helion.pl
KSIĘGARNIA INTERNETOWA WYDAWNICTWO

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

C i C++. Bezpieczne programowanie. Receptury

Autorzy: John Viega, Matt Messier
Tłumaczenie: Bartłomiej Garbacz (rozdz. 8 – 13), Krzysztof Mieśniak (rozdz. 6), Mikołaj Szczepaniak (przedmowa, rozdz. 1 – 5, 7)
ISBN: 83-7361-684-5
Tytuł oryginału: Secure Programming Cookbook for C and C++
Format: B5, stron: 784

„C i C++. Bezpieczne programowanie. Receptury" to kompletne źródło wiedzy dla programistów, którzy chcą udoskonalić swoje umiejętności z zakresu tworzenia bezpiecznego kodu. Przedstawia gotowe rozwiązania zagadnień programistycznych, takich jak bezpieczna inicjalizacja aplikacji, kryptografia, uwierzytelnianie użytkowników, wymiana kluczy, zapobieganie penetracji i wielu innych. Każde zagadnienie jest przedstawione w postaci kodu źródłowego w języku C i C++ oraz obszernego opisu, co ułatwia dostosowanie go do własnych potrzeb.

- Bezpieczne uruchamianie aplikacji
- Kontrola dostępu do plików i aplikacji
- Sprawdzanie poprawności danych wejściowych oraz ochrona przed atakami typu XSS i SQL Injection
- Generowanie i obsługa kluczy symetrycznych
- Wykorzystywanie szyfrowania symetrycznego
- Stosowanie klucza publicznego
- Bezpieczna komunikacja sieciowa
- Liczby losowe
- Zapobieganie penetracjom oraz obsługa błędów

Książka zawiera wszystkie informacje niezbędne do zabezpieczenia aplikacji przed hakerami.

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

Spis treści

Przedmowa 11

Wstęp 15

1. Bezpieczna inicjalizacja 25
1.1. Zabezpieczanie środowiska pracy programu 25
1.2. Ograniczanie uprawnień w systemach Windows 32
1.3. Rezygnacja z uprawnień w programach setuid 40
1.4. Ograniczanie ryzyka związanego z separacją uprawnień 45
1.5. Bezpieczne zarządzanie deskryptorami plików 48
1.6. Bezpieczne tworzenie procesu potomnego 50
1.7. Bezpieczne uruchamianie programów zewnętrznych w systemach Unix 53
1.8. Bezpieczne uruchamianie zewnętrznych programów w systemach Windows 58
1.9. Wyłączanie zrzutów pamięci w przypadku wystąpienia błędu 60

2. Kontrola dostępu 63
2.1. Model kontroli dostępu w systemach Unix 63
2.2. Model kontroli dostępu w systemach Windows 66
2.3. Określanie, czy dany użytkownik ma dostęp do danego pliku w systemie Unix 68
2.4. Określanie, czy dany katalog jest bezpieczny 70
2.5. Bezpieczne usuwanie plików 73
2.6. Bezpieczne uzyskiwanie dostępu do informacji o pliku 79
2.7. Ograniczone prawa dostępu do nowych plików w systemach Unix 80
2.8. Blokowanie plików 83
2.9. Synchronizacja dostępu procesów do zasobów w systemach Unix 85
2.10. Synchronizacja dostępu procesów do zasobów w systemach Windows 89
2.11. Tworzenie plików tymczasowych 91
2.12. Ograniczanie dostępu do systemu plików w systemach Unix 94
2.13. Ograniczanie dostępu do systemu plików i sieci w systemie FreeBSD 95

3. Sprawdzanie poprawności danych wejściowych 97
3.1. Podstawowe techniki sprawdzania poprawności danych 98
3.2. Zapobieganie atakom z wykorzystaniem funkcji formatujących 102
3.3. Zapobieganie przepełnieniom bufora 105

3.4. Stosowanie biblioteki SafeStr 113
3.5. Zapobieganie koercji liczb całkowitych i problemowi przekroczenia zakresu 116
3.6. Bezpieczne stosowanie zmiennych środowiskowych 120
3.7. Sprawdzanie poprawności nazw plików i ścieżek 125
3.8. Obsługa kodowania URL 127
3.9. Sprawdzanie poprawności adresów poczty elektronicznej 129
3.10. Ochrona przed atakami typu cross-site scripting (XSS) 131
3.11. Ochrona przed atakami typu SQL injection 135
3.12. Wykrywanie nieprawidłowych znaków UTF-8 138
3.13. Zapobieganie przepełnieniom deskryptorów plików podczas stosowania funkcji select() 140

4. Podstawy kryptografii symetrycznej ... 145

4.1. Reprezentacje kluczy wykorzystywanych w algorytmach kryptograficznych 146
4.2. Generowanie losowych kluczy symetrycznych 148
4.3. Szesnastkowe reprezentacje kluczy binarnych (lub innych nieprzetworzonych danych) 149
4.4. Przekształcanie szesnastkowych kluczy ASCII (lub innych szesnastkowych danych ASCII) na postać binarną 151
4.5. Kodowanie Base64 152
4.6. Dekodowanie łańcucha zakodowanego zgodnie ze standardem Base64 154
4.7. Reprezentowanie kluczy (lub dowolnych innych danych binarnych) w postaci tekstu zapisanego w języku angielskim 157
4.8. Przekształcanie kluczy tekstowych na klucze binarne 159
4.9. Stosowanie argumentów salt, jednorazowych identyfikatorów i wektorów inicjalizacji 161
4.10. Generowanie kluczy symetrycznych na bazie haseł 165
4.11. Algorytmiczne generowanie kluczy symetrycznych na bazie jednego tajnego klucza głównego 171
4.12. Szyfrowanie okrojonego zbioru znaków 175
4.13. Bezpieczne zarządzanie materiałem klucza 178
4.14. Badanie czasu działania algorytmów kryptograficznych 179

5. Szyfrowanie symetryczne ... 185

5.1. Podejmowanie decyzji w kwestii stosowania wielu algorytmów szyfrujących 185
5.2. Wybór najlepszego algorytmu szyfrującego 186
5.3. Wybór właściwej długości klucza 190
5.4. Wybór trybu pracy szyfru blokowego 193
5.5. Stosowanie podstawowych operacji szyfru blokowego 203
5.6. Stosowanie ogólnej implementacji trybu CBC 207
5.7. Stosowanie ogólnej implementacji trybu CFB 217
5.8. Stosowanie ogólnej implementacji trybu OFB 224

5.9. Stosowanie ogólnej implementacji trybu CTR 228
5.10. Stosowanie trybu szyfrowania CWC 233
5.11. Ręczne dodawanie i sprawdzanie dopełniania szyfru 237
5.12. Wyznaczanie z góry strumienia klucza w trybach OFB, CTR, CCM i CWC (oraz w szyfrach strumieniowych) 239
5.13. Zrównoleglanie szyfrowania i deszyfrowania w trybach, które na takie działania zezwalają (bez wprowadzania ewentualnych niezgodności) 240
5.14. Zrównoleglanie szyfrowania i deszyfrowania w dowolnych trybach (a więc z możliwością wprowadzania ewentualnych niezgodności) 244
5.15. Szyfrowanie zawartości plików lub całych dysków 245
5.16. Stosowanie wysokopoziomowych, odpornych na błędy interfejsów API dla operacji szyfrowania i deszyfrowania 249
5.17. Konfiguracja szyfru blokowego (dla trybów szyfrowania CBC, CFB, OFB oraz ECB) w pakiecie OpenSSL 254
5.18. Stosowanie szyfrów ze zmienną długością klucza w pakiecie OpenSSL 259
5.19. Wyłączanie mechanizmu dopełniania w szyfrach pakietu OpenSSL pracujących w trybie CBC 260
5.20. Dodatkowa konfiguracja szyfrów w pakiecie OpenSSL 261
5.21. Sprawdzanie właściwości konfiguracji szyfru w pakiecie OpenSSL 262
5.22. Wykonywanie niskopoziomowego szyfrowania i deszyfrowania w pakiecie OpenSSL 264
5.23. Konfiguracja i stosowanie szyfru RC4 267
5.24. Stosowanie szyfrów z kluczem jednorazowym 270
5.25. Stosowanie szyfrowania symetrycznego z wykorzystaniem CryptoAPI firmy Microsoft 271
5.26. Tworzenie obiektu klucza interfejsu CryptoAPI na bazie dowolnych danych klucza 277
5.27. Uzyskiwanie surowych danych klucza z obiektu klucza interfejsu CryptoAPI 280

6. Funkcje skrótu i uwierzytelnianie wiadomości 283

6.1. Zrozumienie podstaw funkcji skrótu i kodu uwierzytelniającego wiadomość MAC 283
6.2. Decydowanie, czy obsługiwać wiele skrótów wiadomości lub kodów MAC 287
6.3. Wybór kryptograficznego algorytmu skrótu 288
6.4. Wybór kodu uwierzytelnienia wiadomości 292
6.5. Przyrostowe tworzenie skrótów danych 296
6.6. Tworzenie skrótu z pojedynczego łańcucha znaków 300
6.7. Używanie skrótu kryptograficznego 302
6.8. Wykorzystanie identyfikatora jednorazowego do obrony przed atakami wykorzystującymi paradoks dnia urodzin 303
6.9. Sprawdzanie spójności wiadomości 307
6.10. Używanie HMAC 309
6.11. Używanie OMAC (prostego kodu MAC opartego na szyfrze blokowym) 312

6.12. Używanie HMAC lub OMAC z identyfikatorem jednorazowym 317
6.13. Używanie kodu MAC, który jest wystarczająco szybki w realizacji programowej i sprzętowej 318
6.14. Używanie kodu MAC zoptymalizowanego do szybszego działania w realizacji programowej 319
6.15. Konstruowanie funkcji skrótu z szyfru blokowego 322
6.16. Używanie szyfru blokowego do budowy mocnej funkcji skrótu 325
6.17. Używanie mniejszych znaczników MAC 329
6.18. Szyfrowanie z zachowaniem spójności wiadomości 329
6.19. Tworzenie własnego kodu MAC 331
6.20. Szyfrowanie za pomocą funkcji skrótu 332
6.21. Bezpieczne uwierzytelnianie kodu MAC (obrona przed atakami związanymi z przechwytywaniem i powtarzaniem odpowiedzi) 334
6.22. Przetwarzanie równoległe kodu MAC 335

7. Kryptografia z kluczem publicznym 337
7.1. Określanie sytuacji, w których należy stosować techniki kryptografii z kluczem publicznym 339
7.2. Wybór algorytmu z kluczem publicznym 342
7.3. Wybór rozmiarów kluczy publicznych 343
7.4. Przetwarzanie wielkich liczb 346
7.5. Generowanie liczby pierwszej i sprawdzanie czy dana liczba jest liczbą pierwszą 355
7.6. Generowanie pary kluczy szyfru RSA 358
7.7. Oddzielanie kluczy publicznych i prywatnych w pakiecie OpenSSL 361
7.8. Konwertowanie łańcuchów binarnych na postać liczb całkowitych na potrzeby szyfru RSA 362
7.9. Przekształcanie liczb całkowitych do postaci łańcuchów binarnych na potrzeby szyfru RSA 363
7.10. Podstawowa operacja szyfrowania za pomocą klucza publicznego algorytmu RSA 364
7.11. Podstawowa operacja deszyfrowania za pomocą klucza prywatnego algorytmu RSA 368
7.12. Podpisywanie danych za pomocą klucza prywatnego szyfru RSA 370
7.13. Weryfikacja cyfrowo podpisanych danych za pomocą klucza publicznego algorytmu RSA 374
7.14. Bezpieczne podpisywanie i szyfrowanie danych za pomocą algorytmu RSA 376
7.15. Wykorzystywanie algorytmu DSA 381
7.16. Reprezentowanie kluczy publicznych i certyfikatów w postaci łańcuchów binarnych (zgodnie z regułami kodowania DER) 386
7.17. Reprezentowanie kluczy i certyfikatów w postaci tekstu (zgodnie z regułami kodowania PEM) 390

8. **Uwierzytelnianie i wymiana kluczy 397**
8.1. Wybór metody uwierzytelniania 397
8.2. Uzyskiwanie informacji o użytkownikach i grupach w systemach uniksowych 407
8.3. Uzyskiwanie informacji o użytkownikach i grupach w systemach Windows 410
8.4. Ograniczanie dostępu na podstawie nazwy maszyny lub adresu IP 413
8.5. Generowanie losowych haseł i wyrażeń hasłowych 420
8.6. Sprawdzanie odporności haseł na ataki 424
8.7. Monitowanie o hasło 425
8.8. Kontrola nad nieudanymi próbami uwierzytelnienia 430
8.9. Uwierzytelnianie oparte na hasłach z użyciem funkcji crypt() 432
8.10. Uwierzytelnianie oparte na hasłach z użyciem funkcji MD5-MCF 434
8.11. Uwierzytelnianie oparte na hasłach z użyciem funkcji PBKDF2 439
8.12. Uwierzytelnianie przy użyciu modułów PAM 442
8.13. Uwierzytelnianie za pomocą systemu Kerberos 445
8.14. Uwierzytelnianie z wykorzystaniem mechanizmu HTTP Cookies 449
8.15. Uwierzytelnianie oraz wymiana kluczy oparte na hasłach 452
8.16. Przeprowadzanie uwierzytelnionej wymiany klucza przy użyciu algorytmu RSA 459
8.17. Użycie podstawowego protokołu uzgadniania klucza metodą Diffiego-Hellmana 461
8.18. Wspólne użycie metody Diffiego-Hellmana i algorytmu DSA 466
8.19. Minimalizacja okresu podatności na ataki w przypadku uwierzytelniania bez użycia infrastruktury PKI 467
8.20. Zapewnianie przyszłego bezpieczeństwa w systemie symetrycznym 473
8.21. Zapewnianie przyszłego bezpieczeństwa w systemie z kluczem publicznym 474
8.22. Potwierdzanie żądań za pomocą wiadomości poczty elektronicznej 476

9. **Komunikacja sieciowa 483**
9.1. Tworzenie klienta SSL 484
9.2. Tworzenie serwera SSL 486
9.3. Używanie mechanizmu buforowania sesji w celu zwiększenia wydajności serwerów SSL 489
9.4. Zabezpieczanie komunikacji sieciowej na platformie Windows przy użyciu interfejsu WinInet API 492
9.5. Aktywowanie protokołu SSL bez modyfikowania kodu źródłowego 496
9.6. Używanie szyfrowania standardu Kerberos 498
9.7. Komunikacja międzyprocesowa przy użyciu gniazd 503
9.8. Uwierzytelnianie przy użyciu uniksowych gniazd domenowych 509
9.9. Zarządzanie identyfikatorami sesji 512
9.10. Zabezpieczanie połączeń bazodanowych 513

9.11. Używanie wirtualnych sieci prywatnych w celu zabezpieczenia połączeń sieciowych 516
9.12. Tworzenie uwierzytelnionych bezpiecznych kanałów bez użycia SSL 517

10. Infrastruktura klucza publicznego ... 527
10.1. Podstawy infrastruktury klucza publicznego 527
10.2. Otrzymywanie certyfikatu 538
10.3. Używanie certyfikatów głównych 543
10.4. Podstawy metodologii weryfikacji certyfikatów X.509 546
10.5. Przeprowadzanie weryfikacji certyfikatów X.509 przy użyciu OpenSSL 548
10.6. Przeprowadzanie weryfikacji certyfikatów X.509 przy użyciu interfejsu CryptoAPI 553
10.7. Weryfikowanie certyfikatu pochodzącego od partnera komunikacji SSL 558
10.8. Dodawanie mechanizmu sprawdzania nazwy hosta do procesu weryfikacji certyfikatu 562
10.9. Używanie list akceptacji w celu weryfikowania certyfikatów 566
10.10. Pobieranie list unieważnionych certyfikatów przy użyciu OpenSSL 569
10.11. Pobieranie list unieważnionych certyfikatów przy użyciu CryptoAPI 576
10.12. Sprawdzanie stanu unieważnienia poprzez protokół OCSP przy wykorzystaniu OpenSSL 582

11. Liczby losowe ... 587
11.1. Określanie charakteru liczb losowych, których należy użyć 587
11.2. Używanie ogólnego interfejsu API dla obsługi losowości i entropii 592
11.3. Używanie standardowej infrastruktury losowości w systemach uniksowych 594
11.4. Używanie standardowej infrastruktury losowości w systemach Windows 598
11.5. Używanie generatora poziomu aplikacji 600
11.6. Ponowna inicjalizacja ziarna generatora liczb pseudolosowych 609
11.7. Używanie rozwiązania kompatybilnego z demonem zbierania entropii 612
11.8. Zbieranie entropii lub wartości pseudolosowych przy użyciu pakietu EGADS 616
11.9. Używanie interfejsu API obsługi liczb losowych biblioteki OpenSSL 620
11.10. Otrzymywanie losowych wartości całkowitych 622
11.11. Otrzymywanie losowych wartości całkowitych z zadanego przedziału 623
11.12. Otrzymywanie losowych wartości zmiennopozycyjnych o rozkładzie jednorodnym 625
11.13. Otrzymywanie wartości zmiennopozycyjnych o rozkładzie niejednorodnym 626
11.14. Otrzymywanie losowych drukowalnych ciągów znaków ASCII 627
11.15. Uczciwe tasowanie 628
11.16. Kompresowanie danych z entropią do postaci ziarna o ustalonym rozmiarze 629
11.17. Zbieranie entropii w momencie uruchamiania systemu 630
11.18. Testowanie statystyczne liczb losowych 632

11.19. Szacowanie i zarządzanie entropią 637
11.20. Zbieranie entropii na podstawie interakcji z klawiaturą 645
11.21. Zbieranie entropii na podstawie zdarzeń związanych z obsługą myszy w systemie Windows 653
11.22. Zbieranie entropii na podstawie pomiarów czasowych wątków 657
11.23. Zbieranie entropii na podstawie stanu systemu 659

12. Zapobieganie ingerencji 661
12.1. Podstawowe kwestie dotyczące problemu ochrony oprogramowania 662
12.2. Wykrywanie modyfikacji 667
12.3. Zaciemnianie kodu 672
12.4. Przeprowadzanie zaciemniania na poziomie bitów i bajtów 677
12.5. Przeprowadzanie przekształceń na zmiennych z użyciem wartości stałych 679
12.6. Scalanie zmiennych skalarnych 680
12.7. Rozdzielanie zmiennych 681
12.8. Ukrywanie wartości logicznych 682
12.9. Używanie wskaźników do funkcji 683
12.10. Zmiana struktury tablic 684
12.11. Ukrywanie ciągów znaków 689
12.12. Wykrywanie programów uruchomieniowych 691
12.13. Wykrywanie programów uruchomieniowych w systemie Unix 693
12.14. Wykrywanie programów uruchomieniowych w systemie Windows 695
12.15. Wykrywanie programu SoftICE 696
12.16. Przeciwdziałanie deasemblacji 698
12.17. Używanie kodu samomodyfikującego 703

13. Inne zagadnienia........ 709
13.1. Obsługa błędów 709
13.2. Bezpieczne usuwanie danych z pamięci 713
13.3. Zapobieganie stronicowaniu pamięci na dysku 716
13.4. Poprawne używanie argumentów zmiennych 717
13.5. Poprawna obsługa sygnałów 720
13.6. Ochrona przed atakami rozbicia w systemie Windows 724
13.7. Ochrona przed uruchomieniem zbyt wielu wątków 726
13.8. Ochrona przed tworzeniem zbyt wielu gniazd sieciowych 731
13.9. Ochrona przed atakami wyczerpania zasobów w systemie Unix 734
13.10. Ochrona przed atakami wyczerpania zasobów w systemie Windows 737
13.11. Korzystanie ze sprawdzonych praktyk dotyczących rejestrowania nadzorczego 740

Skorowidz 745

ROZDZIAŁ 9.
Komunikacja sieciowa

Obecnie większość aplikacji jest związana z uczestniczeniem w pewnego rodzaju aktywności sieciowej. Niestety, wielu programistów nie wie, w jaki sposób należy uzyskiwać dostęp do sieci w sposób bezpieczny. Receptury prezentowane w niniejszym rozdziale mają na celu pomóc w wykorzystywaniu sieci we własnych programach. Dla wielu programistów bezpieczeństwo sieciowe postrzegane z punktu widzenia aplikacji oznacza użycie protokołu Secure Socket Layer (SSL), jednak SSL nie stanowi magicznego rozwiązania. Niekiedy może być trudno użyć go w sposób prawidłowy, w wielu sytuacjach stanowi nadmierne obciążenie, a niekiedy jest rozwiązaniem niewystarczającym. W niniejszym rozdziale zaprezentowano receptury opisujące użycie pakietu OpenSSL w celu tworzenia klientów i serwerów obsługujących protokół SSL, jak również receptury dotyczące komunikacji sieciowej i międzyprocesowej odbywającej się bez użycia SSL.

W przypadku platformy Windows z wyjątkiem użycia SSL w celu szyfrowania ruchu HTTP (co omówiono w recepturze 9.4) autorzy zdecydowali się ograniczyć receptury poświęcone protokołowi SSL tylko do pakietu OpenSSL, który jest dostępny za darmo i jest przenośny na wiele platform, w tym również właśnie Windows.

W systemie Windows firma Microsoft zapewnia dostęp do implementacji protokołu SSL poprzez interfejs SSPI (ang. *Security Support Provider Interface*). SSPI jest dobrze udokumentowany, ale niestety, użycie samego SSL — nie. Co więcej, implementacja klienta lub serwera wykorzystującego SSL za pomocą SSPI w systemie Windows jest o wiele bardziej skomplikowana niż użycie OpenSSL. Interfejs SSPI jest zaskakująco niskopoziomowy, wymaga od wykorzystujących go programów wykonywania wielu zadań związanych z wymianą komunikatów protokołu. Ze względu na fakt, że SSL trudno jest używać poprawnie, pożądanym rozwiązaniem jest ukrycie szczegółów protokołu za implementacją wysokopoziomową (taką jak OpenSSL). Stąd też autorzy będą unikać używania interfejsu SSPI.

Jeżeli Czytelnik jest bardziej zainteresowany interfejsami SSPI oraz SSL, warto sięgnąć do dokumentacji Microsoft Developer's Network (MSDN) oraz po przykłady zawarte w pakiecie Microsoft Windows Platform SDK, który jest dostępny pod adresem *http://www.microsoft.com/msdownload/platformsdk/sdkupdate/*. Odpowiednie fragmenty przykładowego kodu można znaleźć z katalogu *Microsoft SDK\Samples\Security\SSPI\SSL*, skąd instaluje się je w swoim systemie (zazwyczaj w katalogu *Program Files* na dysku startowym).

9.1. Tworzenie klienta SSL

Problem

Chcemy zestawić połączenia klienta z serwerem zdalnym przy użyciu protokołu SSL.

Rozwiązanie

Zestawianie połączenia z serwerem zdalnym przy użyciu protokołu SSL nie różni się bardzo od zestawiania połączenia bez jego użycia, a przynajmniej nie musi się wiele różnić. Wymaga to nieco większego nakładu sił w kwestii konfiguracji i w głównej mierze polega na utworzeniu obiektu `spc_x509store_t` (patrz receptura 10.5), który zawiera informacje potrzebne do dokonania weryfikacji serwera. Kiedy zostanie to zrobione, należy utworzyć obiekt `SSL_CTX` i dodać go do połączenia. Za pozostałe działania odpowiedzialny jest pakiet OpenSSL.

Przed lekturą niniejszej receptury należy dobrze poznać podstawy infrastruktury klucza publicznego (patrz receptura 10.1).

Analiza

Po utworzeniu obiektu `spc_x509store_t` poprzez załadowanie go z odpowiednimi certyfikatami i listami CRL (informacje na temat otrzymywania list CRL[1] można znaleźć w recepturach 10.10 oraz 10.11), połączenie się ze zdalnym serwerem przy użyciu protokołu SSL może polegać tylko na wywołaniu funkcji `spc_connect_ssl()`. Opcjonalnie można samodzielnie utworzyć obiekt `SSL_CTX`, używając funkcji `spc_create_sslctx()` lub interfejsu API OpenSSL. Można również wykorzystać obiekt już istniejący, utworzony dla innych połączeń, lub pozostawić to w gestii funkcji `spc_connect_ssl()`. W tym drugim przypadku połączenie zostanie zestawione, zaś utworzony obiekt `SSL_CTX` zostanie zwrócony jako wskaźnik na wskaźnik do obiektu `SSL_CTX` przekazany jako argument funkcji.

```
#include <openssl/bio.h>
#include <openssl/ssl.h>

BIO *spc_connect_ssl(char *host, int port, spc_x509store_t *spc_store,
                     SSL_CTX **ctx) {
  BIO *conn = 0;
  int our_ctx = 0;

  if (*ctx) {
    CRYPTO_add(&((*ctx)->references), 1, CRYPTO_LOCK_SSL_CTX);
    if (spc_store && spc_store != SSL_CTX_get_app_data(*ctx)) {
      SSL_CTX_set_cert_store(*ctx, spc_create_x509store(spc_store));
      SSL_CTX_set_app_data(*ctx, spc_store);
    }
  } else {
    *ctx = spc_create_sslctx(spc_store);
    our_ctx = 1;
  }
```

[1] Lista unieważnionych certyfikatów (ang. *Certificate Revocation List*) — *przyp. tłum.*

```
    if (!(conn = BIO_new_ssl_connect(*ctx))) goto error_exit;
    BIO_set_conn_hostname(conn, host);
    BIO_set_conn_int_port(conn, &port);

    if (BIO_do_connect(conn) <= 0) goto error_exit;
    if (our_ctx) SSL_CTX_free(*ctx);
    return conn;

  error_exit:
    if (conn) BIO_free_all(conn);
    if (*ctx) SSL_CTX_free(*ctx);
    if (our_ctx) *ctx = 0;
    return 0;
  }
```

Powyżej zaprezentowano dodatkową funkcję, która obsługuje różnice między łączeniem się z serwerem zdalnym przy użyciu SSL oraz bez jego użycia. W obu przypadkach zwracany jest obiekt `BIO`, którego można używać w ten sam sposób bez względu na to, czy zestawiono połączenie SSL czy nie. Jeżeli znacznik `ssl` w wywołaniu tej funkcji ma wartość zerową, argumenty `spc_store` oraz `ctx` są ignorowane, ponieważ mają zastosowanie wyłącznie w przypadku połączeń SSL.

OpenSSL często korzysta z obiektów `BIO` i wiele funkcji interfejsu API również pobiera argumenty `BIO`. Pojawia się pytanie, czym są te obiekty. Ujmując rzecz skrótowo, obiekty `BIO` stanowią abstrakcję dla operacji wejścia-wyjścia, która oferuje jednolity, częściowo niezależny interfejs. Obiekty `BIO` są tworzone dla plikowych operacji wejścia-wyjścia, operacji dokonywanych na gniazdach oraz w pamięci. Ponadto specjalne obiekty `BIO`, znane jako ***filtry BIO*** (ang. *BIO filters*), mogą być używane w celu filtrowania danych przed ich zapisaniem lub odczytaniem z odpowiedniego nośnika. Filtry BIO są tworzone dla operacji takich jak kodowanie base64 oraz szyfrowanie przy użyciu szyfru symetrycznego.

Interfejs API OpenSSL bazuje na obiektach `BIO` i specjalny rodzaj filtra zajmuje się szczegółami związanymi z SSL. Filtr SSL `BIO` jest najbardziej przydatny wówczas, gdy stosuje się go wspólnie z obiektem `BIO` gniazd, ale może również być używany do bezpośredniego łączenia dwóch obiektów `BIO` razem (jeden dla operacji odczytu, drugi — zapisu) lub w celu opakowywania potoków lub innego rodzaju podstawowych mechanizmów komunikacyjnych związanych z połączeniami.

```
  BIO *spc_connect(char *host, int port, int ssl, spc_x509store_t *spc_store,
                   SSL_CTX **ctx) {
    BIO *conn;
    SSL *ssl_ptr;

    if (ssl) {
      if (!(conn = spc_connect_ssl(host, port, spc_store, ctx))) goto error_exit;
      BIO_get_ssl(conn, &ssl_ptr);
      if (!spc_verify_cert_hostname(SSL_get_peer_certificate(ssl_ptr), host))
        goto error_exit;
      if (SSL_get_verify_result(ssl_ptr) != X509_V_OK) goto error_exit;
      return conn;
    }

    *ctx = 0;
    if (!(conn = BIO_new_connect(host))) goto error_exit;
    BIO_set_conn_int_port(conn, &port);
    if (BIO_do_connect(conn) <= 0) goto error_exit;
    return conn;
```

```
error_exit:
  if (conn) BIO_free_all(conn);
  return 0;
}
```

Jak stwierdzono wcześniej, funkcja `spc_connect()` podejmuje popołączeniową próbę przeprowadzenia weryfikacji certyfikatu zdalnego klienta. Jeśli zamiast tego chce się przeprowadzić weryfikację za pomocą ***listy akceptacji*** (ang. *whitelist*) lub w ogóle zrezygnować z jej przeprowadzania, należy wprowadzić odpowiednie zmiany w kodzie, wykorzystując recepturę 10.9 w zakresie weryfikacji za pomocą listy akceptacji.

Jeżeli połączenie zostanie zestawione poprawnie, zostanie zwrócony obiekt `BIO` bez względu na to, czy użyto funkcji `spc_connect_ssl()` czy `spc_connect()`. Dzięki temu obiektowi można później używać funkcji `BIO_read()` w celu czytania danych oraz `BIO_write()` w celu ich zapisywania. Można również używać innych funkcji `BIO`, takich jak `BIO_printf()`. Po zakończeniu działań, kiedy chce się zamknąć połączenie, zawsze należy użyć funkcji `BIO_free_all()` zamiast `BIO_free()` w celu usunięcia wszelkich dowiązanych filtrów `BIO`. Jeśli obiekt `BIO` z obsługą SSL otrzymano z którejś z powyższych funkcji, w zbiorze powiązań zawsze występują co najmniej dwa takie filtry: jeden dla filtra SSL oraz jeden dla połączenia gniazdowego.

Zobacz również

- Witryna główna OpenSSL: *http://www.openssl.org/*.
- Receptury 10.1, 10.5, 10.9, 10.10 oraz 10.11.

9.2. Tworzenie serwera SSL

Problem

Chcemy napisać serwer sieciowy, który będzie akceptował połączenia SSL z klientami.

Rozwiązanie

Utworzenie serwera komunikującego się za pomocą protokołu SSL nie różni się zbytnio od utworzenia adekwatnego klienta (patrz receptura 9.1). Wymagane jest przeprowadzenie pewnych dodatkowych działań konfiguracyjnych. W szczególności należy utworzyć obiekt `spc_x509store_t` (patrz receptura 10.5) z certyfikatem oraz kluczem prywatnym. Informacje zawarte w tym obiekcie są przesyłane do klientów w czasie wstępnej wymiany potwierdzeń. Ponadto znacznik `SPC_X509STORE_USE_CERTIFICATE` musi być ustawiony w obiekcie `spc_x509store_t`. Po jego utworzeniu należy wykonać wywołania służące do utworzenia nasłuchującego obiektu `BIO`, wprowadzenia go w stan nasłuchiwania oraz akceptowania nowych połączeń (krótkie omówienie obiektów `BIO` zawarto w recepturze 9.1).

Analiza

Po utworzeniu i pełnym zainicjalizowaniu obiektu `spc_x509store_t` pierwszym etapem tworzenia serwera SSL jest wywołanie funkcji `spc_listen()`. Nazwę hosta można podać jako `NULL`, co określa, że utworzone gniazdo powinno zostać powiązane ze wszystkimi interfejsami. Wszystkie inne informacje należy podać w formie ciągu znaków jako adres IP interfejsu powiązania. Przykładowo, ciąg `127.0.0.1` spowoduje, że obiekt `BIO` serwera zostanie powiązany jedynie z lokalnym interfejsem pseudosieci.

```
#include <stdlib.h>
#include <string.h>
#include <openssl/bio.h>
#include <openssl/ssl.h>

BIO *spc_listen(char *host, int port) {
  BIO   *acpt = 0;
  int   addr_length;
  char  *addr;

  if (port < 1 || port > 65535) return 0;
  if (!host) host = "*";
  addr_length = strlen(host) + 6;  /* 5 dla wartości typu int, 1 dla znaku dwukropka */
  if (!(addr = (char *)malloc(addr_length + 1))) return 0;
  snprintf(addr, addr_length + 1, "%s:%d", host, port);

  if ((acpt = BIO_new(BIO_s_accept())) != 0) {
    BIO_set_accept_port(acpt, addr);
    if (BIO_do_accept(acpt) <= 0) {
      BIO_free_all(acpt);
      acpt = 0;
    }
  }

  free(addr);
  return acpt;
}
```

Wywołanie funkcji `spc_listen()` tworzy obiekt `BIO`, który posiada odpowiednie gniazdo pozostające w trybie nasłuchiwania. Nie występują tu żadne działania faktycznie związane z SSL, ponieważ połączenie SSL powstaje dopiero wówczas, gdy zostanie utworzone nowe połączenie gniazdowe. Wywołanie funkcji `spc_listen()` jest nieblokujące i natychmiast zwraca wynik.

Kolejnym etapem jest wywołanie funkcji `spc_accept()` w celu zestawienia nowego gniazda oraz potencjalnego połączenia SSL między serwerem a zgłaszającym się klientem. Funkcja ta powinna być wywoływana w sposób powtarzalny w celu ciągłego przyjmowania połączeń, jednak należy pamiętać, że powoduje ona powstanie blokady, jeśli nie istnieją żadne połączenia oczekujące. Wywołanie funkcji `spc_accept()` zwraca nowy obiekt `BIO`, który jest połączeniem z nowym klientem lub wartość `NULL`, która określa, że w trakcie zestawiania połączenia wystąpił pewien błąd.

Funkcja `spc_accept()` automatycznie tworzy obiekt `SSL_CTX` w ten sam sposób co funkcja `spc_connect()` (patrz receptura 9.1). Jednak ze względu na sposób działania funkcji `spc_accept()` (jest ona wywoływana w sposób powtarzalny przy użyciu tego samego nadrzędnego obiektu `BIO` dla przyjmowania nowych połączeń) funkcję `spc_create_ssl()` należy wywołać samodzielnie w celu utworzenia pojedynczego obiektu `SSL_CTX`, który będzie współużytkowany przez wszystkie akceptowane połączenia.

```
BIO *spc_accept(BIO *parent, int ssl, spc_x509store_t *spc_store, SSL_CTX **ctx) {
  BIO *child = 0, *ssl_bio = 0;
  int our_ctx = 0;
  SSL *ssl_ptr = 0;

  if (BIO_do_accept(parent) <= 0) return 0;
  if (!(child = BIO_pop(parent))) return 0;

  if (ssl) {
    if (*ctx) {
      CRYPTO_add(&((*ctx)->references), 1, CRYPTO_LOCK_SSL_CTX);
      if (spc_store && spc_store != SSL_CTX_get_app_data(*ctx)) {
        SSL_CTX_set_cert_store(*ctx, spc_create_x509store(spc_store));
        SSL_CTX_set_app_data(*ctx, spc_store);
      }
    } else {
      *ctx = spc_create_sslctx(spc_store);
      our_ctx = 1;
    }

    if (!(ssl_ptr = SSL_new(*ctx))) goto error_exit;
    SSL_set_bio(ssl_ptr, child, child);
    if (SSL_accept(ssl_ptr) <= 0) goto error_exit;

    if (!(ssl_bio = BIO_new(BIO_f_ssl()))) goto error_exit;
    BIO_set_ssl(ssl_bio, ssl_ptr, 1);
    child = ssl_bio;
    ssl_bio = 0;
  }

  return child;

error_exit:
  if (child) BIO_free_all(child);
  if (ssl_bio) BIO_free_all(ssl_bio);
  if (ssl_ptr) SSL_free(ssl_ptr);
  if (*ctx) SSL_CTX_free(*ctx);
  if (our_ctx) *ctx = 0;
  return 0;
}
```

Kiedy zostaje przyjęte nowe połączenie gniazdowe, wywoływana jest funkcja `SSL_accept()` w celu przeprowadzenia procesu uzgadniania protokołu SSL. Certyfikat serwera (być może wraz z certyfikatami nadrzędnymi w łańcuchu certyfikatów, w zależności od sposobu skonfigurowania obiektu `spc_x509store_t`) zostaje przesłany, a jeśli certyfikat klienta jest potrzebny i zostanie pobrany, następuje jego weryfikacja. W razie zakończenia powodzeniem operacji uzgadniania zwrócony obiekt `BIO` zachowuje się dokładnie tak samo jak obiekt `BIO` zwracany przez funkcję `spc_connect()` lub `spc_connect_ssl()`. Bez względu na to, czy nowe połączenie zostało udanie zestawione, nasłuchujący obiekt `BIO` przekazany do funkcji `SSL_accept()` będzie gotowy do akceptacji następnego połączenia dla kolejnego wywołania tej funkcji.

Zobacz również

Receptury 9.1, 10.5.

9.3. Używanie mechanizmu buforowania sesji w celu zwiększenia wydajności serwerów SSL

Problem

Posiadamy parę klient-serwer, w której komunikacja odbywa się przy wykorzystaniu protokołu SSL. Ten sam klient często tworzy kilka połączeń z tym samym serwerem w krótkim czasie. Potrzebny jest sposób przyspieszenia procesu ponownego przyłączania klienta do serwera bez naruszenia schematu zabezpieczeń.

Rozwiązanie

Pojęcia ***sesji SSL*** oraz ***połączenia SSL*** często bywają mylone lub używane zamiennie, choć w rzeczywistości są to dwie różne rzeczy. Sesja SSL to zbiór parametrów i kluczy szyfrowania utworzonych w wyniku przeprowadzenia procesu uzgadniania protokołu SSL. Połączenie SSL to aktywna konwersacja między dwoma węzłami korzystającymi z sesji SSL. W normalnej sytuacji, kiedy zostanie zestawione połączenie SSL, proces uzgadniania służy do wynegocjowania parametrów, które stają się sesją. To właśnie ów proces negocjowania sprawia, że tworzenie połączeń SSL jest tak kosztowne.

Na szczęście istnieje możliwość buforowania sesji. Kiedy klient połączy się z serwerem i z powodzeniem zakończy normalny proces uzgadniania, zarówno klient, jak i serwer posiadają dostęp do parametrów sesji, tak więc następnym razem, kiedy klient nawiąże połączenie z serwerem, może po prostu ponownie wykorzystać tę samą sesję, co pozwala uniknąć narzutu związanego z negocjowaniem nowych parametrów oraz kluczy szyfrowania.

Analiza

Buforowanie sesji zwykle nie jest domyślnie aktywowane, jednak jest to dość proste zadanie do wykonania. OpenSSL wykonuje większość działań za użytkownika, choć niemal wszystkie ustawienia domyślne można zmodyfikować (można, na przykład, utworzyć własny mechanizm buforowania po stronie serwera). Domyślnie OpenSSL używa mechanizmu buforowania wykorzystującego pamięć fizyczną, jednak jeśli ma być buforowana duża liczba sesji lub jeśli sesje mają pozostawać trwałe między przeładowaniami systemu, można wykorzystać pewien mechanizm buforowania dyskowego.

Większość zadań związanych z aktywowaniem buforowania sesji należy wykonać po stronie serwera, jednak nie jest ich zbyt dużo.

1. Ustawiamy kontekst identyfikatora sesji. Jego celem jest zapewnienie, że sesja będzie wykorzystywana ponownie w tym samym celu, w jakim została utworzona. Przykładowo, sesja utworzona dla serwera WWW SSL nie powinna automatycznie uwzględniać połączeń SSL serwera FTP. Kontekstem identyfikatora sesji mogą być dowolne dane binarne liczące maksymalnie 32 bajty długości. Nie ma ograniczeń co do postaci tych danych oprócz tego, że powinny one być unikatowe w zakresie usług świadczonych przez serwer — nie do zaakceptowania jest sytuacja, w której serwer przyjmuje sesje innych serwerów.

2. Ustawiamy czas ważności sesji. W przypadku OpenSSL domyślnie jest to 300 sekund, co w przypadku większości aplikacji jest wartością rozsądną. Kiedy sesja traci ważność, nie zostaje od razu usunięta z bufora serwera, jednak nie będzie akceptowana w przypadku prezentacji przez klienta. Jeżeli klient podejmie próbę użycia wygasłej sesji, serwer usunie ją ze swojej pamięci podręcznej.
3. Ustawiamy tryb buforowania. OpenSSL obsługuje wiele możliwych opcji trybów określanych za pomocą masek bitowych:

`SSL_SESS_CACHE_OFF`
: Ustawienie tego trybu wyłącza mechanizm buforowania sesji. Jeżeli chce się to zrobić, wystarczy określić tylko ten znacznik — nie ma potrzeby ustawiania kontekstu identyfikatora sesji ani czasu ważności sesji.

`SSL_SESS_CACHE_SERVER`
: Ustawienie tego trybu powoduje, że sesje generowane przez serwer są buforowane. Jest to tryb domyślny i powinien być uwzględniany zawsze, kiedy określa się inne opisywane tu znaczniki, oprócz `SSL_SESS_CACHE_OFF`.

`SSL_SESS_CACHE_NO_AUTO_CLEAR`
: Domyślnie bufor sesji jest sprawdzany pod względem wygasłych wpisów co każde 255 zestawionych połączeń. Niekiedy może to powodować niepożądane opóźnienie i może się wówczas okazać przydatne dezaktywowanie takiego automatycznego czyszczenia bufora. W razie ustawienia tego trybu należy zapewnić, aby okresowo była wywoływana funkcja `SSL_CTX_flush_sessions()`.

`SSL_SESS_CACHE_NO_INTERNAL_LOOKUP`
: Jeżeli chce się zastąpić wewnętrzny mechanizm buforowania OpenSSL własnym, należy ustawić ten tryb.

W celu aktywowania mechanizmu buforowania po stronie serwera można używać opisanych poniżej funkcji pomocniczych. Jeżeli zamierza się ich używać wraz z funkcjami serwera SSL przedstawionymi w recepturze 9.2, należy samodzielnie utworzyć obiekt `SSL_CTX` przy użyciu funkcji `spc_create_sslctx()`. Następnie należy wywołać funkcję `spc_enable_sessions()`, używając obiektu `SSL_CTX` i przekazać go do funkcji `spc_accept()`, tak aby nie został automatycznie utworzony nowy taki obiekt. Bez względu na to, czy się aktywuje mechanizm buforowania sesji czy nie, dobrym pomysłem jest utworzenie własnego obiektu `SSL_CTX` przed wywołaniem funkcji `spc_accept()`, tak aby nowy taki obiekt nie był tworzony dla każdego połączenia klienckiego.

```
#include <openssl/bio.h>
#include <openssl/ssl.h>

void spc_enable_sessions(SSL_CTX *ctx, unsigned char *id, unsigned int id_len,
                         long timeout, int mode) {
  SSL_CTX_set_session_id_context(ctx, id, id_len);
  SSL_CTX_set_timeout(ctx, timeout);
  SSL_CTX_set_session_cache_mode(ctx, mode);
}
```

Aktywowanie buforowania sesji po stronie klienta jest jeszcze prostsze. Wszystko, czego potrzeba, to ustawienie obiektu `SSL_SESSION` w obiekcie `SSL_CTX` przed faktycznym zestawieniem połączenia. Poniższa funkcja `spc_reconnect()` stanowi reimplementację funkcji `spc_connect_ssl()` poprzez wprowadzenie zmian potrzebnych do aktywowania buforowania sesji po stronie klienta.

```
BIO *spc_reconnect(char *host, int port, SSL_SESSION *session,
                   spc_x509store_t *spc_store, SSL_CTX **ctx) {
  BIO *conn = 0;
  int our_ctx = 0;
  SSL *ssl_ptr;

  if (*ctx) {
    CRYPTO_add(&((*ctx)->references), 1, CRYPTO_LOCK_SSL_CTX);
    if (spc_store && spc_store != SSL_CTX_get_app_data(*ctx)) {
      SSL_CTX_set_cert_store(*ctx, spc_create_x509store(spc_store));
      SSL_CTX_set_app_data(*ctx, spc_store);
    }
  } else {
    *ctx = spc_create_sslctx(spc_store);
    our_ctx = 1;
  }

  if (!(conn = BIO_new_ssl_connect(*ctx))) goto error_exit;
  BIO_set_conn_hostname(conn, host);
  BIO_set_conn_int_port(conn, &port);

  if (session) {
    BIO_get_ssl(conn, &ssl_ptr);
    SSL_set_session(ssl_ptr, session);
  }
  if (BIO_do_connect(conn) <= 0) goto error_exit;
  if (!our_ctx) SSL_CTX_free(*ctx);
  if (session) SSL_SESSION_free(session);
  return conn;

error_exit:
  if (conn) BIO_free_all(conn);
  if (*ctx) SSL_CTX_free(*ctx);
  if (our_ctx) *ctx = 0;
  return 0;
}
```

Zestawienie połączenia SSL w roli klienta może być równie proste jak ustawienie obiektu `SSL_SESSION` w obiekcie `SSL_CTX`, jednak pojawia się pytanie, skąd się wziął ów tajemniczy obiekt `SSL_SESSION`. W momencie zestawiania połączenia OpenSSL tworzy obiekt sesji SSL i ukrywa go w obiekcie `SSL`, który normalnie jest ukryty w obiekcie `BIO` zwracanym przez funkcję `spc_connect_ssl()`. Można go pobrać, wywołując funkcję `spc_getsession()`.

```
SSL_SESSION *spc_getsession(BIO *conn) {
  SSL *ssl_ptr;

  BIO_get_ssl(conn, &ssl_ptr);
  if (!ssl_ptr) return 0;
  return SSL_get1_session(ssl_ptr);
}
```

Obiekt `SSL_SESSION` zwracany przez funkcję `spc_getsession()` posiada zwiększony licznik odwołań, tak więc należy zapewnić wywołanie w pewnym momencie funkcji `SSL_SESSION_free()`w celu zwolnienia tego odwołania. Obiekt `SSL_SESSION` można otrzymać od razu po udanym zestawieniu połączenia, jednak ponieważ wartość może ulec zmianie między momentem zestawienia połączenia a momentem jego zakończenia ze względu na proces renegocjacji, obiekt `SSL_SESSION` zawsze należy pobierać tuż przed zakończeniem połączenia. W ten sposób można zapewnić sobie posiadanie najnowszego obiektu sesji.

Zobacz również

Receptura 9.2.

9.4. Zabezpieczanie komunikacji sieciowej na platformie Windows przy użyciu interfejsu WinInet API

Problem

Opracowujemy program na platformie Windows, który musi łączyć się z serwerem HTTP przy użyciu SSL. Chcemy użyć interfejsu firmy Microsoft WinInet API w celu prowadzenia komunikacji z tym serwerem.

Rozwiązanie

Interfejs WinInet API firmy Microsoft został wprowadzony wraz z przeglądarką Internet Explorer 3.0. Oferuje on zestaw funkcji pozwalających programom na łatwe uzyskiwanie dostępu do serwerów FTP, Gopher, HTTP oraz HTTPS. W przypadku tych ostatnich szczegóły użycia protokołu SSL są ukryte przed programistą, co pozwala mu skoncentrować się na danych, które muszą być wymienione, a nie samych szczegółach protokołu.

Analiza

WinInet API to bogaty interfejs, który znacznie ułatwia klientom interakcję z serwerami FTP, Gopher, HTTP oraz HTTPS. Jednak podobnie jak w przypadku większości innych interfejsów API systemu Windows wciąż wymagane jest pisanie sporych porcji kodu. Ze względu na bogactwo dostępnych opcji poniżej nie zostanie zaprezentowany pełny przykład działającego kodu. Zamiast tego zostanie omówiony sam interfejs oraz zaprezentowane będą przykłady kodu dla tych jego części, które są interesujące z punktu widzenia kwestii bezpieczeństwa. Autorzy zachęcają Czytelnika do sięgnięcia po dokumentację firmy Microsoft poświęconą temu interfejsowi w celu zapoznania się z jego możliwościami.

Jeżeli zamierza się zestawić połączenie z serwerem sieciowym przy użyciu protokołu SSL z wykorzystaniem interfejsu WinInet, pierwszą rzeczą, jaką należy zrobić, jest utworzenie sesji internetowej poprzez wywołanie funkcji `InternetOpen()`. Inicjalizuje ona i zwraca uchwyt do obiektu wymaganego do faktycznego zestawienia połączenia. Należy zadbać o takie szczegóły jak prezentacja użytkownikowi interfejsu użytkownika mechanizmu łączenia komutowanego, jeżeli nie jest on jeszcze podłączony do internetu, a system został tak skonfigurowany. Chociaż pojedyncza aplikacja może wykonać nieograniczoną liczbę wywołań funkcji `InternetOpen()`, zwykle wymagane jest tylko jednokrotne jej wywołanie. Zwracany uchwyt można używać wielokrotnie.

```
#include <windows.h>
#include <wininet.h>
```

```
HINTERNET hInternetSession;
LPSTR     lpszAgent       = "C i C++. Bezpieczne programowanie. Receptury, receptura 9.4";
DWORD     dwAccessType    = INTERNET_OPEN_TYPE_PROXY;
LPSTR     lpszProxyName   = 0;
LPSTR     lpszProxyBypass = 0;
DWORD     dwFlags         = 0;

hInternetSession = InternetOpen(lpszAgent, dwAccessType, lpszProxyName,
                                lpszProxyBypass, dwFlags);
```

Jeżeli jako wartość `dwAccessType` ustawi się `INTERNET_OPEN_TYPE_PROXY`, `lpszName` — 0, zaś `lpszProxyBypass` — 0, użyte zostaną domyślne ustawienia systemowe dla protokołu HTTP. Jeżeli system skonfigurowano tak, aby korzystał z serwera pośredniczącego, zostanie on użyty zgodnie z wymaganiami. Argument `lpszAgent` jest przekazywany do serwerów jako ciąg znaków agenta HTTP klienta. Może to być dowolna wartość lub ten sam ciąg znaków, który określona przeglądarka internetowa przesyła do serwera sieciowego w momencie zgłoszenia żądania.

Kolejnym etapem jest połączenie się z serwerem. Dokonuje się tego, wywołując funkcję `InternetConnect()`, która zwraca nowy uchwyt do obiektu przechowującego wszystkie odpowiednie informacje o połączeniu. Dwa narzucające się wymagane parametry dla tej funkcji to nazwa serwera, z którym ma zostać nawiązane połączenie, oraz port połączenia. Nazwę serwera można podać w formie nazwy hosta lub adresu IP z rozdzielonymi kropkami wartościami dziesiętnymi. Port można określić jako liczbę lub użyć stałej `INTERNET_DEFAULT_HTTPS_PORT` w celu połączenia się z domyślnym portem SSL o numerze 443.

```
HINTERNET     hConnection;
LPSTR         lpszServerName = "www.helion.pl";
INTERNET_PORT nServerPort    = INTERNET_DEFAULT_HTTPS_PORT;
LPSTR         lpszUsername   = 0;
LPSTR         lpszPassword   = 0;
DWORD         dwService      = INTERNET_SERVICE_HTTP;
DWORD         dwFlags        = 0;
DWORD         dwContext      = 0;

hConnection = InternetConnect(hInternetSession, lpszServerName, nServerPort,
                              lpszUsername, lpszPassword, dwService, dwFlags,
                              dwContext);
```

Wywołanie funkcji `InternetConnect()` zestawia połączenie z serwerem zdalnym. Jeżeli próba połączenia zakończy się z pewnego powodu niepowodzeniem, zwracaną wartością jest `NULL`, zaś kod błędu można pobrać przy użyciu funkcji `GetLastError()`. W przeciwnym razie zostaje zwrócony nowy uchwyt do obiektu. Jeżeli wymagane jest zgłoszenie kilku żądań względem tego samego serwera, należy używać tego samego uchwytu w celu uniknięcia narzutu związanego z tworzeniem wielu połączeń.

Po zestawieniu połączenia z serwerem należy zbudować obiekt żądania. Obiekt ten jest kontenerem przechowującym różne informacje: zasób, którego będzie dotyczyć żądanie, nagłówki, które zostaną przesłane, zestaw znaczników, które określają zachowanie żądania, informacje nagłówkowe zwrócone przez serwer po przesłaniu żądania oraz inne. Nowy obiekt żądania konstruuje się, wywołując funkcję `HttpOpenRequest()`.

```
HINTERNET hRequest;
LPSTR     lpszVerb        = "GET";
LPSTR     lpszObjectName  = "/";
LPSTR     lpszVersion     = "HTTP/1.1";
LPSTR     lpszReferer     = 0;
LPSTR     lpszAcceptTypes = 0;
```

```
DWORD     dwFlags          = INTERNET_FLAG_SECURE |
                             INTERNET_FLAG_IGNORE_REDIRECT_TO_HTTP |
                             INTERNET_FLAG_IGNORE_REDIRECT_TO_HTTPS;
DWORD     dwContext        = 0;

hRequest = HttpOpenRequest(hConnection, lpszVerb, lpszObjectName, lpszVersion,
                           lpszReferer, lpszAcceptTypes, dwFlags, dwContext);
```

Argument `lpszVerb` steruje typem żądania, które zostanie przesłane — może to być dowolne poprawne żądanie protokołu HTTP, takie jak `GET` lub `POST`. Argument `lpszObjectName` określa zasoby, których dotyczy żądanie, i zazwyczaj stanowi część adresu URL występującą po nazwie serwera, czyli rozpoczynającą się od znaku ukośnika i kończącą przed ciągiem zapytania (przed znakiem pytajnika). Określenie wartości argumentu `lpszAcceptTypes` jako 0 informuje serwer, że akceptowane są wszelkiego rodzaju dokumenty tekstowe. Jest to równoważne typowi MIME `text/*`.

Najbardziej interesującym argumentem przekazywanym do funkcji `HttpOpenRequest()` jest `dwFlags`. Definiuje on wiele znaczników, ale tylko kilka z nich ma bezpośredni związek z użyciem protokołu HTTP poprzez SSL.

`INTERNET_FLAG_IGNORE_CERT_CN_INVALID`
: W normalnej sytuacji jako element procesu weryfikacji certyfikatu serwera WinInet sprawdza, czy nazwa hosta jest zawarta w polu `commonName` lub rozszerzeniu `subjectAltName` certyfikatu. W razie określenia tego znacznika sprawdzenie nazwy hosta nie odbywa się (w recepturach 10.4 oraz 10.8 omówiono znaczenie przeprowadzania sprawdzeń nazw hostów w przypadku certyfikatów).

`INTERNET_FLAG_IGNORE_CERT_DATE_INVALID`
: Istotną częścią procesu weryfikacji poprawności certyfikatu X.509 jest sprawdzenie dat jego obowiązywania. Jeżeli bieżąca data jest datą spoza poprawnego zakresu dat certyfikatu, powinien on być traktowany jako niepoprawny. W razie określenia tego znacznika sprawdzenie poprawności dat certyfikatu nie odbywa się. Opcja ta nie powinna być nigdy używana w przypadku finalnej wersji danego produktu.

`INTERNET_FLAG_IGNORE_REDIRECT_TO_HTTP`
: W razie określenia tego znacznika, kiedy serwer podejmuje próbę przekierowania klienta pod adres niewykorzystujący protokołu SSL, przekierowanie takie zostanie zignorowane. Zawsze należy uwzględniać ten znacznik, tak aby zapewnić, że dane, które powinny być chronione, nie będą przesyłane w postaci jawnej.

`INTERNET_FLAG_IGNORE_REDIRECT_TO_HTTPS`
: W razie określenia tego znacznika, kiedy serwer podejmuje próbę przekierowania klienta pod adres wykorzystujący protokół SSL, przekierowanie takie zostanie zignorowane. Jeżeli można oczekiwać, że komunikacja będzie się odbywała tylko w ramach serwerów pozostających pod naszą kontrolą, można ją pominąć. W przeciwnym wypadku warto wziąć pod uwagę jej uwzględnienie, aby zapobiec przekierowaniom do miejsc innych niż oczekiwane.

`INTERNET_FLAG_SECURE`
: Jest to bardzo istotny znacznik. Jego określenie powoduje aktywowanie użycia protokołu SSL w ramach połączenia. Bez niego SSL nie jest używany i wszystkie dane są przesyłane w postaci jawnej. Oczywiście należy go uwzględniać.

Po skonstruowaniu obiektu żądania należy przesłać żądanie do serwera. Robi się to, wywołując funkcję `HttpSendRequest()` z obiektem żądania. Można dołączyć dodatkowe nagłówki,

jak również opcjonalne dane przesyłane po nagłówkach. Takie dane przesyła się w przypadku wykonywania operacji `POST`. Dodatkowe nagłówki i opcjonalne dane określa się jako ciągi znaków oraz długości tych ciągów.

```
BOOL  bResult;
LPSTR lpszHeaders      = 0;
DWORD dwHeadersLength  = 0;
LPSTR lpszOptional     = 0;
DWORD dwOptionalLength = 0;

bResult = HttpSendRequest(hRequest, lpszHeaders, dwHeadersLength, lpOptional,
                          dwOptionalLength);
```

Po przesłaniu żądania można odebrać odpowiedź serwera. W ramach procesu przesyłania żądania WinInet pobiera nagłówki odpowiedzi z serwera. Informacje dotyczące odpowiedzi można pobrać przy użyciu funkcji `HttpQueryInfo()`. Pełną listę informacji, jakie mogą być dostępne, można znaleźć w dokumentacji interfejsu WinInet, jednak dla naszych celów istotna jest jedynie długość treści. Serwer nie musi odsyłać nagłówka długości treści w ramach swojej odpowiedzi, tak więc musimy zapewnić sobie możliwość obsłużenia również takiej sytuacji, w której nie zostanie on przesłany. Dane odpowiedzi przesyłane przez serwer po jej nagłówkach można pobrać, wywołując funkcję `InternetReadFile()` tyle razy, ile jest to konieczne w celu pobrania wszystkich danych.

```
DWORD dwContentLength, dwIndex, dwInfoLevel;
DWORD dwBufferLength, dwNumberOfBytesRead, dwNumberOfBytesToRead;
LPVOID lpBuffer, lpFullBuffer, lpvBuffer;

dwInfoLevel    = HTTP_QUERY_CONTENT_LENGTH;
lpvBuffer      = (LPVOID)&dwContentLength;
dwBufferLength = sizeof(dwContentLength);
dwIndex        = 0;
HttpQueryInfo(hRequest, dwInfoLevel, lpvBuffer, &dwBufferLength, &dwIndex);
if (dwIndex != ERROR_HTTP_HEADER_NOT_FOUND) {
  /* Długość treści jest znana. Odczytujemy tylko taką ilość danych */
  lpBuffer = GlobalAlloc(GMEM_FIXED, dwContentLength);
  InternetReadFile(hRequest, lpBuffer, dwContentLength, &dwNumberOfBytesRead);
} else {
  /* Długość treści nie jest znana. Odczytujemy do momentu napotkania znaku EOF */
  dwContentLength = 0;
  dwNumberOfBytesToRead = 4096;
  lpFullBuffer = lpBuffer = GlobalAlloc(GMEM_FIXED, dwNumberOfBytesToRead);
  while (InternetReadFile(hRequest, lpBuffer, dwNumberOfBytesToRead,
                          &dwNumberOfBytesRead)) {
    dwContentLength += dwNumberOfBytesRead;
    if (dwNumberOfBytesRead != dwNumberOfBytesToRead) break;
    lpFullBuffer = GlobalReAlloc(lpFullBuffer, dwContentLength +
                                 dwNumberOfBytesToRead, 0);
    lpBuffer = (LPVOID)((LPBYTE)lpFullBuffer + dwContentLength);
  }
  lpFullBuffer = lpBuffer = GlobalReAlloc(lpFullBuffer, dwContentLength, 0);
}
```

Po odczytaniu danych za pomocą funkcji `InternetReadFile()` zmienna `lpBuffer` będzie zawierała treść odpowiedzi serwera, zaś zmienna `dwContentLength` będzie zawierała liczbę bajtów zawartych w buforze odpowiedzi. W tym momencie żądanie jest zakończone i obiekt odpowiedzi należy zniszczyć, wywołując funkcję `InternetCloseHandle()`. Jeżeli są wymagane dodatkowe żądania względem tego samego połączenia, można utworzyć nowy obiekt żądania i użyć tego samego uchwytu połączenia pochodzącego z wywołania funkcji `InternetConnect()`. Jeśli dane połączenie nie będzie już potrzebne do przesyłania żądań, należy użyć

funkcji `InternetCloseHandle()` w celu zamknięcia połączenia. Wreszcie, kiedy obiekt sesji internetowej utworzonej przez funkcję `InternetConnect()`nie jest już używany, należy wywołać funkcję `InternetCloseHandle()` w celu usunięcia także tego obiektu.

```
InternetCloseHandle(hRequest);
InternetCloseHandle(hConnection);
InternetCloseHandle(hInternetSession);
```

Zobacz również

Receptury 10.4, 10.8.

9.5. Aktywowanie protokołu SSL bez modyfikowania kodu źródłowego

Problem

Posiadamy klienta lub serwer, który nie obsługuje protokołu SSL, a chcemy zapewnić obsługę SSL bez konieczności modyfikowania kodu źródłowego.

Rozwiązanie

Stunnel to program wykorzystujący pakiet OpenSSL w celu tworzenia tuneli SSL między klientami a serwerami, które nie zapewniają wewnętrznej obsługi protokołu SSL. W czasie pisania niniejszej książki jego najnowsza wersja nosiła numer 4.04 i była dostępna dla systemów Unix oraz Windows pod adresem *http://www.stunnel.org*. W przypadku serwerów nasłuchuje on na innym gnieździe połączeń SSL i przekazuje dane dwukierunkowo do prawdziwego serwera poprzez połączenie nieobsługujące SSL. Klienty obsługujące SSL mogą wówczas łączyć się przez port nasłuchu programu Stunnel i komunikować z serwerem, który nie obsługuje SSL. W przypadku klientów nasłuch odbywa się na gnieździe obsługującym połączenia niewykorzystujące SSL, a dane są przekazywane dwukierunkowo do serwera w ramach połączenia chronionego przez SSL.

Stunnel powstał kilka lat temu i początkowo wykorzystywał przełączniki określane w wierszu poleceń w celu sterowania swoim działaniem. Zmieniło się to dopiero od wersji 4.00. Obecnie Stunnel wykorzystuje w tym celu plik konfiguracyjny i wszystkie wcześniej obsługiwane przełączniki wiersza poleceń zostały usunięte. Niniejsza receptura odnosi się do wersji 4.04.

Analiza

Chociaż w niniejszej recepturze nie zawarto żadnego kodu, znalazła się ona w książce dlatego, że autorzy sądzą, iż Stunnel to narzędzie warte omówienia, szczególnie w sytuacji, gdy opracowuje się klienty i serwery wykorzystujące protokół SSL. Podjęcie próby opracowania od podstaw i usunięcia błędów z oprogramowania klientów i serwerów wykorzystujących SSL może okazać się bardzo frustrującym przeżyciem, szczególnie jeśli nie posiada się doświadczenia w zakresie programowania z użyciem SSL. Program Stunnel może pomóc w usuwaniu błędów z kodu SSL.

Plik konfiguracyjny programu Stunnel jest podzielony na sekcje. Każda z nich zawiera zestaw kluczy, zaś każdy klucz posiada związaną z nim wartość. Sekcje i klucze są nazwane i wielkość liter nie ma znaczenia. Plik konfiguracyjny jest analizowany od początku, sekcje są rozdzielone wierszami zawierającymi ich nazwy ujęte w nawiasy kwadratowe. Pozostałe wiersze zawierają pary klucz-wartość, które należą do sekcji bieżącej. Ponadto przed pierwszą nazwaną sekcją występuje opcjonalna globalna sekcja nienazwana. Klucze i wartości oddziela znak równości (=).

Komentarze mogą się rozpoczynać tylko od początku wiersza (dopuszczalne jest występowanie najpierw znaków odstępu) i ich pierwszym znakiem jest krzyżyk (#); cały taki wiersz jest traktowany jako komentarz. Wszelkie wiodące lub kończące znaki odstępu otaczające klucz lub wartość są pomijane. Wszelkie inne znaki odstępu mają znaczenie, w tym wiodące lub kończące znaki odstępu otaczające nazwę sekcji (występującą w nawiasach kwadratowych). Przykładowo, zapis `[ moja_sekcja ]` nie jest równoważny zapisowi `[moja_sekcja]`. Dokumentacja dołączona do programu Stunnel szczegółowo opisuje obsługiwane klucze, więc tutaj pominiemy ich opis.

Jedną z przydatnych cech pliku konfiguracyjnego w porównaniu ze starym interfejsem wiersza poleceń jest to, że każda sekcja definiuje albo klienta, albo serwer, tak więc pojedyncza instancja programu Stunnel może być używana w celu uruchamiania wielu klientów lub serwerów. Jeżeli chce się uruchamiać zarówno klienty, jak i serwery, potrzebne jest uruchomienie dwóch instancji programu Stunnel, ponieważ znacznik określający, w jakim trybie ma on działać, jest opcją globalną. W przypadku interfejsu wiersza poleceń wymaganych było wiele instancji programu — po jednej dla każdego klienta lub serwera, którego chciało się uruchomić. Stąd, w przypadku chęci używania Stunnel dla połączeń serwerów POP3, IMAP oraz SMTP, należało uruchomić trzy instancje programu.

Nazwa każdej sekcji definiuje nazwę usługi, która będzie używana z mechanizmem opakowania protokołu TCP oraz w celach rejestrowania. Zarówno w przypadku klientów, jak i serwerów należy określić klucze `accept` oraz `connect`. Klucz `accept` określa port, na którym Stunnel będzie nasłuchiwać połączeń przychodzących, zaś klucz `connect` określa port, którym Stunnel będzie podejmował próby łączenia się w przypadku połączeń wychodzących. Klucze te muszą co najmniej określać numer portu, ale mogą również opcjonalnie zawierać nazwę hosta lub adres IP. W celu ich uwzględnienia należy poprzedzić numer portu nazwą hosta lub adresem IP i rozdzielić je znakiem dwukropka (`:`).

Tryb działania programu Stunnel można aktywować na dwa sposoby.

Tryb serwera
: W celu aktywowania trybu serwera należy ustawić klucz opcji globalnej `client` na wartość `no`. W czasie działania w trybie serwera Stunnel oczekuje, że połączenia przychodzące będą się komunikować z użyciem protokołu SSL, zaś połączenia wychodzące będą obsługiwane bez niego. Należy również ustawić dwie opcje globalne `cert` oraz `key` na nazwy plików zawierających certyfikat oraz używany klucz.

Tryb klienta
: W celu aktywowania trybu klienta należy ustawić klucz opcji globalnej `client` na wartość `yes`. W tym trybie Stunnel oczekuje, że połączenia przychodzące nie będą obsługiwać protokołu SSL, zaś połączenia wychodzące będą szyfrowane za pomocą tego protokołu. Można również określić certyfikat i klucz, ale nie jest to wymagane.

Poniższy przykład powoduje uruchomienie dwóch serwerów. Pierwszy z nich to IMAP wykorzystujący SSL, który nasłuchuje połączeń SSL na porcie 993 i przekierowuje ruch bez użycia SSL do połączenia na porcie 143. Drugim jest serwer POP3 wykorzystujący SSL, który nasłuchuje połączeń SSL tylko na porcie 995 interfejsu maszyny lokalnej (127.0.0.1). Połączenia wychodzące są przeprowadzane za pomocą portu 110 w ramach interfejsu lokalnego.

```
client = no
cert   = /home/mmessier/ssl/servercert.pem
key    = /home/mmessier/ssl/serverkey.pem

[impas]
accept  = 993
connect = 143

[pop3]
accept  = localhost:995
connect = localhost:110
```

W poniższym przykładzie Stunnel działa w trybie klienta. Nasłuchuje połączeń przychodzących w ramach interfejsu lokalnego na porcie 25 i przekierowuje ruch do portu 465 na serwerze o adresie *smtp.secureprogramming.com*. Przykład ten może być przydatny dla klientów poczty elektronicznej, które nie obsługują protokołu SMTP szyfrowanego za pomocą SSL.

```
client = yes

[smtp]
accept  = localhost:25
connect = smtp.secureprogramming.com
```

Zobacz również

Witryna internetowa programu Stunnel: *http://www.stunnel.com*.

9.6. Używanie szyfrowania standardu Kerberos

Problem

Musimy użyć szyfrowania w kodzie, który wykorzystuje już standard uwierzytelniania Kerberos.

Rozwiązanie

Kerberos to w głównej mierze usługa uwierzytelniająca stosowana w przypadku usług sieciowych. Efektem ubocznym wymagań związanych z przeprowadzaniem procesu uwierzytelniania jest to, że Kerberos oferuje interfejs API dla szyfrowania i deszyfrowania, aczkolwiek liczba obsługiwanych szyfrów jest znacznie mniejsza niż ma to miejsce w przypadku innych protokołów kryptograficznych. Proces uwierzytelniania powoduje utworzenie kryptograficznie silnego klucza sesji, który może być używany jako klucz szyfrowania.

Bieżąca receptura może być stosowana w przypadku systemów Unix oraz Windows z implementacjami standardu Kerberos Heimdal lub MIT. Prezentowany tu kod nie będzie działał poprawnie w systemach Windows, które oferują wewnętrzne wsparcie dla standardu Kerberos, ponieważ Windows nie udostępnia API Kerberos w sposób umożliwiający funkcjo-

```
#else
  if (key->keytype == ETYPE_DES_CBC_CRC || key->keytype == ETYPE_DES_CBC_MD4 ||
      key->keytype == ETYPE_DES_CBC_MD5 || key->keytype == ETYPE_DES_CBC_NONE ||
      key->keytype == ETYPE_DES_CFB64_NONE || key->keytype == ETYPE_DES_PCBC_NONE)
    return 1;
#endif
  return 0;
}
```

Następnie obiekty `krb5_context` oraz `krb5_keyblock` mogą zostać wspólnie użyte jako argumenty wywołania funkcji `spc_krb5_encrypt()`, którą implementujemy poniżej. Funkcja wymaga również bufora, który będzie przechowywał dane do zaszyfrowania, rozmiaru tego bufora, jak również wskaźnika w celu pobrania dynamicznie alokowanego bufora, który będzie zawierał zaszyfrowane dane, oraz wskaźnika w celu pobrania rozmiaru bufora zaszyfrowanych danych.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <krb5.h>

int spc_krb5_encrypt(krb5_context ctx, krb5_keyblock *key, void *inbuf,
                     size_t inlen, void **outbuf, size_t *outlen) {
#ifdef KRB5_GENERAL__
  size_t        blksz, newlen;
  krb5_data     in_data;
  krb5_enc_data out_data;

  if (krb5_c_block_size(ctx, key->enctype, &blksz)) return 0;
  if (!(inlen % blksz)) newlen = inlen + blksz;
  else newlen = ((inlen + blksz - 1) / blksz) * blksz;

  in_data.magic  = KV5M_DATA;
  in_data.length = newlen;
  in_data.data   = malloc(newlen);
  if (!in_data.data) return 0;

  memcpy(in_data.data, inbuf, inlen);
  spc_add_padding((unsigned char *)in_data.data + inlen, inlen, blksz);

  if (krb5_c_encrypt_length(ctx, key->enctype, in_data.length, outlen)) {
    free(in_data.data);
    return 0;
  }

  out_data.magic   = KV5M_ENC_DATA;
  out_data.enctype = key->enctype;
  out_data.kvno    = 0;
  out_data.ciphertext.magic  = KV5M_ENCRYPT_BLOCK;
  out_data.ciphertext.length = *outlen;
  out_data.ciphertext.data   = malloc(*outlen);
  if (!out_data.ciphertext.data) {
    free(in_data.data);
    return 0;
  }

  if (krb5_c_encrypt(ctx, key, 0, 0, &in_data, &out_data)) {
    free(in_data.data);
    return 0;
  }

  *outbuf = out_data.ciphertext.data;
  free(in_data.data);
```

```
  return 1;
#else
  int           result;
  void          *tmp;
  size_t        blksz, newlen;
  krb5_data     edata;
  krb5_crypto   crypto;

  if (krb5_crypto_init(ctx, key, 0, &crypto) != 0) return 0;

  if (krb5_crypto_getblocksize(ctx, crypto, &blksz)) {
    krb5_crypto_destroy(ctx, crypto);
    return 0;
  }
  if (!(inlen % blksz)) newlen = inlen + blksz;
  else newlen = ((inlen + blksz - 1) / blksz) * blksz;
  if (!(tmp = malloc(newlen))) {
    krb5_crypto_destroy(ctx, crypto);
    return 0;
  }
  memcpy(tmp, inbuf, inlen);
  spc_add_padding((unsigned char *)tmp + inlen, inlen, blksz);

  if (!krb5_encrypt(ctx, crypto, 0, tmp, inlen, &edata)) {
    if ((*outbuf = malloc(edata.length)) != 0) {
      result = 1;
      memcpy(*outbuf, edata.data, edata.length);
      *outlen = edata.length;
    }
    krb5_data_free(&edata);
  }

  free(tmp);
  krb5_crypto_destroy(ctx, crypto);
  return result;
#endif
}
```

Funkcja deszyfrowania działa dokładnie tak jak funkcja szyfrowania. Należy pamiętać, że DES oraz Triple-DES to szyfry pracujące w trybie blokowym, tak więc może okazać się koniecznym uzupełnienie szyfrowanych danych, jeżeli ich rozmiar nie jest wielokrotnością rozmiaru bloku. Biblioteka Kerberos dokonuje wszelkich tego rodzaju uzupełnień automatycznie, jednak polega to na dodaniu bajtów zerowych, co nie jest zbyt dobrym rozwiązaniem. Dlatego też wykonujemy uzupełnianie we własnym zakresie, korzystając z kodu przedstawionego w recepturze 5.11 i używając w tym celu uzupełnienia blokowego PKCS.

```
#include <stdlib.h>
#include <string.h>
#include <krb5.h>

int spc_krb5_decrypt(krb5_context ctx, krb5_keyblock *key, void *inbuf,
                     size_t inlen, void **outbuf, size_t *outlen) {
#ifdef KRB5_GENERAL__
  int           padding;
  krb5_data     out_data;
  krb5_enc_data in_data;

  in_data.magic    = KV5M_ENC_DATA;
  in_data.enctype = key->enctype;
  in_data.kvno    = 0;
  in_data.ciphertext.magic  = KV5M_ENCRYPT_BLOCK;
  in_data.ciphertext.length = inlen;
  in_data.ciphertext.data   = inbuf;
```

```
  out_data.magic  = KV5M_DATA;
  out_data.length = inlen;
  out_data.data   = malloc(inlen);
  if (!out_data.data) return 0;

  if (krb5_c_block_size(ctx, key->enctype, &blksz)) {
    free(out_data.data);
    return 0;
  }
  if (krb5_c_decrypt(ctx, key, 0, 0, &in_data, &out_data)) {
    free(out_data.data);
    return 0;
  }

  if ((padding = spc_remove_padding((unsigned char *)out_data.data +
                             out_data.length - blksz, blksz)) == -1) {
    free(out_data.data);
    return 0;
  }

  *outlen = out_data.length - (blksz - padding);
  if (!(*outbuf = realloc(out_data.data, *outlen))) *outbuf = out_data.data;
  return 1;
#else
  int         padding, result;
  void        *tmp;
  size_t      blksz;
  krb5_data   edata;
  krb5_crypto crypto;

  if (krb5_crypto_init(ctx, key, 0, &crypto) != 0) return 0;
  if (krb5_crypto_getblocksize(ctx, crypto, &blksz) != 0) {
    krb5_crypto_destroy(ctx, crypto);
    return 0;
  }
  if (!(tmp = malloc(inlen))) {
    krb5_crypto_destroy(ctx, crypto);
    return 0;
  }
  memcpy(tmp, inbuf, inlen);
  if (!krb5_decrypt(ctx, crypto, 0, tmp, inlen, &edata)) {
    if ((padding = spc_remove_padding((unsigned char *)edata.data + edata.length -
                              blksz, blksz)) != -1) {
      *outlen = edata.length - (blksz - padding);
      if ((*outbuf = malloc(*outlen)) != 0) {
        result = 1;
        memcpy(*outbuf, edata.data, *outlen);
      }
    }
    krb5_data_free(&edata);
  }

  free(tmp);
  krb5_crypto_destroy(ctx, crypto);
  return result;
#endif
}
```

Zobacz również

Receptury 5.11, 5.25, 8.13.

9.7. Komunikacja międzyprocesowa przy użyciu gniazd

Problem

Posiadamy dwa lub większą liczbę procesów działających na tej samej maszynie, które muszą się ze sobą komunikować.

Rozwiązanie

Współczesne systemy operacyjne obsługują różnorodne elementarne mechanizmy komunikacji międzyprocesowej, które różnią się w przypadku różnych systemów. Jeżeli chce się zapewnić przenośność programu między różnymi platformami, czy wręcz różnymi implementacjami systemu Unix, najlepszym rozwiązaniem jest wykorzystanie gniazd. Wszystkie współczesne systemy operacyjne obsługują co najmniej interfejs gniazd standardu Berkeley dla protokołu TCP/IP, zaś większość — o ile nie wszystkie — implementacji Uniksa obsługuje również uniksowe gniazda domenowe.

Analiza

Wiele systemów operacyjnych obsługuje różne metody pozwalające dwóm lub większej liczbie procesów na komunikowanie się ze sobą. Większość systemów (w tym Unix i Windows) obsługuje potoki anonimowe i nazwane. Wiele systemów uniksowych (w tym BSD) obsługuje również kolejki komunikatów, których początki sięgają systemu uniksowego AT&T System V. Systemy Windows posiadają podobną konstrukcję, noszącą nazwę ***szczelin wysyłkowych*** (ang. *mailslots*). Systemy uniksowe posiadają także gniazda domenowe, które współdzielą interfejs gniazd standardu Berkeley z gniazdami TCP/IP. Poniżej przedstawiono przegląd najczęściej spotykanych mechanizmów.

Potoki anonimowe

Potoki anonimowe są przydatne w zakresie komunikacji między procesami nadrzędnym a potomnym. Proces nadrzędny może utworzyć dwa punkty końcowe potoku przed uruchomieniem procesu potomnego, zaś ten ostatni dziedziczy po nim deskryptory plików. Zarówno w systemie Unix, jak i Windows istnieją sposoby zapewnienia wymiany deskryptorów plików między dwoma pod innymi względami niepowiązanymi procesami, jednak jest to rzadko stosowane. W systemie Unix można skorzystać z gniazd domenowych, z kolei w systemie Windows można użyć funkcji interfejsu Win32 API `OpenProcess()` oraz `DuplicateHandle()`.

Potoki nazwane

Zamiast używania potoków anonimowych między niepowiązanymi procesami lepszym rozwiązaniem może okazać się użycie potoków nazwanych. W ich przypadku proces może utworzyć potok, który posiada skojarzoną ze sobą nazwę. Inny proces, który zna nazwę potoku, może następnie go otworzyć. W systemie Unix potoki nazwane stanowią w rzeczywistości pliki specjalne tworzone w systemie plików i nazwą potoku jest nazwa takiego pliku specjalnego. System Windows wykorzystuje specjalną przestrzeń nazw w jądrze i w rzeczywistości w ogóle nie używa systemu plików, choć ograniczenia co do nazwy nadawanej potokowi są podobne do tych obowiązujących w przypadku plików. Po-

toki sprawdzają się dobrze w sytuacji, gdy komunikacja dotyczy tylko dwóch procesów, gdyż dodawanie kolejnych procesów szybko komplikuje cały schemat. Potoków nie zaprojektowano z myślą o użyciu przez więcej niż dwa procesy naraz i w żadnej mierze nie zaleca się podejmowania prób takiego ich wykorzystywania.

Kolejki komunikatów (Unix)

Uniksowe kolejki komunikatów posiadają nazwy w postaci dowolnych wartości całkowitych nazywanych ***kluczami***. Często tworzony jest plik, którego i-węzeł jest używany jako klucz dla kolejki komunikatów. Dowolny proces, który ma prawo czytania z kolejki komunikatów, może to zrobić. Podobnie każdy proces posiadający odpowiednie uprawnienia może pisać do kolejki komunikatów. Kolejki komunikatów wymagają współpracy między procesami, które je wykorzystują. Złośliwy program może z łatwością naruszyć tę współpracę i wykraść komunikaty z kolejki. Kolejki komunikatów są również ograniczone pod tym względem, że potrafią obsługiwać dość niewielkie porcje danych.

Szczeliny wysyłkowe (Windows)

Szczeliny wysyłkowe systemu Windows mogą być nazywane tak jak ma to miejsce w przypadku potoków nazwanych, aczkolwiek można wyróżnić dwie oddzielne przestrzenie nazw. Szczeliny wysyłkowe stanowią jednokierunkowy mechanizm komunikacji. Tylko proces, który tworzy szczelinę, może z niej czytać. Inne procesy mogą jedynie zapisywać do niej. Szczeliny wysyłkowe sprawdzają się dobrze w sytuacji, gdy mamy do czynienia z pojedynczym procesem, który musi pobierać dane od innych procesów, jednak nie musi nic do nich odsyłać.

Gniazda

W dzisiejszych czasach trudno znaleźć system operacyjny, który nie obsługiwałby interfejsu gniazd standardu Berkeley dla gniazd TCP/IP. Większość połączeń TCP/IP zestawia się w ramach sieci między dwiema maszynami, jednak istnieje również możliwość połączenia przy użyciu protokołu TCP/IP dwóch procesów działających na jednej maszynie bez generowania jakiegokolwiek ruchu sieciowego. W systemach uniksowych ten sam interfejs może być używany również dla uniksowych gniazd domenowych, które są szybsze, jak i może służyć do wymiany deskryptorów plików oraz może być używany w celu wymiany danych uwierzytelniających (patrz receptura 9.8).

Używanie gniazd TCP/IP dla celów ***komunikacji międzyprocesowej*** (ang. *interprocess communication*, IPC) nie różni się zbytnio od używania ich dla celów komunikacji sieciowej. W rzeczywistości można ich używać w dokładnie taki sam sposób i wszystko powinno funkcjonować prawidłowo, jednak jeśli jest się zainteresowanym ich użyciem wyłącznie dla celów lokalnej komunikacji międzyprocesowej, istnieje kilka dodatkowych działań, jakie należy wykonać, co zostanie omówione poniżej.

Jeżeli dla celów lokalnej komunikacji międzyprocesowej używa się gniazd TCP/IP, najważniejszą rzeczą, jaką trzeba wiedzieć, jest to, że ***zawsze*** należy używać adresu pseudosieci. Kiedy dokonuje się powiązania gniazda, nie należy tego robić dla adresu `INADDR_ANY`, lecz dla 127.0.0.1. W przeciwnym razie będzie możliwe łączenie się z portem jedynie przy użyciu adresu 127.0.0.1. Oznacza to, że serwer będzie nieosiągalny dla innych maszyn bez względu na to, czy port będzie lub nie zablokowany przez zaporę sieciową.

W przypadku systemów Windows przedstawiony poniżej kod wykorzystuje wyłącznie gniazda TCP/IP, jednak w przypadku systemów uniksowych wprowadzono usprawnienie polegające na użyciu gniazd uniksowych, o ile jest używany adres pseudosieci 127.0.0.1. Utworzono

również kod opakowujący deskryptor gniazda, który nadzoruje rodzaj gniazda (uniksowe lub TCP/IP) oraz adres, z którym zostało ono powiązane. Informacje te są następnie używane w wywołaniach funkcji `spc_socket_accept()`, `spc_socket_sendto()` oraz `spc_socket_recvfrom()` działających jako kod opakowujący dla funkcji, odpowiednio, `accept()`, `sendto()` oraz `recvfrom()`.

Należy pamiętać, że w przypadku systemu Windows należy wywołać funkcję `WSAStartup()`, zanim będzie można używać jakichkolwiek funkcji gniazd. Należy również zapewnić wywołanie funkcji `WSACleanup()` po zakończeniu używania gniazd w swoim programie.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#ifndef WIN32
#include <errno.h>
#include <netdb.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#define INVALID_SOCKET  -1
#define closesocket(x)  close((x))
#else
#include <windows.h>
#include <winsock2.h>
#endif

#define SPC_SOCKETFLAG_BOUND  0x1
#define SPC_SOCKETFLAG_DGRAM  0x2

typedef struct {
#ifdef WIN32
  SOCKET          sd;
#else
  int             sd;
#endif
  int             domain;
  struct sockaddr *addr;
  int             addrlen;
  int             flags;
} spc_socket_t;

void spc_socket_close(spc_socket_t *);

static int make_sockaddr(int *domain, struct sockaddr **addr, char *host,
                         int port) {
  int                addrlen;
  in_addr_t          ipaddr;
  struct hostent     *he;
  struct sockaddr_in *addr_inet;

  if (!host) ipaddr = INADDR_ANY;
  else {
    if (!(he = gethostbyname(host))) {
      if ((ipaddr = inet_addr(host)) == INADDR_NONE) return 0;
    } else ipaddr = *(in_addr_t *)he->h_addr_list[0];
    endhostent();
  }
```

```
#ifndef WIN32
  if (inet_addr("127.0.0.1") == ipaddr) {
    struct sockaddr_un  *addr_unix;

    *domain = PF_LOCAL;
    addrlen = sizeof(struct sockaddr_un);
    if (!(*addr = (struct sockaddr *)malloc(addrlen))) return 0;
    addr_unix = (struct sockaddr_un *)*addr;
    addr_unix->sun_family = AF_LOCAL;
    snprintf(addr_unix->sun_path, sizeof(addr_unix->sun_path),
             "/tmp/127.0.0.1:%d", port);
#ifndef linux
    addr_unix->sun_len = SUN_LEN(addr_unix) + 1;
#endif
    return addrlen;
  }
#endif

  *domain = PF_INET;
  addrlen = sizeof(struct sockaddr_in);
  if (!(*addr = (struct sockaddr *)malloc(addrlen))) return 0;
  addr_inet = (struct sockaddr_in *)*addr;
  addr_inet->sin_family = AF_INET;
  addr_inet->sin_port = htons(port);
  addr_inet->sin_addr.s_addr = ipaddr;
  return addrlen;
}

static spc_socket_t *create_socket(int type, int protocol, char *host, int port) {
  spc_socket_t  *sock;

  if (!(sock = (spc_socket_t *)malloc(sizeof(spc_socket_t)))) return 0;
  sock->sd    = INVALID_SOCKET;
  sock->addr  = 0;
  sock->flags = 0;
  if (!(sock->addrlen = make_sockaddr(&sock->domain, &sock->addr, host, port)))
    goto error_exit;
  if ((sock->sd = socket(sock->domain, type, protocol)) == INVALID_SOCKET)
    goto error_exit;
  return sock;

error_exit:
  if (sock) spc_socket_close(sock);
  return 0;
}

void spc_socket_close(spc_socket_t *sock) {
  if (!sock) return;
  if (sock->sd != INVALID_SOCKET) closesocket(sock->sd);
  if (sock->domain == PF_LOCAL && (sock->flags & SPC_SOCKETFLAG_BOUND))
    remove(((struct sockaddr_un *)sock->addr)->sun_path);
  if (sock->addr) free(sock->addr);
  free(sock);
}

spc_socket_t *spc_socket_listen(int type, int protocol, char *host, int port) {
  int           opt = 1;
  spc_socket_t  *sock = 0;

  if (!(sock = create_socket(type, protocol, host, port))) goto error_exit;
  if (sock->domain == PF_INET) {
    if (setsockopt(sock->sd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt)) == -1)
      goto error_exit;
```

```
    if (bind(sock->sd, sock->addr, sock->addrlen) == -1) goto error_exit;
  } else {
    if (bind(sock->sd, sock->addr, sock->addrlen) == -1) {
      if (errno != EADDRINUSE) goto error_exit;
      if (connect(sock->sd, sock->addr, sock->addrlen) != -1) goto error_exit;
      remove(((struct sockaddr_un *)sock->addr)->sun_path);
      if (bind(sock->sd, sock->addr, sock->addrlen) == -1) goto error_exit;
    }
  }
  sock->flags |= SPC_SOCKETFLAG_BOUND;
  if (type == SOCK_STREAM && listen(sock->sd, SOMAXCONN) == -1) goto error_exit;
  else sock->flags |= SPC_SOCKETFLAG_DGRAM;
  return sock;

error_exit:
  if (sock) spc_socket_close(sock);
  return 0;
}

spc_socket_t *spc_socket_accept(spc_socket_t *sock) {
  spc_socket_t  *new_sock = 0;

  if (!(new_sock = (spc_socket_t *)malloc(sizeof(spc_socket_t))))
    goto error_exit;
  new_sock->sd      = INVALID_SOCKET;
  new_sock->domain  = sock->domain;
  new_sock->addrlen = sock->addrlen;
  new_sock->flags   = 0;
  if (!(new_sock->addr = (struct sockaddr *)malloc(sock->addrlen)))
    goto error_exit;

  if (!(new_sock->sd = accept(sock->sd, new_sock->addr, &(new_sock->addrlen))))
    goto error_exit;
  return new_sock;

error_exit:
  if (new_sock) spc_socket_close(new_sock);
  return 0;
}

spc_socket_t *spc_socket_connect(char *host, int port) {
  spc_socket_t  *sock = 0;

  if (!(sock = create_socket(SOCK_STREAM, 0, host, port))) goto error_exit;
  if (connect(sock->sd, sock->addr, sock->addrlen) == -1) goto error_exit;
  return sock;

error_exit:
  if (sock) spc_socket_close(sock);
  return 0;
}

int spc_socket_sendto(spc_socket_t *sock, const void *msg, int len, int flags,
                      char *host, int port) {
  int             addrlen, domain, result = -1;
  struct sockaddr *addr = 0;

  if (!(addrlen = make_sockaddr(&domain, &addr, host, port))) goto end;
  result = sendto(sock->sd, msg, len, flags, addr, addrlen);

end:
  if (addr) free(addr);
  return result;
}
```

```
int spc_socket_recvfrom(spc_socket_t *sock, void *buf, int len, int flags,
                        spc_socket_t **src) {
  int result;

  if (!(*src = (spc_socket_t *)malloc(sizeof(spc_socket_t)))) goto error_exit;
  (*src)->sd      = INVALID_SOCKET;
  (*src)->domain  = sock->domain;
  (*src)->addrlen = sock->addrlen;
  (*src)->flags   = 0;
  if (!((*src)->addr = (struct sockaddr *)malloc((*src)->addrlen)))
    goto error_exit;
  result = recvfrom(sock->sd, buf, len, flags, (*src)->addr, &((*src)->addrlen));
  if (result == -1) goto error_exit;
  return result;

error_exit:
  if (*src) {
    spc_socket_close(*src);
    *src = 0;
  }
  return -1;
}

int spc_socket_send(spc_socket_t *sock, const void *buf, int buflen) {
  int nb, sent = 0;

  while (sent < buflen) {
    nb = send(sock->sd, (const char *)buf + sent, buflen - sent, 0);
    if (nb == -1 && (errno == EAGAIN || errno == EINTR)) continue;
    if (nb <= 0) return nb;
    sent += nb;
  }

  return sent;
}

int spc_socket_recv(spc_socket_t *sock, void *buf, int buflen) {
  int nb, recvd = 0;

  while (recvd < buflen) {
    nb = recv(sock->sd, (char *)buf + recvd, buflen - recvd, 0);
    if (nb == -1 && (errno == EAGAIN || errno == EINTR)) continue;
    if (nb <= 0) return nb;
    recvd += nb;
  }

  return recvd;
}
```

Zobacz również

Receptura 9.8.

9.8. Uwierzytelnianie przy użyciu uniksowych gniazd domenowych

Problem

Przy użyciu uniksowego gniazda domenowego chcemy odkryć informacje o procesie, który znajduje się po drugiej stronie połączenia, takie jak identyfikator jego użytkownika lub grupy.

Rozwiązanie

Większość implementacji uniksowych gniazd domenowych zapewnia obsługę mechanizmu pobierania danych uwierzytelniających od procesów związanych z danym połączeniem. Używając tych informacji, można sprawdzić identyfikator użytkownika oraz grupy procesu znajdującego się po drugiej stronie połączenia. Dane uwierzytelniające nie są przekazywane automatycznie. W przypadku wszystkich implementacji odbierający musi jawnie poprosić o takie informacje. W przypadku niektórych implementacji informacje te muszą zostać przesłane jawnie. Ogólnie rzecz biorąc, kiedy projektuje się system, który będzie wymieniał dane uwierzytelniające, należy zapewnić po stronach połączenia koordynację przesyłania żądań i samych danych uwierzytelniających.

Niniejsza receptura dotyczy systemów FreeBSD, Linux oraz NetBSD. Niestety, nie wszystkie uniksowe implementacje gniazd domenowych oferują obsługę danych uwierzytelniających. W momencie pisania tej książki brak ten dotyczył jądra systemu Darwin (MacOS X), OpenBSD oraz Solaris.

Analiza

Oprócz wspomnianych powyżej ograniczeń co do obsługi mechanizmu uwierzytelniania w przypadku różnych platform drugim problemem jest to, że różne implementacje wymieniają dane na różne sposoby. Na przykład w przypadku systemów FreeBSD informacje muszą zostać jawnie przesłane i odbierający musi być w stanie obsłużyć ich odbiór. W systemach Linux informacje są przesyłane automatycznie, jeżeli odbierający poprosi o nie.

Trzecim problemem są różnice w zakresie przesyłanych danych w przypadku różnych implementacji. Linux przekazuje identyfikator procesu, identyfikator użytkownika oraz identyfikator grupy procesu przesyłającego. FreeBSD uwzględnia wszystkie grupy, do których należy proces, ale nie dotyczy to identyfikatora procesu. W najgorszym przypadku należy oczekiwać otrzymania tylko identyfikatorów użytkownika oraz grupy procesu.

```
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <sys/param.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/uio.h>
#if !defined(linux) && !defined(__NetBSD__)
#include <sys/ucred.h>
#endif
```

```
#ifndef SCM_CREDS
#define SCM_CREDS SCM_CREDENTIALS
#endif

#ifndef linux
#  ifndef __NetBSD__
#    define SPC_PEER_UID(c)   ((c)->cr_uid)
#    define SPC_PEER_GID(c)   ((c)->cr_groups[0])
#  else
#    define SPC_PEER_UID(c)   ((c)->sc_uid)
#    define SPC_PEER_GID(c)   ((c)->sc_gid)
#  endif
#else
#  define SPC_PEER_UID(c)   ((c)->uid)
#  define SPC_PEER_GID(c)   ((c)->gid)
#endif

#ifdef __NetBSD__
typedef struct sockcred spc_credentials;
#else
typedef struct ucred spc_credentials;
#endif

spc_credentials *spc_get_credentials(int sd) {
  int             nb, sync;
  char            ctrl[CMSG_SPACE(sizeof(struct ucred))];
  size_t          size;
  struct iovec    iov[1] = { { 0, 0 } };
  struct msghdr   msg = { 0, 0, iov, 1, ctrl, sizeof(ctrl), 0 };
  struct cmsghdr  *cmptr;
  spc_credentials *credentials;

#ifdef LOCAL_CREDS
  nb = 1;
  if (setsockopt(sd, 0, LOCAL_CREDS, &nb, sizeof(nb)) == -1) return 0;
#else
#ifdef SO_PASSCRED
  nb = 1;
  if (setsockopt(sd, SOL_SOCKET, SO_PASSCRED, &nb, sizeof(nb)) == -1) return 0;
#endif
#endif

  do {
    msg.msg_iov->iov_base = (void *)&sync;
    msg.msg_iov->iov_len  = sizeof(sync);
    nb = recvmsg(sd, &msg, 0);
  } while (nb == -1 && (errno == EINTR || errno == EAGAIN));
  if (nb == -1) return 0;

  if (msg.msg_controllen < sizeof(struct cmsghdr)) return 0;
  cmptr = CMSG_FIRSTHDR(&msg);
#ifndef __NetBSD__
  size = sizeof(spc_credentials);
#else
  if (cmptr->cmsg_len < SOCKCREDSIZE(0)) return 0;
  size = SOCKCREDSIZE(((cred *)CMSG_DATA(cmptr))->sc_ngroups);
#endif
  if (cmptr->cmsg_len != CMSG_LEN(size)) return 0;
  if (cmptr->cmsg_level != SOL_SOCKET) return 0;
  if (cmptr->cmsg_type != SCM_CREDS) return 0;

  if (!(credentials = (spc_credentials *)malloc(size))) return 0;
  *credentials = *(spc_credentials *)CMSG_DATA(cmptr);
  return credentials;
}
```

```
int spc_send_credentials(int sd) {
  int             sync = 0x11223344;
  struct iovec    iov[1] = { { 0, 0, } };
  struct msghdr   msg = { 0, 0, iov, 1, 0, 0, 0 };

#if !defined(linux) && !defined(__NetBSD__)
  char            ctrl[CMSG_SPACE(sizeof(spc_credentials))];
  struct cmsghdr  *cmptr;

  msg.msg_control    = ctrl;
  msg.msg_controllen = sizeof(ctrl);

  cmptr = CMSG_FIRSTHDR(&msg);
  cmptr->cmsg_len   = CMSG_LEN(sizeof(spc_credentials));
  cmptr->cmsg_level = SOL_SOCKET;
  cmptr->cmsg_type  = SCM_CREDS;
  memset(CMSG_DATA(cmptr), 0, sizeof(spc_credentials));
#endif

  msg.msg_iov->iov_base = (void *)&sync;
  msg.msg_iov->iov_len  = sizeof(sync);

  return (sendmsg(sd, &msg, 0) != -1);
}
```

Na wszystkich platformach istnieje możliwość otrzymania danych uwierzytelniających w dowolnym momencie połączenia, jednak często najlepszym rozwiązaniem jest pobranie tych danych tuż po nawiązaniu połączenia. Przykładowo, jeżeli serwer musi pobierać dane uwierzytelniające każdego klienta, który się łączy, jego kod mógłby mieć postać podobną do podanej poniżej.

```
typedef void (*spc_client_fn)(spc_socket_t *, spc_credentials *, void *);

void spc_unix_server(spc_client_fn callback, void *arg) {
  spc_socket_t    *client, *listener;
  spc_credentials *credentials;

  listener = spc_socket_listen(SOCK_STREAM, 0, "127.0.0.1", 2222);
  while ((client = spc_socket_accept(listener)) != 0) {
    if (!(credentials = spc_get_credentials(client->sd))) {
      printf("Nie można pobrać danych uwierzytelniających od łączącego się
klienta!\n");
      spc_socket_close(client);
    } else {
      printf("Dane uwierzytelniające klienta:\n\tuid: %d\n\tgid: %d\n",
             SPC_PEER_UID(credentials), SPC_PEER_GID(credentials));
      /* tu wykonanie pewnych działań związanych z danymi uwierzytelniającymi i połączeniem ... */
      callback(client, credentials, arg);
    }
  }
}
```

Odpowiedni kod klienta mógłby mieć postać jak poniżej.

```
spc_socket_t *spc_unix_connect(void) {
  spc_socket_t *conn;

  if (!(conn = spc_socket_connect("127.0.0.1", 2222))) {
    printf("Nie można nawiązać połączenia z serwerem!\n");
    return 0;
  }
  if (!spc_send_credentials(conn->sd)) {
    printf("Nie można przesłać danych uwierzytelniających do serwera!\n");
    spc_socket_close(conn);
    return 0;
  }
```

```
    printf("Dane uwierzytelniające zostały poprawnie przesłane do serwera.\n");
    return conn;
  }
```

Należy również zauważyć, że choć istnieje możliwość otrzymania danych uwierzytelniających w dowolnym momencie połączenia, wiele implementacji przesyła je tylko raz. Jeżeli dostęp do tych danych jest wymagany w więcej niż jednym momencie w czasie konwersacji, należy zapewnić, aby otrzymane informacje zostały zapisane przy pierwszym razie.

9.9. Zarządzanie identyfikatorami sesji

Problem

Aplikacja sieciowa wymaga, aby użytkownicy logowali się, zanim będą mogli wykonywać znaczące transakcje w ramach aplikacji. Kiedy użytkownik jest zalogowany, trzeba śledzić jego sesję aż do momentu, gdy się wyloguje.

Rozwiązanie

Rozwiązanie tego problemu jest proste. Jeżeli użytkownik poda poprawne hasło, generujemy identyfikator sesji i zwracamy go do klienta poprzez mechanizm cookie. Kiedy sesja jest aktywna, klient przesyła identyfikator sesji z powrotem do serwera, serwer weryfikuje go względem wewnętrznej tabeli sesji, która zawiera odpowiednie informacje o użytkowniku związane z każdym identyfikatorem sesji. Pozwala to serwerowi na kontynuowanie działań bez konieczności każdorazowego wymagania od klienta przesyłania nazwy użytkownika i hasła. W celu zapewnienia maksymalnego poziomu bezpieczeństwa całość komunikacji powinna się odbywać w ramach połączenia SSL.

Jedyny problem polega na tym, że identyfikator powinien być duży i kryptograficznie losowy w celu zapobieżenia atakom przejmowania sesji.

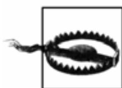
Analiza

Niestety, niewiele można zrobić w zakresie zapobiegania przejmowaniu sesji, jeżeli napastnik może w pewien sposób uzyskać dostęp do identyfikatora sesji generowanego dla użytkownika w razie jego poprawnego zalogowania się. W normalnej sytuacji cookie używane w celu przechowywania identyfikatora sesji nie powinno być trwałe (tzn. powinno ulegać wygaśnięciu w momencie zamknięcia przeglądarki przez użytkownika), tak więc większość przeglądarek nigdy nie przechowuje go na dysku, a tylko w pamięci. Choć nie zapobiega to całkowicie uzyskaniu przez napastnika dostępu do identyfikatora sesji, z pewnością znacznie to utrudnia.

Kwestia ta podkreśla znaczenie poprawnego użycia protokołu SSL, co zwykle nie stanowi problemu w przypadku komunikacji między przeglądarkami a serwerami sieciowymi. Trzeba to jednak wziąć pod uwagę w przypadku innych aplikacji wykorzystujących SSL. Jeżeli certyfikaty nie są weryfikowane poprawnie, co pozwala napastnikowi na przeprowadzenie ataku metodą *man-in-the-middle*, identyfikator sesji może zostać przechwycony. W takiej sytuacji nie ma to jednak prawie żadnego znaczenia. Jeżeli taki atak jest możliwy, napastnik może zrobić o wiele groźniejsze rzeczy, niż tylko przechwycić identyfikator sesji.

Jedynym wymogiem związanym z generowaniem identyfikatora sesji jest zapewnienie, aby był on unikatowy i nieprzewidywalny. Kryptograficznie silna liczba losowa zakodowana w formacie base64 zwykle powinna wystarczyć, ale istnieje wiele innych sposobów osiągnięcia tego samego rezultatu. Przykładowo, można użyć funkcji skrótu na liczbie losowej lub zaszyfrować pewne dane przy użyciu klucza symetrycznego. Każdy sposób jest dobry, o ile otrzymana wartość jest unikatowa i nieprzewidywalna. Zawsze potrzebny jest pewien element losowy w identyfikatorze sesji, więc zaleca się każdorazowe używanie ***co najmniej*** 64-bitowej, kryptograficznie silnej liczby losowej.

W zależności od sposobu generowania identyfikatora sesji może okazać się potrzebna tablica przeglądowa o kluczach stanowiących identyfikatory sesji. W takiej tablicy jest przechowywana przynajmniej nazwa użytkownika powiązana z identyfikatorem sesji, tak aby było wiadomo, o którego użytkownika w danym momencie chodzi. Można również dołączyć dane czasowe w celu umożliwiania przeprowadzania procesu wygasania sesji. Jeżeli nie chce się posuwać tak daleko i wszystko, czego trzeba, to nazwa użytkownika lub pewien wewnętrzny identyfikator użytkownika, dobrym rozwiązaniem jest zaszyfrowanie tych informacji wraz z innymi. W takim przypadku należy zapewnić dołączenie ***identyfikatora jednorazowego*** (ang. *nonce*) oraz odpowiednio uwierzytelnić i zaszyfrować dane (np. w trybie CWC opisanym w recepturze 5.10 lub zgodnie z opisem z receptury 6.18). Otrzymanym wynikiem będzie identyfikator sesji. W pewnych przypadkach można również chcieć zawrzeć w cookie adres IP.



Kuszącym rozwiązaniem może wydawać się zawieranie adresu IP klienta w identyfikatorze sesji. Należy jednak dobrze przemyśleć takie rozwiązanie, ponieważ klienci często zmieniają adresy IP, szczególnie wówczas, gdy znajdują się w ruchu lub łączą się z serwerem poprzez serwer pośredniczący, który w rzeczywistości stanowi zespół maszyn posiadających różne adresy IP. Dwa połączenia pochodzące od tego samego klienta nie zawsze muszą posiadać ten sam adres IP.

Zobacz również

Receptury 5.10, 6.18.

9.10. Zabezpieczanie połączeń bazodanowych

Problem

W aplikacji używamy bazy danych i chcemy zapewnić, aby ruch sieciowy między aplikacją a serwerem bazy danych był zabezpieczony za pomocą SSL.

Rozwiązanie

MySQL 4.00, PostgreSQL 7.1 oraz nowsze wersje każdego z tych serwerów obsługują połączenia SSL między klientami a serwerami. Jeżeli używa się starszej wersji lub innego serwera, który nie posiada wbudowanej obsługi SSL, można wykorzystać program Stunnel (patrz receptura 9.5) w celu zabezpieczenia połączeń z takim serwerem.

Analiza

Poniżej zostaną omówione różne kwestie związane z serwerami MySQL oraz PostgreSQL.

MySQL

Domyślnie, w trakcie konsolidacji serwera MySQL obsługa protokołu SSL jest wyłączona. W celu zapewnienia obsługi pakietu OpenSSL należy określić opcje `--with-vio` oraz `--with-openssl` w wierszu poleceń dla skryptu konfiguracyjnego. Kiedy posiada się zainstalowany i działający serwer MySQL z obsługą SSL, można to zweryfikować przy użyciu następującego polecenia SQL:

```
SHOW VARIABLES LIKE 'have_openssl'
```

Jeżeli wynikiem polecenia będzie wartość `yes`, oznacza to, że SSL jest obsługiwany.

W przypadku wersji serwera MySQL z obsługą SSL można używać polecenia `GRANT` w celu określenia wymagań związanych z SSL względem dostępu użytkownika do określonej bazy danych lub tabeli. Każdy klient może określić, że chce się łączyć z serwerem przy użyciu SSL, ale w przypadku polecenia `GRANT` będzie to wymagane.

Pisząc kod wykorzystujący interfejs C API serwera MySQL w celu zestawienia połączenia z serwerem, należy używać funkcji `mysql_real_connect()`zamiast funkcji `mysql_connect()`, która przestała być obsługiwana. Wszystko, czego zwykle potrzeba w celu zestawienia połączenia SSL klienta z serwerem, to określenie znacznika `CLIENT_SSL` w wywołaniu funkcji `mysql_real_connect()`.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mysql.h>

int spc_mysql_real_connect(MYSQL *mysql, const char *host, const char *pw,
                           const char *db, unsigned int flags) {
  int         port = 0, result = 0;
  char        *host_copy = 0, *p;
  const char  *socket = 0, *user = 0;

  if (host) {
    if (!(host_copy = strdup(host))) return 0;
    if ((p = strchr(host_copy, '@')) != 0) {
      user = host_copy;
      *p++ = 0;
      host = p;
    }
    if ((p = strchr((p ? p : host_copy), ':')) != 0) {
      *p++ = 0;
      port = atoi(p);
    }
    if (*host == '/') {
      socket = host;
      host = 0;
    }
  }

  /* poniższy znacznik wystarczy do aktywowania obsługi protokołu SSL w ramach połączeń */
  flags |= CLIENT_SSL;

  if (mysql_real_connect(mysql, host, user, pw, db, port, socket, flags))
    result = 1;
```

```
    if (host_copy) free(host_copy);
    return result;
}
```

Jeżeli serwer skonfigurowano tak, aby wymagany był certyfikat, może on wraz z kluczem zostać określony w pliku *my.cnf* i należy wówczas użyć funkcji `mysql_options()` z opcją `MYSQL_READ_DEFAULT_GROUP` w celu odczytania odpowiedniej grupy konfiguracji dla swojej aplikacji. Opcje związane z używanym certyfikatem i kluczem to, odpowiednio, `ssl-cert` oraz `ssl-key`. Ponadto można użyć opcji `ssl-ca` oraz `ssl-capath` w celu określenia pliku lub katalogu zawierającego zaufane certyfikaty, które mają być używane w czasie procesu weryfikacji. Ostatnią opcją jest `ssl-cipher`, której można użyć w celu określenia używanego szyfru lub zestawu szyfrów. Wszystkie te opcje mają również zastosowanie w przypadku konfiguracji serwera.

Innym rozwiązaniem jest użycie nieudokumentowanej funkcji `mysql_ssl_set()` w celu ustawienia klucza, certyfikatu, pliku zaufanego certyfikatu, katalogu zaufanego certyfikatu oraz szyfru. Ze względu na fakt, że funkcja ta jest nieudokumentowana, jest prawdopodobne, że zostanie ona w przyszłości usunięta lub zmieniona bez ostrzeżenia[2]. Prototyp tej funkcji zdefiniowano w pliku *mysql.h* i ma on następującą postać:

```
int STDCALL mysql_ssl_set(MYSQL *mysql, const char *key, const char *cert,
                          const char *ca, const char *capath, const char *cipher);
```

Wreszcie należy zauważyć, że przejrzenie kodu źródłowego *MySQL-4.0.10.-gamma* (najnowszej wersji w czasie pisania tej książki) pozwala odkryć, że jeśli ustawi się certyfikat używając opcji pliku konfiguracyjnego lub nieudokumentowanej funkcji `mysql_ssl_set()`, klient będzie podejmował próby łączenia się z serwerem z wykorzystaniem SSL bez względu na określenie lub nie znacznika `CLIENT_SSL` przekazywanego do funkcji `mysql_real_connect()`.

PostgreSQL

Domyślnie w trakcie konsolidacji serwera PostgreSQL obsługa protokołu SSL jest wyłączona. W celu zapewnienia obsługi pakietu OpenSSL należy określić opcję `--with-openssl` w wierszu poleceń dla skryptu konfiguracyjnego. Nawet w przypadku serwera PostgreSQL skonsolidowanego z opcją obsługi SSL domyślnym postępowaniem jest nieuwzględnianie tego protokołu. W tym celu należy ustawić parametr `ssl` na wartość `on` w pliku konfiguracyjnym *postgresql.conf*. W razie aktywacji protokołu SSL należy zapewnić, aby pliki *server.key* oraz *server.crt* zawierały, odpowiednio, klucz prywatny oraz certyfikat serwera. PostgreSQL będzie poszukiwał tych dwóch plików w słowniku danych i muszą one być obecne, aby serwer mógł wystartować.

W przypadku konfiguracji domyślnej PostgreSQL nie wymaga od klientów, aby łączyły się z serwerem poprzez protokół SSL — jego użycie to opcja ściśle związana z klientem. Jednakże można wymagać od klientów użycia SSL, wykorzystując format rekordu `hostssl` w pliku *pg_hba.conf*.

Funkcja `PGconnectdb()` interfejsu API C serwera PostgreSQL wymaga, aby obiekt `conninfo` był wypełniony oraz przekazany do niej w celu zestawienia połączenia z serwerem. Jednym z pól w strukturze `conninfo` jest pole całkowitoliczbowe o nazwie `requiressl`, które pozwala

[2] Wersje MySQL wcześniejsze niż 4.00 wydają się przynajmniej częściowo obsługiwać połączenia SSL, jednak nie istnieją żadne opcje konfiguracyjne, które pozwoliłyby na ich aktywowanie. Funkcja `mysql_ssl_set()` istnieje w wersji 3.23 i prawdopodobnie również we wcześniejszych wersjach, ale jej sygnatura różni się od występującej w wersji 4.00.

zdecydować klientowi, czy w ramach połączenia powinien być używany protokół SSL. W razie ustawienia jego wartości na 1 połączenie zakończy się niepowodzeniem, jeżeli serwer nie będzie obsługiwał SSL. W przeciwnym razie użycie SSL zostanie wynegocjowane w trakcie procesu wymiany potwierdzeń. W tym ostatnim przypadku protokół SSL będzie używany tylko wówczas, gdy w pliku *pg_hba.conf* istnieje rekord `hostssl` wymuszający używanie przez klientów protokołu SSL.

Zobacz również

Receptura 9.5.

9.11. Używanie wirtualnych sieci prywatnych w celu zabezpieczenia połączeń sieciowych

Problem

Nasz program działa w sieci i współpracuje z istniejącą infrastrukturą, która nie zapewnia żadnego wsparcia dla bezpiecznej komunikacji, takiej jak w ramach SSL. Jest pewne, że program będzie używany tylko przez określoną grupę użytkowników i zachodzi potrzeba zabezpieczenia ruchu sieciowego przed atakami podsłuchu i przechwytywania połączeń.

Rozwiązanie

W przypadku tego rodzaju problemów wystarczy użycie rozwiązania tunelującego SSL (takiego jak program Stunnel), jednak wymagania odnośnie do certyfikatów oraz ograniczone opcje weryfikacji oferowane przez Stunnel mogą nie spełniać stawianych wymagań. Ponadto niektóre protokoły sieciowe nie dopuszczają tunelowania SSL (takim protokołem jest na przykład FTP, gdyż może używać losowych portów w przypadku komunikacji w obu kierunkach). Alternatywnym rozwiązaniem jest użycie ***wirtualnej sieci prywatnej*** (ang. *virtual private network*, VPN) w zakresie usług sieciowych, których wymaga program.

Analiza

Zadanie konfigurowania i uruchomienia wirtualnych sieci prywatnych może niekiedy okazać się niebanalne. Może występować wiele problemów związanych ze współpracą różnych platform, jednak sieci VPN stanowią eleganckie rozwiązanie o tyle, że wymagają mniejszej liczby modyfikacji reguł zapory sieciowej (szczególnie, jeśli wchodzi w grę wiele niezabezpieczonych usług sieciowych), wiążą się z mniejszymi kosztami związanymi z wdrożeniem oprogramowania tunelującego oraz mniejszymi wymaganiami co do konserwacji. Dodanie lub usunięcie usług stanowi kwestię jej włączenia lub wyłączenia — nie są wymagane żadne zmiany w konfiguracji zapory sieciowej lub mechanizmu tunelowania. Kiedy sieć VPN zostanie skonfigurowana i uruchomiona, zasadniczo sama dba o swoje prawidłowe działanie.

Choć warto rozważyć możliwość użycia sieci VPN w przypadku, gdy inne zaprezentowane dotąd rozwiązania nie wchodzą w rachubę, pełne omówienie tego rodzaju metody znacznie

wykracza poza ramy niniejszej książki. Zagadnieniu temu poświęcono całe tomy i najlepszym rozwiązaniem jest tu sięgnięcie po któryś z nich. Dobrym punktem wyjścia może być pozycja *Building & Managing Virtual Private Networks* autorstwa Dave'a Kosiura (John Wiley & Sons).

9.12. Tworzenie uwierzytelnionych bezpiecznych kanałów bez użycia SSL

Problem

Chcemy szyfrować komunikację między dwoma węzłami bez użycia protokołu SSL oraz związanego z tym narzutu. Ze względu na fakt, że zwykle błędem jest szyfrowanie bez kontroli spójności (w celu uniknięcia ataków takich jak *man-in-the-middle*, przechwycenia i powtórzenia lub zamiany bitów w strumieniu szyfru) chcemy również zastosować pewien rodzaj sprawdzania spójności danych, aby móc stwierdzić, czy w czasie przesyłania dane nie zostały zmienione.

Zakładamy ponadto, że nie chcemy używać pełnej infrastruktury klucza publicznego, a zamiast tego bardziej tradycyjnego modelu kont użytkowników zarządzanych na każdej maszynie oddzielnie.

Rozwiązanie

Należy wykorzystać mechanizm uwierzytelniający wymiany kluczy z rozdziału 8. oraz użyć otrzymanego klucza sesji w rozwiązaniu szyfrowania z uwierzytelnianiem, przeprowadzając równocześnie odpowiednie działania zarządcze w odniesieniu do kluczy oraz identyfikatorów jednorazowych.

W niniejszej recepturze zostanie przedstawiona infrastruktura dla prostego bezpiecznego kanału, który może być używany po przeprowadzeniu procesu uwierzytelniania i wymiany kluczy.

Analiza

Biorąc pod uwagę narzędzia omówione we wcześniejszych recepturach związanych z uwierzytelnianiem, wymianą kluczy oraz tworzeniem bezpiecznych kanałów, opracowanie całościowego rozwiązań nie powinno być zbyt trudne. Mimo wszystko istnieją pewne potencjalne pułapki, o których nie można zapominać.

W przypadku protokołów, takich jak SSL/TLS, zestawianie połączenia jest nieco bardziej skomplikowane niż w przypadku samego uwierzytelniania i wymiany kluczy. W szczególności, takie protokoły zwykle stosują negocjowanie używanej wersji protokołu oraz algorytmu kryptograficznego i rozmiarów kluczy.

W takich sytuacjach istnieje groźba ***ataku wycofania*** (ang. *rollback attack*), który ma miejsce, kiedy napastnik ingeruje w przesyłane komunikaty w czasie zestawiania połączenia i podstępnie przekonuje obie strony do wynegocjowania niebezpiecznego zbioru parametrów (na przykład użycia starej, działającej niepoprawnie wersji protokołu).

Dobry protokół uwierzytelniania i wymiany kluczy, taki jak PAX lub SAX (patrz receptura 8.15), zapewnia, że nie istnieje możliwość przeprowadzenia ataku wycofania w kontekście protokołu. Jeżeli nie ma się komunikatów, które przychodzą przed wymianą kluczy oraz jeśli natychmiast rozpoczyna się używanie klucza szyfrowania po dokonaniu wymiany przy użyciu uwierzytelnionego mechanizmu szyfrowania, można przeprowadzać negocjacje innego rodzaju (takie jak uzgodnienie protokołu) i nie martwić się o atak wycofania.

Z drugiej strony, jeżeli przesyła się komunikaty przed dokonaniem wymiany kluczy lub tworzy własny protokół (nie są to zalecane rozwiązania), zachodzi potrzeba zabezpieczenia się we własnym zakresie przed atakami metodą powtórzenia. W tym celu po zestawieniu połączenia każda ze stron powinna uwierzytelnić każdy komunikat, który pojawił się w czasie zestawiania połączenia. Jeżeli klient przesyła swój ***kod uwierzytelniający wiadomość*** (ang. *message authentication code*, MAC) jako pierwszy, a serwer przeprowadza jego walidację, serwer powinien uwierzytelniać w ten sposób nie tylko komunikaty zestawiania, ale również wartość MAC przesłaną przez klienta. Podobnie, jeżeli serwer przesyła MAC jako pierwszy, klient powinien w swojej odpowiedzi zawrzeć MAC otrzymany od serwera.

Ogólne zalecenie jest takie, że w ramach własnych mechanizmów kryptograficznych nie należy wprowadzać możliwości konfiguracyjnych podobnych do SSL. Jeżeli, na przykład, używa się protokołu PAX, jedyną opcją dostępną w całym procesie wymiany klucza i uwierzytelniania jest rozmiar klucza, który podlega wymianie. Zaleca się używanie klucza w ramach silnego, uwierzytelnionego schematu szyfrowania bez mechanizmu negocjacji. Jeżeli uzna się, że negocjowanie algorytmów absolutnie musi być uwzględnione, zaleca się wykorzystanie bardzo ostrożnych ustawień domyślnych, których używanie rozpoczyna się od razu po dokonaniu wymiany kluczy, na przykład algorytmu AES w trybie CWC z kluczami 256-bitowymi, oraz umożliwienie renegocjowania.

Zgodnie z treścią receptury 6.21 należy używać licznika komunikatów wraz z kodem MAC w celu zapobieżenia atakom przechwycenia i powtórzenia. Liczniki komunikatów mogą również być pomocne w określeniu, kiedy komunikaty przychodzą w zmienionej kolejności lub są gubione, o ile za każdym razem sprawdza się, czy numer komunikatu zwiększył się dokładnie o jeden (standardowe wykrywanie przechwycenia i powtórzenia polega jedynie na sprawdzaniu, czy numer komunikatu uległ zwiększeniu).

Należy zauważyć, że jeżeli wykorzystuje się niezawodny mechanizm transportu danych, taki jak protokół TCP, zyskuje się wstępne zabezpieczenie przed zmianą kolejności komunikatów oraz ich zgubieniem. Ochrona protokołu TCP przed tego rodzaju problemami nie jest jednak kryptograficznie bezpieczna. Zdolny napastnik wciąż może przypuścić tego rodzaju atak w sposób niemożliwy do wykrycia przez warstwę TCP.

W niektórych środowiskach kolejność i gubienie komunikatów nie odgrywa zbyt dużego znaczenia. Są to środowiska, w których w normalnej sytuacji używa się zawodnego protokołu, takiego jak UDP. Ogólnie rzecz biorąc, protokoły silne kryptograficznie mogą być w stanie tolerować zgubienia, jednak nie powinny tolerować zmiany kolejności, gdyż oznaczałoby to zrezygnowanie ze standardowego mechanizmu zapobiegania powtórzeniom przechwytywania. Zawsze można usunąć komunikaty przesłane w niepoprawnej kolejności lub jawnie śledzić numery ostatnich komunikatów, jakie nadeszły, a następnie usuwać wszelkie duplikaty lub komunikaty o numerach nienależących do takiego przedziału.

Szczególnie w przypadku, gdy używa się protokołu TCP, jeżeli kryptograficzne uwierzytelnienie komunikatu zakończy się niepowodzeniem, jego odtworzenie jest zadaniem bardzo trudnym. Przypadkowe błędy niemal zawsze są wychwytywane na poziomie TCP i można założyć,

że jeżeli zostanie to wykryte kryptograficznie, świadczy o wystąpieniu ataku. W takim przypadku napastnik może spowodować wystąpienie zablokowania usługi. Zwykle najłatwiej jest zamknąć wówczas połączenie, przesyłając być może najpierw z powrotem pakiet błędu.

Często błędy niemożliwe do naprawienia powodują generowanie komunikatów o błędach w formie tekstu jawnego. W takich sytuacjach nie należy przesyłać żadnych informacji mogących wskazywać przyczynę powstania błędu. Istnieją sytuacje w przypadku znanych protokołów, w których pełny opis błędu może prowadzić do ujawnienia istotnych informacji.

Projektując protokół dla komunikacji klient-serwer należy uwzględnić sekwencję komunikatów przesyłanych między obiema stronami, która będzie określać normalne zamknięcie połączenia. W ten sposób w przypadku przedwczesnego zerwania połączenia obie strony będą miały możliwość stwierdzenia, czy było to normalnym działaniem czy być może wskazuje na wystąpienie ataku. W tym drugim przypadku można podjąć odpowiednie działania. Przykładowo, jeżeli połączenie zostanie przedwcześnie zerwane w trakcie wykonywania pewnych działań na bazie danych, można wycofać wszelkie wprowadzone zmiany.

Kolejną kwestią wartą rozważenia jest używany format komunikatów. Ogólnie rzecz biorąc, komunikat rozpoczyna się od tekstu jawnego, pola o stałej długości, które koduje długość pozostałej części komunikatu. Dalej mogą, ale nie muszą, występować inne wartości jawne, takie jak numer komunikatu (numer komunikatu może również być zawarty w tekście zaszyfrowanym, jednak często jest przydatny w obliczaniu identyfikatora jednorazowego zamiast jego przyjmowania). Na końcu występuje tekst zaszyfrowany oraz wartość MAC (mogą one stanowić jedną całość w zależności od tego, czy używa się uwierzytelniającego trybu szyfrowania, takiego jak CWC).

Wszelkie niezaszyfrowane dane w komunikacie powinny zostać uwierzytelnione w bezpieczny sposób wraz z danymi zaszyfrowanymi. Tryby, takie jak CWC i CCM pozwalają na uwierzytelnianie zarówno tekstu jawnego, jak i zaszyfrowanego przy użyciu pojedynczej wartości MAC. Tryb CMAC posiada te same możliwości. W przypadku innych wartości MAC można symulować takie zachowanie poprzez utworzenie MAC dla długości tekstu jawnego scalonej z samym tekstem jawnym oraz scalonej z tekstem zaszyfrowanym. Aby zrobić to poprawnie, należy jednak zawsze uwzględniać długość tekstu jawnego, nawet, gdy wynosi zero.

Załóżmy, że zestawiliśmy połączenie TCP i wymieniliśmy 128-bitowy klucz przy użyciu protokołu takiego jak PAX (zgodnie z recepturą 8.15). Pojawia się pytanie, co mamy teraz zrobić z kluczem. Odpowiedź zależy od kilku czynników. Po pierwsze, możemy potrzebować oddzielnych kluczy dla szyfrowania oraz tworzenia kodu MAC, jeżeli nie używamy trybu podwójnego przeznaczenia takiego jak CWC. Po drugie, klient i serwer mogą przesyłać komunikaty na przemian lub asynchronicznie. Jeżeli mamy do czynienia z drugim przypadkiem, możemy użyć oddzielnego klucza dla każdego kierunku transmisji lub (w przypadku użyciu trybu szyfrowania identyfikatora jednorazowego) zarządzać dwoma identyfikatorami jednorazowymi, zapewniając jednocześnie , aby te identyfikatory — klienta i serwera — zawsze były różne (zostanie to wykorzystane w poniższym kodzie).

Jeżeli zachodzi konieczność użycia wielu kluczy, można wykorzystać wymieniony klucz i używać go w celu generowania kluczy pochodnych, co omówiono w recepturze 4.11. W takim przypadku wymienionego klucza należy używać wyłącznie w celu generowania kluczy pochodnych.

W tym momencie po każdej stronie połączenia powinniśmy posiadać otwarty deskryptor pliku oraz wszelkie wymagane klucze. Załóżmy, że korzystamy z trybu CWC (korzystając z API określonego w recepturze 5.10), nasza komunikacja ma charakter synchroniczny, deskryptor pliku znajduje się w trybie blokującym i klient przesyła pierwszy komunikat. Używamy losowego klucza sesji, więc nie musimy tworzyć klucza pochodnego, jak ma to miejsce w recepturze 5.16.

Pierwszą rzeczą, jaką musimy zrobić, jest ustalenie, w jaki sposób określimy 11-bajtowy identyfikator jednorazowy dostępny w trybie CWC. Pierwszego bajtu użyjemy w celu rozróżnienia strony wysyłającej na wypadek, gdybyśmy w przyszłości chcieli przejść do transmisji w trybie asynchronicznym. Klient przesyła dane z najstarszym bajtem ustawionym na wartość `0x80`, zaś serwer przesyła z tym bajtem ustawionym na wartość `0x00`. Dalej mamy związaną z sesją 40-bitową (5-bajtową) wartość losową wybraną przez klienta, po której występuje 5-bajtowy licznik.

Elementy komunikatu stanowią: bajt stanu, identyfikator jednorazowy o stałym rozmiarze, długość tekstu zaszyfrowanego zakodowana jako 32-bitowa wartość z najstarszym bajtem jako pierwszym oraz tekst zaszyfrowany CWC (wraz z wartością uwierzytelnienia). Bajt, identyfikator jednorazowy oraz pole długości są przesyłane w postaci jawnej.

Bajt stanu zawsze ma wartość `0x00`, chyba że zamykamy połączenie, kiedy to przesyłamy wartość `0xff`. Jeżeli po stronie nadawcy wystąpi błąd, po prostu usuwamy połączenie, zamiast przesyłać status błędu). Jeżeli otrzymamy jakąkolwiek wartość niezerową, zamykamy połączenie. Jeżeli wartość jest różna od `0x00` i `0xff`, wskazuje to na prawdopodobne wystąpienie ataku.

Tworząc kod MAC, nie musimy brać pod uwagę identyfikatora jednorazowego, ponieważ stanowi on nieodłączny element walidacji komunikatu CWC. Podobnie pole długości jest niejawnie uwierzytelniane w czasie deszyfrowania CWC. Bajt stanu również powinien być uwierzytelniany i możemy go przekazać do CWC w formie danych powiązanych.

Teraz posiadamy już wszystkie narzędzia potrzebne do utworzenia naszego uwierzytelnionego bezpiecznego kanału. Najpierw tworzymy abstrakcję połączenia składającą się z kontekstu szyfrowania CWC, informacji stanu o identyfikatorze jednorazowym oraz deskryptora pliku, przez który się komunikujemy.

```
#include <stdlib.h>
#include <errno.h>
#include <cwc.h>

#define SPC_CLIENT_DISTINGUISHER 0x80
#define SPC_SERVER_DISTINGUISHER 0x00
#define SPC_SERVER_LACKS_NONCE   0xff

#define SPC_IV_IX   1
#define SPC_CTR_IX  6
#define SPC_IV_LEN  5
#define SPC_CTR_LEN 5

#define SPC_CWC_NONCE_LEN (SPC_IV_LEN + SPC_CTR_LEN + 1)

typedef struct {
  cwc_t         cwc;
  unsigned char nonce[SPC_CWC_NONCE_LEN];
  int           fd;
} spc_ssock_t;
```

Po zakończeniu procedury wymiany kluczy klient będzie posiadał klucz oraz deskryptor pliku połączony z serwerem. Możemy użyć tych informacji w celu zainicjalizowania struktury `spc_ssock_t`.

```
/* keylen to wartość określana w bajtach. Należy zauważyć, że w przypadku
 * wystąpienia błędów wywoływana jest funkcja abort(), choć w realnej sytuacji
 * zwykle pożądane będzie przeprowadzenie obsługi błędów, co omówiono w
 * recepturze 13.1. W każdym bądź razie informacja o błędzie nigdy nie jest
 * przekazywana drugiej stronie; następuje po prostu odrzucenie połączenia
 * (poprzez wyjście). W przypadku poprawnego zamykania przesyłany jest komunikat.
 */

void spc_init_client(spc_ssock_t *ctx, unsigned char *key, size_t klen, int fd) {
  if (klen != 16 && klen != 24 && klen != 32) abort();

  /* Trzeba pamiętać, że funkcja cwc_init() czyści przekazywany klucz! */
  cwc_init(&(ctx->cwc), key, klen * 8);

  /* Wybieramy 5 losowych bajtów i umieszczamy pierwszy na pozycji nonce[1].
   * Używamy interfejsu API z receptury 11.2.
   */
  spc_rand(ctx->nonce + SPC_IV_IX, SPC_IV_LEN);

  /* Ustawiamy 5 przeciwległych bajtów na wartość 0, przez co określamy, że
   * nie przesłaliśmy żadnego komunikatu. */
  memset(ctx->nonce + SPC_CTR_IX, 0, SPC_CTR_LEN);
  ctx->fd = fd;

  /* Poniższa wartość zawsze określa ostatnią osobę, do której przesłaliśmy
   * komunikat. Jeżeli klient prześle komunikat, a ten zostanie przesłany do
   * SPC_CLIENT_DISTINGUISHER, wówczas wiemy, że wystąpił błąd.
   */
  ctx->nonce[0] = SPC_SERVER_DISTINGUISHER;
}
```

Klient może teraz przesłać komunikat do serwera, używając poniższej funkcji, która pobiera tekst jawny i szyfruje go przed przesłaniem.

```
#define SPC_CWC_TAG_LEN     16
#define SPC_MLEN_FIELD_LEN  4
#define SPC_MAX_MLEN        0xffffffff

static unsigned char spc_msg_ok  = 0x00;
static unsigned char spc_msg_end = 0xff;

static void spc_increment_counter(unsigned char *, size_t);
static void spc_ssock_write(int, unsigned char *, size_t);
static void spc_base_send(spc_ssock_t *ctx, unsigned char *msg, size_t mlen);

void spc_ssock_client_send(spc_ssock_t *ctx, unsigned char *msg, size_t mlen) {
  /* Jeżeli nie nasza kolej nadawania, anulujemy. */
  if (ctx->nonce[0] != SPC_SERVER_DISTINGUISHER) abort();

  /* Ustawiamy element wyróżniający, a następnie zwiększamy licznik przed faktycznym rozpoczęciem przesyłania. */
  ctx->nonce[0] = SPC_CLIENT_DISTINGUISHER;
  spc_increment_counter(ctx->nonce + SPC_CTR_IX, SPC_CTR_LEN);
  spc_base_send(ctx, msg, mlen);
}

static void spc_base_send(spc_ssock_t *ctx, unsigned char *msg, size_t mlen) {
  unsigned char encoded_len[SPC_MLEN_FIELD_LEN];
  size_t        i;
  unsigned char *ct;
```

```
  /* Jeżeli nie nasza kolej nadawania, anulujemy. */
  if (ctx->nonce[0] != SPC_SERVER_DISTINGUISHER) abort();

  /* Najpierw zapisujemy bajt stanu, później identyfikator jednorazowy. */
  spc_ssock_write(ctx->fd, &spc_msg_ok, sizeof(spc_msg_ok));
  spc_ssock_write(ctx->fd, ctx->nonce, sizeof(ctx->nonce));

  /* Następnie zapisujemy długość tekstu zaszyfrowanego, która będzie
   * rozmiarem tekstu jawnego powiększonym o SPC_CWC_TAG_LEN bajtów
   * zajmowanych przez znacznik. Anulujemy, jeżeli ciąg znaków liczy ponad
   * 2^32-1 bajtów. Robimy to w sposób zwykle niezależny od rozmiaru słowa.
   */
  if (mlen > (unsigned long)SPC_MAX_MLEN ||  mlen < 0) abort( );
  for (i = 0;  i < SPC_MLEN_FIELD_LEN;   i++)
    encoded_len[SPC_MLEN_FIELD_LEN - i - 1] = (mlen >> (8 * i)) & 0xff;
  spc_ssock_write(ctx->fd, encoded_len, sizeof(encoded_len));

  /* Teraz przeprowadzamy szyfrowanie CWC i przesyłamy wynik. Należy zauważyć,
   * że jeżeli przesyłanie zakończy się niepowodzeniem i nie anuluje się działania,
   * tak jak ma to miejsce w poniższym kodzie, trzeba pamiętać o zwolnieniu pamięci
   * zajmowanej przez bufor komunikatów.
   */
  mlen += SPC_CWC_TAG_LEN;
  if (mlen < SPC_CWC_TAG_LEN) abort(); /* Komunikat za długi, przepełnienie mlen. */
  if (!(ct = (unsigned char *)malloc(mlen))) abort(); /* Brak pamięci. */
  cwc_encrypt_message(&(ctx->cwc),  &spc_msg_ok, sizeof(spc_msg_ok),  msg,
                      mlen - SPC_CWC_TAG_LEN, ctx->nonce, ct);
  spc_ssock_write(ctx->fd, ct, mlen);
  free(ct);
}

static void spc_increment_counter(unsigned char *ctr, size_t len) {
  while (len--) if (++ctr[len]) return;
  abort(); /* Licznik przekręcony, co oznacza wystąpienie błędu! */
}

static void spc_ssock_write( int fd, unsigned char *msg, size_t mlen) {
  ssize_t w;

  while (mlen) {
    if ((w = write(fd, msg, mlen)) == -1) {
      switch (errno) {
        case EINTR:
          break;
        default:
          abort();
      }
    } else {
      mlen -= w;
      msg += w;
    }
  }
}
```

Teraz spójrzmy na resztę połączenia po stronie klienta, zanim skupimy uwagę na serwerze. Kiedy klient chce zakończyć połączenie w sposób bezpieczny, przesyła komunikat pusty, ale jako bajt stanu przekazuje wartość `0xff`. Wciąż musi przesłać poprawny identyfikator jednorazowy oraz zaszyfrować komunikat o zerowej długości (co umożliwia schemat CWC). Można tego dokonać przy użyciu kodu bardzo podobnego do przedstawionego powyżej, więc nie będziemy marnować miejsca na jego powtarzanie.

Teraz spójrzmy na zdarzenia zachodzące w momencie otrzymania przez klienta komunikatu. Bajt stanu powinien mieć wartość 0x00. Identyfikator jednorazowy otrzymany od serwera powinien być niezmieniony w porównaniu z przesłanym przez nas poza tym, że pierwszy bajt powinien mieć wartość SPC_SERVER_DISTINGUISHER. Jeżeli identyfikator jednorazowy jest niepoprawny, anulujemy po prostu dalsze działania, choć można by również odrzucić komunikat (jest to jednak nieco problematyczne, ponieważ trzeba wówczas w pewien sposób dokonać resynchronizacji połączenia).

Następnie odczytujemy wartość długości i dynamicznie przydzielamy bufor, który będzie w stanie pomieścić tekst zaszyfrowany. Prezentowany kod nigdy nie przydziela więcej niż $2^{32} - 1$ bajtów pamięci. W praktyce należy określić maksymalną długość komunikatu i sprawdzać, czy pole długości nie przekracza tej wartości. Takie sprawdzenie może zapobiec przeprowadzeniu ataku zablokowania usług, kiedy to napastnik prowokuje przydzielenie takiej ilości pamięci, która spowalnia działanie maszyny.

Wreszcie wywołujemy funkcję cwc_decrypt_message() i sprawdzamy, czy kod uwierzytelniający komunikat jest poprawny. Jeśli tak, zwracamy komunikat. W przeciwnym wypadku anulujmy.

```
static void spc_ssock_read(int, unsigned char *, size_t);
static void spc_get_status_and_nonce(int, unsigned char *, unsigned char *);
static unsigned char *spc_finish_decryption(spc_ssock_t *, unsigned char,
                                            unsigned char *, size_t *);

unsigned char *spc_client_read(spc_ssock_t *ctx,  size_t *len, size_t *end) {
  unsigned char status;
  unsigned char nonce[SPC_CWC_NONCE_LEN];

  /* Jeżeli kolej nadawania klienta, anulujemy. */
  if (ctx->nonce[0] != SPC_CLIENT_DISTINGUISHER) abort();
  ctx->nonce[0] = SPC_SERVER_DISTINGUISHER;
  spc_get_status_and_nonce(ctx->fd, &status, nonce);
  *end = status;
  return spc_finish_decryption(ctx, status, nonce, len);
}

static void spc_get_status_and_nonce(int fd, unsigned char *status,
                                     unsigned char *nonce) {
  /* Odczytujemy bajt stanu. Jeżeli jego wartością jest 0x00 lub 0xff, musimy
   * sprawdzić resztę komunikatu, w przeciwnym wypadku kończymy od razu.
   */
  spc_ssock_read(fd,  status, 1);
  if (*status != spc_msg_ok && *status != spc_msg_end) abort( );
  spc_ssock_read(fd, nonce, SPC_CWC_NONCE_LEN);
}

static unsigned char *spc_finish_decryption(spc_ssock_t *ctx, unsigned char status,
                                            unsigned char *nonce, size_t *len) {
  size_t        ctlen = 0, i;
  unsigned char *ct, encoded_len[SPC_MLEN_FIELD_LEN];

  /* Sprawdzamy identyfikator jednorazowy. */
  for (i = 0;  i < SPC_CWC_NONCE_LEN;  i++)
    if (nonce[i] != ctx->nonce[i]) abort();

  /* Odczytujemy pole długości. */
  spc_ssock_read(ctx->fd, encoded_len, SPC_MLEN_FIELD_LEN);
  for (i = 0;  i < SPC_MLEN_FIELD_LEN;  i++) {
    ctlen <<= 8;
    ctlen += encoded_len[i];
  }
```

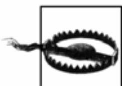
```
  /* Odczytujemy tekst zaszyfrowany. */
  if (!(ct = (unsigned char *)malloc(ctlen))) abort(); /* Brak pamięci. */
  spc_ssock_read(ctx->fd, ct, ctlen);

  /* Odszyfrowujemy szyfrogram i anulujemy, jeżeli proces ten kończy się.
   * niepowodzeniem. Odszyfrowujemy do tego samego bufora, w którym już
   * znajduje się tekst zaszyfrowany.
   */
  if (!cwc_decrypt_message(&(ctx->cwc), &status, 1, ct, ctlen, nonce, ct)) {
    free(ct);
    abort();
  }

  *len = ctlen - SPC_CWC_TAG_LEN;
  /* Unikamy konieczności późniejszego wywołania funkcji realloc(),
   * pozostawiając o SPC_CWC_TAG_LEN dodatkowych bajtów więcej na końcu bufora.
   */
  return ct;
}

static void spc_ssock_read(int fd, unsigned char *msg, size_t mlen) {
  ssize_t r;

  while (mlen) {
    if ((r = read(fd, msg, mlen)) == -1) {
      switch (errno) {
        case EINTR:
          break;
        default:
          abort();
      }
    } else {
      mlen -= r;
      msg += r;
    }
  }
}
```



Klient jest odpowiedzialny za zwolnienie pamięci przydzielonej dla komunikatów. Zaleca się wcześniejsze bezpieczne czyszczenie komunikatów, co omówiono w recepturze 13.2. Ponadto należy w bezpieczny sposób zamazywać kontekst `spc_ssock_t`, kiedy nie jest już potrzebny.

W przypadku klienta to wszystko. Teraz możemy skupić się na serwerze. Serwer może współużytkować typ `spc_ssock_t` wykorzystywany przez klienta, jak również wszystkie funkcje pomocnicze, takie jak `spc_ssock_read()` i `spc_ssock_write()`. Jednak interfejs API dla operacji inicjalizacji, czytania oraz zapisu muszą ulec zmianie.

Poniżej przedstawiono funkcję inicjalizacji wykorzystywaną po stronie serwera, która powinna zostać wywołana po zakończeniu procedury wymiany kluczy, ale przed odczytaniem pierwszego komunikatu od klienta.

```
void spc_init_server(spc_ssock_t *ctx, unsigned char *key, size_t klen,  int fd) {
  if (klen != 16 && klen != 24 && klen != 32) abort();

  /* należy pamiętać, że funckja cwc_init() czyści przekazany klucz! */
  cwc_init(&(ctx->cwc), key, klen * 8);

  /* Musimy poczekać na losowy fragment identyfikatora jednorazowego od klienta.
   * Fragment licznika można zainicjalizować wartością zero. Element wyróżniający
   * ustawiamy na wartość SPC_SERVER_LACKS_NONCE, dzięki czemu będziemy wiedzieć,
```

```
 * że należy skopiować losowy fragment identyfikatora jednorazowego w momencie
 * otrzymania komunikatu.
 */
 ctx->nonce[0] = SPC_SERVER_LACKS_NONCE;
 memset(ctx->nonce + SPC_CTR_IX, 0, SPC_CTR_LEN);
 ctx->fd = fd;
}
```

Pierwszą rzeczą wykonywaną przez serwer jest odczytanie danych z gniazda klienta. W praktyce poniższy kod nie jest przeznaczony dla jednowątkowego serwera wykorzystującego funkcję `select()` w celu określenia, który klient posiada dane do odczytania. Jest tak dlatego, że kiedy rozpoczniemy odczyt danych, kontynuujemy go do momentu pobrania całego komunikatu, a wszystkie odczyty mają charakter blokujący. Prezentowany kod nie jest przeznaczony do użycia w środowisku nieblokującym.

Zamiast tego powinniśmy użyć przedstawionego kodu w wątku lub wykorzystać tradycyjny model uniksowy, w którym dla każdego połączenia klienta tworzone jest odgałęzienie za pomocą funkcji `fork()`. Można również po prostu przeorganizować kod, tak aby dane czytać przyrostowo bez blokowania.

```
unsigned char *spc_server_read(spc_ssock_t *ctx,  size_t *len, size_t *end) {
  unsigned char nonce[SPC_CWC_NONCE_LEN], status;

  /* Jeżeli kolej serwera na nadawanie, anulujemy. Wiemy, że kolej serwera na
   * nadawanie, jeżeli pierwszy bajt identyfikatora jednorazowego ma wartość
   * elementu wyróżniającego CLIENT.
   */
  if (ctx->nonce[0] != SPC_SERVER_DISTINGUISHER &&
      ctx->nonce[0] != SPC_SERVER_LACKS_NONCE) abort();

  spc_get_status_and_nonce(ctx->fd, &status, nonce);
  *end = status;

  /* Jeżeli to konieczne, kopiujemy losowy bajt identyfikatora jednorazowego. */
  if (ctx->nonce[0] == SPC_SERVER_LACKS_NONCE)
    memcpy(ctx->nonce + SPC_IV_IX, nonce + SPC_IV_IX, SPC_IV_LEN);

  /* Teraz ustawiamy pole wyróżniające na klienta i zwiększamy o jeden naszą
   * kopię identyfikatora jednorazowego.
   */
  ctx->nonce[0] = SPC_CLIENT_DISTINGUISHER;
  spc_increment_counter(ctx->nonce + SPC_CTR_IX, SPC_CTR_LEN);

  return spc_finish_decryption(ctx, status, nonce, len);
}
```

Teraz musimy jedynie obsłużyć przesłanie komunikatu po stronie serwera, co wymaga niewielu działań.

```
void spc_ssock_server_send(spc_ssock_t *ctx, unsigned char *msg, size_t mlen) {
  /* Jeżeli nie nasza kolej nadawania, anulujemy. Wiemy, że nasza kolej
   * nadawania, jeżeli jako ostatni nadawał klient.
   */
  if (ctx->nonce[0] != SPC_CLIENT_DISTINGUISHER) abort();

  /* Ustawiamy element wyróżniający, ale nie zwiększamy licznika, ponieważ
   * zrobiliśmy to już, kiedy otrzymaliśmy komunikat od klienta.
   */
  ctx->nonce[0] = SPC_SERVER_DISTINGUISHER;
  spc_base_send(ctx, msg, mlen);
}
```

Trzeba pamiętać o jeszcze jednej kwestii. W pewnych sytuacjach, w których ma się do czynienia z bardzo długimi komunikatami, nie ma sensu określanie ilości danych, jakie będą zawarte w komunikacie przed rozpoczęciem jego przesyłania. Wymagałoby to buforowania dużych ilości danych, co nie zawsze jest możliwe, szczególnie w przypadku urządzeń wbudowanych.

W takich przypadkach trzeba sobie zapewnić możliwość przyrostowego odczytywania komunikatu, a jednocześnie posiadanie pewnego wskaźnika końca komunikatu, tak aby móc w odpowiednim momencie zakończyć proces deszyfrowania. Taki scenariusz wymaga określenia specjalnego formatu komunikatu.

Zaleca się wówczas przesyłanie danych w „ramkach" o jednakowym rozmiarze. Na końcu każdej ramki znajduje się pole określające długość danych zawartych w tej ramce oraz pole wskazujące, czy ramka reprezentuje koniec komunikatu. W przypadku niepełnych ramek bajty leżące między końcem danych a polami informacyjnymi powinny być ustawione na wartość 0.

Zobacz również

Receptury 4.11, 5.10, 5.16, 6.21, 8.15 oraz 13.2.