IDŹ DO

PRZYKŁADOWY ROZDZIAŁ

SPIS TREŚCI

KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK

DODAJ DO KOSZYKA

CENNIK I INFORMACJE

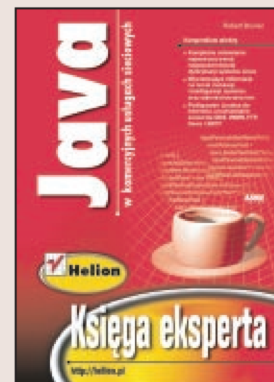
ZAMÓW INFORMACJE O NOWOŚCIACH

ZAMÓW CENNIK

CZYTELNIA

FRAGMENTY KSIĄŻEK ONLINE

Java w komercyjnych usługach. Księga eksperta



Autor: Robert Bruner
Tłumaczenie: Adam Fiącek, Cezary Welsyng
ISBN: 83-7197-779-4
Tytuł oryginału: Java Web Services Unleashed
Format: B5, stron: 660
Przykłady na ftp: 6323 kB

Usługi WWW to ostatni krzyk programistycznej mody. W największym skrócie polegają one na udostępnianiu w sieci Internet dynamicznych aplikacji sieciowych. Ten jednozdaniowy opis nie przekazuje całego potencjału usług WWW. Jest on ogromny. Dzięki usługom WWW aplikacje łączą się na niespotykaną do tej pory, globalną skalę. Już obecnie skorzystać można z setek usług WWW. Dzięki nim uzyskasz między innymi informacje finansowe, przeszukasz Internet, przeprowadzisz konwersję między różnymi formatami dokumentów, a nawet rozegrasz partię szachów z komputerem.

Java — dzięki doskonałemu wsparciu dla XML-a jest jedną z najlepszych platform do implementacji usług WWW. Książka, którą trzymasz w ręku, to wyczerpujące, dokładne i — co równie ważne — aktualne kompendium, zawierające informacje niezbędne, by wykorzystać istniejące i stworzyć nowe usługi WWW.

W książce przedstawiono między innymi:

- Przykłady zastosowań usług WWW
- Biznesowe aspekty tworzenia usług WWW
- WDSL — język opisu usług sieciowych
- SOAP — protokół wymiany komunikatów
- UDDI — format katalogowania usług WWW
- Wykorzystanie pakietu JAX do tworzenia usług WWW
- Zagadnienia związane z bezpieczeństwem
- Dodatkowe formaty używane w kontekście usług WWW: WSFL, WSIF
- Przykładowe implementacje usług WWW

Księga eksperta kierowana jest średnio zaawansowanym i zaawansowanych programistom, którzy pragną poznać najnowocześniejsze technologie, przedstawione w sposób kompletny i wyczerpujący.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

O Autorach ... 13

Wstęp ... 17

Część I Wprowadzenie do usług sieciowych ... 21

Rozdział 1. Czym są usługi sieciowe? ... 23

B2B to tak naprawdę A2A ... 23
Składanie elementów w jedną całość ... 24
Ideologiczne wojny bez zwycięzców ... 24
Zgodność operacyjna dla każdego ... 25
Wszystko się zmienia ... 25
Czym są usługi sieciowe? ... 26
Prosta definicja ... 26
Szersza perspektywa ... 27
Zasięg usług sieciowych ... 28
Znaczenie technologii usług sieciowych ... 29
XML (eXtensible Markup Language) ... 29
WSDL (Web Services Definition Language) ... 30
UDDI (Universal Description Discovery and Integration) ... 31
SOAP (Simple Object Access Protocol) ... 32
ebXML (Electronic Business XML) ... 33
Podsumowanie ... 34

Rozdział 2. Internet i usługi sieciowe — nowe środowisko biznesowe ... 35

Nowe aspekty znanych rozwiązań ... 36
Tworzenie metamodelu przedsiębiorstwa ... 37
Techniczny aspekt atrakcyjności usług sieciowych ... 38
Uniwersalne spoiwo ... 38
Sprzężenie a niezależność ... 38
Tworzenie definicji z zastosowaniem metadanych ... 39
Przysłanianie zabezpieczeń ... 40
Role biznesowe ... 41
Zamawiający ... 42
Broker ... 42
Usługodawca ... 43
Zagadnienia bezpieczeństwa ... 44
Wpływ na modele biznesowe ... 44
Analiza biznesowa ... 45
Co decyduje o powodzeniu inwestycji? ... 46
Usługi sieciowe a B2B ... 47

Kierunki i perspektywy rozwoju biznesu 48
Integracja łańcuchów usług 48
Usługodawca WSP 50
Ontologie pionowe 53
Podsumowanie 55

Rozdział 3. Jak zostać usługodawcą WSP 57

Praktyczne przykłady realizacji usług sieciowych 58
EDI 58
Koncepcje wymiany danych elektronicznych 59
Porównanie EDI i usług sieciowych 60
Dostępność usług sieciowych 61
Internet i przejrzyste rynki 61
Rozwiązania dla małych przedsiębiorstw 62
Przejrzysta ekonomia usług sieciowych 63
Nowe zastosowania usług sieciowych 63
Poszukiwanie nowych możliwości 64
Dlaczego jest to tak istotne 65
Różni usługodawcy WSP 66
Usługi sieciowe dla mniejszych przedsiębiorstw 66
Perspektywy 67
Dla konsultantów i programistów 67
Małe przedsiębiorstwa 68
Wielkie korporacje 68
Producenci oprogramowania 68
Usługodawcy ASP i ISP 69
Podsumowanie 69

Rozdział 4. Konstrukcja usług sieciowych opartych na Javie 71

Architektura usług sieciowych 72
Komponenty Javy 74
Pobranie i instalacja serwera Tomcat 74
Pierwsze kroki z serwerem Tomcat 76
Budowanie usług 78
Zastosowanie stron JavaServer Pages oraz serwletów 78
Interfejsy użytkownika 84
Narzędzia Java 87
Przyszłość usług sieciowych opartych na Javie 88
Podsumowanie 89

Rozdział 5. Przykład prostej usługi sieciowej z zastosowaniem języka Java 91

Aplikacja 92
Platforma usług sieciowych 93
Zestaw narzędzi SOAP 93
Parser XML 94
Testowanie instalacji 95
Wywołanie usługi SOAP 97
Deskryptor WSDL dla usługi HelloWorldService 97
Generowanie klienta HelloWorldService 98
Za kulisami 103
Generowanie usług sieciowych 105

Publikowanie i wyszukiwanie usług sieciowych 108
Podsumowanie 108

Rozdział 6. Tworzenie usługi sieciowej JSP 109

Aplikacja 109
Menu 110
Tworzenie usługi sieciowej 121
Usługa menu restauracji Thai Palace 121
Dokument WSDL dla menu 123
Klient usługi sieciowej menu restauracji Thai Palace 126
Serwer ByteGourmet 132
Uruchomienie usługi sieciowej 133
Serwlet 134
Strony JavaServer Pages 136
Podsumowanie 137

Część II Narzędzia usług sieciowych 139

Rozdział 7. Koncepcja protokołu SOAP 141

Historia SOAP 141
Zdalne wywołania RPC a warstwy pośrednie 142
RSS, RDF i strony WWW 143
Zgłaszanie żądań: XML-RPC 145
Od XML-RPC do SOAP 147
Podstawy SOAP 149
Architektura SOAP 150
Bloki i elementy SOAP 151
Style kodowania 153
Protokoły transportowe 154
RPC 154
Platforma komunikacyjna 155
Zgodność wersji 155
Nagłówek 156
Komunikat błędu SOAP 157
Kodowanie SOAP 158
Pola kodujące 158
Typy proste 158
Typy złożone: struktury 159
Typ złożony: tablice 160
Protokoły transportowe 162
Wiązanie do HTTP 162
Wiązania do SMTP 163
Załączniki do komunikatu SOAP 164
Podsumowanie 165

Rozdział 8. Podstawy SOAP 167

Tworzenie węzłów SOAP 167
Samodzielne tworzenie węzła 168
Biblioteki SOAP 171
Prosta usługa sieciowa: rezerwacja 171
Schemat usługi 173
Instalacja przykładowej aplikacji 173

Prezentacja przykładowej aplikacji 174
Śledzenie komunikatów SOAP 177
Rekompilacja projektu 178
Interfejs administracyjny 181
Interfejs publiczny — aspekty projektowania 187
Kwestie bezpieczeństwa 187
Opóźnienia sieci 188
Wydajność 190
Niezależność 191
Apache SOAP dla RPC 191
Tworzenie aplikacji usługodawcy 191
Uruchomienie aplikacji usługodawcy 196
Wywoływanie aplikacji usługodawcy 199
Podsumowanie 205

Rozdział 9. UDDI 207

Znaczenie UDDI dla usług sieciowych 207
Podstawy UDDI 208
Korzystanie z UDDI 209
Standaryzacja 210
Typowe zastosowania 211
Role UDDI 212
Białe strony 212
Żółte strony 213
Zielone strony 214
Podsumowanie 216

Rozdział 10. Szczegółowa prezentacja UDDI 217

Zapytanie: wyszukiwanie pozycji 217
find_business 218
find_relatedBusiness 221
find_binding 221
find_service 222
find_tModel 223
Zapytanie: gromadzenie danych szczegółowych 225
get_bindingDetail 225
get_businessDetail oraz get_businessDetailExt 226
get_serviceDetail 228
get_tModelDetail 230
Rejestracja 232
Uwierzytelnianie 232
Zapisywanie i usuwanie typów danych UDDI 232
Potwierdzenia 234
Replikacja 235
Podsumowanie 235

Rozdział 11. WSDL 237

Wprowadzenie do WSDL 237
Procesy komunikacyjne 238
Typy 241
Krótki przegląd schematu XML 242
Tworzenie typu Adres dla dokumentu WSDL 247

Komunikaty ... 248
Tworzenie komunikatów z zastosowaniem postaci elementowej ... 249
Tworzenie komunikatów z zastosowaniem typów ... 250
Operacje ... 251
Typy operacji ... 252
Tworzenie operacji jednokierunkowej ... 253
Tworzenie operacji typu żądanie/odpowiedź ... 253
Tworzenie powiadomienia ... 253
Tworzenie operacji typu pytanie/odpowiedź ... 254
Typ portu ... 254
Tworzenie typu portu z operacją jednokierunkową ... 254
Wiązanie ... 255
Wiązanie metody mojaMetoda do SOAP z wykorzystaniem HTTP ... 255
Wiązanie metody mojaMetoda do wielu protokołów transportowych ... 256
Port ... 257
Definicja portu ... 257
Usługa ... 258
Deklaracja usługi ... 258
Łączenie typów portu w ramach usługi ... 259
Tworzenie dokumentów WSDL na podstawie klas Javy ... 260
Deklaracja HelloWorldWSDL ... 260
Generowanie pliku WSDL za pomocą serwera AXIS ... 261
Zastosowanie narzędzia Java2WSDL ... 261
Wynik działania Java2WSDL ... 262
Pozostałe opcje Java2WSDL ... 264
Dostęp do usługi za pośrednictwem dokumentu WSDL ... 265
Tworzenie klas pośredniczących WSDL ... 265
Tworzenie klasy pośredniczącej dla HelloWorldWSDL za pomocą WSDL2Java ... 266
Pozostałe opcje WSDL2Java ... 268
Podsumowanie ... 270

Część III Pakiet JAX Pack ... 271

Rozdział 12. JAXP ... 273

Komponenty XML ... 273
Dokument XML ... 274
Opis typu dokumentu DTD ... 274
Walidacja dokumentu ... 276
Techniki analizy syntaktycznej ... 276
Początki ... 277
Analizator syntaktyczny SAX (Simple API for XML) ... 277
Tworzenie programu obsługi dokumentu ... 278
Korzystanie z parsera SAX ... 278
Analizator syntaktyczny DOM (Document Object Model) ... 289
Tworzenie drzewa XML w pamięci ... 289
Odczyt drzewa XML ... 290
Wyprowadzanie drzewa XML do strumienia wyjściowego ... 295
Operacje na drzewie XML ... 295
XSLT ... 299
Szablony stylów XSL ... 300
Przykład ... 301
JAXP a usługi sieciowe ... 305
Podsumowanie ... 306

Rozdział 13. JAXB ... 307

Przygotowanie ... 307
Terminologia JAXB ... 308
Wiązanie schematu XML i klasy ... 308
XJC w działaniu ... 309
Szeregowanie (marshalling) ... 312
Rozszeregowanie (unmarshalling) ... 313
Skaner XML ... 314
Wykorzystanie klas wbudowanych JAXB ... 317
Odczyt dokumentu XML ... 317
Modyfikacja pliku XML ... 318
Zapis dokumentu XML ... 319
Przykład ... 319
Zaawansowane schematy wiązania ... 321
Definiowanie typów podstawowych ... 323
Definiowanie typów pochodnych ... 323
Definiowanie typów wyliczeniowych ... 324
Podklasy klas wygenerowanych ... 327
Podsumowanie ... 327

Rozdział 14. JAXR ... 329

Potrzeba istnienia API rejestrów ... 329
Podstawowe klasy JAXR ... 330
Interfejs Connection ... 330
Interfejs RegistryClient ... 331
Interfejs RegistryService ... 331
Model danych JAXR ... 331
Stosowanie JAXR ... 333
Tworzenie połączenia ... 333
Przeszukiwanie rejestru ... 334
Aktualizowanie danych w rejestrze ... 338
Podsumowanie ... 342

Rozdział 15. JAXM ... 343

Wprowadzenie do JAXM ... 343
Czym jest, a czym nie jest JAXM ... 344
Obszary zastosowań JAXM ... 346
Architektura ... 347
Modele komunikacji ... 347
Łączniki ... 347
Profile komunikacyjne ... 349
Implementacja ... 350
Układ pakietów ... 350
Obiekty zależne ... 350
Najistotniejsze aspekty prezentowanych przykładów ... 351
Podstawowe czynności ... 351
Połączenia ... 353
Punkty końcowe ... 354
Komunikaty ... 355
Wycieczka po hierarchii komunikatów ... 356
Tworzenie komunikatu ... 357

Klasa SOAPPart 358
Komponenty XML 359
Nadawcy i odbiorcy komunikatu 368
Łączenie wszystkich elementów w całość....370
Hello World....371
Prosty serwer plików....377
Podsumowanie....394

Rozdział 16. JAX-RPC 395

Po co nam kolejny interfejs API?....395
Od zastrzeżonych do otwartych interfejsów API....396
Zasięg JAX-RPC....396
Status JAX-RPC....397
Odwzorowanie danych....398
Standardowe odwzorowanie dla Javy....398
Odwzorowanie nazw elementów....399
Odwzorowanie typów XML....404
Odwzorowanie usług....405
Odwzorowanie usług do Javy....405
Odwzorowanie portów do Javy....406
Zastosowanie JAX-RPC w tworzeniu aplikacji klienckich....407
Wywołania dynamiczne....408
Tworzenie węzłów SOAP....409
Pułapki prostoty....410
Porównanie JAX-RPC i innych technologii rozproszonych....410
Różnice między JAX-RPC, RMI, DCOM i CORBA....411
Obszary zastosowania JAX-RPC i JAXM....412
Podsumowanie....413

Część IV Dopracowywanie usług sieciowych....415

Rozdział 17. Zabezpieczenia w usługach sieciowych.... 417

Dlaczego szyfrowanie jest istotne....417
Szyfrowanie z kluczem prywatnym....418
Szyfrowanie z kluczem publicznym....418
Warstwa zabezpieczeń łączy SSL....419
Podpisy cyfrowe....420
Szyfrowanie w Javie....421
Cyfrowe podpisywanie danych....421
Szyfrowanie danych....425
Zastosowanie SSL i SOAP....428
Szyfrowanie w XML....433
Podsumowanie....437

Rozdział 18. Przepływy w usługach sieciowych (WSFL).... 439

Przepływ w usługach i sprzęganie usług....440
Koncepcje modelowania przepływu....441
Czynności....442
Przepływ sterowania....443
Przepływ danych....445
Przepływ....446

Przepływy jako proces sprzęgania usług sieciowych 446
Usługodawcy i ich typy 447
Lokatory 448
Implementacja czynności zlecanych w outsourcing — eksport....449
Udostępnianie przepływów jako usług sieciowych....450
Model przepływu jako usługodawca....450
Czynności eksportowane....451
Operacje zarządzania cyklem życia przepływu, dane wejściowe i wyjściowe przepływu....452
Sprzęganie zagnieżdżone 454
Przepływy publiczne i prywatne....454
Opis zachowania usługi....454
Implementacja usługi jako przepływ — przepływy prywatne a przepływy publiczne....455
Tworzenie przepływu publicznego 456
Modele globalne 457
Wtyki....458
Modele globalne w WSFL 459
Źródła....462
Podsumowanie....462

Rozdział 19. Platforma wywoływania usług sieciowych (WSIF)............ 463

Opis serwera synchronizacji czasu 465
Pakiety i ścieżki....465
Kod usługi synchronizacji czasu 466
Uruchomienie 468
Aplikacja kliencka Apache SOAP....469
Plik WSDL 472
Dynamiczna aplikacja kliencka WSIF 475
Port SOAP....476
Dynamiczny moduł wywołujący....480
Generowanie pniaków Java 481
Korzystanie z generatora....482
Klasy wygenerowane 482
Testowanie usługi....486
Port Java 487
Zmiany w deskryptorze implementacyjnym WSDL....487
Dostęp do portu Java z poziomu aplikacji klienckiej....489
Dostęp do portu Java za pomocą dynamicznego modułu wywołującego....489
Dostęp do portu Java za pomocą klas wygenerowanych 490
Podsumowanie....490

Część V Implementacja usług sieciowych......................................491

Rozdział 20. Zarządzanie magazynem.. 493

Architektura....494
Witryna WWW 494
Obiekty rozproszone 495
Usługa sieciowa....497
Magazyn w wersji WWW 498
Baza danych 498
Dostęp do magazynu ze strony WWW 499

Witryna hurtownika 501
Kilka uwag na temat projektu 501
Usługa Magazyn 504
Deskryptor uruchomieniowy 505
Witryna sprzedawcy 506
Magazyn w sklepie 506
Lokalna baza danych 507
Aplikacja Sklep 508
Kompilacja i uruchamianie projektu 513
Instalacja przykładu 514
Rekompilacja projektu 515
Podsumowanie 519

Rozdział 21. Handel akcjami — wykorzystanie technologii EJB 521

Identyfikacja użytkownika 522
Architektura aplikacji 522
SOAP i EJB 523
Oprogramowanie 523
Obiekt SOAP o nazwie StockTrading 524
Ziarno sesji Trading 529
Ziarna encji 543
Przykładowy klient 549
Podsumowanie 552

Rozdział 22. Testowanie usług sieciowych 553

Schemat projektowania usług sieciowych 554
Schemat optymalizujący wydajność i skalowalność 556
Strategie testowania 560
Testowanie usług sieciowych przy użyciu programu TestMaker 561
Nowa technologia usług sieciowych — nowa metodologia testowania 563
Testowanie przeszukujące 564
Projektowanie i testowanie zespołowe 564
Testowanie jednostkowe 565
Testowanie systemowe 566
Agenty testujące 567
Testowanie skalowalności i wydajności 569
Testy dla pojedynczego użytkownika 571
Tworzenie agentów testujących za pomocą programu TestMaker 573
Języki skryptowe i agenty testujące 574
Monitorowanie usług sieciowych w celu zagwarantowania odpowiedniego ich poziomu 581
Zasoby 581
Podsumowanie 582

Rozdział 23. Narzędzia do tworzenia usług sieciowych 583

Przegląd narzędzi do tworzenia usług sieciowych 584
Dostarczenie usługi 585
Tworzenie 585
Uruchamianie 586
Testowanie 586
Udostępnianie 586
Wykorzystanie usługi 587
Wyszukiwanie 587
Uzyskiwanie dostępu 587

Krótki przegląd narzędzi do tworzenia usług sieciowych 588
Tworzenie usługi TemperatureConverter 588
Uruchamianie usługi na bazie klasy 590
Tworzenie aplikacji internetowej 592
Narzędzia do tworzenia usług sieciowych 594
Metoda „góra-dół” 595
Metoda „dół-góra” 597
Metoda „środka” 601
Narzędzia do uruchamiania usług sieciowych 602
Narzędzia do testowania usług sieciowych 603
Narzędzia do udostępniania usług sieciowych 606
Narzędzia do wyszukiwania usług sieciowych 607
Narzędzia dostępu do usług sieciowych 608
Podsumowanie 615

Rozdział 24. Tworzenie usług sieciowych za pomocą pakietu WebLogic 617

Usługi sieciowe w środowisku WebLogic 618
Zdalne wywołania procedur 618
Wywołania bazujące na komunikatach 619
Jak to działa? 619
Architektura usług sieciowych na platformie WebLogic 620
Żądania SOAP oparte na RPC 621
Żądania SOAP bazujące na komunikatach 621
Cykl projektowy a środowisko 622
Usługa Curmudgeon 624
Dokument WSDL dla usługi Curmudgeon 624
Kod ziarna dla usługi Curmudgeon 625
Uruchamianie EJB na platformie WebLogic 629
Kompilowanie ziarna EJB 630
Automatyzowanie procesu kompilacji przy użyciu narzędzia Ant 630
Testowanie usługi Curmudgeon 633
Usługi sieciowe następnej generacji 635
Narzędzie WebLogic Workshop 636
Obiekty sterujące w programie WebLogic Workshop 636
Pliki usług sieciowych Java (.JWS) 637
Platforma programu Weblogic Workshop 637
Zasoby 639
Podsumowanie 640

Dodatki 641

Skorowidz 643

Rozdział 17.
Zabezpieczenia w usługach sieciowych

Mark Wutka

W tym rozdziale:

- Dlaczego szyfrowanie jest istotne
- Szyfrowanie w Javie
- Zastosowanie SSL i SOAP
- Szyfrowanie w XML

Internet zasadniczo jest siecią publiczną, a zatem handel elektroniczny może budzić obawy. Nikt nie zechce przesyłać numerów kart kredytowych, jeśli istnieje możliwość ich przechwycenia. Powszechnym rozwiązaniem tego problemu jest oczywiście szyfrowanie — taki sposób kodowania danych, by tylko prawomocny ich adresat mógł je odkodować.

W zakresie przeglądania treści oferowanych na stronach internetowych najpopularniejszym mechanizmem szyfrowania jest protokół SSL (*Secure Sockets Layer*). Jeżeli adres URL danej witryny rozpoczyna się sekwencją znaków `https:`, oznacza to, że zastosowano tu SSL w celu szyfrowania danych. SSL sam w sobie nie jest metodą kodowania, jest platformą umożliwiającą przesyłanie zaszyfrowanych komunikatów za pośrednictwem sieci. W niniejszym rozdziale opisano SSL i mechanizmy szyfrowania, które mogą okazać się pomocne w zabezpieczaniu usług sieciowych.

Dlaczego szyfrowanie jest istotne

Bezpieczeństwo w sieci jest bardzo rozległą dziedziną — nigdy nie można przewidzieć, czego będą w stanie dokonać ludzie chcący obejść czyjeś środki bezpieczeństwa. Przykładowo, w przypadku zakupów w sklepach internetowych, istnieją sposoby oszukiwania niczego nie podejrzewającego użytkownika. Można na przykład przekierować przeglądarkę pod adres *amazzon.com* zamiast *amazon.com*, a następnie zażądać od użytkownika jego numeru karty kredytowej i zachować go do przyszłego wykorzystania. Tak naprawdę *amazon.com* nie może zrobić zupełnie nic, żeby zapobiec tego typu oszustwom

— użytkownik musi sam na podstawie adresu w swojej przeglądarce upewnić się, że znajduje się tam, gdzie chciał.

Chociaż aplikacji nie można oszukać prostymi sztuczkami ze zmianą pisowni adresu, to jednak istnieją sposoby, żeby tego dokonać. Aplikacja nie posługuje się żadną intuicją — w całości polega na środkach programistycznych, które mogą pomóc w wykryciu oszustwa. Dlatego warto wykorzystywać szyfrowanie i podpisy cyfrowe (specjalny rodzaj szyfrowania), co umożliwia rozwiązywanie wielu problemów.

Szyfrowanie z kluczem prywatnym

Szyfrowanie z kluczem prywatnym jest właśnie tym mechanizmem, który większość ludzi ma na myśli mówiąc o szyfrowaniu. Dana osoba dysponuje pewnego rodzaju poufnym hasłem, do którego ma dostęp tylko ona sama oraz odbiorca wiadomości. Za pomocą tego hasła (czyli klucza) wiadomość jest szyfrowana i wysyłana. Adresat odkodowuje otrzymaną wiadomość za pomocą tego samego klucza, który został wykorzystany do jej zakodowania. Oczywiście każdy, kto zna klucz może odczytać wiadomość, dlatego właśnie musi być on zachowany w tajemnicy — stąd określenie *klucz prywatny*.

Szyfrowanie z kluczem prywatnym liczy sobie już kilka tysięcy lat. Grecy i Rzymianie znani byli z szyfrowania przesyłanych komunikatów za pomocą rozmaitych metod. Metoda klucza prywatnego odegrała nawet znaczącą rolę w trakcie II wojny światowej, gdy alianci zdołali odkodować strategiczne wiadomości niemieckie i japońskie, o których sądzono, że są niemożliwe do odszyfrowania.

Era komputerów weszła przebojem w dziedzinę szyfrowania — prymitywne algorytmy szyfrowania za pomocą papieru i ołówka oraz proste maszyny szyfrujące nie miały najmniejszych szans z programowanym komputerem. Specjaliści od kryptografii opracowali nowe algorytmy szyfrowania z zastosowaniem komputera, a także metody ich łamania. Kryptografia cyfrowa jest dzisiaj niezwykle złożoną dziedziną, dysponującą mnóstwem algorytmów i zadziwiająco przemyślnymi sposobami ich łamania.

Szyfrowanie z kluczem publicznym

W porównaniu z szyfrowaniem metodą klucza prywatnego, szyfrowanie z kluczem publicznym jest bardzo młodą dziedziną. Sposób ten opracowano w połowie lat ’70 w celu rozwiązania problemu przesyłania kluczy prywatnych. Kłopotliwe było przekazywanie adresatowi swojego klucza stosując szyfrowanie z kluczem prywatnym. W przeszłości organizacje militarne musiały opierać się na książkach kodów i kurierach doręczających odpowiednie klucze. W dobie komputerów niezbędny jest sposób przesyłania adresatowi klucza prywatnego niemal na żądanie.

Rozwiązaniem tego problemu jest zastosowanie pary klucz publiczny-klucz prywatny. Mówiąc ogólnie, chcąc otrzymywać zaszyfrowane wiadomości należy za pomocą złożonego procesu matematycznego utworzyć specjalną parę kluczy. Jednym z elementów tej pary jest klucz prywatny, który musi pozostać utajniony. Drugi z elementów nosi nazwę klucza publicznego, ponieważ można go przekazać dowolnej osobie. Gdy ktoś chce

przesłać posiadaczowi klucza prywatnego poufną wiadomość, po prostu szyfruje ją za pomocą klucza publicznego tej osoby. Na skutek matematycznych właściwości klucza i samego procesu szyfrowania jedyną osobą, która może odczytać wiadomość jest posiadacz klucza prywatnego. Na podstawie klucza publicznego nie można wyliczyć klucza publicznego, nie można także odszyfrować wiadomości.

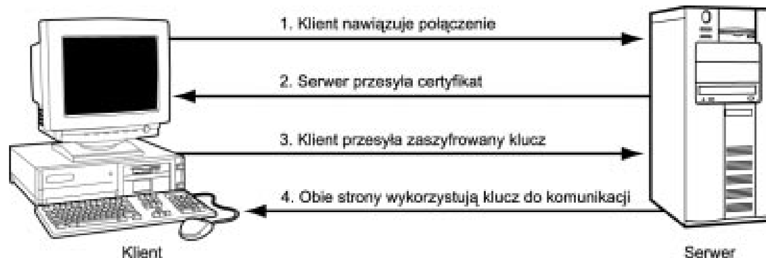
Ta forma szyfrowania określana jest także mianem *szyfrowania z kluczem asymetrycznym*, ponieważ klucz szyfrujący jest różny od deszyfrującego. Typowa metoda szyfrowania z kluczem prywatnym jest mechanizmem symetrycznym — ten sam klucz służy do szyfrowania i odszyfrowywania wiadomości. Z teoretycznego punktu widzenia można skonstruować asymetryczny algorytm z kluczem prywatnym, gdzie wykorzystuje się dwa różne klucze, które obydwa pozostają prywatne, ale w praktyce nie stosuje się tego rozwiązania. Za to niemożliwe jest stworzenie użytecznego symetrycznego algorytmu szyfrowania z kluczem publicznym. Jeżeli wszyscy mają klucz publiczny i klucz ten może być wykorzystany do odkodowania wiadomości, wszyscy mogą odczytać tę wiadomość.

Warstwa zabezpieczeń łączy SSL

Warstwa zabezpieczeń łączy SSL (*Secure Sockets Layer*) jest znakomitym przykładem szyfrowania za pomocą klucza publicznego i prywatnego. Jest to protokół wymiany szyfrowanych danych za pośrednictwem sieci. Klient SSL łączy się z serwerem SSL, który następnie przesyła klientowi certyfikat zawierający klucz publiczny serwera.

Następnie klient tworzy losowy symetryczny klucz prywatny i szyfruje go za pomocą klucza publicznego serwera. Następnie przesyła tak zakodowany klucz serwerowi, który go odszyfrowuje. W tym momencie zarówno klient, jak i serwer znają symetryczny klucz prywatny i mogą wymieniać między sobą dane.

Na rysunku 17.1 pokazano schemat przebiegu typowej wymiany klucza SSL.



Rysunek 17.1. *SSL wykorzystuje klucz publiczny do transmisji klucza prywatnego*

Można zastanawiać się, dlaczego klient i serwer przeprowadzają pozornie bardziej skomplikowany proces szyfrowania za pomocą klucza prywatnego. Wydawałoby się, że do transmisji danych można wykorzystywać po prostu mechanizm z kluczem publicznym. Z technicznego punktu widzenia nie ma przeszkód, ponieważ klient i serwer znają nawzajem swoje klucze publiczne. Problem tkwi w tym, że algorytmy szyfrowania z kluczem publicznym są bardzo złożone i wymagają dużo większej liczby obliczeń niż algorytmy z kluczem prywatnym. Większość metod szyfrowania z kluczem publicznym opiera się na podnoszeniu liczby do bardzo wielkiej potęgi. Może to być, przykładowo, podnoszenie jakiejś liczby do potęgi o wykładniku, którego długość przekracza 300 cyfr!

Podpisy cyfrowe

Szyfrowanie z kluczem publicznym stało się podstawą innej techniki szyfrowania, która jest absolutnie niezbędna w handlu elektronicznym — podpisów cyfrowych. Podpis cyfrowy jest zasadniczo migawką danych, zakodowaną w sposób umożliwiający określenie, kto sygnował te dane oraz czy zostały one zmodyfikowane po umieszczeniu podpisu cyfrowego nadawcy.

W celu utworzenia podpisu cyfrowego najpierw należy zastosować algorytm zwany mieszaniem zabezpieczonym (*secure hash*). Wynik mieszania stanowi niewielki „odcisk palca" danych — zazwyczaj o długości ok. 1024 bitów. Algorytm mieszający musi dawać pewność, że takie zmodyfikowanie danych źródłowych, które spowoduje otrzymanie tej samej wartości mieszanej, jest w zasadzie niemożliwe. W przypadku typowego algorytmu mieszającego nawet zmiana jednego bitu spowoduje radykalną zmianę wartości mieszanej. Prawdopodobieństwo takiej zmiany niewielkiej liczby bitów, by otrzymać początkową wartość mieszaną jest znikome.

Po wyliczeniu zabezpieczonej wartości mieszanej jest ona szyfrowana z wykorzystaniem algorytmu z kluczem publicznym, ale tym razem za pomocą klucza prywatnego. Przydatną właściwością dobranej pary klucz publiczny-klucz prywatny jest nie tylko możliwość użycia klucza publicznego do zaszyfrowania wiadomości, którą można odszyfrować jedynie za pomocą klucza prywatnego, ale także własność odwrotna — wiadomość zaszyfrowana kluczem prywatnym może być odszyfrowana jedynie za pomocą klucza publicznego. Oczywiście druga z tych technik jest nieprzydatna do wysyłania poufnych wiadomości. Szyfrowanie wiadomości w sposób umożliwiający jej odczytanie przez kogokolwiek nie ma sensu.

Okazuje się jednak, że jest to dokładnie taki mechanizm, jaki jest potrzebny w przypadku podpisu cyfrowego. Każdy może odczytać zaszyfrowaną wartość mieszaną. Adresat podpisanego cyfrowo dokumentu wylicza dla niego wartość mieszaną i porównuje ją z zaszyfrowanym podpisem dołączonym do dokumentu. Nadawca jest jedynym posiadaczem prywatnej części klucza, a zatem tylko on mógł zaszyfrować wartość mieszaną, która może zostać zdekodowana za pomocą jego klucza publicznego, co daje pewność, że jest on osobą, która przesłała dany dokument.

Zabezpieczonego algorytmu mieszającego nie można oszukać niewielkimi zmianami w dokumencie, a zatem jeżeli wartość mieszana danego dokumentu jest taka sama jak ta dołączona przez nadawcę, z bardzo dużym prawdopodobieństwem można powiedzieć, że nikt niepowołany nie manipulował przy dokumencie.

Podpisy cyfrowe pozwalają zweryfikować zarówno spójność danych, jak i ich pochodzenie — przynajmniej w stopniu, w jakim można zaufać nadawcy wiadomości i jego dyskrecji w kwestii przechowywania klucza prywatnego. Jedyną rzeczą, której nie zapewnia podpis cyfrowy, jest poufność. Chociaż algorytmy cyfrowego podpisywania dokumentów obejmują szyfrowanie, to nie kodują one samych danych. Podpis cyfrowy zawsze dołączany jest do oryginalnych danych. Aby zapewnić poufność danych, należy je zaszyfrować.

Instytucje certyfikacyjne

Podpisy cyfrowe odgrywają również ważną rolę w zabezpieczeniach SSL. Gdy serwer przesyła klientowi swój klucz publiczny, wysyła go po w postaci certyfikatu cyfrowego. Certyfikat ten w rzeczywistości jest kluczem publicznym i podpisem cyfrowym. Uruchamiając zabezpieczony serwer WWW należy wygenerować parę klucz publiczny-klucz prywatny. Następnie klucz publiczny należy przesłać do instytucji certyfikacyjnej (*CA — Certificate Authority*). Zazwyczaj do klucza załączyć trzeba jeszcze dodatkowe dane identyfikacyjne, dające instytucji CA możliwość weryfikacji podmiotu ubiegającego się o certyfikat. Następnie instytucja ta cyfrowo podpisuje klucz i odsyła nadawcy certyfikat, składający się klucza nadawcy oraz podpisu należącego do CA.

Przeglądarka internetowa lub biblioteka SSL przechowuje listę zaufanych instytucji CA. Jeśli serwer wysyła certyfikat, przeglądarka nie tylko weryfikuje podpis w certyfikacie, ale także sprawdza, czy został on podpisany przez wiarygodną instytucję. Bez udziału instytucji CA przeglądarka nie ma możliwości zweryfikowania tożsamości serwera WWW. Można sobie wyobrazić, że ktoś mógłby wprowadzić przeglądarkę w błąd, podając się za *amazon.com*, co więcej — mógłby przedstawić certyfikat, że naprawdę tak jest! Jeśli nie ma możliwości odwołania się do CA, przeglądarka nie może sprawdzić, czy dany certyfikat jest autentyczny. Choć ktoś o złych intencjach w dalszym ciągu może wprowadzić w błąd przeglądarkę, to raczej nie ma możliwości, by ktokolwiek inny mógł otrzymać certyfikat potwierdzający, że jego witryna to *amazon.com*. Niestety, w przeszłości bywały przypadki oszukiwania instytucji CA.

Warunkiem właściwego funkcjonowania mechanizmu uwierzytelniającego opartego na instytucjach CA jest utworzenie z wyprzedzeniem w przeglądarce lub bibliotece SSL listy zaufanych instytucji. Gdyby przeglądarka opierała się jedynie na informacjach pobranych z sieci, byłaby podatna na atak typu przechwycenie połączenia (*man-in-the-middle attack*), który polega na dostarczeniu przeglądarce odpowiednio spreparowanych danych. Jeżeli jednak przeglądarka wyposażona jest w dane niepochodzące z Internetu (listę zaufanych instytucji certyfikacyjnych), jej użytkownik może uniknąć tego typu ataków.

Szyfrowanie w Javie

Dopóki w biblioteki Java-XML nie zostaną wbudowane algorytmy szyfrowania, trzeba polegać na rozszerzeniach Java Cryptography Extensions. Pakiet JDK został już wyposażony w obsługę podpisów cyfrowych, zaś JDK 1.4 obsługuje protokół SSL, zapewniając bezpieczną komunikację w sieci.

Cyfrowe podpisywanie danych

Do utworzenia cyfrowego podpisu danych w Javie jest potrzebny obiekt `Signature`, uzyskiwany za pomocą metody `Signature.getInstance()` z podaniem nazwy odpowiedniego algorytmu zabezpieczającego. Nazwy algorytmów wykorzystywanych do podpisywania dokumentów składają się z nazwy wybranego zabezpieczonego algorytmu mieszającego i algorytmu szyfrowania z kluczem publicznym.

Dwoma najczęściej stosowanymi algorytmami mieszającymi są SHA1 (*Secure Hash Algorithm 1*) oraz MD5 (*Message Digest 5*). Z kolei najczęściej wykorzystywanymi algorytmami szyfrowania z kluczem publicznym są RSA (od nazwisk jego twórców — Rivest, Shamir i Adleman) oraz DSA (*Digital Signature Algorithm*). Ogólnym formatem pełnej nazwy algorytmu jest `HHHwithCCC`, gdzie `HHH` to nazwa zabezpieczonego algorytmu mieszającego, zaś `CCC` to kod algorytmu szyfrowania z kluczem publicznym. Można zatem wykorzystać np. `SHA1withDSA` lub `MD5withRSA`. Ponieważ Java umożliwia zastosowanie algorytmu szyfrującego dostarczonego przez dowolnego producenta, nie ma jednej listy obsługiwanych algorytmów. Różni producenci opierają swoje oprogramowanie na różnych algorytmach. Pakiet JDK w wersji 1.3 oferuje `SHA1withDSA`, `SHA1withRSA`, `MD5withRSA` oraz `MD5withDSA`.

Oprócz danych, które mają być podpisane, niezbędny jest także klucz prywatny. Zazwyczaj klucz taki jest wczytywany ze specjalnej bazy kluczy o nazwie `keystore`. JDK zawiera narzędzie `keytool`, umożliwiające zarządzanie bazą kluczy.

Na listingu 17.1 przedstawiono prosty program wczytujący klucz z bazy `keystore`, który następnie wykorzystano do utworzenia podpisu cyfrowego danych zawartych w pliku.

Listing 17.1. *Kod źródłowy SignData.java*

```
import java.security.*;
import java.security.interfaces.*;
import java.io.*;

/** Obliczenie podpisu cyfrowego dla pliku */

public class SignData
{
    public static void main(String[] args)
    {
        try
        {
            String keystorePassword = "mykspass";
            String testAlias = "signkey";
            String testKeyPassword = "mykeypass";

// Tworzenie bazy kluczy keystore.
            KeyStore keystore = KeyStore.getInstance("JKS");

// Sprawdzenie gdzie znajduje się baza keystore użytkownika.
            String keystoreFilename = System.getProperty("user.home")+
                File.separator+".keystore";

// Pobranie keystore z pliku.
            keystore.load(new FileInputStream(keystoreFilename),
                keystorePassword.toCharArray());

// Lokalizacja klucza ze wskazaną nazwą i hasłem.
            Key testkey = keystore.getKey(testAlias,
                testKeyPassword.toCharArray());

// Wskazanie, że klucz ten jest kluczem RSA.
            RSAPrivateKey pvtKey = (RSAPrivateKey) testkey;
```

```
// Pobranie certyfikatu dla tego klucza (nie na potrzeby wyliczenia podpisu,
// ale do zapisania w oddzielnym pliku do późniejszej weryfikacji podpisu).
            java.security.cert.Certificate cert =
                keystore.getCertificate(testAlias);

// Tworzenie i inicjalizacja obiektu podpisu stosującego Md5 i RSA.
            Signature signer = Signature.getInstance("MD5withRSA");
            signer.initSign(pvtKey);

// Otwarcie pliku z danymi.
            FileInputStream in = new FileInputStream(args[0]);

// Utworzenie bloku bajtowego do wczytania pliku.
            byte[] buffer = new byte[4096];
            int len;

// Wczytanie bloku pliku i dodanie go do obiektu podpisu.
            while ((len = in.read(buffer)) > 0)
            {
                signer.update(buffer, 0, len);
            }
            in.close();

// Wyliczenie podpisu cyfrowego.
            byte signatureBytes[] = signer.sign();

// Zapisanie podpisu w pliku.
            FileOutputStream out = new FileOutputStream(args[0]+".sig");
            out.write(signatureBytes);
            out.close();

// Zapisanie zaszyfrowanego certyfikatu w pliku.
            out = new FileOutputStream(args[0]+".cer");
            out.write(cert.getEncoded());
            out.close();
        }
        catch (Exception exc)
        {
            exc.printStackTrace();
        }
    }
}
```

Przed uruchomieniem programu tworzącego podpis cyfrowy należy zbudować parę klucz publiczny-klucz prywatny o nazwie `signkey` (nazwa wykorzystywana w programie `SignData`). Trzeba pamiętać, że `SignData` opiera się na algorytmie `RSAwithMD5`, dlatego tworząc parę kluczy jako algorytm należy wskazać RSA. Klucze można zabezpieczyć dodatkowo hasłem (baza `keystore` także jest zabezpieczona hasłem), a ponieważ `SignData` oczekuje podania hasła dla klucza, w czasie tworzenia tego klucza trzeba podać hasło. Klucz niezbędny do działania programu `SignData` można utworzyć za pomocą następującego polecenia:

```
keytool -genkey -alias signKey -keyalg RSA -keypass hasło
```

Uruchamiając po raz pierwszy narzędzie `keytool`, użytkownik jest monitowany o podanie hasła. Jako hasło należy podać `changeit` lub zmienić hasło w programie `SignData`,

tak by odpowiadało podanemu. Uruchamiając zakodowany za pomocą SSL, przykładowy SOAP, który szerzej zaprezentowano nieco dalej w tym rozdziale, jako hasło do bazy `keystore` trzeba podać `changeit`.

Na listingu 17.2 przedstawiono kod prostego programu weryfikującego podpis cyfrowy, także za pomocą bazy `keystore` i obiektu `Signature`.

Listing 17.2. *Kod źródłowy VerifyData.java*

```
import java.io.*;
import java.security.*;
import java.security.cert.*;
import java.security.interfaces.*;

/** Program wykorzystuje klucz publiczny z certyfikatu (czyli plik z
 *  oryginalnymi danymi oraz dwa pliki wygenerowane przez program SignData)
 *  do porównania podpisu cyfrowego z plikiem.
 */

public class VerifyData
{
    public static void main(String[] args)
    {
        try
        {
            if (args.length < 3)
            {
                System.out.println(
                    "Proszę podać plik z danymi, plik podpisu "+
                    "oraz plik certyfikatu.");
                System.exit(1);
            }

            FileInputStream certFile = new FileInputStream(args[2]);

// Tworzenie konstruktora certyfikatu, odczytującego certyfikat X.509.
            CertificateFactory certFact = CertificateFactory.
                getInstance("X.509");

// Wczytanie certyfikatu z odpowiedniego pliku.
            java.security.cert.Certificate cert =
                certFact.generateCertificate(certFile);

            certFile.close();

// Ekstrakcja klucza publicznego z certyfikatu (przy założeniu, że jest to klucz RSA).
            RSAPublicKey pubKey = (RSAPublicKey) cert.getPublicKey();

// Tworzenie i inicjalizacja obiektuvSignature dla MD5 i RSA.
            Signature sigVerifier = Signature.getInstance("MD5withRSA");
            sigVerifier.initVerify(pubKey);

// Otwarcie pliku z danymi.
            FileInputStream dataFile = new FileInputStream(args[0]);

// Utworzenie bufora do odczytu pliku.
            byte[] buffer = new byte[4096];
```

```
            int len;

// Odczyt bloku bajtów z pliku i dołączenie go do podpisu.
            while ((len = dataFile.read(buffer)) > 0)
            {
                sigVerifier.update(buffer, 0, len);
            }

            dataFile.close();

// Tworzenie bufora dla oryginalnego podpisu.
            File sigFile = new File(args[1]);
            byte sigBytes[] = new byte[(int) sigFile.length()];

// Odczyt podpisu z pliku.
            FileInputStream sigIn = new FileInputStream(sigFile);
            sigIn.read(sigBytes);
            sigIn.close();

// Porównanie podpisu z pliku z plikiem danych
// i kluczem publicznym z certyfikatu.
            if (sigVerifier.verify(sigBytes))
            {
                System.out.println("Podpis pasuje.");
            }
            else
            {
                System.out.println(
                    "Podpis nie pasuje.");
            }
        }
        catch (Exception exc)
        {
            exc.printStackTrace();
        }
    }
}
```

Szyfrowanie danych

Aby zapewnić sobie możliwość korzystania z rozszerzeń JCE (*Java Cryptography Extensions*) do szyfrowania danych, warto zaktualizować pakiet JDK do wersji 1.4. W przeciwnym przypadku trzeba pobrać i zainstalować pakiet JCE osobno, a jego instalacja jest nieco bardziej skomplikowana niż dołączenie plików do ścieżki `CLASSPATH`.

W sytuacji dołączania JCE do pakietu JDK 1.2 lub 1.3 należy pobrać i rozpakować najnowszą wersję JCE. Wtedy w katalogu *lib* pakietu JCE powinny się pojawić się cztery pliki: *jce1_2_1.jar*, *subjce_provider.jar*, *local_policy.jar* oraz *US_export_policy.jar*. Pliki te należy skopiować do katalogu *jre/lib/ext* w katalogu instalacyjnym narzędzi JDK. W następnej kolejności należy zmodyfikować plik *jre/lib/security/java.security*. W pliku tym znajduje się następująca sekcja:

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

Jeżeli w pliku *java.security* znajduje się inna liczba wpisów konfigurujących obsługę oprogramowania szyfrującego różnych producentów, wystarczy po prostu dołączyć wiersz o kolejnym numerze większym niż najwyższy z już istniejących. Czyli np. gdy wierszem o najwyższym numerze jest `security.provider.5`, należy dodać wiersz `security.provider.6`.

Po zapisaniu zmian w pliku *java.security* rozszerzenia JCE będą dostępne w pakiecie JDK. Jak widać, instalacja jest nieco kłopotliwa. Tym samym JDK 1.4 z wbudowaną obsługą JCE jest atrakcyjną alternatywą.

Do szyfrowania danych należy zastosować obiekt `Cipher` — w ten sam sposób, co obiekt `Signature` do tworzenia podpisu cyfrowego. Zasadniczo za pomocą metody `Cipher.getInstance()` jest tworzony obiekt `Cipher`, a następnie za pomocą metod `update()` i `doFinal()` do obiektu są przekazywane dane. Metoda `doFinal()` zwraca zaszyfrowane lub odszyfrowane dane w postaci tablicy bajtów.

Obiekt `Cipher` zarówno szyfruje, jak i deszyfruje dane. Pożądaną operację należy wskazać za pomocą metody `init()`, do której jako parametr należy przekazać `Cipher.ENCRYPT_MODE` (dla trybu szyfrowania) lub `Cipher.DECRYPT_MODE` (dla trybu deszyfrowania danych) wraz z odpowiednim kluczem.

Na listingu 17.3 zaprezentowano kod prostego programu szyfrującego dane pobrane z pliku.

Listing 17.3. *Kod źródłowy EncryptData.java*

```
import java.io.*;
import javax.crypto.*;
import javax.crypto.spec.*;

public class EncryptData
{

/* Szyfrowanie pliku za pomocą potrójnego algorytmu DES (DESede).
   Nazwa pliku jest przekazywana jako parametr w wierszu poleceń. Wynik
   jest zapisywany w pliku o tej samej nazwie, co oryginalny, ale z rozszerzeniem
   ".enc" */

    public static void main(String[] args)
    {
        try
        {
// Tworzenie tablicy, w której zostanie umieszczony klucz.
            byte[] encryptKey = "This is a test DESede key".getBytes();

// Tworzenie specyfikacji klucza DESede na podstawie klucza.
            DESedeKeySpec spec = new DESedeKeySpec(encryptKey);

// Tworzenie konstruktora kluczy poufnych, który posłuży do generowania kluczy DESede.
            SecretKeyFactory keyFactory = SecretKeyFactory.getInstance(
                "DESede");

// Tworzenie obiektu DESede SecretKey.
            SecretKey theKey = keyFactory.generateSecret(spec);
```

```
// Tworzenie obiektu DESede Cipher.
            Cipher cipher = Cipher.getInstance("DESede/ECB/PKCS5Padding");

// Inicjalizacja obiektu Cipher i ustawianie trybu szyfrowania.
            cipher.init(Cipher.ENCRYPT_MODE, theKey);

// Otwarcie pliku do szyfrowania.
            FileInputStream in = new FileInputStream(args[0]);

// Otwarcie pliku wynikowego.
            FileOutputStream out = new FileOutputStream(args[0]+".enc");

            byte[] buffer = new byte[7];
            int len;

// Odczytywanie jednego bloku DES w każdym przebiegu.
            while ((len = in.read(buffer)) > 0)
            {
                // Dołączanie porcji bajtów do instancji Cipher.
                byte[] encrypted = cipher.doFinal(buffer, 0, len);
                out.write(encrypted);
            }

            in.close();
            out.close();
        }
        catch (Exception exc)
        {
            exc.printStackTrace();
        }
    }
}
```

Na listingu 17.4 pokazano kod programu deszyfrującego plik zaszyfrowany za pomocą programu z listingu 17.3

Listing 17.4. *Kod źródłowy DecryptData.java*

```
import java.io.*;
import javax.crypto.*;
import javax.crypto.spec.*;

public class DecryptData
{

/* Deszyfrowanie pliku za pomocą potrójnego algorytmu DES (DESede)..
   Nazwę pliku należy podać w wierszu poleceń
   pomijając rozszerzenie .enc */

    public static void main(String[] args)
    {
        try
        {
// Tworzenie tablicy, w której zostanie umieszczony klucz.
            byte[] encryptKey = "Klucz testowy DESede".getBytes();

// Tworzenie specyfikacji klucze DESede na podstawie klucza.
            DESedeKeySpec spec = new DESedeKeySpec(encryptKey);
```

```
// Tworzenie konstruktora kluczy poufnych, który posłuży do generowania kluczy DESede.
            SecretKeyFactory keyFactory = SecretKeyFactory.getInstance(
                "DESede");

// Generowanie obiektu DESede SecretKey.
            SecretKey theKey = keyFactory.generateSecret(spec);

// Tworzenie obiektu DESede Cipher.
            Cipher cipher = Cipher.getInstance("DESede/ECB/PKCS5Padding");

// Inicjalizacja obiektu Cipher i ustawianie trybu deszyfrowania.
            cipher.init(Cipher.DECRYPT_MODE, theKey);

// Otwarcie pliku do odszyfrowania.
            FileInputStream in = new FileInputStream(args[0]+".enc");

// Otwieranie pliku wynikowego.
            FileOutputStream out = new FileOutputStream(args[0]);

            byte[] buffer = new byte[8];
            int len;

// Wczytywanie jednego bloku DES w każdym przebiegu.
            while ((len = in.read(buffer)) > 0)
            {
                // Add the bytes to the cipher
                byte[] encrypted = cipher.doFinal(buffer, 0, len);
                out.write(encrypted);
            }

            in.close();
            out.close();
        }
        catch (Exception exc)
        {
            exc.printStackTrace();
        }
    }
}
```

Zastosowanie SSL i SOAP

Chociaż rozszerzenia JCE umożliwiają zaszyfrowanie i odszyfrowanie dowolnych danych, protokół SSL (*Secure Sockets Layer*) daje możliwość przesyłania zaszyfrowanych danych za pośrednictwem połączenia sieciowego. SSL posiada zasadniczą zaletę — uwalnia od konieczności samodzielnego szyfrowania danych. Dane są przesyłane za pośrednictwem zabezpieczonego gniazda w taki sam sposób, jak za pomocą zwykłego gniazda. Do szyfrowana służy implementacja SSL. Gniazda zabezpieczone działają tak samo jak zwykłe, a zatem współpracują ze wszystkimi klasami wbudowanymi Javy obsługującymi połączenia sieciowe. Można na przykład wykorzystać klasy `URL` oraz `URLconnection` do komunikacji z adresami `HTTPS` — wystarczy, że stosowany pakiet JDK obsługuje SSL.

Rozszerzenie JSSE (*Java Secure Sockets Extension*), wbudowane w pakiet JDK 1.4 obsługuje gniazda zabezpieczone z poziomu Javy. Aby zastosować narzędzia JSSE z pakietami JDK w wersjach 1.2 lub 1.3, trzeba je samodzielnie pobrać i zainstalować. Niestety, instalacja jest utrudniona w podobny sposób, jak w przypadku JCE. Należy pobrać JSSE spod adresu *http://java.sun.com/products/jsse* i zdekompresować pakiet instalacyjny. Następnie pliki *jse.jar*, *jcer.jar* oraz *jnet.jar* z katalogu JSSE *lib* należy skopiować do katalogu JDK *jre/lib.ext*. Na końcu należy zmodyfikować plik *jre/lib/security/java.security*, tak samo jak w przypadku instalacji JCE. Wiersz konfiguracji JSSE będzie wyglądał następująco:

```
security.provider.5=com.sun.net.ssl.internal.ssl.Provider
```

Podobnie, numer wiersza konfiguracji obsługi oprogramowania zabezpieczającego należy dopasować do aktualnego kształtu pliku *java.security* (powinien być on o jeden większy od bieżącego najwyższego numeru). Podobnie jak w przypadku JCE — JDK w wersji 1.4 obejmuje rozszerzenia JSSE.

Zazwyczaj implementacja SOAP opiera się na mechanizmie obsługi SSL po stronie serwera WWW. Chcąc stosować SSL z SOAP, należy wyposażyć serwer WWW w obsługę tego protokołu. Wiele komercyjnych serwerów WWW obsługuje SSL automatycznie. Jeżeli są zainstalowane rozszerzenia JSSE, także Apache Tomcat 4.0 obsługuje ten protokół. Wystarczy zmodyfikować plik *server.xml* w katalogu *conf*. Domyślnie w pliku tym sekcja sterująca obsługą SSL jest ujęta w znaki komentarza. Wygląda ona następująco:

```
<!--
<Connector className="org.apache.catalina.connector.http.HttpConnector"
    port="8443" minProcessors="5" maxProcessors="75"
    enableLookups="true"
    acceptCount="10" debug="0" scheme="https" secure="true">
    <Factory className="org.apache.catalina.net.SSLServerSocketFactory"
            clientAuth="false" protocol="TLS"/>
</Connector>
-->
```

Do aktywacji SSL wystarczy usunąć znaczniki komentarza `<!-- -->`. Jednak zanim serwer Tomcat będzie mógł obsługiwać SSL, trzeba mu zapewnić własny klucz szyfrujący. Klucz ten można utworzyć specjalnie dla tego serwera za pomocą następującego polecenia:

```
keytool -genkey -alias tomcat -keyalg RSA
```

Domyślnym ustawieniem serwera Tomcat jest, że zarówno baza `keystore`, jak i sam klucz są zabezpieczane hasłem `changeit`. Po wygenerowaniu klucza i ponownym uruchomieniu serwera obsługa SSL powinna działać. Aby sprawdzić, czy tak jest rzeczywiście, wystarczy w przeglądarce wpisać adres URL *https://tomcathostname:8443*. Domyślna konfiguracja SSL dla serwera Tomcat wykorzystuje port 8443. Ustawienie to można zmienić na standardowy port 443 przez edytowanie pliku *server.xml* — numer portu znajduje się w sekcji sterującej obsługą SSL. Trzeba jednak pamiętać, że w przypadku systemów Unix czy Linux w celu uruchomienia serwera na porcie o numerze niższym niż 1024 trzeba mieć najwyższe uprawnienia.

Na listingu 17.5 pokazano kod bardzo prostej klasy Java, którą można wywołać za pomocą SOAP.

Listing 17.5. *Kod źródłowy ZłóżZamówienie.java*

```
public class ZłóżZamówienie
{
    public ZłóżZamówienie()
    {
    }

    public String złóż(String nazwaKlienta, String adres, String numerCzęści,
                       int wielkość, String nazwaKartyKredytowej,
                       String numerKartyKredytowej)
    {
// W tym miejscu znajdzie się kod obsługi zamówienia. Na potrzeby przykładu
// po prostu wyświetlimy wszystkie wartości.
        System.out.println("Zamówienie od: "+nazwaKlienta);
        System.out.println(adres);
        System.out.println("Część" "+numerCzęści);
        System.out.println("Wielkość zamówienia: "+wielkość);
        System.out.println("Karta kredytowa: "
                           +nazwaKartyKredytowej+numerKartyKredytowej);
        return "Zamówienie złożone";
    }
}
```

Z powyższego wynika, że klasa ta nie zawiera żadnego kodu obsługi SSL. Całość obsługi tego protokołu jest zadaniem serwera WWW. Na listingu 17.6 pokazano kod klienta Apache SOAP, który korzystając z adresu URL `https` wywołuje obiekt SOAP za pośrednictwem SSL.

Listing 17.6. *Kod źródłowy WyślijZamówienie.java*

```
import java.util.*;
import java.net.*;
import apache.soap.*;
import apache.soap.encoding.*;
import apache.soap.encoding.soapenc.*;
import apache.soap.rpc.*;
import apache.soap.util.xml.*;

public class WyślijZamówienie
{
    public static void main(String[] args)
    {
// Tworzenie URL dla serwera SOAP z wykorzystaniem https zamiast zwykłego http.
        URL url = null;

        try
        {
            url = new URL("https//localhost:8443/soap/servlet/rpcrouter");
        }
        catch (Exception exc)
        {
            exc.printStackTrace();
```

```
            System.exit(0);
        }
// Tworzenie nowego wywołania.
        Call call = new Call();

// Wskazanie SOAP, którego obiektu ma użyć.
        call.setTargetObjectURI("urn:OrderApp");

// Wskazanie SOAP, którą metodę ma wywołać.
        call.setMethodName("złóż");

// Polecenie wykorzystania standardowego stylu kodowania SOAP.
        call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);

// Utworzenie wektora, w którym zostaną umieszczone parametry.
        Vector params = new Vector();

// Umieszczenie parametrów w wektorze.
        params.addElement(new Parameter("nazwaKlienta", String.class,
            "Katarzyna Woźniak", null));
        params.addElement(new Parameter("adres", String.class,
            "ul. Przyjazna 3, 50-560 Radostów", null));
        params.addElement(new Parameter("numerCzęści", String.class,
            "BD-015", null));
        params.addElement(new Parameter("wielkość", Integer.class,
            new Integer(8), null));
        params.addElement(new Parameter("nazwaKartyKredytowej", String.class,
            "SamCard", null));
        params.addElement(new Parameter("numerKartyKredytowej", String.class,
            "40123-557799-12", null));

// Umieszczenie parametrów w żądaniu.
        call.setParams(params)

        Response resp=null;

// Wywołanie metody.
        try
        {
            resp = call.invoke(url, "");
        }
        catch SOAPException exc)
        {
            exc.printStackTrace();
            System.exit();
        }

// Sprawdzenie, czy odpowiedź nie zawiera błędu.
        if (!resp.generatedFault())
        {
// Jeżeli nie, pobranie zwróconej wartości
            Parameter retValue = resp.getReturnedValue();

            System.out.println(retValue.getValue());
        }
        else
        {
            Fault fault = resp.getFault();
```

```
            System.err.println("Wystąpił błąd:");
            System.err.println("  Kod błędu = "+fault.getFaultCode());
            System.err.println("  Komunikat błędu = "fault.getFaultString());
        }
    }
}
```

Istotne jest, że w przypadku klasy `WyślijZamówienie` klasa `URL` stosuje oparty na SSL prefiks `https` zamiast `http`. Program ten bez problemów powinien działać w środowisku JDK 1.4, ale w przypadku wersji 1.2 lub 1.3 należy przekazać do platformy Java odpowiednią informację o zastosowaniu protokołu SSL (prefiks `https`) i podać specjalny parametr systemowy (za pomocą opcji `-D`). Program ten można uruchomić za pomocą następującego polecenia:

```
java -Djava.protocol.handler.pkgs=com.sun.net.ssl.internal.www.protocol
WyślijZamówienie
```

Klasy Java `URL` i `URLConnection` w rzeczywistości nie implementują żadnego z popularnych protokołów internetowych, takich jak HTTP czy FTP. Zamiast tego opierają się one na specjalnych klasach obsługi protokołów. Gdy klasa URL napotyka żądanie dla określonego protokołu, przeszukuje listę pakietów Java w poszukiwaniu klasy obsługi dla tego protokołu. Klasa `java.protocol.handler.pkgs` rozszerza tę listę o dodatkowe protokoły.

Można zauważyć, że podczas uruchomienia programu `WyślijZamówienie` z implementacją SOAP działającą na serwerze Tomcat, pojawia się błąd wejścia/wyjścia. Po aktywowaniu SSL dla serwera Tomcat wygenerowano parę klucz publiczny-klucz prywatny za pomocą polecenia `keytool`. Certyfikat ten początkowo nie jest rozpoznawany jako zaufany — gdyż nie został sygnowany przez instytucję CA. W takiej sytuacji istnieją dwie możliwości — przesłać klucz publiczny do CA w celu autoryzacji lub wymusić na JDK jego uznawanie mimo braku autoryzacji.

Aby wymusić uznanie klucza publicznego przez JDK, należy wyeksportować certyfikat do pliku za pomocą następującego polecenia:

```
keytool -export -file tomcat.cert -alias tomcat
```

W wyniku wykonania tego polecenia jest tworzony plik *tomcat.cert*, zawierający certyfikat cyfrowy, który wykorzystuje serwer Tomcat. Teraz certyfikat ten należy zaimportować do bazy kluczy `keystore`, akceptowanych przez JDK. Baza kluczy znajduje się w pliku JDK *jre/lib./security/cacerts*. W przypadku komputera działającego, na przykład, pod kontrolą systemu Linux, należy zaimportować certyfikat do pliku *cacerts* za pomocą polecenia:

```
keytool -import -keystore /usr/java/jre/lib/security/cacerts -file tomcat.cert
```

W przypadku komputera pracującego pod kontrolą Windows stosuje się polecenie:

```
keytool -import -keystore c:\jdk1.3.1\jre\lib\security\cacerts -file tomcat.cert
```

Po zaimportowaniu certyfikatu opisywany klient SOAP nie powinien mieć żadnych problemów z uzyskaniem dostępu do serwera za pośrednictwem SSL.

Ważne jest, aby zapamiętać, że w kontekście stosowania SOAP z SSL nie trzeba podejmować żadnych dodatkowych działań do obsługi SSL — wystarczy zmienić prefiks adresu URL z `http` na `https`.

Szyfrowanie w XML

Podstawowym problemem dotyczącym szyfrowania w XML jest fakt, że XML jest językiem, którego format miał być czytelny dla człowieka, podczas gdy zaszyfrowane dane trudno nazwać czytelnymi — wydają się one wielką masą przypadkowych znaków.

W przypadku podpisów cyfrowych jest jeszcze gorzej. Czytelnik zapewne pamięta, że nawet zmiana pojedynczego bitu w oryginalnych danych powoduje zmianę wartości mieszanej. Podczas przesyłania pliku XML z jednego systemu do drugiego lub jego analizy za pomocą parsera, mogą zachodzić w nim niewielkie zmiany. Na przykład niektóre systemy jako ogranicznik wiersza stosują znak nowego wiersza, podczas gdy w przypadku innych systemów jest to znak nowego wiersza, występujący po znaku powrotu karetki. Jeżeli oryginalny podpis został utworzony dla dokumentu, w którym występują jedynie znaki nowego wiersza, a adresat otrzyma dokument ze znakami powrotu karetki i nowego wiersza, podpis nie będzie się zgadzał. Także kolejność atrybutów danego elementu nie ma znaczenia dla większości programów, ale ma wpływ na wartość podpisu cyfrowego.

Specyfikacja podpisu elektronicznego dla XML, czyli XML-DSIG (*XML Digital Signature*) formułuje kanoniczną postać XML, która ma rozwiązać ten problem. Zawiera ona następujące ograniczenia:

- Należy usunąć wszystkie nadmiarowe spacje, które nie są częścią danych tekstowych (czyli wszystkie spacje wewnątrz samego znacznika).
- Dla wszystkich brakujących atrybutów należy podać wartości domyślne.
- Jako ogranicznik wiersza należy stosować jedynie znak nowego wiersza.
- Należy rozwinąć wszystkie odsyłacze encji oraz odsyłacze znakowe.

Po opracowaniu kanonicznej formy dokumentu można wyliczyć dla niego podpis cyfrowy. Specyfikacja XML-DSIG definiuje format znacznika `<Signature>`, który zawiera informacje niezbędne do weryfikacji podpisu.

Ogólna postać podpisu XML wygląda następująco:

```
<Signature>
    <SignedInfo>
        (CanonicalizationMethod)
        (SignatureMethod)
        <{Reference (URI=)? >
            (Transforms)?
            (DigestMethod)
            (DigestValue)
        </Reference>)+
    </SignedInfo>
```

```
(SignatureValue)
    (KeyInfo)?
    (Object)*
</Signature>
```

Sekcja `SignedInfo` opisuje różne algorytmy zastosowane do wygenerowania podpisu. Można na przykład zastosować różne metody tworzenia kanonicznej formy dokumentu XML. Metody te pozwalają na opracowanie ogólnego formatu dokumentu. Chociaż XML sam jest formatem ogólnym, różne metody tworzenia postaci kanonicznej dokumentu obejmują różne zasady traktowania nieznaczących spacji oraz innych fragmentów, dając pewność, że dwa dokumenty zawierające te same dane są rzeczywiście identyczne, nie tylko na poziomie strukturalnym czy formalnym, ale także pod względem zapisu binarnego danych dokumentów. Należy także wskazać metodę mieszania (tworzenia skrótu wiadomości) oraz metodę tworzenia podpisu cyfrowego (RSA, DSA etc.). Oczywiście, trzeba także podać wartość samego podpisu, czyli końcowego wyniku algorytmu tworzenia podpisu. Należy także wiedzieć, że znacznik `<Signature>` nie zawiera danych, dla których jest tworzony podpis, ale sam podpis.

Wartość podpisu, wartość skrótu oraz informacje na temat klucza są podawane zazwyczaj w specjalnym formacie o nazwie `base-64`, który pozwala na reprezentowanie danych binarnych za pomocą zestawu 64 drukowalnych znaków. Trzy wartości 8-bitowe są kodowane za pomocą 4 znaków, co zwiększa objętość końcowego bloku danych o ok. 33%.

Za pomocą odsyłaczy do odpowiednich specyfikacji W3C można wskazać różne metody kodowania. Na listingu 17.7 zaprezentowano kod programu Java odczytującego dokument XML, a następnie tworzącego podpis cyfrowy dla tego dokumentu.

Listing 17.7. *Kod źródłowy SingXML.java*

```
import java.security.*;
import java.security.interfaces.*;
import java.io.*;

/** Tworzenie podpisu cyfrowego dla dokumentu XML */

public class SignXML
{
    public static void main(String[] args)
    {
        try
        {
            String keystorePassword = "changeit";
            String testAlias = "signkey";
            String testKeyPassword = "changeit";

// Tworzenie bazy keystore.
            KeyStore keystore = KeyStore.getInstance("JKS");

// Sprawdzenie lokalizacji bazy keystore użytkownika.
            String keystoreFilename = System.getProperty("user.home")+
                File.separator+".keystore";

// Wczytanie bazy keystore z odpowiedniego pliku.
            keystore.load(new FileInputStream(keystoreFilename),
                keystorePassword.toCharArray());
```

```
// Lokalizacja klucza ze wskazaną nazwą i hasłem.
            Key testkey = keystore.getKey(testAlias,
                testKeyPassword.toCharArray());

// Założenie, że dany klucz jest kluczem RSA.
            RSAPrivateKey pvtKey = (RSAPrivateKey) testkey;

// Pobranie certyfikatu dla tego klucza (nie na potrzeby wyliczenia podpisu,
// ale do zapisania w oddzielnym pliku do późniejszej weryfikacji podpisu).
            java.security.cert.Certificate cert =
                keystore.getCertificate(testAlias);

// Pobranie klucza publicznego z certyfikatu.
            RSAPublicKey pubKey = (RSAPublicKey) cert.getPublicKey();

// Tworzenie i inicjalizacja obiektu podpisu korzystającego z MD5 i RSA.
            Signature signer = Signature.getInstance("MD5withRSA");
            signer.initSign(pvtKey);

// Tworzenie skrótu wiadomości — podpis CML musi zawierać
// skrót wiadomości.
            MessageDigest digest = MessageDigest.getInstance("MD5");

// Otwieranie pliku XML.
            FileInputStream in = new FileInputStream(args[0]);

// Tworzenie bloku bajtowego do odczytu pliku.
            byte[] buffer = new byte[4096];
            int len;

// Odczyt bloku pliku i dołączenie go do obiektu podpisu.
            while ((len = in.read(buffer)) > 0)
            {
                signer.update(buffer, 0, len);
                digest.update(buffer, 0, len);
            }
            in.close();

// Wyliczenie podpisu cyfrowego.
            byte signatureBytes[] = signer.sign();

            byte digestBytes[] = digest.digest();

// Zapisanie podpisu do pliku.
            PrintWriter ps = new PrintWriter(
                new BufferedWriter(
                new FileWriter(args[0]+".sig.xml")));

            ps.println("<?xml version=\"1.0\" encoding=\"UTF-8\"?>");
            ps.println("<Signature Id=\"SampleSignature\"
                xmlns=\"http://www.w3.org/2000/09/xmldsig#\">");
            ps.println("  <SignedInfo>");
            ps.println("    <CanonicalizationMethod
                Algorithm=\"http://www.w3.org/TR/2001/REC-xml-c14n-20010315\"/>");
            ps.println("    <Reference URI=\"http://www.w3.org/TR/2000/REC-xhtml1-
                2000-126/\">");
            ps.println("      <Transforms>");
```

```
            ps.println("       <Transform Algorithm=\"http://www.w3.org/TR/2001/REC-
                xml-c14n-20010315\"/>");
            ps.println("      </Transforms>");
            ps.println("      <DigestMethod
                Algorithm=\"http://www.w3.org/2000/09/xmldsig#md5\"/>");
            ps.println("      <DigestValue>"+Base64.toBase64(digestBytes)+
                "</DigestValue>");
            ps.println("    </Reference>");
            ps.println("  </SignedInfo>");
            ps.println("  <SignatureValue>");
            ps.println("    "+Base64.toBase64(signatureBytes));
            ps.println("  </SignatureValue>");
            ps.println("  <KeyInfo>");
            ps.println("    <KeyValue>");
            ps.println("      <RSAKeyValue>");
            ps.println("        <Exponent>");
            ps.println("          "+Base64.toBase64(pubKey.getPublicExponent().
                toByteArray()));
            ps.println("        </Exponent>");
            ps.println("        <Modulus>");
            ps.println("          "+Base64.toBase64(pubKey.getModulus().
                toByteArray()));
            ps.println("        </Modulus>");
            ps.println("      </RSAKeyValue>");
            ps.println("    </KeyValue>");
            ps.println("  </KeyInfo>");
            ps.println("</Signature>");
            ps.close();
        }
        catch (Exception exc)
        {
            exc.printStackTrace();
        }
    }
}
```

Na listingu 17.8 zaprezentowano wynik działania programu `SignXML` dla prostego pliku XML.

Listing 17.8. *Wynik działania SignXML.java*

```
<?xml version="1.0" encoding="UTF-8"?>
<Signature Id="SampleSignature" xmlns="http://www.w3.org/2000/09/xmldsig#">
  <SignedInfo>
    <CanonicalizationMethod Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-
        20010315"/>
    <Reference URI="http://www.w3.org/TR/2000/REC-xhtml1-2000-126/">
      <Transforms>
        <Transform Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
      </Transforms>
      <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#md5"/>
      <DigestValue>tYL+Xgo3exn7dEta8gTW2A==</DigestValue>
    </Reference>
  </SignedInfo>
  <SignatureValue>
```

```
    TTdqAXD83GzdyBa6XB8HVMDckb4+zM4oxhgnvePMOIY8zI9WXnHx+FuF4a/SNHJxoHuMv0N51MMrzOw
    JwsCQAZ6GKfKiebMCyZINXbctKPCXJ+fjiYTXeYa5fg5rJsmwCMv8NYVZiynxItk8uAHzgIJmarU3J
    BeIjOc/dkp1dU4=
  </SignatureValue>
  <KeyInfo>
    <KeyValue>
      <RSAKeyValue>
        <Exponent>
          AQAB
        </Exponent>
        <Modulus>
          /7KpRkbNo3y35VzK4RraLcuG4kdaRWlIT4lpMRkDMxchICNhm43pIVokRmWQl43bovIHZmao
          K7660BGzIaylXeK4z5FXaiUOb6edmtzkgSdQLPskthwZWaDCS6W9iUkgbAyTZ6ZSABApxMJ75Xl
          bryRdlfOVInLNqEJ1kOvoUY+J
        </Modulus>
      </RSAKeyValue>
    </KeyValue>
  </KeyInfo>
</Signature>
```

XML-DSIG jest względnie nowym standardem i wciąż się rozwija. Gdy osiągnie stadium dojrzałości, można spodziewać się bibliotek Java ułatwiających podpisywanie dokumentów XML i weryfikację podpisów cyfrowych XML.

Podsumowanie

Z treści tego rozdziału wynika, że istnieje kilka metod poprawy bezpieczeństwa tworzonych aplikacji. Można zastosować SSL do szyfrowania transmisji albo JCE do zaszyfrowania danych przed wysłaniem ich w żądaniu SOAP lub zaszyfrowania całego żądania XML.

W rozdziale 18. zaprezentowano pewne informacje dotyczące języka przepływu w usługach sieciowych WSFL (*Web Services Flow Language*), dzięki któremu można ustalić, w jaki sposób różne metody SOAP będą współpracować ze sobą.