstrakcyjne zawierają definicje inwariantów (metod, atrybutów i właściwości) właściwych dla reprezentowanego pojęcia.

2. drugi poziom hierarchii tworzą klasy abstrakcyjne dla konkretnych algorytmów, takich jak np. Rijandel, czy SHA1.
   Klasy zdefiniowane na tym poziomie rozszerzają definicje klas ogólnych reprezentujących poszczególne mechanizmy o właściwości charakteryzujące konkretne algorytmy.

Krótkie wprowadzenie (2)

3. na samym dole znajdują się właściwe implementacje konkretnych algorytmów, np. `RijandelManaged`, czy `SHA1CryptoServiceProvider`.

   Warto zwrócić uwagę, że nazwy poszczególnych typów korespondują ze sposobem implementacji danego mechanizmu przez konkretną klasę. Przytoczona nazwa `RijandelManaged` oznacza, że implementacja algorytmu AES została całkowicie oparta o kod zarządzany. Z kolei `SHA1CryptoServiceProvider` świadczy o tym, że dana implementacja SHA1 została zrealizowana poprzez odwołania do Microsoft Win32 CryptoAPI.

.NET Framework umożliwia konfigurację implementacji mechanizmów kryptograficznych – każdy mechanizm lub konkretny algorytm posiada ściśle określoną domyślną implementację.

Przykładowo domyślną implementacją algorytmu `Rijandel` jest klasa `RijandelManaged`.

Dopuszczalna długość kluczy wykorzystywanych przez algorytmy kryptograficzne jest zależna od używanej wersji Windows oraz czy został zainstalowany tzw. *high encryption update* – dostępny domyślnie począwszy od Windows XP.

123

Szyfrowanie (1)

Jak wiemy szyfrowanie symetryczne (*symetric encryption*) umożliwia nam zabezpieczenie poufności danych w oparciu o:
- algorytm szyfrowania symetrycznego (*cipher*) oraz
- klucz tajny współdzielony przez strony biorące udział w wymianie informacji.

Oczywiście przy tego rodzaju komunikacji największe wyzwanie stanowi zapewnienie poufności podczas dystrybucji oraz przechowywanie klucza tajnego umożliwiającego zarówno:
- tworzenie szyfrogramu na podstawie tekstu jawnego, jak i
- odtwarzanie tekstu jawnego z szyfrogramu.

Ze względu na problemy związane z przechowywaniem i dystrybucją częstą praktyką jest uzyskiwanie klucza tajnego na podstawie hasła podanego przez użytkownika.

Przytoczone rozwiązanie nie jest niestety pozbawione poważnych wad – przeciętny człowiek nie jest w stanie łatwo zapamiętać hasła, które mogłoby stanowić ziarno dla mocnego klucza.

124

Szyfrowanie (2)

Można podnieść jakość opisanego zabezpieczenia poprzez wymuszanie na użytkownikach podawania haseł składających się różnych znaków drukowalnych, jak również nie poddających się atakom słownikowym.

Innym rozwiązaniem jest „doklejenie” do hasła losowego ciągu znaków, lub wygenerowanie klucza w oparciu hasło użytkownika oraz sól.

W obu sytuacjach użytkownik musi dostarczyć oba elementy, na podstawie których jest uzyskiwane jest hasło, co wiąże się z koniecznością:
- zapisania przez użytkownika losowo wygenerowanego członu na kartce, tak by mógł on wprowadzić ten ciąg podczas deszyfrowania wiadomości, lub też
- dołączenie tego ciągu w postaci niezabezpieczonej do właściwego szyfrogramu.

Jak wiemy na siłę hasła wpływ mają zarówno długość hasła, jak również wielkość zbioru użytych znaków.
Przykładowo efektywna długość 7-znakowego hasło, którego weryfikacja nie uwzględnia wielkości liter (*case insensitive*) wynosi zaledwie 33 bity (!) – zamiast teoretycznie 56.

Szyfrowanie (3)

W Microsoft .NET – podobnie jak w przypadku platformy Java™ – powszechnie używany generator liczb losowych (**`System.Random`**) jest de facto generatorem liczb pseudolosowych, dla którego ziarnem jest pewna ściśle określona liczba.

Z tego względu przestrzeń nazw **`System.Security.Cryptography`** została uzupełniona o implementację generatora liczb losowych – **`RandomNumberGenerator`**.

```
/// examplary verification of password strength
using System.Text.RegularExpresssions;
...
const string REGEX_PATTERN =
   @"^(?=.*[0-9])(?=.*[a-z])(?=.*[A-Z])(?=.*\W)";

bool verifyPasswordStrength (string password) {
   Regex r = new Regex(REGEX_PATTERN, RegexOptions.Singleline);
   return (password.Length >= 9 && r.IsMatch(password))
      ? true : false;
}
```

Szyfrowanie (4)

```
/// generate salt --- a cryptographically random sequence of
/// bytes, which is basically responsible for the strength of
/// a derived key
const int SALT_LEN = 16;

RandomNumberGenerator rand = RandomNumberGenerator.Create();
byte[] salt = new byte[SALT_LEN];
rand.GetBytes(salt);
```

```
/// derive key from a user provided password and a salt
const int KEY_LEN = 16;

PasswordDerivedBytes bytes =
   new PasswordDerivedBytes(password, salt);
byte[] key = bytes.GetBytes(KEY_LEN);

/// to be continued on the consecutive slides
```

127

Szyfrowanie (5)

Właściwe szyfrowanie danych za pomocą algorytmu symetrycznego wymaga w pierwszej kolejności skonfigurowania obiektu reprezentującego wybrany algorytm.

```
/// create and configure an instance of symmetric algorithm
Rijandel algorithm = Rijandel.Create();
/// assign the byte array prepared on the previous slides
algorithm.Key = key;
```

Kolejnym krokiem jest utworzenie strumienia umożliwiającego zapis danych w postaci zaszyfrowanej.

Zanim jednak rozpoczniemy właściwy zapis do strumienia szyfrującego musimy w pierwszej kolejności utrwalić w postaci niezabezpieczonej sól oraz wektor inicjalizujący (*initialization vector*).

Wektor inicjalizujący przechowuje stan algorytmu przed rozpoczęciem właściwego szyfrowania, z kolei sól umożliwia odtworzenie klucza, za pomocą którego odbiorca będzie mógł odszyfrować otrzymaną wiadomość.

128

Szyfrowanie (6)

```
/// create a FileStream instance
FileStream fs = new FileStream(filename, FileMode.Create);
/// write down previously prepared salt the FileStream
fs.Write(salt, 0, salt.Length);
/// write down initialization vector to the FileStream
fs.Write(algorithm.IV, 0, algorithm.IV.Length);
```

Ostatnim krokiem wieńczącym cały proces jest utworzenie instancji **CryptoStream** oraz zapewnienie poufności poprzez zapis danych do tego strumienia.

W trakcie korzystania z instancji **CryptoStream** należy pamiętać, by po zapisaniu całości danych wywołać metodę **FlushFinalBlock** zapisującą ostatni blok szyfrowanych danych.

```
/// encrypt the data, flush the last block and close the stream
CryptoStream cs = new CryptoStream(
   fs, algorithm.CreateEncryptor(), CryptoStreamMode.Write);
cs.Write(buffer, 0, buffer.Length);
cs.FlushFnalBlock();
cs.Close();
```

129

Szyfrowanie (7)

Proces deszyfrowania wiadomości jest co prawda procesem odwrotnym w stosunku do szyfrowania, ale przebiega w sposób dość analogiczny do tego drugiego.

W pierwszej kolejności należy:

- **odzyskać klucz w oparciu o sól i hasło współdzielone przez nadawcę i odbiorcę** wiadomości, oraz
- **skonfigurować** wcześniej utworzoną instancję algorytmu za pomocą **wektora inicjalizującego**

```
/// create an instance of a symmetric algorithm
Rijandel algorithm = Rijandel.Create();
/// open encrypted file (or any other input stream)
FileStream fs = new FileStream(filename, FileMode.Open);
/// extract salt and initialization vector
byte[] salt = new byte[SALT_LEN];
       iv = new byte[algorithm.IV.Length];
fs.Read(salt, 0, salt.Length);
fs.Read(iv, 0, iv.Length);

/// to be continued in the next slides
```

130

Szyfrowanie (8)

```
/// regenerate key from password and salt
PasswordDerivedBytes passBytes = new PasswordDerivedBytes(
   password, salt);
byte[] key = passBytes.GetBytes(KEY_LENGTH);
/// set the decryption key to the algorithm instance
algorithm.Key = key;
/// set up the initial algorithm state with the initialization
/// vector
algorithm.IV = iv;
```

Ostatnim krokiem procesu deszyfrowania jest **odczyt danych ze strumienia za pomocą instancji** `CryptoStream`.

```
CryptoStream cs = new CryptoStream(
   fs, algorithm.CreateDecryptor(), CryptoStreamMode.Read);
int bytesRead = 0; const int BUFSIZ = 4096;
byte[] buffer = new byte[BUFSIZ];
while ((bytesRead = cs.Read(buffer, 0, BUFSIZ) > 0)
   /// buffer now contains decrypted data
}
cs.Close(); /// close stream when finished
```

131

Algorytmy asymetryczne – informacje ogólne (1)

Stosowane w praktyce kryptosystemy są wciąż oparte o opracowany w roku 1977 algorytm RSA, który powoli jest wypierany przez uważany za mocniejszy DSA.

O ile czas wygenerowania podpisu przy pomocy klucza o tej samej długości jest w przypadku obu algorytmów zbliżony, o tyle czas algorytmicznej weryfikacji podpisu stworzonego przy użyciu DSA jest do kilkudziesięciu razy dłuższy. który został zatwierdzony jako standard federalny dla podpisu elektronicznego (DSS) w roku

DSA jest algorytmem nieco młodszym, opracowanym na przełomie lat 80-tych i 90-tych ubiegłego stulecia.

W roku 1991 amerykański National Institute of Standards and Technology (NIST) przedstawił propozycję, by DSA stał się standardem dla podpisu cyfrowego wykorzystywanym przez administrację federalną.
Procedura zatwierdzania DSA jako Digital Signature Standard (DSS) została zakończona w roku 1993 i od tego czasu specyfikacja DSA dołączyła do grona Federal Information Processing Standards (FIPS) jako FIPS 186.

DSA w odróżnieniu od RSA był opracowywany z myślą o składaniu podpisów, ale istnieje możliwość wykorzystania DSA również do szyfrowania danych.

132

Algorytmy asymetryczne – informacje ogólne (2)

Przewiduje się, że powszechnie stosowane RSA oparte o problem faktoryzacji (Integer Factorization Cryptography), oraz DSA zaliczany do algorytmów ciała skończonego (Finite-Field Cryptography) z czasem ustąpią miejsca algorytmom z rodziny Ellyptic Curve Cryptography (ECC).

Algorytmy ECC charakteryzują się znacznie krótszymi kluczami – przy tym samym poziomie zabezpieczeń – doświadczenia wskazują, że klucz ECC o długości 163 bitów zapewnia poziom bezpieczeństwa zbliżony do klucza RSA o długości 1024 bitów.

Użycie kryptografii klucza publicznego (1)

Jak wiemy kryptografia asymetryczna poprzez wykorzystanie pary kluczy daje nam możliwość zapewnienia w bardzo elegancki sposób: (1) poufności, (2) integralności, oraz (3) niezaprzeczalności danych.

Niestety obecnie istniejące algorytmy asymetryczne mają jedną zasadniczą wadę, która jednocześnie stanowi zaletę z punktu widzenia poziomu bezpieczeństwa danych.
Współczesne algorytmy asymetryczne są bardzo nieefektywne w porównaniu z szyframi – DES jest ponad 100-krotnie (!) bardziej wydajny od algorytmu RSA.

Wyjątkowo niska wydajność całkowicie uniemożliwiająca samodzielne stosowanie algorytmów asymetrycznych w praktyce wymusiła konieczność tworzenia rozwiązań hybrydowych, które:
- łączą zalety kryptografii symetrycznej i asymetrycznej, przy jednoczesnym
- minimalizowaniu wad dwóch wymienionych rodzajów algorytmów szyfrowania

Użycie kryptografii klucza publicznego (2)

Jak wiemy w kryptografii asymetrycznej wykorzystujemy parę kluczy: (1) prywatny i (2) publiczny, które są wykorzystywane podczas wykonywania operacji wzajemnie odwrotnych.

Zapewnienie poufności (krótkie przypomnienie):
- do szyfrowania danych jest wykorzystywany klucz, bądź klucze publiczne odbiorców
- do deszyfrowania danych przez danego odbiorcę wykorzystywany jest jego klucz prywatny odpowiadający kluczowi publicznemu
  publicznym odbiorcy i deszyfrowane za pomocą jego klucza prywatnego

Zapewnienie integralności i niezaprzeczalności (krótkie przypomnienie):
- podpis składany jest przy użyciu klucza prywatnego nadawcy
- do weryfikacji podpisu po stronie odbiorcy jest wykorzystywany klucz publiczny nadawcy

Implementacje algorytmów asymetrycznych w .NET

W aplikacjach tworzonych na platformę .NET Framework wykorzystanie kryptografii klucza publicznego w aplikacji zawsze w pierwszej kolejności wiąże się z utworzeniem instancji algorytmu szyfrowania.

Klasy konkretne reprezentujące algorytmy asymetryczne udostępniane przez poszczególnych dostawców rozszerzają klasę **`AsymetricAlgorithm`**.

```
/// instantiation of RSACryptoServiceProvider --- a default
/// implementation of RSA encryption algorithm
/// RSACryptoServiceProvider extends abstract class RSA
/// encapsulating the features of RSA algorithm
RSA algorithm = RSA.Create();
```

Oczywiście przed właściwym użyciem instancja algorytmu musi być odpowiednio zainicjalizowana – w tym przede wszystkim parą kluczy.

Składowanie kluczy kryptograficznych (1)

Z myślą o parametryzacji dostawcy bezpieczeństwa wprowadzono w .NET Framework klasę **`CspParameters`**.

```
/// providerType: determines the type of service a particular
/// Cryptographic Service Provider supplies, e.g. 1 for RSA,
/// 13 for Diffie-Hellman key exchange, 5 for Microsoft Office
/// compatible services
/// more values available at:
/// HKEY_LOCAL_MACHINE\Software\Microsoft\Cryptography\
/// Defaults\Provider Types
string PROVIDER_TYPE_RSA = 1;

/// providerName: determines the name of the Cryptographic
/// Service Provider, e.g. ActivCard Gold Cryptographic Service
/// Provider
/// more values available at:
/// HKEY_LOCAL_MACHINE\Software\Microsoft\Cryptography\
/// Defaults\Provider
string PROVIDER_NAME_SAFESIGN = "SafeSign CSP Version 1.0";
```

137

Składowanie kluczy kryptograficznych (2)

```
string CONTAINER_NAME = "Arbitrary Container Name";

CspParameters paramsCsp = new CspParameters(PROVIDER_TYPE_RSA,
   PROVIDER_NAME_SAFESIGN, CONTAINER_NAME);

/// create an instance of a concrete class complying with the
/// interface specified by RSA abstract class
/// the instance will be the implementation of RSA supplied by
/// SafeSign CSP Version 1.0 provider which enables access to
/// the functionality of a SafeSign smart card
RSA algorithm = new RSACryptographicServiceProvider(paramsCsp);
```

138

Składowanie kluczy kryptograficznych (3)

Jak wiemy do przechowywania pary kluczy (a dokładnie pary składającej się z klucza prywatnego i łańcucha certyfikatów) służą tzw. magazyny kluczy (*key stores*).

Platforma .NET jest silnie zintegrowana z systemem operacyjnym Microsoft Windows i z tego względu aplikacje tworzone w oparciu o to środowisko wykorzystują systemowy magazyn kluczy.

Począwszy od wersji 2.0 systemowy magazyn kluczy jest reprezentowany przez klasę **`X509Store`**.

```
/// create an instance of key store which holds the personal
/// certificates of the currently logged on user
X509Store store = new X509Store(StoreName.My,
  StoreLocation.CurrentUser);
store.Open(OpenFlags.ReadWrite);
```


Składowanie kluczy kryptograficznych (4)

Poszczególne wpisy (*entries*) w ramach systemowego magazynu kluczy są reprezentowane przez klasę o wyjątkowo nieintuicyjnej i błędnej nazwie: **`X509Certificate2`**.

Brak rozróżnienia w interfejsie pozycji klucza/certyfikatu od właściwego w interfejsie programistycznym sugerowałoby wręcz brak zrozumienia co tak naprawdę kryje się pod pojęciem certyfikatu cyfrowego.

```
X509Store store = new X509Store(StoreName.My,
  StoreLocation.CurrentUser);
store.Open(OpenFlags.ReadWrite);
foreach (X509Certificate2 cert in store.Certificates) {
  if (cert.HasPrivateKey) {
    Console.WriteLine("key entry");
  } else {
    Console.WriteLine("certificate entry");
  }
  Console.WriteLine("dn: " + cert.Subject);
}
```

Składowanie kluczy kryptograficznych (5)

Inną metodą przechowywania kluczy kryptograficznych wspieranym przez .NET Framework jest zapis w postaci dokumentu XML – z oczywistych względów nie jest to sposób zalecany w rozwiązaniach wdrożonych.

Ten rodzaj magazynu kluczy stworzono wyłącznie z myślą o etapie rozwoju oprogramowania.

```
/// create a file which holds a key pair
private void CreateTestKey (FileStream file) {
  RSA algorithm = RSA.Create();
  string key = algorithm.ToXmlString(true);
  Writer writer = new StreamWriter(file);
  writer.Write(key);
  writer.Close();
}
/// to be continued on the consecutive slide
```

141

Składowanie kluczy kryptograficznych (6)

```
/// continuation of the code snippet from the previous slide

/// load a key pair from a given XML file
private RSA LoadTestKey (FileStream file) {
  Reader reader = new StreamReader(file);
  string key = reader.ReadToEnd();
  reader.Close();
  RSA algorithm = RSA.Create();
  algorithm.FromXmlString(key);
  return algorithm;
}
```

Chcąc przechowywać w pliku jedynie klucz publiczny musielibyśmy ręcznie usunąć z pliku wszystkie podelementy `RSAKeyValue` poza `Modulus` i `Exponent`.

142

Szyfrowanie algorytmami asymetrycznymi (1)

Najprostszym sposobem na zaszyfrowanie danych przy pomocy algorytmu asymetrycznego jest skorzystanie bezpośrednio z interfejsu programistycznego udostępnianego przez instancję klasy implementującej dany algorytm.

```
private static byte[] StringToByteArray(string input) {
   ASCIIEncoding encoder = new ASCIIEncoding();
   return encoder.GetBytes(input);
}
private static void Main () {
  byte[] plaintext = StringToByteArray("top secret data");
  RSACryptoServiceProvider rsa
    = new RSACryptoServiceProvider();
  boolean PADDING_OAEP = true;
  boolean PADDING_PKCS1 = false;
  byte[] ciphertext = rsa.Encrypt(plaintext, PADDING_PKCS1);
}
```

Klasa **`RSACryptoServiceProvider`** wspiera dwa schematy uzupełnienia bloków danych (*padding scheme*):

- **Optimal Asymmetric Encryption Padding (OAEP),** oraz
- standardowego sposobu uzupełniania **opisanego w standardzie PKCS#1**

143

Szyfrowanie algorytmami asymetrycznymi (2)

Optimal Asymmetric Encryption Padding (OAEP) polega na uzupełnieniu tekstu jawnego (*plaintext*) losowym ciągiem przed właściwym zaszyfrowaniem.

OAEP definiuje tzw. bezpieczny sposób dołączenia losowego ciągu bitów do tekstu jawnego dla deterministycznej funkcji z „oknem wyłazowym” (*trapdoor function*).

Innymi słowy uzupełnienie tekstu jawnego losowym ciągiem znaków nie wpływa na wynik deszyfrowania – oczywiście przy założeniu, że odbiorca szyfrogramu ma świadomość jakiego rodzaju schemat uzupełniania był wykorzystany w trakcie szyfrowania.

Dzięki dodaniu losowego ciągu do oryginalnej wiadomości uzyskujemy efekt mapowania jednego tekstu jawnego na wiele szyfrogramów przekształcając tym samym **schemat deterministyczny (*deterministic scheme*)** w tzw. **schemat probabilistyczny (*probabilistic scheme*)**.

Przejście ze schematu deterministycznego na probabilistyczny efektywnie zwiększa tzw. bezpieczeństwo semantyczne (*semantic security*), czyli poziom zabezpieczenia tekstu jawnego przy założeniu, że przeciwnik (*adversary*):

- posiada dostęp szyfrogramu, oraz
- zna zasady funkcjonowania schematu szyfrowania.

144

Szyfrowanie algorytmami asymetrycznymi (3)

Poza możliwością szyfrowania programista ma również możliwość wykonania na rzecz instancji danego algorytmu operacji odwrotnej – deszyfrowania.

Oczywiście warunkiem pomyślnego deszyfrowania jest użycie tego samego schematu uzupełnienia, co podczas szyfrowania.

```
private static void Main () {
  byte[] ciphertext;
  /// retrieve ciphertext
  ...
  RSACryptoServiceProvider rsa
    = new RSACryptoServiceProvider();
  boolean PADDING_OAOP = true;
  boolean PADDING_PKCS1 = false;
  byte[] plaintext = rsa.Decrypt(ciphertext, PADDING_PKCS1);
}
```

Na koniec należy pamiętać, że wybór rodzaju uzupełniania tekstu jawnego określa maksymalny rozmiar wiadomości, jaki możemy zabezpieczyć za pomocą algorytmu asymetrycznego.

145

Szyfrowanie algorytmami asymetrycznymi (4)

.NET Framework dostarcza tzw. klasy „pomocnicze” ułatwiające zarówno szyfrowanie, jak i deszyfrowanie informacji za pomocą algorytmów asymetrycznych.

Jak wiemy spotykane w praktyce rozwiązania (SSL/TLS, PKCS#7, S/MIME, PGP) są tzw. kryptosystemami hybrydowymi wykorzystującymi zarówno kryptografię klucza tajnego, jak i publicznego.

Klasy **`AsymmetricKeyExchangeFormatter`** oraz **`AsymmetricKeyExchangeDeformatter`** stanowią abstrakcyjną reprezentację mechanizmów ułatwiających zapewnienie poufności podczas wymiany kluczy symetrycznych, które będą wykorzystane do właściwego szyfrowania danych.

Konkretnymi implementacjami powyższych klas są np. pary:

- **`RSAOAEPKeyExchangeFormatter`**, **`RSAOAPKeyExchangeDeformatter`**, czy
- **`RSAPKCS1KeyExchangeFormatter`**, **`RSAPKCSKeyExchangeDeformatter`**

146

Szyfrowanie algorytmami asymetrycznymi (5)

```
/// ensure confidentiality while key exchange with RSA

private byte[] encryptKey (RSA alg, byte[] key) {
  RSAPKCS1KeyExchangeFormatter formatter
    = new RSAPKCS1KeyExchangeFormatter(alg);
  return formatter.CreateKeyExchange(key);
}

private byte[] decryptKey (RSA alg, byte[] encryptedKey) {
  RSAPKCS1KeyExchangeFormatter formatter
    = new RSAPKCS1KeyExchangeFormatter(alg);
  return formatter.DecryptKeyExchange(encryptedKey);
}
```

147

Funkcje skrótu (1)

Wyliczenie skrótu wiadomości – podobnie jak w przypadku innych usług kryptograficznych – wymaga w pierwszej kolejności utworzenia obiektu reprezenującego prymityw funkcji jednokierunkowej (*one-way function*).

Abstrakcyjną reprezentacją funkcji skrótu, której konkretne implementacje są udostępniane przez poszczególnych dostawców jest klasa **`HashAlgorithm`**.

Najprostszym sposobem na wyliczenie skrótu wiadomości jest wywołanie metody **`ComputeHash`** na rzecz implementacji konkretnego algorytmu skrótu przekazując jako parametr treść komunikatu w postaci tablicy bajtów, bądź strumienia.

```
private static byte[] ComputeSHA1MessageDigest(byte[] data) {
   HashAlgorithm hash = SHA1.Create();
   return hash.ComputeHash(data);
}
private static byte[] ComputeMD5MessageDigest(Stream input) {
   HashAlgorithm hash = MD5.Create();
   return hash.ComputeHash(input);
}
```

148

Funkcje skrótu (2)

Czasami wyliczenie skrótu na podstawie strumienia, a tym bardziej na podstawie tablicy bajtów jest niemożliwe, bądź całkowicie nieefektywne.

W takich okolicznościach należy dołączyć usługę wyliczania skrótu do obiektu CryptoStream, który w miarę możliwości będzie dostarczał dane do obiektu reprezentującego przekształcenie kryptograficzne (**`ICryptoTransform`**).

```
Stream input = new FileStream("input.dat", FileMode.Open);
HashAlgorithm hash = SHA512.Create();
CryptoStream cryptoStream = new CryptoStream(
  input, hash, CryptoStreamMode.Read);
Stream output = new FileStream(
  "input-with-hash.dat", FileMode.Create);
int bytesRead = 0;
int BUFFER_SIZE = 4096, END_OF_FILE = 0;
byte[] buffer = new byte[BUFFER_SIZE];
do { /// copy data as well as update the current digest value
  byteRead = cryptoStream.Read(buffer, 0, BUFFER_SIZE);
  output.Write(buffer, 0, bytesRead);
} while (bytesRead > END_OF_FILE); /// to be continued ...
```

149

Funkcje skrótu (3)

```
/// continuation of the code snippet from the previous slide
cryptoStream.Close();

/// append computed hash value to the output stream
output.Write(hash.Hash, 0, hash.Hash.Length);
output.Close();
```

150

Podpis cyfrowy (1)

Implementacje algorytmów RSA (**RSACryptoServiceProvider**) oraz DSA (**DSA**) dostarczane wraz z .NET Framework posiadają metody umożliwiające wygenerowanie podpisu cyfrowego dla danych przekazanych w postaci tablicy bajtów, bądź strumienia wejściowego.

```
const string OID_SHA1 = "1.3.14.3.2.26";
const strign OID_MD5 = "1.2.840.113549.2.5";
private byte[] GenerateRSASignature1 (byte[] data,
  RSAParameters key)
{
  HashAlgorithm hash = SHA1.Create();
  RSACryptoServiceProvider rsa
    = new RSACryptoServiceProvider();
  rsa.ImportParameters(key);
  byte[] hashValue = hash.ComputeHash(data);
  byte[] signature = rsa.SignHash(hashValue, OID_SHA1);
  return signature;
}
```

151

Podpis cyfrowy (2)

```
/// another method for generating RSA signature
private byte[] GenerateRSASignature2 (byte[] data,
  RSAParameters key)
{
  HashAlgorithm hash = SHA1.Create();
  RSACryptoServiceProvider rsa
    = new RSACryptoServiceProvider();
  rsa.ImportParameters(key);
  byte[] signature = rsa.SignData(data,hash);
  return signature;
}
```

Nazwy algorytmów skrótu mogą być przekazywane w postaci identyfikatorów ASN.1 zarejestrowanych w hierarchii obiektów opisanej w rodzinie standardów X.66x oraz X.67x, bądź w postaci tzw. prostych nazw: **MD5**, czy **SHA1**.

Rejestracją specyfikacji w drzewie standardów zajmują się podkomitety International Telecommunication Union (ITU), bądź International Organization for Standardization (ISO) właściwe dla dziedziny, której dany standard dotyczy.

152

Podpis cyfrowy (3)

Oczywiście implementacja podpisu cyfrowego dostarczana przez .NET Framework byłaby połowiczna, gdyby nie dawała możliwości weryfikacji wygenerowanego podpisu cyfrowego.

```
const string OID_SHA1 = "1.3.14.3.2.26";
const strign OID_MD5 = "1.2.840.113549.2.5";
private bool verifyRSASignature (byte[] data,
  byte[] signature, RSAParameters publicKey)
{
  HashAlgorithm hash = SHA1.Create();
  RSACryptoServiceProvider rsa
    = new RSACryptoServiceProvider();
  rsa.ImportParameters(publicKey);
  byte[] hashValue = hash.ComputeHash();
  return rsa.VerifyHash(hashValue, OID_SHA1, signature);
}
```

153

Podpis cyfrowy (4)

Istnieje również możliwość generowania i weryfikacji podpisów za pomocą prostszego interfejsu – niewymagającego jawnego tworzenia skrótów.

```
const string SIMPLE_NAME_SHA1 = "SHA1";
const strign SIMPLE_NAME_MD5 = "MD5";
private byte[] GenerateRSASignature (byte[] data,
  RSAParameters privateKey)
{
  RSACryptoServiceProvider rsa
    = new RSACryptoServiceProvider();
  rsa.ImportParameters(privateKey);
  return rsa.SignData(data, SIMPLE_NAME_SHA1);
}
private bool VerifyRSASignature (byte[] data,
  RSAParameters publicKey, byte[] signature)
{
  RSACryptoServiceProvider rsa
    = new RSACryptoServiceProvider();
  rsa.ImportParameters(publicKey);
  return rsa.VerifyData(data, signature);
}
```

154

Podpis cyfrowy (5)

Tytuł najpowszechniej wykorzystywanego algorytmu asymetrycznego wciąż należy do RSA, który raczej powoli ustępuje miejsca nieco młodszemu DSA.

Microsoft .NET Framework – podobnie jak Java™ – daje programiście możliwość wykorzystania algorytmu DSA we własnych aplikacjach.

DSA jest algorytmem opracowanym przez NIST specjalnie z myślą o składaniu i weryfikacji podpisów.

Zapewne to jest przyczyną, czemu API .NET Framework nie przewiduje wykorzystania tego algorytmu do zapewnienia poufności, mimo iż zostały opracowane mechanizmy umożliwiające wykorzystanie DSA do szyfrowania.

155

Podpis cyfrowy (6)

```
private byte[] GenerateDSASignature (byte[] data,
  DSAParameters privateKey)
{
  SHA1 sha1 = SHA1.Create();
  DSA dsa = new DSACryptoServiceProvider();
  dsa.ImportParameters(privateKey);
  byte[] hashValue = sha1.ComputeHash(data);
  return dsa.CreateSignature(hashValue);
}
private bool VerifyDSASignature (byte[] data, byte[] signature,
  DSAParameters publicKey)
{
  SHA1 sha1 = SHA1.Create();
  DSA dsa = new DSACryptoServiceProvider();
  dsa.ImportParameters(publicKey);
  byte[] hashValue = sha1.ComputeHash(data);
  return dsa.VerifySignature(hashValue, signature);
}
```

156

XML-Signature i XML-Encryption

.NET Framework daje twórcy aplikacji możliwość (1) podpisywania oraz (2) szyfrowania wybranych elmentów dokumentów XML.

Tak zabezpieczone fragmenty treści zostaną utrwalone w formacie zgodnym z opracowanymi w ramach konsorcjum W3C standardami XML-Signature oraz XML-Encryption.

Specyfikacje XML-Signature i XML-Encryption stanowią fundament dla:
- specyfikacji rozszerzających funkcjonalność Simple Object Access Protocol (SOAP) realizujących pozafunkcjonalne wymaganie związane z bezpieczeństwem, takich jak WS-Security, WS-SecureConversation, WS-Trust, ...
- standardu XML Advanced Electronic Signature (XAdES) opisującego format zapisu danych podpisanych (*signed data*), czy koperty cyfrowej (*enveloped data*). XAdES stanowi wśród specyfikacji W3C odpowiednik standardu PKCS#7, bądź jego rozszerzenia opublikowanego w postaci RFC2630 Cryptographic Message Syntax.

157

SOAP – fałszywy przełom (1)

XML stał się fundamentem dla protokołu SOAP stanowiącego uniwersalny mechanizm komunikacji umożliwiający integrację rozwiązań heterogenicznych.

SOAP nie jest pierwszą próbą zmierzenia się problemem heterogeniczności. Wręcz przeciwnie można odnieść wrażenie, że specyfikacje tworzone wokół SOAP są ponowną próbą „wynalezienia koła” – jedyna różnica polega na użyciu innego „materiału” do jego budowy.

Bardzo spójnym rozwiązaniem problemu heterogeniczności – bolączki współczesnych systemów informatycznych jest np. OMG CORBA. Przyczyn niepowodzenia CORBA nie należy szukać w błędach w specyfikacji.

W przypadku SOAP wykorzystanie XML jako formatu zapisu komunikatów również budzi wiele kontrowersji i stało się przyczyną podjęcia prac nad tzw. Fast Web Services, w których w roli „medium transportowego” miejsce XML zajęła leciwa notacja ASN.1.

158

SOAP – fałszywy przełom (2)

Zasadnicza różnica między SOAP a CORBA polega na pewnym triku użytym podczas tworzenia specyfikacji.

Twórcy CORBA podeszli do opracowania specyfikacji w sposób tradycyjny, tj. wszystkie elementy zamieścili w treści jednego standardu, który z czasem rozrósł się do niebotycznych rozmiarów.

I to właśnie przesądziło o upadku CORBA – powstała opasła specyfikacja, dla której nikt nie podjął się stworzyć pełnej implementacji, co przyczyniło się do ogólnego zniechęcenia do tego standardu.

Popularność SOAP wyrosła właśnie na bazie rozczarowania specyfikacją CORBA.

Doświadczeni przykładem CORBA twórcy SOAP najprawdopodobniej zdawali sobie sprawę, że każda specyfikacja techniczna mająca ambicje objąć jak najwięcej aspektów jest prędzej, czy później skazana na przegraną.

Z drugiej strony problem heterogeniczności jest na tyle złożony, że chyba tylko osoba całkowicie niezdająca sobie sprawy ze skali tego zagadnienia może liczyć, że uda się go rozwiązać prostymi mechanizmami.

159

SOAP – fałszywy przełom (3)

Chyba dlatego autorzy SOAP posłużyli się pewnym wybiegiem.
Postanowili bowiem stworzyć możliwie elastyczną i prostą specyfikację podstawową, na bazie której będą powstawać standardy odnoszące się do kolejnych problemów związanych z heterogenicznością.

W ten sposób w kopercie (*envelope*) SOAP pojawiły się dwa główne elementy
- tzw. ciało (*body*) zawierające właściwą treść komunikatu (*payload*) – czyli wywołanie zdalnej metody, oraz
- nagłówek (*header*) – bliżej nieokreślona zawartość, która z czasem zacznie uwzględniać kolejne aspekty problemu heterogeniczności.

Zamiast jednego spójnego standardu zaczęły dookoła SOAP powstawać kolejne specyfikacje odnoszące się do różnych problemów wynikających z niejednorodności systemów informatycznych.

Wszystkie rozszerzenia SOAP określają wyłącznie zawartość nagłówka, bez ingerowania w treść wywołania.

160

SOAP – fałszywy przełom (4)

Wobec w ten sposób opracowywanych specyfikacji tworzonych pod egidą konsorcjum W3C używa się wspólnej nazwy: WS-* specifications.

Obecnie WS-* specifications to cała rodzina wzajemnie zależnych specyfikacji odnoszących się do różnych problemów związanych z integracją systemów rozproszonych:

- bezpieczeństwa: XML-Encryption, XML-Signature, WS-Security, WS-SecureConversation, WS-Trust, WS-Federation, ...;
- niezawodnego transportu i wywołań asynchronicznych: WS-ReliableMessaging, WS-Reliability;
- transakcyjności: WS-Transaction, WS-AtomicTransaction, WS-Coordination, WS-BusinessActivity, ...;
- wsparcia dla paradygmatu zdarzeniowego: WS-BaseNotification, WS-Eventing;
- opisu sposobu dostępu do udostępnianej funkcjonalności: WS-Policy, WS-MetadataExchange, ...;
- ...

Model komunikacji SOAP

Model komunikacji opisany w specyfikacji SOAP zakłada zestawianie połączeń między dwoma „końcami” (*end-to-end conversations*).

Zasadniczą różnicą między modelem komunikacji SOAP a tradycyjnych połączeniach między dwoma punktami (*point-to-point communication*) jest możliwość ingerencji w treść koperty podczas transportu.

Tzw. węzły pośredniczące (*intermediaries*) mogą modyfikować zawartość koperty SOAP – ale tylko część nagłówkową decydującą o zapewnieniu wymagań pozafunkcjonalnych określonych w politykach poszczególnych usług.

Ciało komunikatu powinno w niezmienionej postaci dotrzeć do końcowego odbiorcy (*ultimate receiver*) komunikatu.

Analogicznie treść odpowiedzi usługi powinna dotrzeć bez zmian do początkowego nadawcy (original sender).

W3C Exclusive XML Canonicalization

Tzw. postać kanoniczna to dokument XML pozbawiony elementów nie wpływających na reprezentację w postaci struktury w pamięci operacyjnej.

Postać kanoniczna zakłada m.in. (1) usunięcie zbędnych białych znaków, (2) informacji, że poszczególne węzły zawierają wartości domyślne, lub opcjonalnie (3) usunięcie komentarzy.

Celem kanoninizacji jest zapewnienie możliwości poruszania się po dokumencie XML za pomocą wyrażeń ścieżkowych XPath.

XML-Signature lokalizuje podpisane elementy za pomocą wyrażeń ścieżkowych XPath.

Umieszczenie dokumentu podpisanego wewnątrz innego elementu – np. w wyniku przetwarzania nagłówka SOAP przez węzeł pośredniczący spowoduje, że zmieni się kontekst, w którym został umieszczony oryginalny dokument
Tym samym nie będzie możliwa weryfikacja podpisu.

Wspomiany problem znalazł rozwiązanie dzięki rekomendacji W3C Exclusive XML Canonicalization, która uniezależnia składanie oraz weryfikację podpisu od kontekstu użycia.

163

Public Key Cryptography Standards (1)

Public Key Cryptography Standards (PKCS) to rodzina w chwili obecnej kilkunastu standardów rozwijanych przez firmę RSA Security Inc. przy współpracy z innymi liczącymi się na świecie dostawcami oprogramowania takimi jak: Microsoft, Sun Microsystems, Apple, ...

RSA Security Inc. została założona przez twórców algorytmu RSA: Rona Rivesta, Adiego Shamira, oraz Lena Adlemana i jest właścicielem patentu na algorytm RSA.

Celem PKCS jest upowszechnienie i standaryzacja wykorzystania kryptografii wśród dostawców oprogramowania.

Należy liczyć się, że każdy, kto w swojej pracy zawodowej musi uwzględniać bezpieczeństwo informacji prędzej, czy później zetknie się ze standardami z rodziny PKCS.

Treść specyfikacji należących do PKCS dotyczy:
- algorytmów,
- formatu wymiany danych między systemami informatycznymi,
- interfejsu dostępu do urządzeń kryptograficznych.

164

Public Key Cryptography Standards (2)

PKCS#1	mechanizmy szyfrowania i podpisywania przy pomocy RSA
PKCS#3	algorytm uzgadniania kluczy Diffie-Hellman
PKCS#5	schemat szyfrowania przy użyciu hasła (*password-based encryption*) – tworzenie kluczy tajnych na podstawie hasła
PKCS#6	format certyfikatu klucza publicznego – zastąpiony przez standard X.509
PKCS#7	format wiadomości kryptograficznych – rozszerzony przez RFC 2630 (CMS) oraz wykorzystany w RFC 2634 (S/MIME)
PKCS#8	format zapisu klucza prywatnego
PKCS#9	specyfikacja atrybutów dołączanych do danych kryptograficznych wykorzystywanych w pozostałych standardach
PKCS#10	format wniosku certyfikacji (*certification signing request*)
PKCS#11	specyfikacja interfejsu dostępu do tzw. żetonów przechowujących wrażliwe dane kryptograficzne (kart chipowych, bądź HSM)

165

Public Key Cryptography Standards (3)

PKCS#12	format wymiany tzw. osobistych danych kryptograficznych (kluczy prywatnych, bądź łańcuchów certyfikatów)
PKCS#13	specyfikacja mechanizmów szyfrowania i podpisywania za pomocą algorytmów zaliczanych do kryptografii krzywych eliptycznych ECC (Ellyptic Curve Cryptography)
PKCS#14	specyfikacja mechanizmu generatora liczb pseudolosowych (Pseudo-Random Number Generator – PRNG)
PKCS#15	zawartość oraz format zapisu informacji przechowywanej na żetonach kryptograficznych – zwłaszcza danych wykorzystywanych w procesie uwierzytelnienia użytkowników

Standardy PKCS#2 oraz PKCS#4 stały się z czasem częścią specyfikacji PKCS#1.

166

Cryptographic Message Syntax (1)

Format zapisu wiadomości kryptograficznej został opisany w specyfikacji Cryptographic Message Syntax opatrzonej wśród standardów PKCS numerem 7.

Standard PKCS#7 w wersji 1.5 został zaadoptowany jako oficjalny standard społeczności internetowej (The Internet Society) i opublikowany w postaci RFC 2315.

Z czasem PKCS#7 (RFC 2315) doczekał się uzupełnienia opracowanego przez jedną z grup roboczych zaangażowanych w proces standaryzacyjny, które zostało opublikowane jako RFC 2360.

Standard CMS definiuje dwa główne rodzaje wiadomości kryptograficznych: (1) dane podpisane (*signed data*), oraz (2) kopertę cyfrową (*enveloped data*).

Część specyfikacji CMS poświęcona danym podpisanym określa format zapisu podpisu cyfrowego zarówno nierozerwalnie zintegrowanego z danymi, jak również występującego w postaci oddzielonej od danych.

167

Cryptographic Message Syntax (2)

Koperta cyfrowa określa format zapisu danych zaszyfrowanych z wykorzystaniem hybrydowego schematu szyfrowania.

Zgodnie ze standardem koperty cyfrowej CMS właściwa treść jest zabezpieczona przy użyciu klucza tajnego, który z kolei jest dołączany do zawartości koperty w postaci zaszyfrowanej kluczem publicznym umieszczonym w certyfikacie odbiorcy wiadomości.

Ta sama koperta cyfrowa CMS może być adresowana do wielu odbiorców bez jakiejkolwiek redundancji danych.

Dzieje się tak, ponieważ treść wiadomości jest zaszyfrowana przy pomocy jednego klucza tajnego.

Każda koperta CMS zawiera informację o adresatach (*recipient info*) stanowiącą sekwencję ściśle określonej struktury danych obejmującą m.in.

- certyfikat X.509 danego odbiorcy, oraz
- klucz tajny wykorzystany wcześniej do zabezpieczenia koperty zaszyfrowany kluczem publicznym umieszczonym w certyfikacie odbiorcy.

168

Cryptographic Message Syntax (3)

Oczywiście samo zabezpieczanie treści wiadomości powinno być wykonywane w trybie utrudniającym przeprowadzenie ataku statystycznego (*repeat attack*).

Trybem najczęściej wykorzystywanym podczas szyfrowania tekstu jawnego komunikatu zabezpieczanego za pomocą koperty CMS jest Cipher Block Chaining Mode (CBC).

Poza tradycyjną kopertą cyfrową standard CMS definiuje również kopertę podpisaną umożliwiającą zapewnienie zarówno poufności, jak również integralności i niezaprzeczalności danych.

Poza wymienionymi rodzajami komunikatów CMS specyfikuje również format zapisu:
- danych wzbogaconych o skrót wiadomości (*digested data*), oraz
- danych zaszyfrowanych (*encrypted data*) – bez jakiejkolwiek informacji o kluczu użytym podczas zabezpieczania treści.

Wsparcie dla XML-Encryption w .NET (1)

W celu zaszyfrowania dowolnego elementu dokumentu XML należy:
- za pomocą metody `Load` zbudować w pamięci strukturę odpowiadającą zawartości dokumentu XML;
- wybrać węzeł drzewa DOM, którego zawartość chcemy zabezpieczyć;
- zaszyfrować treść węzła;
- podmienić oryginalny węzeł elementem zaszyfrowanym

```
/// load XML document content
XmlDocument xml = new XmlDocument();
xml.Load("personal-data.xml");

/// search for the XML document element you want to encrypt
XmlElement elem
  = xml.GetElementsByTagName("Name")[0] as XmlElement;

/// instantiate cipher implementation
Rijandel aes = new RijandelManaged();

/// to be continued on the following slide
```

Wsparcie dla XML-Encryption w .NET (2)

```
/// reveal the content secured with XML-Encryption
bool REVEAL_ENCRYPTED_CONTENT = true;

/// create an XML document element which constitutes the
/// encrypted version of the the selected node
EncryptedXml encEl = new EncryptedXml(xml);
byte[] encDataRaw = encEl.EncryptData(elem, aes,
  REVEAL_ENCRYPTED_CONTENT);
EncryptedData encData = new EncryptedData();
encData.Type = EncryptedXml.XmlEncElementUrl;
encData.EncryptionMethod = new EncryptionMethod(
  EncryptedXml.XmlEncAES256Url);
encData.CipherData.CipherValue = encDataRaw;

/// replace the original element with the encrypted one
EncryptedXml.ReplaceElement(el, encData,
  REVEAL_ENCRYPTED_CONTENT);

/// store modified XML document
xml.Save("personal-data-secured.xml");
```

171

Wsparcie dla XML-Encryption w .NET (3)

Deszyfrowanie zabezpieczonych elementów przebiega w sposób symetryczny:

- w pierwszej kolejności należy załadować dokument XML;
- odpowiednio zainicjalizować instancję implementacji algorytmu szyfrowania;
- odszukać element zabezpieczony za pomocą XML-Encryption;
- zdeszyfrować zawartość wybranego węzła;
- zastąpić węzeł zabezpieczony jego wersją zdeszyfrowaną.

```
/// load XML document
XmlDocument xml = new XmlDocument();
xml.Load("personal-data-secured.xml");

/// initialize encryption algorithm
Rijandel aes = new RijandelManaged();
aes.Key = key;

/// select encrypted element
XmlElement encEl
  = xml.GetElementByTagName("EncryptedData")[0] as XmlElement;

/// to be continued on the following slide
```

172

Wsparcie dla XML-Encryption w .NET (4)

```
/// get content of the selected encrypted element
EncryptedData encData = new EncryptedData();
encData.LoadXml(encEl);
EncryptedXml encXml = new EncryptedXml();

/// decrypt secured content
byte[] elData = encXml.DecryptData(encData, aes);

/// replace the secured node with its decrypted version
encXml.ReplaceData(encEl, elData);

/// store changes
xml.Save("personal-data.xml");
```


Wsparcie dla XML-Encryption w .NET (5)

Zabezpieczenie zawartości wybranego elementu, bądź elmentów z wykorzystaniem hybrydowego schematu szyfrowania jest nieco bardziej złożone ze względu na konieczność dołączenia zaszyfrowanego klucza tajnego.

```
/// load XML document
XmlDocument xml = new XmlDocument();
xml.Load("personal-data.xml");

/// initialize asymmetric encryption algorithm
string KEY_CONTAINER_NAME = "Key Container";
CspParameters params = new CspParameters();
params.KeyContainerName = KEY_CONTAINER_NAME;
RSA rsa = new RSACryptoServiceProvider(params);

/// initialize cipher
Rijandel aes = new RijandelManaged();
aes.Key = key;

/// to be continued on the following slide
```

Wsparcie dla XML-Encryption w .NET (6)

```
/// select encrypted element
XmlElement encEl
  = xml.GetElementByTagName("Name")[0] as XmlElement;

/// create an XML document element which constitutes the
/// encrypted version of the the selected node
EncryptedXml encEl = new EncryptedXml(xml);
byte[] encDataRaw = encEl.EncryptData(elem, aes,
  REVEAL_ENCRYPTED_CONTENT);
EncryptedData encData = new EncryptedData();
encData.Type = EncryptedXml.XmlEncElementUrl;
encData.EncryptionMethod = new EncryptionMethod(
  EncryptedXml.XmlEncAES256Url);
encData.CipherData.CipherValue = encDataRaw;

/// to be continued on the following slide
```


Wsparcie dla XML-Encryption w .NET (7)

```
/// create an encrypted element which holds secured secret key
EncryptedKey encKey = new EncryptedKey();
byte[] encKeyData = EncryptedXml.EncryptKey(aes.Key, rsa,
  false);
encKey.EncryptionMethod = new EncryptionMethod(
  EncryptedXml.XmlEncRSA15Url);

/// attach public key identifier
string ID_KEY_PAIR = "alias";
KeyInfoName keyInfo = new KeyInfoName();
keyInfo.Value = ID_PUBLIC_KEY;
encKey.KeyInfo.AddClause(keyInfo);

/// attach the secret key to the secured element
encData.KeyInfo.AddClause(new KeyInfoEncryptedElement(encKey));

/// replace the original element with the encrypted one
EncryptedXml.ReplaceElement(el, encData,
  REVEAL_ENCRYPTED_CONTENT);
/// store modified XML document
```

Wsparcie dla XML-Encryption w .NET (8)

Ten sam klucz tajny możne być użyty do zabezpieczenia wielu elementów w dokumencie – w takim wypadku podczas szyfrowania pierwszego elementu należy dodać do klucza identyfikator.

```
...

/// create an encrypted element which holds secured secret key
/// and set the key id so that it could be reused while
/// encrypting other document nodes
EncryptedKey encKey = new EncryptedKey();
byte[] encKeyData = EncryptedXml.EncryptKey(aes.Key, rsa,
  false);
encKey.EncryptionMethod = new EncryptionMethod(
  EncryptedXml.XmlEncRSA15Url);
string SECRET_KEY_ID = "Secret Key";
encKey.Id = SECRET_KEY_ID;

/// to be continued on the following slide
```

177

Wsparcie dla XML-Encryption w .NET (9)

Kolejny element jest wyszukiwany i zabezpieczany w ten sam sposób co dotychczas.

```
...
/// select next document node you want to encrypt
encEl
  = xml.GetElementByTagName("Address")[0] as XmlElement;

/// create an XML document element which constitutes the
/// encrypted version of the the selected node
encEl = new EncryptedXml(xml);
byte[] encDataRaw = encEl.EncryptData(elem, aes,
  REVEAL_ENCRYPTED_CONTENT);
encData = new EncryptedData();
encData.Type = EncryptedXml.XmlEncElementUrl;
encData.EncryptionMethod = new EncryptionMethod(
  EncryptedXml.XmlEncAES256Url);

/// to be continued on the next slide
```

178

Wsparcie dla XML-Encryption w .NET (10)

Tym razem nie tworzymy elementu przechowującego klucz tajny, a jedynie odnosimy się do wcześniej utworzonego węzła.

Bez zmian odbywa się również dodanie informacji o kluczu tajnym oraz zastąpienie oryginalnego węzła.

```
/// refer to the existing node which holds secured secret key
EncryptedKey encKeyRef = new EncryptedKey();
DataReference refData = new DataReference();
string HASH = "#";
refData.Uri = HASH + encKey.Id;
encKeyRef.AddReference(refData);

/// attach the secret key to the secured element
encData.KeyInfo.AddClause(new KeyInfoEncryptedElement(encKey));

/// replace the original element with the encrypted one
EncryptedXml.ReplaceElement(el, encData,
  REVEAL_ENCRYPTED_CONTENT);
```

179

Wsparcie dla XML-Encryption w .NET (11)

Deszyfrowanie tak zabezpieczonego dokumentu jest wyjątkowo proste – po utworzeniu drzewa DOM odpowiadającego zawartości dokumentu XML wystarczy jedynie wywołać metodę **`DecryptDocument`**.

```
/// load XML document
XmlDocument xml = new XmlDocument();
xml.Load("personal-data-secured.xml");

/// create an instance representing an XML document secured
/// with XML-Encryption
EncryptedXml encXml = new EncryptedXml(xml);

/// decrypt the content of the XML document
/// NOTE: you need to initialize the RSA instance with an
/// appropriate private key first
encXml.AddKeyElementMapping(ID_PUBLIC_KEY, rsa);
encXml.DecryptDocument();

/// store decrypted XML document
xml.Save("personal-data.xml");
```

180

Wsparcie dla XML-Encryption w .NET (12)

W praktyce źródłem kluczy publicznych, za pomocą których zabezpieczana jest treść dokumentów są najczęściej certyfikaty X.509 zapisane w magazynie kluczy nadawcy wiadomości.

Przyjmując za główne kryterium łatwość użycia interfejsu programistycznego można uznać, że szyfrowanie za pomocą kluczy publicznych zapisanych w certyfikatach X.509 jest zasadniczo podstawowym sposobem zabezpieczania treści dokumentów w .NET Framework.

```
/// (1) load XML document, (2) select the node you want to
/// secure, (3) extract X.509 certificate either from the
/// system store, or from a file
...
EncryptedXml encXml = new EncryptedXml();
EncryptedData encData = encXml.Encrypt(el, x509Cert);
EncryptedXml.ReplaceElement(el, encData, false);

/// store the document
```

181

Wsparcie dla XML-Encryption w .NET (13)

Deszyfrowanie fragmentów dokumentu zabezpieczonych przy użyciu certyfikatu X.509 nastręcza jeszcze mniej problemów niż deszyfrowanie za pomocą klucza prywatnego.

Nie ma potrzeby inicjalizacji instancji algorytmu asymetrycznego – .NET Framework sam podejmie próbę pobrania odpowiedniego klucza prywatnego z osobistego magazynu kluczy bieżąco zalogowanego użytkownika.

```
XmlDocument xml = new XmlDocument():
xml.Load("personal-data-secured.xml");
xml.DecryptDocument();
xml.Save("personal-data.xml");
```

182

Wsparcie dla XML-Signature w .NET (1)

Poza szyfrowaniem wybranych elementów dokumentu XML .NET Framework daje twórcy rozwiązania możliwość fragmentów dokumentu zgodnie ze specyfikacją XML-Signature.

```
XmlDocument xml = new XmlDocument();
xml.Load("personal-data.xml");
xml.PreserveWhitespace = false;

RSA rsa = new RSACryptoServiceProvider();
/// create representation of a signed XML (sub-) document
SignedXml signedXml = new SignedXml(xml);
signedXml.SigningKey = rsa;

/// determine the element to sign and add transform
Reference ref = new Reference();
reference.Uri = "";
XmlDsigEnvelopedSignatureTransform transform
  = new XmlDsigEnvelopedSignatureTransform();
ref.AddTransform(tranform);
signedXml.AddReference(ref);
```

183

Wsparcie dla XML-Signature w .NET (2)

```
/// set key info
KeyInfo keyInfo = new KeyInfo();
keyInfo.AddClause(new RSAKeyValue(rsa));
signedXml.KeyInfo = keyInfo;

/// generate signature
signedXml.ComputeSignature();

/// extract signature
XmlElement signature = signedXml.GetXml();

/// append signature element to the document
xml.DocumentElement.AppendChild(xml.ImportNode(signature,
  true));

/// and finally store the XML document
```

Niestety umieszczenie dokumentu XML podpisanego w sposób opisany powyżej w tzw. kopercie uniemożliwi pomyślną weryfikację jego autentyczności.

184

Wsparcie dla XML-Signature w .NET (3)

Jak wiemy problem weryfikacji podpisów złożonych na fragmentach dokumentu umieszczonego już po podpisaniu wewnątrz tzw. koperty został rozwiązany w rekomendacji określającą „odłączoną” postać kanoniczną.

Interfejs programistyczny .NET Framework daje programiście możliwość zapisu podpisanego dokumentu XML również w formacie zgodnym z Exclusive XML Canonicalization.

```
...
/// create representation of a signed XML (sub-) document
SignedXml signedXml = new SignedXml(xml);
signedXml.SigningKey = rsa;
signedXml.SignedInfo.CanonicalizationMethod
  = SignedXml.XmlDsigExcC14NTTransformUrl;
XmlDsigExcC14NTTransform excTranform
  = signedXml.SignedInfo.CanonicalizationMethodObject
    as XmlDsigExcC14NTTransform;
...

/// to be continued on the following slide
```

185

Wsparcie dla XML-Signature w .NET (4)

```
...
/// determine the element to sign and add transform
Reference ref = new Reference();
reference.Uri = "";
XmlDsigEnvelopedSignatureTransform transform
  = new XmlDsigEnvelopedSignatureTransform();
ref.AddTransform(excTranform);
ref.AddTransform(tranform);
signedXml.AddReference(ref);
...
```

186

Wsparcie dla CMS w .NET (1)

Interfejs programistyczny .NET Framework umożliwia programiście zabezpieczenie treści w postaci wiadomości kryptograficznych zgodnych z formatem zdefiniowanym w standardzie Cryptographic Message Syntax.

Twórca aplikacji może – w zależności od konkretnych potrzeb:

- zapewnić integralność i niezaprzeczalność danych poprzez utworzenie wiadomości podpisanej (*signed data*) CMS reprezentowanej w API .NET Framework przez klasę **`SignedCms`**, bądź
- zagwarantować poufność treści poprzez umieszczenie jej w kopercie cyfrowej (*enveloped data*) CMS reprezentowanej przez klasę **`EnvelopedCms`**.

187

Wsparcie dla CMS w .NET (2)

```
/// create CMS signed data
private byte[] CreateCMSSignedData (byte[] input,
  X509Certificate2 signerEntry)
{
  ContentInfo content = new ContentInfo(input);
  SignedCms signedData = new SignedCms(content);
  CmsSigner singer = new CmsSigner(signerEntry);
  signedData.ComputeSignature();
  return signedData.Encode();
}
```

```
/// verify CMS signed data signature
private boolean CreateCMSSignedData (byte[] signedInput) {
  SignedCms signedData = new SignedCms();
  signedData.Decode(signedInput);
  /// NOTE: verification of the signature attached to CMS
  /// message is highly integrated with the system certificate
  /// store
  return signedData.CheckSignature(true);
}
```

188

Wsparcie dla CMS w .NET (3)

Przekazanie wartości **`true`** jako argumentu do metody **`CheckSignature`** oznacza, że nie jest wykonywana zarówno walidacja, ani też weryfikacja ścieżki zaufania do certyfikatu podmiotu, który złożył podpis.

Tworzenie koperty cyfrowej CMS przy użyciu interfejsu programistycznego .NET Framework przebiega równie intuicyjnie jak omówione wcześniej tworzenie wiadomości podpisanej.

```
/// create CMS enveloped data
byte[] CreateCMSEnvelopedData (byte[] input,
  X509Certificate2 recipientEntry)
{
  ContentInfo content = new ContentInfo(input);
  EnvelopedCms envelopedData = new EnvelopedCms(content);
  CmsRecipient recipient = new CmsRecipient(
    SubjectIdentifierType.IssuerAndSerialNumber,
    recipientEntry);
  envelopedData.Encrypt(recipient);
  return envelopedData.Encode();
}
```

189

Wsparcie dla CMS w .NET (4)

.NET Framework daje również możliwość utworzenia koperty adresowanej do wielu odbiorców.

```
/// create CMS enveloped data for several recipients
byte[] CreateCMSEnvelopedData (byte[] input,
  X509Certificate2Collection recipientEntries)
{
  ContentInfo content = new ContentInfo(input);
  EnvelopedCms envelopedData = new EnvelopedCms(content);
  CmsRecipientCollection recipients
    = new CmsRecipientCollection(
        SubjectIdentifierType.IssuerAndSerialNumber,
        recipientEntries);
  envelopedData.Encrypt(recipients);
  return envelopedData.Encode();
}
```

190

Wsparcie dla CMS w .NET (5)

Ze względu na silną integrację implementacji CMS dostępnej w .NET Framework odszyfrowanie zawartości koperty wymaga umieszczenia pary: klucz prywatny, certyfikat w systemowym magazynie przechowującym osobiste certyfikaty bieżąco zalogowanego użytkownika.

```
/// decrypt CMS enveloped data
/// NOTE: decrypting the content of CMS envelope is performed
/// in the same fashion regardless whether the envelope has
/// been prepared for one or many recipients
byte[] CreateCMSEnvelopedData (byte[] envelopedInput)
{
  EnvelopedCms envelopedData = new EnvelopedCms();
  envelopedData.Decode(envelopedInput);
  envelopedData.Decrypt(envelopedData.RecipientInfos[0]);
  return envelopedData.Encode();
}
```

191

Wsparcie dla CMS w .NET (6)

Łącząc funkcjonalność umożliwiającą generowanie wiadomości podpisanej oraz koperty cyfrowej możemy utworzyć kopertę podpisaną zgodną z formatem zdefiniowanym w standardzie CMS.

Tworzenie kopertę podpisaną można utworzyć na dwa sposoby – najpierw zaszyfrować dane, a dopiero później je podpisać, bądź też wykonać wspomniane dwie czynności w kolejności odwrotnej.

O właściwej kolejności kroków powinien decydować efekt, który chcą uzyskać twórcy rozwiązania .

Obie metody tworzenia kopert podpisanych są uważane za równie bezpieczne – co najwyżej może być większy, bądź mniejszy sens ich stosowania w konkretnych okolicznościach.

192

Wsparcie dla CMS w .NET (7)

Tradycyjnie przyjęło się, że najpierw dane są podpisywane, a dopiero potem szyfrowane.

Wynika to z ogólnego zalecenia, by raczej podpisywać tekst jawny dając tym samym odbiorcy możliwość weryfikacji autentyczności danych i walidacji podpisu w oparciu o niezabezpieczoną treść komunikatu.

Dość łatwo możemy sobie jednak wyobrazić sytuację, w której bardziej właściwa może okazać się kolejność odwrotna.

Dane powinniśmy w pierwszej kolejności szyfrować, a dopiero później podpisywać m.in. wówczas, gdy zależy nam na możliwie bezzwłocznym odrzucaniu:

- wszystkich wiadomości, których treść została zmanipulowana, bądź
- wiadomości które zostały podpisane za pomocą nieważnego certyfikatu (błąd walidacji).

193

Wsparcie dla CMS w .NET (8)

```
/// create enveloped signed data complying with CMS
/// specificationdecrypt CMS enveloped data
byte[] CreateCMSEnvelopedSignedData (byte[] input,
  X509Certificate2 signerEntry,
  X509CertificateCollection recipientEntries)
{
  /// create CMS signed data
  ContentInfo content = new ContentInfo(input);
  SignedCms signedData = new SignedCms(content);
  CmsSigner signer = new CmsSigner(signerEntry);
  signedData.ComputeSignature(signer);

  ...
  /// to be continued on the following slide
```

194

Wsparcie dla CMS w .NET (9)

```
  ...

  /// set signed data as the input for the enveloped data
  content = new ContentInfo(signedData.Encode());

  /// create CMS envelope based on already built signed data
  EnvelopedCms envelopedData = new EnvelopedCms(content);
  CmsRecipientCollection recipients =
    new CmsRecipientCollection(
      SubjectIdentifierType.IssuerAndSerialNumber,
      recipientEntries);
  envelopedData.Encrypt(recipients);
  return envelopedData.Encode();
} /// CreateCMSEnvelopedSignedData
```


Wsparcie dla CMS w .NET (10)

Odczyt zawartości koperty podpisanej obejmujący weryfikację autentyczności treści przebiega w sposób symetryczny do jej powstania.

```
/// decrypt and verify authenticity of CMS enveloped signed
/// data
byte[] DecryptAndVerifyCMSEnvelopedSignedData (byte[] input) {

  EnvelopedCms envelopedData = new EnvelopedCms();
  envelopedData.Decode(input);
  envelopedData.Decrypt(envelopedData.RecipientInfos[0]);

  SignedCms signedData = new SignedCms();
  signedData.Decode(envelopedData.Encode());
  return signedData.CheckSignature(true);
}
```