

Atak na sieć okiem hakera

*Wykrywanie i eksploatacja luk
w zabezpieczeniach sieci*



James Forshaw



Helion 

Tytuł oryginału: *Attacking Network Protocols: A Hacker's Guide to Capture, Analysis, and Exploitation*

Tłumaczenie: Andrzej Grażyński

ISBN: 978-83-283-5390-9

Copyright © 2018 by James Forshaw. Title of English-language original: *Attacking Network Protocols: A Hacker's Guide to Capture, Analysis, and Exploitation*, ISBN 978-1-59327-750-5, published by No Starch Press.

Polish-language edition copyright © 2019 by Helion S.A. All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Helion SA dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Helion SA nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Helion SA

ul. Kościuszki 1c, 44-100 Gliwice

tel. 32 231 22 19, 32 230 98 63

e-mail: helion@helion.pl

WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:

<ftp://ftp.helion.pl/przyklady/ataksi.zip>

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/ataksi>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

O AUTORZE	10
O RECENZENCIE	10
PRZEDMOWA	11
PODZIĘKOWANIA	15
WSTĘP	17
Po co czytać tę książkę?	17
Co znajdziesz w tej książce?	18
Jak korzystać z tej książki?	19
Skontaktuj się ze mną	20
I	
PODSTAWY SIECI KOMPUTEROWYCH	21
Architektura sieci i protokoły sieciowe	21
Zestaw protokołów internetowych	23
Enkapsulacja danych	25
Nagłówki, stopki i adresy	25
Transmisja danych	27
Trasowanie w sieci	28
Mój model analizy protokołów sieciowych	29
Podsumowanie	31
2	
PRZECHWYTYWANIE RUCHU SIECIOWEGO	33
Biernie przechwytywanie ruchu sieciowego	34
Wireshark — podstawy	34
Alternatywne techniki biernego przechwytywania	37
Śledzenie wywołań funkcji systemowych	37
Linuksowy program strace	38
Monitorowanie połączeń sieciowych za pomocą programu DTrace	39
Monitor procesów Windows	41

Zalety i wady biernego przechwytywania	43
Czynne przechwytywanie ruchu sieciowego	43
Proxy sieciowe	44
Proxy z przekierowaniem portów	44
Proxy SOCKS	48
Proxy HTTP	53
Proxy przekazujące HTTP	54
Proxy odwrotne HTTP	57
Podsumowanie	60

3

STRUKTURA PROTOKOŁÓW SIECIOWYCH 61

Struktury protokołów binarnych	62
Dane numeryczne	62
Tekst i dane czytelne dla człowieka	68
Binarne dane o zmiennej długości	72
Data i czas	75
Czas w systemach Unix/POSIX	76
Znacznik FILETIME w systemie Windows	76
Wzorzec typ – długość – wartość	76
Multipleksowanie i fragmentacja	77
Informacje sieciowe	79
Strukturalne formaty binarne	80
Struktury protokołów tekstowych	81
Dane numeryczne	81
Wartości logiczne (boolowskie)	82
Data i czas	82
Dane zmiennej długości	82
Strukturalne formaty tekstowe	83
Kodowanie danych binarnych	86
Kodowanie szesnastkowe	86
Base64	87
Podsumowanie	89

4

ZAAWANSOWANE TECHNIKI

PRZECHWYTYWANIA RUCHU SIECIOWEGO 91

Przetrasowywanie ruchu	92
Program Traceroute	92
Tablice trasowania	93
Konfigurowanie routera	95
Włączanie trasowania w Windows	96
Włączanie trasowania w systemach uniksowych	96

NAT — translacja adresów sieciowych	97
Włączanie SNAT	97
Konfigurowanie SNAT w Linuksie	98
Włączanie DNAT	99
Przekierowanie ruchu do bramy	101
Ingerencja w DHCP	101
Infekowanie ARP	104
Podsumowanie	107
5	
ANALIZA „NA DRUCIE”	109
SuperFunkyChat — aplikacja generująca ruch	109
Uruchamianie serwera	110
Uruchamianie aplikacji klienckiej	110
Komunikacja między aplikacjami klienckimi	111
Wireshark na kursie kolizyjnym	112
Generowanie ruchu sieciowego i przechwytywanie pakietów	113
Podstawowa analiza	115
Odczytywanie zawartości sesji TCP	115
Identyfikowanie elementów struktury pakietu na podstawie zrzutu szesnastkowego	117
Podgląd pojedynczych pakietów	117
Odkrywanie struktury protokołu	119
Weryfikowanie założeń	120
Dysekcja protokołu przy użyciu języka Python	121
Dysektory dla Wiresharka w języku Lua	127
Tworzenie dysektora	130
Dysekcja protokołu w języku Lua	131
Parsowanie pakietu komunikatu	132
Czynne analizowanie ruchu za pomocą proxy	135
Konfigurowanie proxy	136
Analiza protokołu przy użyciu proxy	138
Podstawowe parsowanie protokołu	140
Zmiany w zachowaniu się protokołu	142
Podsumowanie	144
6	
INŻYNIERIA WSTECZNA	147
Kompilatory, interpretery i asemblery	148
Języki interpretowane	149
Języki kompilowane	149
Konsolidacja statyczna kontra konsolidacja dynamiczna	150
Architektura x86	150
Zestaw instrukcji maszynowych	151
Rejestry procesora	152
Przepływ sterowania w programie	156

Podstawy systemów operacyjnych	157
Formaty plików wykonywalnych	157
Sekcje pliku wykonywalnego	158
Procesy i wątki	159
Interfejs sieciowy systemu operacyjnego	159
ABI — binarny interfejs aplikacji	162
Stacyczna inżynieria wsteczna	164
Krótki przewodnik po IDA	164
Analiza stosu — zmienne lokalne i parametry	167
Rozpoznawanie kluczowych funkcji	168
Dynamiczna inżynieria wsteczna	174
Punkty przerwania	175
Okna debuggera	175
Gdzie ustawiać punkty przerwania?	177
Inżynieria wsteczna a kod zarządzany	177
Aplikacje .NET	178
Dekompilacja za pomocą ILSpy	179
Aplikacje Javy	182
Inżynieria wsteczna a obfuskacja kodu	184
Zasoby dotyczące inżynierii wstecznej	185
Podsumowanie	185

7

BEZPIECZEŃSTWO PROTOKOŁÓW SIECIOWYCH 187

Algorytmy szyfrowania	188
Szyfry podstawieniowe	189
Szyfrowanie XOR	190
Generatory liczb (pseudo)losowych	191
Kryptografia symetryczna	192
Szyfry blokowe	193
Tryby operacyjne szyfrów blokowych	196
Dopełnianie szyfrowanych bloków	199
Atak na dopełnienie z udziałem wyrocni	200
Szyfry strumieniowe	202
Kryptografia asymetryczna	202
Algorytm RSA	204
Dopełnianie w szyfrowaniu RSA	206
Wymiana kluczy Diffiego-Hellmana	206
Algorytmy podpisów	208
Algorytmy haszowania	208
Asymetryczne algorytmy podpisów	209
Kody uwierzytelniania komunikatów	210
Infrastruktura klucza publicznego	213
Certyfikaty X.509	213
Weryfikacja certyfikatów w łańcuchu	215

Analiza przypadku: protokół TLS	216
Uzgadnianie TLS	216
Negocjowanie wstępne	217
Uwierzelnianie punktów końcowych	218
TLS a wymagania w zakresie bezpieczeństwa	221
Podsumowanie	223

8

IMPLEMENTOWANIE PROTOKOŁU SIECIOWEGO 225

Reprodukcja przechwyconego ruchu sieciowego	226
Przechwytywanie ruchu za pomocą Netcat	226
Reprodukcja przechwyconych pakietów UDP	229
Reimplementacja proxy	230
Ponowne wykorzystywanie kodu wykonywalnego	236
Kod aplikacji .NET	237
Kod aplikacji w Javie	242
Binarny kod maszynowy	243
Neutralizacja szyfrowania w protokole TLS	249
Rozpoznawanie stosowanego algorytmu szyfrowania	250
Deszyfracja danych protokołu TLS	251
Podsumowanie	257

9

IMPLEMENTACYJNE ZAGROŻENIA BEZPIECZEŃSTWA APLIKACJI 259

Kategorie zagrożeń bezpieczeństwa	260
Zdalne wykonywanie kodu	260
Blokada usług	260
Ujawnianie poufnych informacji	260
Omijanie uwierzelniania	261
Omijanie autoryzacji	261
Niszczanie zawartości pamięci	262
Pamięciowe bezpieczeństwo języków programowania	262
Przepełnienie bufora w pamięci	263
Błędy w indeksowaniu tablic	269
Niedozwolona ekspansja danych	270
Błędy dynamicznego przydziału pamięci	270
Domyślne i hardkodowane dane uwierzelniające	271
Enumeracja użytkowników	272
Nieprawidłowy dostęp do zasobów	273
Postać kanoniczna nazwy zasobu	273
Przegadana diagnostyka	275
Wyczerpanie pamięci	276
Wyczerpanie przestrzeni w pamięci masowej	278

Wyczerpanie mocy procesora	278
Kosztowne algorytmy	279
Konfigurowalna kryptografia	281
Niebezpieczne formatowanie łańcuchów	282
Wstrzykiwanie poleceń systemowych	283
Wstrzykiwanie kodu SQL	284
Niebezpieczna konwersja tekstu	286
Podsumowanie	287

10

WYKRYWANIE I EKSPLOATACJA LUK W ZABEZPIECZENIACH 289

Testowanie fazyjne	290
Najprostszy test fazyjny	290
Fuzzer mutacyjny	291
Generowanie przypadków testowych	292
Segregacja luk	292
Debugowanie aplikacji	293
Narzędzia wspomagające analizę awarii	300
Eksploatowanie typowych luk	303
Eksploatowanie nadpisywania pamięci	303
Nieuprawniony zapis do plików wskutek nadpisywania zawartości pamięci	311
Tworzenie kodu powłoki	314
Warsztat	315
Int3 — proste debugowanie	317
Wywołania systemowe	318
Wywoływanie zewnętrznych programów	323
Generowanie kodu powłoki za pomocą Metasploita	325
Zapobieganie eksploatowaniu nadpisywania pamięci	327
DEP — zapobieganie wykonywaniu danych	327
Zapobieganie powrotom pod spreparowane adresy	328
ASLR — randomizacja przestrzeni adresowej	331
Kanarki na stosie	334
Podsumowanie	337

A

NARZĘDZIA WSPOMAGAJĄCE

ANALIZĘ PROTOKOŁÓW SIECIOWYCH 339

Bierne przechwytywanie ruchu sieciowego i jego analiza	340
Microsoft Message Analyzer	340
TCPDump i LibPCAP	341
Wireshark	341
Czynne przechwytywanie ruchu sieciowego i jego analiza	342
Canape	342
Canape Core	343
Mallory	344

Połączenia sieciowe i analizowanie protokołów	344
HPing	344
Netcat	344
NMap	345
Testowanie aplikacji webowych	345
Burp Suite	346
Zed Attack Proxy (ZAP)	347
Mitmproxy	347
Testowanie fazyjne, generowanie pakietów, eksploatacja luk	348
American Fuzzy Lop (AFL)	348
Kali Linux	349
Metasploit	349
Scapy	349
Sulley	350
Podsłuchiwanie sieci i przekierowywanie pakietów	350
DNSMasq	350
Ettercap	350
Inżynieria wsteczna kodu wykonywalnego	351
Java Decompiler (JD)	351
IDA/IDA Pro	352
Hopper	353
ILSpy	353
.NET Reflector	354

SKOROWIDZ 355

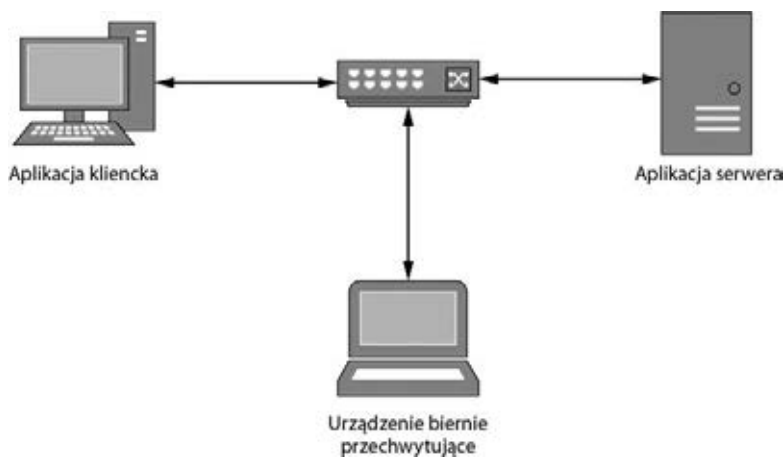
2

Przechwytywanie ruchu sieciowego

Wbrew pozorom efektywne przechwytywanie danych transmitowanych w ramach ruchu sieciowego (efektywne, czyli prowadzone tak, by pozyskane dane okazały się rzeczywiście użyteczne w procesie analizy), nie jest zadaniem banalnym i stanowić może prawdziwe wyzwanie. W tym rozdziale omawiam dwa główne rodzaje przechwytywania: **bierne** (ang. *passive*) i **czynne** (ang. *active*). Przechwytywanie bierne polega na *ekstrakcji przesyłanych danych bezpośrednio z łącza* („drułu”) bez bezpośredniej ingerencji w ruch sieciowy; technika ta jest doskonale znana użytkownikom programu Wireshark i podobnych programów. Przechwytywanie czynne zaś oznacza ingerencję w dane przesyłane między klientem a serwerem i daje znacznie większe możliwości śledzenia niż przechwytywanie bierne, lecz za cenę dodatkowych komplikacji. Wygodnie jest myśleć o przechwytywaniu czynnym w kategoriach dodatkowego obiektu włączającego się w wymianę danych, którym to obiektem może być węzeł proxy lub fizyczny operator (człowiek). W dalszej części rozdziału zajmiemy się szczegółami wymienionych technik.

Bierne przechwytywanie ruchu sieciowego

Przechwytywanie bierne jest techniką raczej nieskomplikowaną, nie wymaga bowiem ani używania dodatkowego sprzętu, ani tworzenia własnego kodu. Dostęp do przesyłanych danych może odbywać się za pośrednictwem jednej z „końcówek” (czyli komputera klienta lub serwera). Można też pozyskiwać dane „z trasy” przez fizyczną ingerencję w łącze (na przykład poprzez nacięcie przewodu i fizyczne przyłączenie się do niego). Na rysunku 2.1 widoczny jest najprostszy wariant przechwytywania ruchu bezpośrednio z łącza ethernetowego.



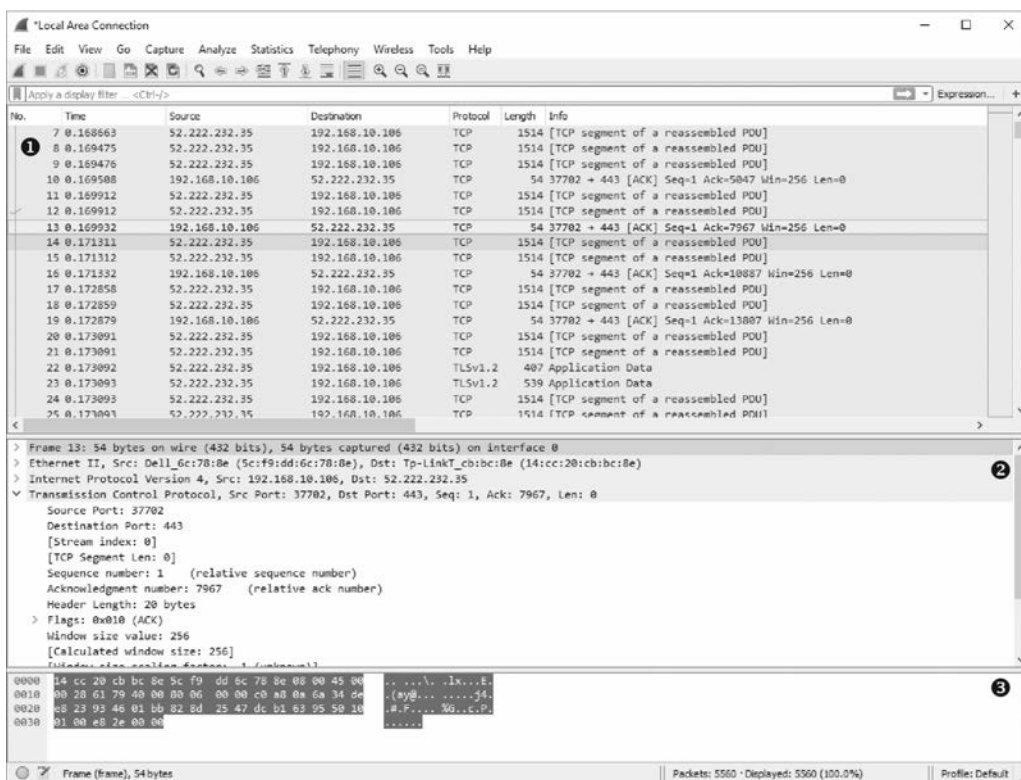
Rysunek 2.1. Przykład biernego przechwytywania ruchu sieciowego

Wireshark — podstawy

Wireshark to najpopularniejsza bodaj aplikacja do „podsluchiwania” pakietów sieciowych. Jest aplikacją międzyplatformową, łatwą w obsłudze, zawierającą wiele wbudowanych mechanizmów przydatnych w analizie działania protokołów sieciowych. W rozdziale 5. pokażę, jak stworzyć własny dysektor (ang. *dissector*) pomagający w analizowaniu protokołów, tutaj ograniczę się do konfigurowania Wiresharka na potrzeby przechwytywania pakietów IP.

Aby przechwytywać ruch ethernetowy (przewodowy lub bezprzewodowy), urządzenie przechwytyjące musi pracować w tak zwanym **trybie nasłuchiwania** (ang. *promiscuous mode*). Urządzenie w tym trybie odbiera i przetwarza każdą ramkę ethernetową, nawet jeśli nie jest adresatem tejże ramki. Przechwytywanie ruchu generowanego przez aplikację działającą na tym samym komputerze co aplikacja przechwytyjąca jest bardzo proste, wystarczy włączyć monitorowanie interfejsu wyjściowego lub lokalnej pętli zwrotnej (ang. *local loopback*), znanej również pod nazwą *localhost*. Podsluchiwanie aplikacji pracującej na innym urządzeniu wymaga użycia dodatkowego sprzętu, takiego jak hub bądź skonfigurowany przełącznik (ang. *switch*), wymuszającego skierowanie ruchu sieciowego do określonego interfejsu.

Na rysunku 2.2 widoczne jest domyślne okno aplikacji Wireshark wykonującej przechwytywanie ruchu sieciowego z interfejsu ethernetowego.



Rysunek 2.2. Domyślne okno aplikacji Wireshark

Okno to podzielone jest na trzy części. W części ❶ widoczna jest chronologiczna lista pakietów „złapanych” w sieci; dla każdego pakietu widoczne są adresy IP (źródłowy i docelowy) i zdekodowana informacja podsumowująca zawartość. Część ❷ to anatomiczny widok pakietu wybranego w części ❶, czyli wynik rozbioru tego pakietu na protokoły, zgodnie z warstwowym modelem sieci OSI-ISO. Ten sam pakiet przedstawiony jest w części ❸ jako surowy strumień bajtów.

Protokół TCP ma naturę strumieniową, a jednym z jego zadań jest odzyskiwanie informacji traconej wskutek gubienia pakietów IP lub uszkodzania przesyłanych danych. Sieci oparte na protokole IP nie gwarantują zachowywania kolejności pakietów — pakiety IP mogą docierać do węzła docelowego w kolejności innej niż kolejność ich wysyłania przez węzeł źródłowy. Piszę o tym tutaj dlatego, że ta okoliczność mogłaby utrudniać identyfikowanie pakietów IP w kontekście wyższej warstwy, której ładunek użyteczny przenoszą. Na szczęście jednak Wireshark wyposażony jest w wiedzę o strukturze i specyfice znanych protokołów, w szczególności więc zapewnia między innymi rekonstrukcję segmentów

TCP do postaci oryginalnego strumienia i prezentowanie kompletnej informacji w jednym miejscu — w tym celu należy podświetlić żądany pakiet w części ❶ i wybrać z menu głównego opcję *Analyze/Follow TCP Stream*. Powinno się wówczas wyświetlić okno podobne do pokazanego na rysunku 2.3, zawierające czytelny widok zdekodowanego strumienia.



Rysunek 2.3. Czytelna prezentacja strumienia TCP w oknie programu Wireshark

Wireshark jest narzędziem wszechstronnym i opisanie wszystkich jego możliwości wykraczałoby poza ramy niniejszej książki. Zainteresowanym Czytelnikom polecić mogę pozycję *Praktyczna analiza pakietów*¹. Wireshark jest nieocenionym narzędziem do analizy ruchu sieciowego, jest dostępny za darmo na zasadach General Public License (GPL).

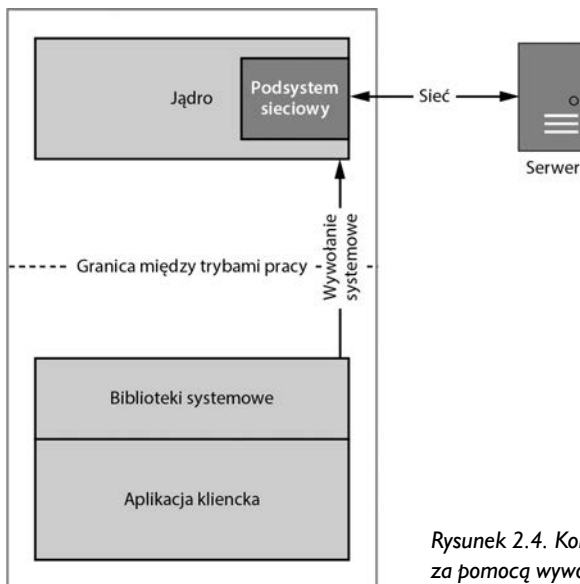
¹ Chris Sanders, *Praktyczna analiza pakietów. Wykorzystanie narzędzia Wireshark do rozwiązywania problemów związanych z siecią*. Wydanie III, Helion, Gliwice 2017.

Alternatywne techniki biernego przechwytywania

Przechwytywanie pakietów może być niekiedy niewykonalne, na przykład w sytuacji, gdy operator nie ma uprawnień do takiego przechwytywania bądź też zabraniają go ograniczone przywileje powłoki urządzenia mobilnego. Ponadto czasami trudno jest oddzielić ruch związany z konkretną aplikacją od ruchu generowanego przez inne aplikacje działające na tym samym urządzeniu. Przydatne stają się wówczas inne techniki pozyskiwania próbek przesyłanych danych, obwiązujące się bez podsłuchiwania pakietów.

Śledzenie wywołań funkcji systemowych

Większość współczesnych systemów operacyjnych realizuje swoje funkcje w dwóch trybach pracy. W **trybie jądra** (ang. *kernel mode*) procesor pracuje w trybie uprzywilejowanym i wykonywać może instrukcje o krytycznym znaczeniu; w tym trybie wykonywany jest kod realizujący funkcje kluczowe dla systemu. W **trybie użytkownika** (ang. *user mode*) procesor może wykonywać jedynie „zwykłe” instrukcje, nie zagrażające integralności systemu ani wykonywanych procesów; w tym trybie wykonywane są typowe aplikacje użytkownika. Gdy aplikacja działa w trybie użytkownika, żąda od systemu wykonania pewnych krytycznych operacji, dokonuje tak zwanego wywołania systemowego (zobacz rysunek 2.4), którego efektem jest przekazanie sterowania programu do kodu funkcji systemowej, zazwyczaj z jednoczesną zmianą trybu pracy na tryb jądra. Funkcje systemowe odpowiedzialne są za realizację różnorodnych usług na rzecz aplikacji, takich jak operacje na systemie plików, tworzenie procesów i wątków oraz — co najistotniejsze w kontekście niniejszego rozdziału — połączeń z sieciami.



Rysunek 2.4. Komunikacja sieciowa realizowana za pomocą wywołania systemowego

Gdy aplikacja kliencka chce połączyć się ze zdalnym serwerem, dokonuje wywołania systemowego mającego na celu otwarcie połączenia. Kiedy połączenie z serwerem zostanie otwarte, aplikacja może za jego pośrednictwem odczytywać dane z serwera i zapisywać na nim dane. Śledzenie takich właśnie wywołań systemowych umożliwi (w niektórych systemach operacyjnych) ich monitorowanie w celu biernej ekstrakcji przesyłanych danych.

W większości systemów uniksowych wywołania systemowe związane z komunikacją sieciową implementują **berkeleyowski model gniazd** (ang. *Berkeley Sockets Model*). Nic w tym dziwnego, gdy uświadomić sobie, że protokół IP zaprojektowany został dla Uniksa w wersji Berkeley Software Distribution (BSD) 4.2. Implementacja owego modelu jest także częścią systemu POSIX, co czyni go standardem de facto. Najważniejsze funkcje tego modelu przedstawiłem w tabeli 2.1.

Tabela 2.1. Podstawowe funkcje berkeleyowskiego modelu gniazd

Nazwa funkcji	Opis
socket	Tworzy nowy deskryptor gniazda.
connect	Dokonuje połączenia gniazda ze wskazanym portem pod wskazanym adresem IP.
bind	Wiąże gniazdo ze wskazanym portem pod lokalnym adresem IP.
recv, read, recvfrom	Dokonuje pobrania danych z sieci za pośrednictwem gniazda. Generyczna funkcja read służy do odczytu danych z dowolnego źródła (np. pliku), funkcje recv i recvfrom są specyficzne dla API gniazd.
send, write, sendto	Wysyła dane do sieci za pośrednictwem gniazda.

Czytelnicy zainteresowani szczegółami działania wymienionych funkcji mogą sięgnąć po książkę *The TCP/IP Guide* (No Starch Press, 2005) bądź do dowolnego podręcznika systemu uniksowego za pomocą polecenia

```
man 2 nazwa_funkcji
```

Mnóstwo informacji dotyczących systemu Unix dostępnych jest ponadto online. Zobaczmy teraz, jak monitoruje się wywołania systemowe.

Linuksowy program strace

W systemie Linux bezpośrednio monitorowanie aplikacji pod kątem wywołań systemowych nie wymaga szczególnych uprawnień, chyba że aplikacja ta została uruchomiona przez uprzywilejowanego użytkownika. Większość dystrybucji Linuksa zawiera poręczny program narzędziowy strace, wykonujący wiele funkcji związanych z takim monitorowaniem. Program ten nie musi być jednak domyślnie zainstalowany, należy go wtedy pobrać za pomocą menedżera pakietów lub skompilować jego kod źródłowy.

W celu rejestrowania wywołań systemowych generowanych przez daną aplikację należy uruchomić program za pomocą polecenia

```
$ strace -e trace=network,read,write ścieżka parametry
```


gdzie *ścieżka* jest kompletną ścieżką do śledzonej aplikacji, a *parametry* są parametrami jej uruchomienia.

Zalóżmy, że chcemy śledzić przykładową aplikację `customapp`, odczytującą i zapisującą kilka łańcuchów tekstowych. Polecenie uruchamiające śledzenie i rezultaty tego śledzenia widoczne są na listingu 2.1.

Listing 2.1. Przykładowy raport ze śledzenia za pomocą programu `strace`

```
$ strace -e trace=network,read,write customapp
. . .
❶ socket(PF_INET, SOCK_STREAM, IPPROTO_TCP) = 3
❷ connect(3, {sa_family=AF_INET, sin_port=htons(5555),
             sin_addr=inet_addr("192.168.10.1")}, 16) = 0
❸ write(3, "Hello World!\n", 13)          = 13
❹ read(3, "Boo!\n", 2048)                 = 5
```

W wierszu ❶ raportu widzimy utworzenie nowego gniazda TCP, któremu system przydziela uchwyt (deskryptor) 3. Wiersz ❷ ilustruje wydanie polecenia `connect` w celu połączenia gniazda identyfikowanego przez uchwyt 3 z portem 5555 węzła o adresie IP 192.168.0.1 Następnie, w wierszu ❸, do tegoż gniazda wysłany zostaje łańcuch tekstowy `Hello World` zakończony znakiem nowego wiersza (`\n`). Wreszcie, w wierszu ❹, ze wspomnianego gniazda odczytywany jest łańcuch znaków `Boo!\n`. Analizując taki lub podobny raport, można się zorientować, co tak naprawdę dzieje się w ramach śledzonej aplikacji — i to bez żadnych specjalnych uprawnień.

Monitorowanie połączeń sieciowych za pomocą programu `DTrace`

`DTrace` to potężne narzędzie w wielu systemach uniksowych, między innymi w Solarisie (dla którego zostało oryginalnie stworzone), MacOS i FreeBSD. Jest narzędziem skryptowym, funkcjonującym w oparciu o próbki (ang. *probes*) i ich dostawców (ang. *providers*) i umożliwia śledzenie rozmaitych aspektów systemu, między innymi wywołań systemowych w aplikacjach. Jego skrypty zapisywane są w języku o składni zbliżonej do języka C. Szczegóły jego obsługi znaleźć można w podręczniku online pod adresem http://wuw.dtracebook.com/index.php/DTrace_Guide.

Na listingu 2.2 widoczny jest przykładowy skrypt `DTrace` uruchamiający monitorowanie wychodzących połączeń IP.

Listing 2.2. Przykładowy skrypt programu `DTrace`

```
traceconnect.d /* traceconnect.d – Prosty skrypt DTrace do monitorowania sieciowych wywołań systemowych */
❶ struct sockaddr_in {
    short sin_family;
    unsigned short sin_port;
    in_addr_t sin_addr;
    char sin_zero[8];
```

```

};

❷ syscall::connect:entry
❸ /arg2 == sizeof(struct sockaddr_in)/
{
    ❹ addr = (struct sockaddr_in*)copyin(arg1, arg2);
    ❺ printf("proces:'%s' %s:%d", execname,
            inet_ntop(2, &addr-sin_addr), ntohs(addr-sin_port));
}

```

Zadaniem tego przykładowego skryptu jest monitorowanie wywołań funkcji systemowej connect i wyjściowych pakietów IPv4 niosących segmenty TCP i datagramy UDP. Funkcję tę wywołuje się z trzema parametrami, reprezentowanymi w skrypcie przez zmienne arg0, arg1 i arg2. Parametr arg0 reprezentuje deskryptor gniazda (akurat w tym skrypcie niewykorzystywany), arg1 jest adresem struktury odzwierciedlającej strukturę gniazda w pamięci aplikacji, arg2 jest rozmiarem tej struktury w bajtach. Rozmiar ten może mieć różną wartość, przykładowo dla protokołu IPv4 jest mniejszy niż w przypadku IPv6.

Wspomniana struktura adresu gniazda definiowana jest w miejscu ❶ (sockaddr_in) w wersji dla IPv4 (notabene takie i podobne definicje można kopiować wprost z nagłówek kodu źródłowego systemu). Rodzaj monitorowanych wywołań systemowych specyfikowany jest w instrukcji ❷. Instrukcja ❸ jest instrukcją warunkową filtrującą śledzone wywołania — uwzględniane są tylko te, dla których rozmiar struktury adresu gniazda jest taki sam jak rozmiar sockaddr_in. Instrukcja ❹ kopiuje wspomnianą strukturę z parametrów wywołania systemowego do lokalnej struktury skryptu w celu jej analizowania — w tym przypadku analiza ogranicza się do wypisania na konsolę nazwy procesu, docelowego adresu IP i docelowego portu, co realizowane jest przez instrukcję ❺.

Aby uruchomić śledzenie, należy zapisać powyższy skrypt w pliku tekstowym (niech będzie to plik traceconnect.d) i po uprzednim zalogowaniu się jako root wydać polecenia

```
dtrace -s traceconnect.d
```

W efekcie otrzymamy raport ze śledzenia podobny do tego z listingu 2.3.

Listing 2.3. Przykładowy raport programu DTrace z przebiegu monitorowania wywołań systemowych

```

proces:'Google Chrome' 173.194.78.125:5222
proces:'Google Chrome' 173.194.66.95:443
proces:'Google Chrome' 217.32.28.199:80
proces:'ntpd' 17.72.148.53:123
proces:'Mail' 173.194.67.109:993
proces:'syncdefaultsd' 17.167.137.30:443
proces:'AddressBookSour' 17.172.192.30:443

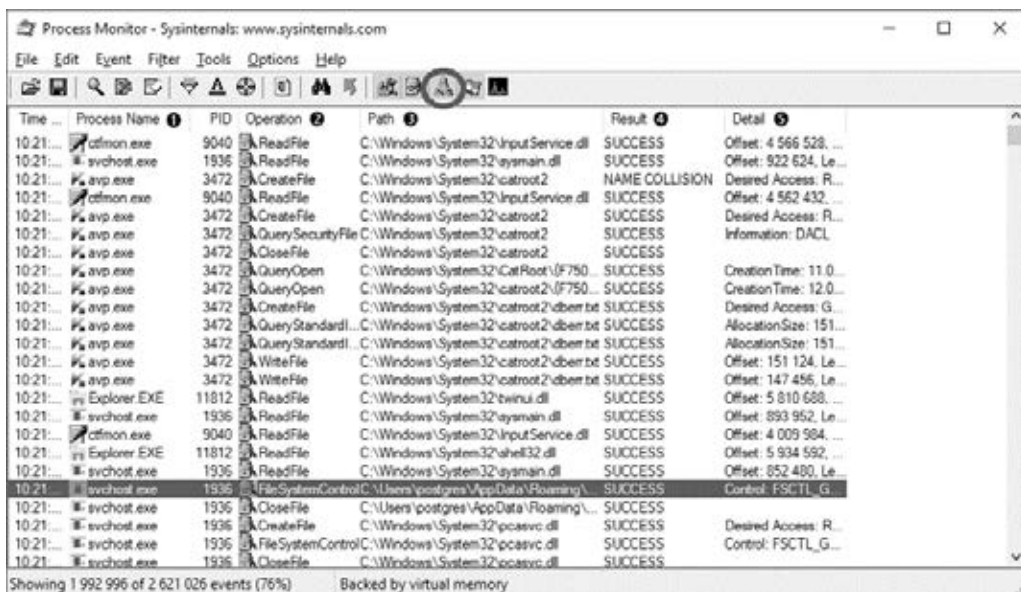
```

Jak widać, dla każdego uwzględnionego wywołania systemowego wyświetlana jest nazwa procesu wywołającego oraz docelowy adres IP i port. Co prawda raport ten nie zawsze jest tak czytelny jak raporty programu *strace*, nie umniejsza to jednak w niczym bogactwa możliwości programu *DTrace*, a zaprezentowany przykład odzwierciedla jedynie drobny ułamek tego bogactwa.

Monitor procesów Windows

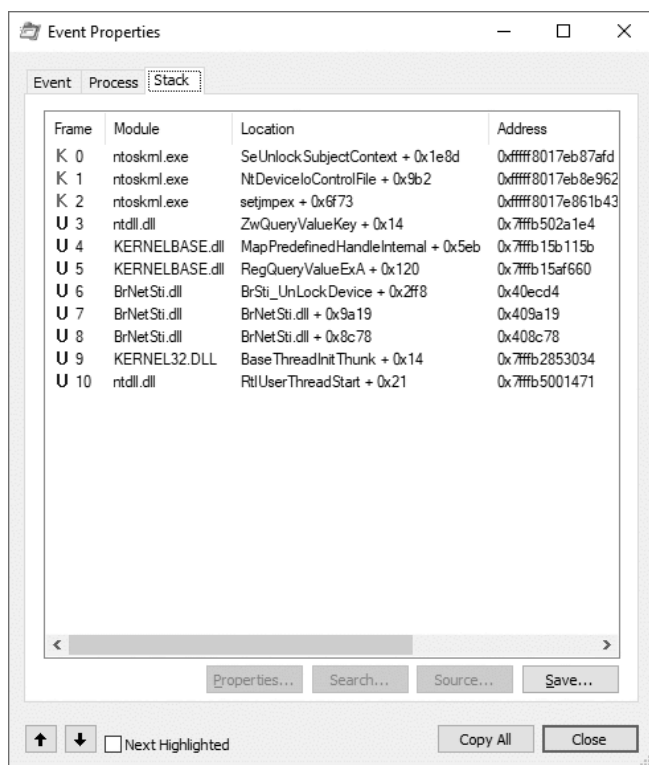
W przeciwieństwie do systemów uniksowych Microsoft Windows implementuje własny zestaw funkcji API związanych z systemem operacyjnym zamiast bezpośrednich wywołań funkcji systemowych. Dostęp do sieci odbywa się poprzez sterownik (ang. *driver*), a współpraca z gniazdami realizowana jest przez „plikowe” funkcje *open*, *read* i *write*. Mimo iż Windows umożliwia monitorowanie aktywności sieciowej w sposób podobny do programu *strace*, to monitorowanie ruchu sieciowego na poziomie podobnym do platform uniksowych staje się znacznie trudniejsze.

Począwszy od wersji Vista, Windows umożliwia monitorowanie aktywności sieciowej za pomocą mechanizmu zdarzeń (ang. *events*). Co prawda implementacja takiego monitorowania jest zadaniem bardzo skomplikowanym, szczęśliwie jednak ktoś już temu zadaniu znakomicie sprostał: na rysunku 2.5 widzimy okno główne aplikacji *Process Monitor*, dostępnej do pobrania pod adresem <https://docs.microsoft.com/pl-pl/sysinternals/downloads/procmon/>. Kliknięcie zaznaczonego na rysunku przycisku na pasku narzędziowym spowodowało uaktywnienie filtra sprawiającego, że widoczne są tylko zdarzenia systemowe związane z połączeniami sieciowymi.



Rysunek 2.5. Okno główne monitora procesów Windows z uwidocznieniem połączeń sieciowych

Dla każdego procesu widoczne są między innymi: jego nazwa i identyfikator ❶, nazwa wykonywanej operacji sieciowej ❷, ścieżka modułu wykonywanego procesu ❸, status operacji ❹ oraz rozmaite szczegóły związane z transmitowanymi danymi ❺. Jakkolwiek *Process Monitor* nie daje żadnego czytelnego podglądu postaci tych danych, to widok połączeń sieciowych nawiązywanych przez poszczególne aplikacje jest bardzo wartościową informacją. Inną cenną informacją, pomocną w śledzeniu, debugowaniu czy inżynierii wstecznej procesu (zajmiemy się nią szczegółowo w rozdziale 6.), jest stan **stosu wywołań** (ang. *call stack*) prowadzących do uruchomienia operacji sieciowej przez proces. By ją wyświetlić, należy z menu kontekstowego podświetlonego procesu (rozwijanego prawym kliknięciem) wybrać opcję *Stack*. Dla przykładowego procesu stos wywołań może wyglądać tak jak na rysunku 2.6.



Rysunek 2.6. Stan stosu wywołań dla wybranego procesu

Mimo iż *Process Monitor* nie jest w śledzeniu wywołań systemowych tak pomocny jak analogiczne narzędzia na innych platformach, to w Windows jest on cenny jako narzędzie umożliwiające identyfikowanie protokołów sieciowych używanych przez poszczególne procesy. I chociaż brak jest obrazu danych wymienianych w ramach tych protokołów, to już sama lista protokołów może być cennym wstępem do innych technik przechwytywania i analizowania ruchu sieciowego.

Zalety i wady biernego przechwytywania

Niewątpliwą zaletą technik biernego przechwytywania ruchu sieciowego jest fakt, iż odbywają się one bez ingerowania w ów ruch, czyli (nazwijmy rzecz po imieniu) bez uszkodzania danych wymienianych między klientem a serwerem. Biernie przechwytywanie odbywa się także bez modyfikowania adresów komunikujących się węzłów, nie wymaga też zmian w konfiguracjach komunikujących się aplikacji.

Biernie przechwytywanie jest ponadto jedyną dostępną metodą w sytuacji, gdy nie ma możliwości kontrolowania komputerów klienta lub serwera — przy niewielkim wysiłku można wówczas zorganizować pobieranie danych bezpośrednio „z drutu”. Mimo ograniczonych możliwości techniki biernego przechwytywania pozwalają zebrać cenne dane, przydatne nie tylko w dalszej analizie, prawdopodobnie z udziałem technik czynnego przechwytywania, ale i już na starcie pomocne w samym wyborze konkretnej z tych technik.

Specyfiką wszystkich metod biernego przechwytywania ruchu sieciowego jest fakt, iż są to metody niskopoziomowe, koncentrujące się raczej na fizycznej postaci danych niż na treści, jaką te dane przenoszą — i to jest podstawowy mankament tychże metod. Co prawda narzędzia w rodzaju Wiresharka przyczyniają się do jego złagodzenia, ale jest to możliwe jedynie w przypadku standardowych protokołów — gdy napotykają niestandardowy protokół, specyficzny dla danej aplikacji, pomóc mogą niewiele, nie znając przecież semantyki tego protokołu.

Drugi podstawowy mankament biernego przechwytywania wynika z jego natury, czyli nietykalności podsłuchiwanego danych. W sytuacji, gdy dane te są zaszyfrowane lub skompresowane, ich analiza może być niewykonalna bez uprzedniego rozszyfrowania czy dekompresji bądź też wyłączenia opcji szyfrowania czy kompresowania w aplikacji wysyłającej. Tego wszystkiego nie da się osiągnąć przez biernie podsłuchiwanie czy nawet wstrzykiwanie dodatkowych pakietów.

Gdy zatem ograniczenia właściwe technikom biernego przechwytywania stają się barierą w uzyskiwaniu zadowalających rezultatów, należy zmienić strategię i zwrócić się ku przechwytywaniu czynnemu.

Czynne przechwytywanie ruchu sieciowego

Podstawową cechą czynnego przechwytywania, odróżniającą je od przechwytywania biernego, jest ingerencja w ruch sieciowy, realizowana najczęściej w formie tak zwanego „ataku z człowiekiem pośrodku” (ang. *man-in-the-middle attack* — MITM), przy czym „człowiek” jest zwykle operatorem urządzenia przechwytyującego. Urządzenie to, ulokowane na drodze przepływu danych między klientem a serwerem, pełni rolę pomostu między nimi. Taka konfiguracja, przedstawiona schematycznie na rysunku 2.7, daje wiele korzyści, na przykład umożliwia wyłączenie kompresji lub szyfrowania danych, czyniąc je wygodniejszymi do analizy lub eksploatacji „dziury” w zabezpieczeniach.



Rysunek 2.7. Atak z „człowiekiem pośrodku”

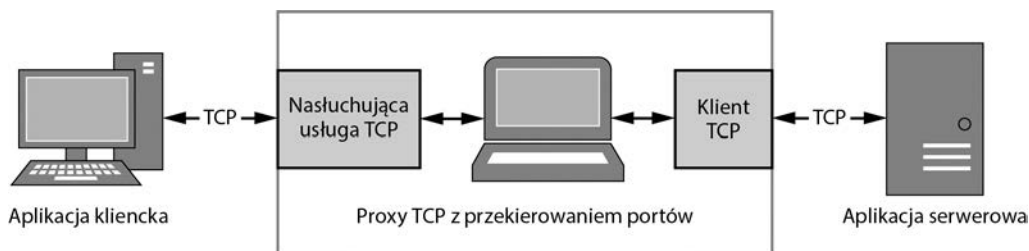
Wykonanie czynnego przechwytywania jest zadaniem znacznie trudniejszym niż bierne podsłuchiwanie pakietów, wymaga bowiem dokonania zmiany trasy przy użyciu urządzenia przechwytyjącego. To z kolei może powodować niezamierzone lub niepożądane efekty: jeżeli na przykład zmienimy w aplikacji adres klienta lub serwera na adres proxy, dane wysyłane przez tę aplikację mogą trafić w niewłaściwe miejsce. Mimo to czynne przechwytywanie jest bodaj najbardziej użyteczną metodą analizowania protokołów sieciowych i eksploatawania luk w zabezpieczeniach.

Proxy sieciowe

Najczęstszą metodą przypuszczania ataków „z człowiekiem pośrodku” jest zmuszenie aplikacji klienckiej do komunikowania się z serwerem za pośrednictwem usługi proxy. Poniżej omawiam zalety i wady najczęściej używanych w tym celu proxy i wyjaśniam, jak można za ich pomocą przechwytywać ruch sieciowy, analizować przechwycone dane i wykorzystywać wyniki analizy do eksploatawania protokołów sieciowych.

Proxy z przekierowaniem portów

Przekierowywanie portów (ang. *port forwarding*) to najprostszy sposób realizacji proxy przechwytyjącego. Jako proxy konfigurujemy serwer (TCP lub UDP) nasłuchujący na określonym porcie i w przypadku otrzymania żądania z aplikacji klienckiej przekazujemy to żądanie do „prawdziwego” serwera, z którym wcześniej proxy nawiązało połączenie. W ten sposób klient i serwer zostają logicznie połączone ze sobą, chociaż fizycznie każde z nich komunikuje się wyłącznie z proxy. Ideę tę przedstawia rysunek 2.8.



Rysunek 2.8. Proxy TCP z przekierowaniem portów

Prosta implementacja

Do zrealizowania opisanego modelu wykorzystamy mechanizm przekierowywania portów TCP wbudowany w bibliotekę *Canape Core*. Kod widoczny na listingu 2.4 należy w tym celu umieścić w pliku skrypcowym C#, zamieniając symbole *LOCALPORT* ❶, *REMOTEHOST* ❷ i *REMOTEPORT* ❸ na konkretne wartości (odpowiednio) portu lokalnego, zdalnego hosta i portu zdalnego.

Listing 2.4. Prosty przykład przekierowania portów TCP

```
PortFormat // PortFormatProxy.csx – proste proxy przekierowujące porty TCP
↳ Proxy.csx // Udostępnij metody ReadLine, WriteLine i Writepackets na poziomie globalnym
using static System.Console;
using static CANAPE.Cli.ConsoleUtils;

// Utworzenie szablonu proxy
var template = new FixedProxyTemplate();
template.LocalPort = ❶ LOCALPORT;
template.Host = ❷ "REMOTEHOST";
template.Port = ❸ REMOTEPORT;

// Utworzenie i uruchomienie instancji proxy
❹ var service = template.Create();
    service.Start();

    WriteLine("Utworzono {0}", service);
    WriteLine("Naciśnij Enter, by zakończyć...");
    ReadLine();
❺ service.Stop();

// Zrzut pakietów
var packets = service.Packets;
WriteLine("Przechwycono {0} pakietów:",
          packets.Count);
❻ WritePackets(packets);
```

Ten prosty skrypt rozpoczyna swe działanie od utworzenia instancji szablonu *FixedProxyTemplate* ❶. *Canape Core* funkcjonuje na bazie modelu szablonów, choć wymaga również operowania niskopoziomowymi funkcjami konfiguracji sieci. Utworzona instancja szablonu inicjalizowana jest następnie niezbędnymi informacjami związanymi z siecią lokalną i siecią docelową. Zainicjalizowana instancja szablonu służy do utworzenia instancji usługi ❹; zasadniczo dokumenty mogą być postrzegane jako szablony dla usług. Instancja usługi zostaje uruchomiona i następuje konfigurowanie połączenia sieciowego. Usługa pozostaje uruchomiona, a wywołanie funkcji *ReadLine()* powoduje oczekiwanie na naciśnięcie klawisza *Enter*, po czym usługa zostaje zatrzymana ❺. Raport o wszystkich przechwyconych pakietach wypisany zostaje na konsolę za pomocą metody *WritePackets()* ❻.

Uruchomienie skryptu powinno spowodować powiązanie z lokalnym portem o numerze `LOCALPORT` interfejsu `localhost`. Gdy zostanie nawiązane połączenie z tym portem, kod proxy powinien spowodować nawiązanie nowego połączenia TCP z portem `REMOTEPORT` hosta `REMOTEHOST` i związać ze sobą oba połączenia.

OSTRZEŻNIE *Związanie proxy ze wszystkimi adresami sieciowymi jest posunięciem ryzykownym z perspektywy bezpieczeństwa sieci, jako że testowe proxy tworzone na użytek testowania protokołów rzadko kiedy zawierają solidne mechanizmy zabezpieczeń. By uniknąć wynikającego stąd ryzyka, należy wiązać testowe proxy wyłącznie z interfejsem lokalnej pętli zwrotnej (ang. loopback) — jak na listingu 2.4 — chyba że ma się pełną kontrolę nad podłączaną siecią albo po prostu nie ma się innego wyboru. Aby proxy z listingu 2.4 związać ze wszystkimi interfejsami sieciowymi, należy ustawić na `TRUE` właściwość `AnyBind`.*

Przekierowywanie ruchu do proxy

Gdy już skompletujemy aplikację proxy, należy skierować do niej ruch sieciowy, którego treść chcemy przechwytywać. Z pomocą przeglądarki WWW jest to bardzo proste: zamiast oryginalnego URL-a w rodzaju `http://www.<domena>`. `com/<zasób>` trzeba wpisać w pasku adresu `http://localhost:<port_lokalny>/<zasób>`.

Z innymi aplikacjami nie jest już tak łatwo, ponieważ konieczna staje się ingerencja w ich ustawienia. I często zdarza się tak, że jedynym ustawieniem dostępnym dla modyfikacji jest docelowy adres (adresy) IP. To prowadzić może do problemu jajka i kury: niewiadomymi mogą być bowiem numery portów TCP lub UDP, które rzeczona aplikacja wykorzystuje pod wskazanym adresem; jest to szczególnie prawdopodobne w aplikacjach wykonujących skomplikowane operacje na bazie połączeń z wieloma usługami. Jest tak między innymi z protokołami zdalnego wywołania procedury (ang. *Remote Procedure Call* — RPC), na przykład z architekturą CORBA (ang. *Common Object Request Broker Architecture*), gdzie nawiązywane jest początkowo połączenie z brokerem, spełniającym rolę katalogu dostępnych usług, dopiero następne połączenia dotyczą portów TCP konkretnych usług, nieznanych a priori.

Dobrym wyjściem z tej kłopotliwej sytuacji jest przeprowadzenie przechwytywania biernego w celu monitorowania jak największej liczby połączeń aplikacji z różnymi usługami. Zdobywając w ten sposób listę portów docelowych używanych typowo przez aplikację, można później próbować odtwarzać poszczególne połączenia z udziałem proxy przekierowującego porty.

Niestety niektóre aplikacje nie pozwalają nawet na konfigurowanie docelowego adresu IP i wtedy trzeba być naprawdę pomysłowym. Prawdopodobnie wspomniana aplikacja odwołuje się do docelowego hosta przez jego nazwę, więc przekierowanie ruchu można uzyskać, instalując ad hoc serwer DNS „przeliczający” nazwę hosta na żądany adres IP proxy. Większość systemów operacyjnych — w tym Windows — umożliwia ponadto wyszczególnienie nazw DNS, które mają być przeliczane w sposób nietypowy.

W procesie przeliczania nazw DNS system operacyjny (lub biblioteka), zanim sięgnie do serwera DNS, analizuje zawartość pliku *hosts* i gdy znajdzie w nim parę adres IP – nazwa, uznaje nazwę za równoważną adresowi IP. Jeżeli więc na przykład plik *hosts* wygląda tak jak na listingu 2.5, to nazwy *www.badgers.com* i *www.domain.com* zostaną potraktowane jako równoważne adresowi 127.0.0.1, oznaczającemu, jak wiadomo, lokalny komputer (*localhost*).

Listing 2.5. Przykładowy plik hosts

```
# Standardowe adresy komputera lokalnego
127.0.0.1    localhost
::1        localhost

# Nazwy przekierowywane do komputera lokalnego
127.0.0.1    www.badgers.com
127.0.0.1    www.domain.com
```

W systemach uniksowych plik *hosts* znajduje się w katalogu */etc/*. W Windows jego domyślną lokalizacją jest podkatalog *\System32\Drivers\etc* katalogu systemowego; można tę lokalizację zmienić, edytując rejestr systemu — jest ona zapisana jako wartość *DataBasePath* w kluczu

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Tcpip\Parameters
```

UWAGA

Niektóre programy antywirusowe i inne produkty zabezpieczające nie zezwalają na modyfikowanie pliku hosts, ponieważ traktują je jako przejaw działania złośliwego oprogramowania. Aby edycję jednak przeprowadzić, należy te produkty tymczasowo wyłączyć.

Zalety i wady proxy z przekierowaniem portów

Główną zaletą proxy z przekierowaniem portów jest prostota jego używania: nasłuchując, oczekujemy na połączenie z klientem, otwieramy nowe połączenie z oryginalnym serwerem i potem już tylko kierujemy ruch sieciowy do jednego lub drugiego. Proxy nie wymaga do działania żadnego dodatkowego protokołu, nie jest także wymagane żadne wsparcie ze strony aplikacji, od i do której ruch mamy przechwytywać.

Proxy z przekierowaniem portów jest ponadto naturalnym sposobem przechwytywania ruchu generowanego przez protokół UDP: jako że jest to protokół bezpołączeniowy, implementacja dedykowanego mu forwardera jest prostsza niż w przypadku TCP.

Ta prostota ma jednak swoją cenę. Przede wszystkim, ponieważ przechwytywamy ruch między *konkretnym* serwerem a *konkretnym* klientem, w sytuacji, gdy aplikacja kliencka wykorzystuje wiele usług na różnych portach, potrzebujemy *wielu instancji* proxy. Jako przykład niech posłuży aplikacja, która łączy się z pojedynczym serwerem (określonym przez adres IP lub nazwę hosta) i chcemy to

połączenie kontrolować bądź to przez zmianę ustawień aplikacji, bądź przez „podszywanie” się pod nazwę hosta (za pośrednictwem pliku *hosts*). Aplikacja ta łączy się z portami TCP 443 i 1234. Ponieważ możemy kontrolować adres docelowy, *ale nie numery portów*, musimy utworzyć dwie instancje proxy, po jednej dla każdego portu.

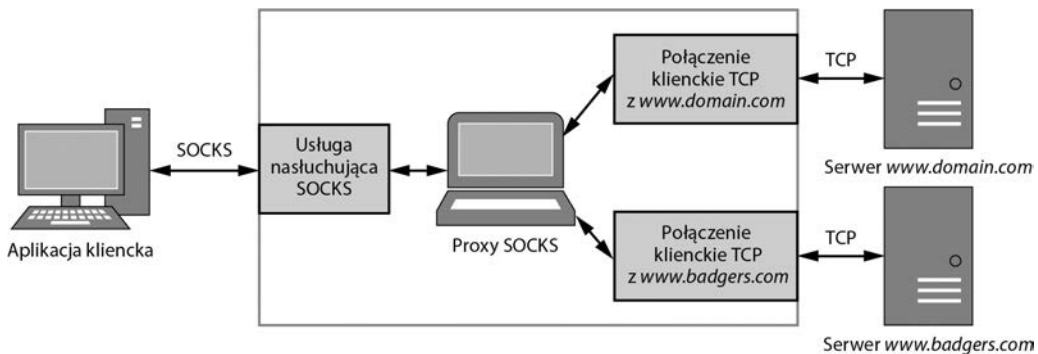
Podobnie jest w sytuacji, gdy dwa połączenia do różnych lokalizacji docelowych prowadzą do *tego samego portu*, powiedzmy 1234. Tworzymy dwie instancje proxy, po jednej dla każdej lokalizacji; obie instancje nasłuchują na porcie 1234. Gdy aplikacja kliencka zażąda połączenia z jedną z lokalizacji, jej żądanie zostanie poprawnie skierowane do serwera. Gdy pojawi się żądanie połączenia z drugą lokalizacją, pojawią się problemy — nie da się poprawnie skierować ruchu do drugiego serwera. Oczywiście do rozwiązania tego problemu konieczna jest zmiana portu na jednym z połączeń. Daje się to łatwo zrobić, gdy numery portów można określać w ustawieniach aplikacji; w przeciwnym razie musimy sięgnąć po inne mechanizmy, takie jak translacja docelowych adresów sieciowych (ang. *Destination Network Address Translation* — DNAT), którą opisuję dokładniej w rozdziale 5. wraz z innymi zaawansowanymi technikami przechwytywania ruchu sieciowego.

Ponadto dany protokół może wykorzystywać określony adres docelowy do swoich prywatnych celów. Przykładowo nagłówek *Host* protokołu HTTP (ang. *Hypertext Transport Protocol*) może być wykorzystywany przez wirtualny host, co może powodować, że protokół przekierowujący będzie funkcjonował niezgodnie z oczekiwaniami albo nie będzie funkcjonował w ogóle dla przekierowanego połączenia. W dalszym ciągu rozdziału, w sekcji „Proxy odwrotne HTTP”, omawiam jeden ze sposobów obejścia tego ograniczenia.

Proxy SOCKS

Proxy SOCKS to w pewnym sensie podrasowane proxy z przekierowaniem portów. Oprócz bowiem przekierowywania połączeń TCP do żądanej lokalizacji sieciowej każde nowe połączenie rozpoczyna się od prostego protokołu uzgadniania wstępnego (ang. *handshaking*) w celu przekazania informacji o lokalizacji docelowej; w klasycznym proxy z przekierowaniem lokalizacja ta była ustalona a priori. Proxy SOCKS realizuje ponadto funkcję nasłuchiwanie połączeń, co jest istotne dla protokołów takich jak FTP (ang. *File Transfer Protocol* — protokół przesyłania plików), który wymaga otwierania nowych portów lokalnych w celu przesyłania plików z serwera. Ogólny schemat proxy SOCKS przedstawiony jest na rysunku 2.9.

W użyciu są obecnie trzy wersje protokołu SOCKS, oznaczone numerami 4, 4a i 5. Wersja 4 ma najpełniejsze wsparcie ze strony produktów sieciowych, lecz obsługuje jedynie połączenia IPv4, czyli adres docelowy musi być 32-bitowy. Wersja 4a to wynik rozszerzenia wersji 4 o możliwość specyfikowania również *nazw* hostów (oprócz, jak dotychczas, adresów IP), co jest bardzo pomocne, gdy nie dysponujemy serwerem DNS. W wersji 5, poza możliwością specyfikowania nazw hostów, wprowadzono obsługę protokołu IPv6 i przekierowywanie portów UDP; ulepszono także mechanizmy uwierzytelniania. Wersja 5 jest ponadto jedyną wersją SOCKS opisaną w dokumentacji RFC (RFC 1928).



Rysunek 2.9. Ogólny schemat proxy SOCKS

W charakterze przykładu przeanalizujemy sytuację, w której klient formułuje żądanie nawiązania połączenia SOCKS z adresem IP 10.0.0.1 na porcie 12345. Format tego żądania widoczny jest na rysunku 2.10. Pole VER zawiera wersję protokołu SOCKS (w tym przypadku 4), wartość 1 w polu CMD oznacza zamiar nawiązania połączenia (wartość 2 oznaczałaby wiązanie z adresem IP), TCP PORT i IP ADDRESS to oczywiście port i adres docelowy. Jedynym sposobem uwierzytelniania w wersji 4 jest uwierzytelnianie w oparciu o nazwę użytkownika, która znajduje się w polu USERNAME; pole to może mieć różną długość, jego koniec sygnalizowany jest przez zerowy oktet.

VER 0x04	CMD 0x01	TCP PORT 12345	IP ADDRESS 0x10000001	USERNAME "james"	Zera 0x00
-------------	-------------	-------------------	--------------------------	---------------------	--------------

Rozmiar w oktetach 1 1 2 4 Zmienny 1

Rysunek 2.10. Żądanie SOCKS w wersji 4

Po pomyślnym nawiązaniu połączenia proxy powinno wysłać odpowiedź w formacie widocznym na rysunku 2.11. VER jest (jak poprzednio) wersją protokołu, RESP zawiera status odpowiedzi, pola TCP PORT i IP ADDRESS zawierają, zgodnie ze swymi nazwami, numer portu docelowego i adres docelowy, ale tylko w przypadku odpowiedzi na żądanie wiązania z adresem; w przypadku innych żądań są wyzerowane. Od tej pory połączenie pozostaje transparentne: klient i serwer bezpośrednio negocjują ze sobą, rola proxy ogranicza się do przekierowywania ruchu w obu kierunkach.

VER 0x04	RESP 0x5A	TCP PORT 0	IP ADDRESS 0
-------------	--------------	---------------	-----------------

Rozmiar w oktetach 1 1 2 4

Rysunek 2.11. Potwierdzenie pomyślnego nawiązania połączenia w wersji 4 protokołu SOCKS

Prosta implementacja

Biblioteka Canape Core zapewnia wsparcie dla wszystkich trzech wersji SOCKS — 4, 4a i 5. Kod widoczny na listingu 2.6 należy umieścić w pliku skrypcowym C#, zamieniając symbol `LOCALPORT` ❷ na konkretną wartość portu lokalnego TCP, na którym proxy ma nasłuchiwać połączeń.

Listing 2.6. Prosta implementacja proxy SOCKS

```
SocksProxy.csx // SocksProxy.csx – proste proxy SOCKS
// Udostępnij metody ReadLine, WriteLine i Writepackets na poziomie globalnym
using static System.Console;
using static CANAPE.Cli.ConsoleUtils;

// Tworzenie szablonu proxy SOCKS
❶ var template = new SocksProxyTemplate();
template.LocalPort = ❷ LOCALPORT;

// Tworzenie i uruchomienie instancji proxy
var service = template.Create();
service.Start();

WriteLine("Utworzono {0}", service);
WriteLine("Naciśnij Enter, by zakończyć...");
ReadLine();
service.Stop();

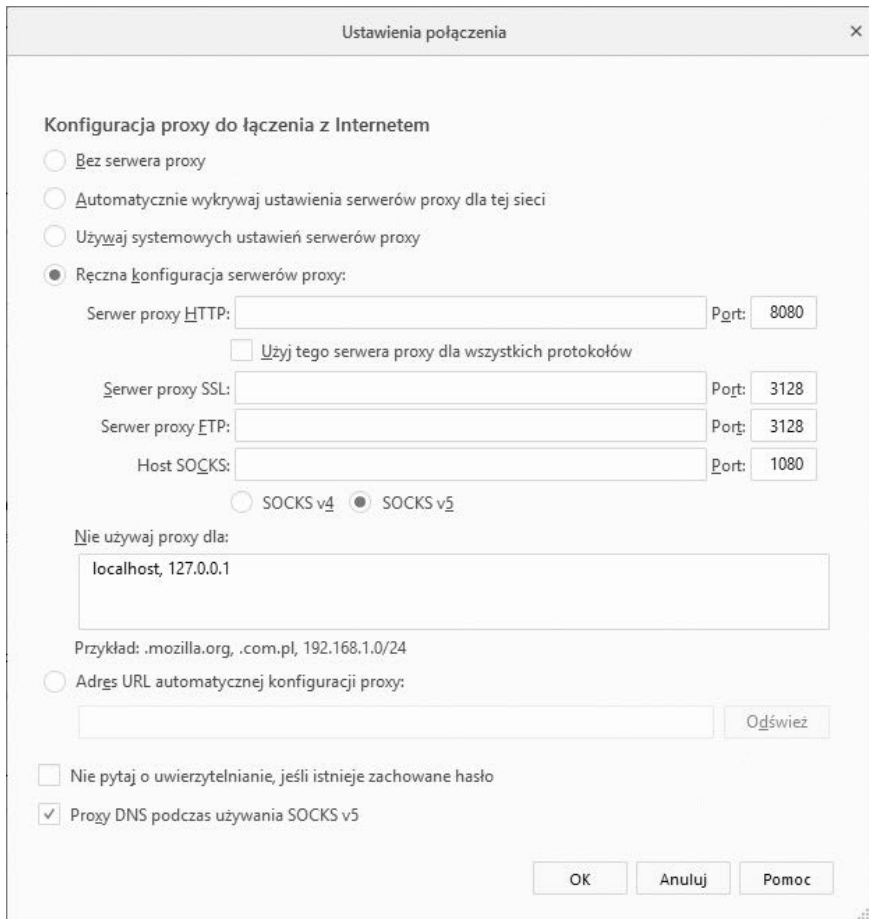
// Zrzut pakietów
var packets = service.Packets;
WriteLine("Przechwycono {0} pakietów:",
          packets.Count);
WritePackets(packets);
```

Kod z listingu 2.6 sporządzony został według tego samego wzorca co kod klasycznego proxy przekierowującego widoczny na listingu 2.4 — z tą różnicą, że w wierszu ❶ tworzony jest szablon SOCKS, reszta kodu jest taka sama.

Przekierowywanie ruchu do proxy

Aby przyjrzeć się bliżej przekierowywaniu ruchu do proxy SOCKS, najlepiej przeanalizować pod tym kątem znane aplikacje. Uruchamiając przeglądarkę Mozilla Firefox i przechodząc do ustawień serwerów proxy, ujrzymy okno dialogowe przedstawione na rysunku 2.12.

W innych aplikacjach wsparcie dla SOCKS może już być mniej oczywiste. Dla aplikacji opartych na Javie biblioteka Java Runtime oferuje opcję włączenia SOCKS dla wszystkich wychodzących połączeń TCP wskutek użycia w wierszu poleceń stosownego parametru. Zobaczmy najpierw kod przykładowej aplikacji, realizującej nawiązywanie połączenia z portem 5555 pod adresem 192.168.10.1 (zobacz listing 2.7).



Rysunek 2.12. Okno ustawień proxy przeglądarki Firefox

Listing 2.7. Prosta aplikacja kliencka w języku Java

```

Socket // SocketClient.java – prosty klient TCP socket
↳ Client.java import java.io.PrintWriter;
import java.net.Socket;

public class SocketClient {
    public static void main(String[] args) {
        try {
            Socket s = new Socket("192.168.10.1", 5555);
            PrintWriter out = new PrintWriter(s.getOutputStream(), true);
            out.println("Hello World!");
            s.close();
        } catch (Exception e) { }
    }
}

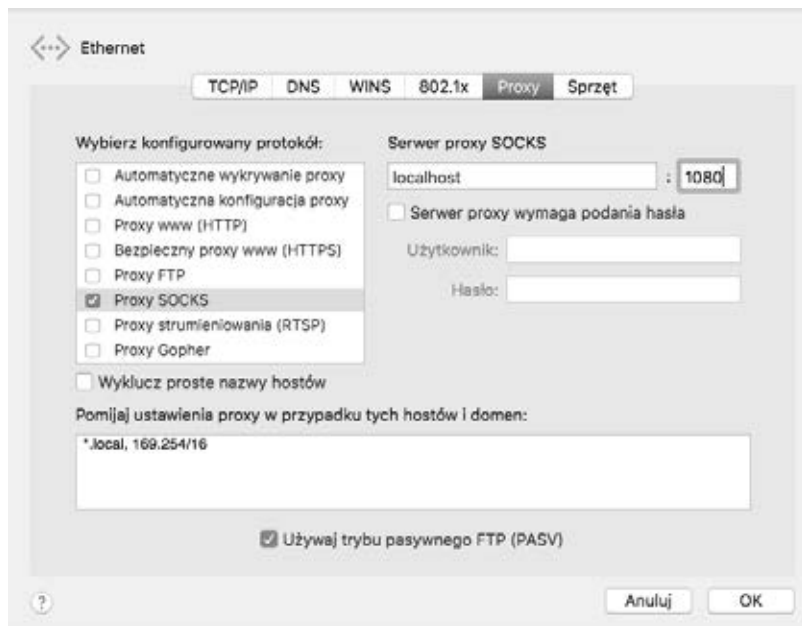
```

Uruchomienie tego kodu w sposób standardowy nie spowoduje jednak przekierowania ruchu, którego mógłby oczekiwać użytkownik. Wystarczy jednak w wierszu poleceń uruchamiającym aplikację ustawić odpowiednio właściwości DsocksProxyHost i DsocksProxyPort, reprezentujące host i port lokalny przechwytywania SOCKS:

```
java -DsocksProxyHost=localhost -DsocksProxyPort=1080 SocketClient
```

by spowodować przechwycenie żądania klienckiego na porcie 1080 lokalnego hosta.

Innym przykładem jest przypadek, gdy SOCKS jest domyślnym proxy systemowym. W systemie MacOS, wybrawszy opcję *Preferencje systemowe/Siec/Zaawansowane.../Proxy*, ujrzymy okno przedstawione na rysunku 2.13. Można tu manipulować ustawieniami SOCKS globalnymi dla systemu i ustawieniami proxy dla różnych protokołów. System daje dość dużą elastyczność pod tym względem, chociaż nie zawsze udaje się osiągnąć pożądane rezultaty po pierwszej próbie.



Rysunek 2.13. Konfigurowanie ustawień proxy w systemie MacOS

Dla wielu aplikacji, które nie zapewniają rodzimej obsługi proxy SOCKS, opracowano rozmaite narzędzia uzupełniające ten brak. Repertuar tych narzędzi obejmuje zarówno produkty darmowe i *open source* — na przykład linuksowy **Dante** (<https://www.inet.no/dante/>) — jak i komercyjne, między innymi **Proxi-fier** w wersjach dla systemów Windows i MacOS (<https://www.proxifier.com/>). Istotą każdego z nich jest wstrzykiwanie do aplikacji kodu modyfikującego obsługę interfejsu gniazd.

Zalety i wady proxy SOCKS

Niewątpliwą zaletą proxy SOCKS, w przeciwieństwie do zwykłego przekierowywania portów, jest możliwość przechwytywania wszystkich połączeń TCP (w wersji 5 SOCKS także ruchu UDP) nawiązywanych przez aplikację. Warunkiem jednakże spożytkowania tej zalety jest możliwość skonfigurowania interfejsu gniazd systemu operacyjnego w taki sposób, by wszystkie połączenia TCP (i ewentualnie UDP) efektywnie kierowane były do proxy.

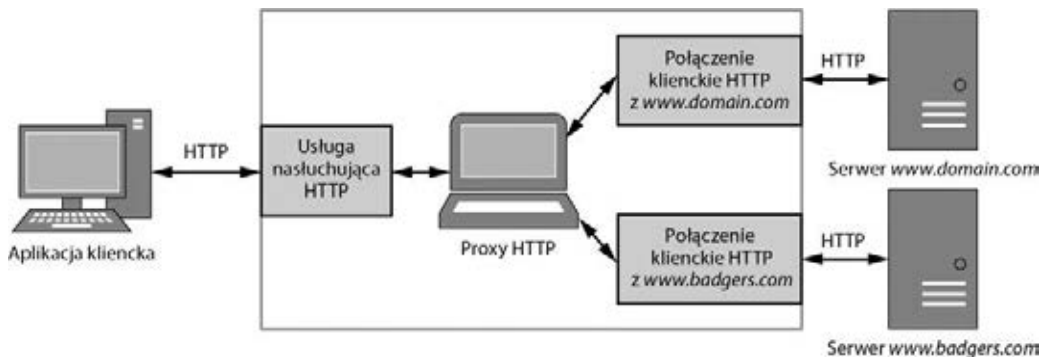
Ponadto proxy SOCKS zasadniczo zachowują tożsamość lokalizacji docelowej z punktu widzenia aplikacji klienckiej. Jeśli więc w treści wysyłanej przez aplikację kliencką znajduje się informacja zależna od tej tożsamości — na przykład adres IP lub numer portu — serwer otrzymuje dane, których faktycznie oczekuje. Niestety nie jest możliwe analogiczne zachowywanie tożsamości *lokalizacji źródłowej*, co powoduje problemy w odniesieniu do protokołów takich jak FTP, które roszczą sobie prawo do otwierania nowych portów po stronie klienckiej. Co prawda protokół SOCKS dostarcza środków do wiązania połączeń nasłuchujących z konkretnymi adresami, lecz odbywa się to za cenę znacznego skomplikowania implementacji, między innymi konieczności utrzymywania wielu strumieni komunikacji między klientem a serwerem.

Jedną ze słabych stron protokołu SOCKS jest brak spójności implementacyjnej między różnymi platformami i aplikacjami. Na przykład Windows oferuje obsługę wyłącznie SOCKS w wersji 4, co ogranicza wsparcie dla tegoż protokołu tylko do IPv4, a jedyną rozpoznawalną nazwą hosta jest localhost. Wersja 4 SOCKS jest ponadto pozbawiona solidnych mechanizmów uwierzytelnienia. Lepsze wsparcie dla SOCKS można w Windows uzyskać za pomocą zewnętrznych narzędzi, te jednak nie zawsze dają gwarancję niezawodnego czy choćby tylko poprawnego działania.

Proxy HTTP

HTTP (ang. *Hypertext Transport Protocol*) to siła napędowa Internetu, mnóstwa usług webowych i protokołów typu REST. Może być również zaadaptowany jako mechanizm transportowy protokołów nieopartych na WWW — należy wśród nich wymienić między innymi RMI (ang. *Java Remote Method Invocation* — zdalne wywołanie metody Javy) i RTMP (ang. *Real Time Messaging Protocol* — protokół [transportu] komunikatów w czasie rzeczywistym). Popularność HTTP jako roboczego mechanizmu transportowego wynika przede wszystkim z zaufania, jakim darzy go większość firewalli, pozwalając mu na bezproblemowe przenikanie („tunelowanie”).

Na rysunku 2.14 widzimy schemat wykorzystywania protokołu HTTP do realizacji proxy przechwytyjącego ruch sieciowy. Zrozumienie tego schematu jest o tyle pożądane, że z pewnością okaże się on użyteczny w analizie protokołów nawet wówczas, gdy testowane aplikacje nie będą mieć nic wspólnego z usługami webowymi. Zewnętrzne narzędzia do testowania aplikacji webowych rzadko kiedy działają idealnie, gdy protokołowi HTTP przychodzi funkcjonować poza jego oryginalnym środowiskiem, i wówczas zaimplementowanie tego protokołu w roli proxy może być jedynym możliwym rozwiązaniem.



Rysunek 2.14. Ogólny schemat proxy HTTP

Istnieją dwa typy proxy HTTP: **przekazujące** (ang. *forwarding*) i **odwrotne** (ang. *reverse*). Oba mają swe zalety i wady z punktu widzenia potencjalnego analizatora protokołów sieciowych.

Proxy przekazujące HTTP

Protokół HTTP opisywany jest w dwóch dokumentach RFC: 1945 (dla wersji 1.0) i 2616 (dla wersji 1.1). Obie wersje oferują proste mechanizmy umożliwiające jego wykorzystywanie jako proxy dla żądań. Na przykład zgodnie ze specyfikacją wersji 1.1 pierwszy pełny wiersz w treści żądania, zwany krótko **wierszem żądania** (ang. *request line*), ma następujący format:

```
❶ GET ❷ /image.jpg HTTP/1.1
```

W żądaniu proxy metoda ❶ określa funkcję żądania za pomocą znanych słów języka angielskiego, takich jak GET, POST i HEAD — identycznie jak w normalnym żądaniu HTTP. Ścieżka ❷ określa zasób będący przedmiotem żądania, w tym przypadku jest to ścieżka absolutna. Ścieżka ta może mieć także formę absolutnego URL-a i wówczas serwer proxy może ustanowić nowe połączenie z lokalizacją docelową, przekierowując do niej cały ruch i zwracając klientowi żądany przez niego zasób. Co więcej, serwer proxy może w ograniczonym zakresie manipulować tym ruchem, na przykład dodając funkcję uwierzytelniania, ukrywając serwery w wersji 1.0 przed klientami w wersji 1.1 czy też kompresując przesyłane dane. Ta elastyczność niesie jednak ze sobą pewne koszty, związane ze zdolnością przetwarzania ruchu HTTP. Spójrzmy na poniższe żądanie pobrania pliku ze zdalnego serwera za pośrednictwem proxy:

```
GET http://www.domain.com/image.jpg HTTP/1.1
```

Uważny Czytelnik z pewnością się domyśla, jakie komplikacje kryją się za tym żądaniem. Ponieważ proxy musi mieć zdolność obsługi ruchu HTTP — co z połączeniami szyfrowanymi (HTTPS), czyli transportem HTTP poprzez szyfrowane po-

łączenia TLS, czyli koniecznością samodzielnego szyfrowania/rozszyfrowywania danych? W normalnym środowisku jest przecież mało prawdopodobne, że klient HTTP zaufa pierwszemu z brzegu certyfikatowi dostarczonemu przez serwer, poza tym TLS celowo został zaprojektowany w taki sposób, by jego użycie w ramach ataku „z człowiekiem pośrodku” było wręcz niemożliwe. Na szczęście twórcy HTTP przewidzieli ten problem i w dokumencie RFC 2817 opisali dwa jego rozwiązania. Po pierwsze, połączenie HTTP może zostać dynamicznie wzbogacone o funkcję szyfrowania (pomiń w tym miejscu szczegóły). Po drugie — i ważniejsze — za pomocą nowego żądania CONNECT możliwe jest tworzenie transparentnych, tunelowanych połączeń poprzez proxy HTTP. I tak, chcąc nawiązać bezpieczne połączenie poprzez proxy za pomocą przeglądarki WWW, należy wpisać w pasku adresu żądanie

```
CONNECT www.domain.com:443 HTTP/1.1
```

Jeżeli proxy zaakceptuje to żądanie, powinno nawiązać nowe połączenie TCP z serwerem i (po pomyślnym jego nawiązaniu) zwrócić przeglądarce odpowiedź

```
HTTP/1.1 200 Connection Established
```

Nowe połączenie z proxy staje się tym samym transparentne i przeglądarka jest w stanie ustanawiać negocjowane połączenia TLS z serwerem, nie przejmując się faktem, że na drodze do serwera znajduje się proxy. Warto w tym miejscu zauważyć, że mało prawdopodobne jest weryfikowanie przez proxy, czy w bieżącym połączeniu rzeczywiście używany jest protokół właśnie TLS, a nie jakiś inny; niektóre aplikacje próbują nadużywać tej okoliczności, tunelując swe własne protokoły binarne poprzez proxy HTTP. Z tego względu powszechnie jest poszukiwanie takich wdrożeń proxy HTTP, które umożliwiają ograniczenie dozwolonych numerów portów do niezbędnego minimum.

Prosta implementacja

Biblioteka Canape Core zawiera prostą implementację proxy HTTP. Niestety implementacja ta nie obsługuje metody CONNECT do tworzenia transparentnych tuneli, lecz nie ma to znaczenia w prezentowanym przykładzie. Kod widoczny na listingu 2.8 należy umieścić w pliku skrypcowym C#, zamieniając symbol `LOCALPORT` na lokalny port TCP, na którym chcemy nasłuchiwać połączeń.

Listing 2.8. Prosty przykład proxy przekazującego HTTP

```
HttpProxy.csx // HttpProxy.csx – proste proxy HTTP
// Udostępnij metody ReadLine, WriteLine i WritePackets na poziomie globalnym
using static System.Console;
using static CANAPE.Cli.ConsoleUtils;
```

```

// Tworzenie szablonu proxy
❶ var template = new HttpProxyTemplate();
template.LocalPort = ❷ LOCALPORT;

// Tworzenie i uruchomienie instancji proxy
var service = template.Create();
service.Start();

WriteLine("Utworzono {0}", service);
WriteLine("Naciśnij Enter, by zakończyć...");
ReadLine();
service.Stop();

// Zrzut pakietów
var packets = service.Packets;
WriteLine("Przechwycono {0} pakietów:", packets.Count);
WritePackets(packets);

```

Jak łatwo zauważyć, powyższy kod stworzony został według tego samego wzorca co przykładowe kody wcześniej prezentowanych proxy.

Przekierowywanie ruchu do proxy

Podobnie jak w przypadku proxy SOCKS, następuje odwołanie do portu aplikacji. Rzadko się zdarza, by aplikacja używająca protokołu HTTP nie oferowała w jakimś zakresie opcji konfigurowania proxy. Jeśli jednak się tak zdarzy, trzeba sięgnąć do konfiguracji proxy na poziomie systemu. Przeprowadza się ją w tym samym miejscu co konfigurację proxy SOCKS, na przykład w Windows dostęp do odpowiednich opcji uzyskuje się w panelu sterowania w sekcji *Opcje internetowe/ Połączenia/Ustawienia sieci LAN* (w Windows 10 są to ustawienia *Sieć i Internet/ Serwer proxy*).

W systemach uniksowych wiele programów uruchamianych z wiersza poleceń, takich jak curl, wget czy apt, realizuje konfigurowanie proxy za pomocą zmiennych środowiskowych: URL identyfikujący serwer proxy (na przykład `http://localhost:3128`) jest treścią zmiennej `http_proxy` dla zwykłych połączeń i `https_proxy` dla połączeń bezpiecznych. Niektóre implementacje oferują obsługę specjalnych postaci URL-i, na przykład `sock4://` jest prefiksem URL-a identyfikującego proxy SOCKS w wersji 4.

Zalety i wady proxy przekazującego

Podstawową zaletą proxy przekazującego jest łatwość konfigurowania aplikacji. Wszystko, co trzeba zrobić w celu dodania do aplikacji obsługi proxy, to zmiana absolutnej ścieżki w wierszu żądania na absolutny URL nasłuchującego serwera proxy. Ponadto rzadko się zdarza, by aplikacja używająca HTTP jako protokołu transportowego nie oferowała w swoich ustawieniach konfigurowania serwerów proxy w jakimś zakresie.

Konieczność zaimplementowania kompletnego parsera w celu obsługi wielu osobliwości protokołu HTTP czyni konstrukcję proxy przekazującego znacząco skomplikowaną. Ta komplikacja może być źródłem problemów z efektywnością przetwarzania, a w najgorszym przypadku może przyczyniać się do powstawania luk w zabezpieczeniach. Ponadto dodanie proxy jako lokalizacji docelowej w ramach protokołu sprawia, że unowocześnianie istniejących aplikacji pod kątem obsługi proxy HTTP musi z reguły odbywać się przy użyciu zewnętrznych technik i narzędzi, co jest zabiegiem trudnym i nie zawsze pewnym. Częściowym antidotum na tę złożoność jest konwersja połączeń z użyciem metody CONNECT (która notabene funkcjonuje poprawnie także dla połączeń nieszyfrowanych).

Ze względu na złożoność zarządzania połączeniami HTTP 1.1 powszechną praktyką w projektowaniu proxy jest rozłączanie klientów po realizacji każdego żądania bądź *downgrade* połączeń do wersji 1.0, gdzie klient z definicji jest odłączany od serwera po odebraniu wszystkich danych w ramach pojedynczej odpowiedzi. Takie rozłączanie może być jednak problematyczne z perspektywy protokołów wysokopoziomowych spodziewających się wersji HTTP 1.1, jest także przeszkodą w realizacji **potokowości** (ang. *pipelining*), czyli przetwarzania jednocześnie kilku żądań w locie w celu poprawienia wydajności lub utrzymania stanu połączenia.

Proxy odwrotne HTTP

Proxy odwrotne stosowane są powszechnie w środowiskach, w których wewnętrzny klient łączy się z zewnętrzną siecią. Funkcjonują jako granice bezpieczeństwa, ograniczając wychodzący ruch do wybranych typów protokołów (zapomnijmy na chwilę o implikacjach używania metody CONNECT). Czasami jednak wskazane jest użycie proxy dla połączeń przychodzących, na przykład dla równoważenia obciążeń serwerów lub w celu oddzielenia serwera od świata zewnętrznego poprzez uniknięcie jego bezpośredniej ekspozycji. W ten jednak sposób traci się kontrolę nad klientem — który nawet nie zdaje sobie sprawy z faktu, że połączony jest z proxy, nie z serwerem. I tu właśnie wkracza do gry **proxy odwrotne HTTP** (ang. *reverse HTTP proxy*).

Zamiast specyfikowania zdalnego hosta w wierszu żądania — jak w przypadku proxy przekazującego — wykorzystuje się fakt, że aplikacje klienckie zgodne z HTTP 1.1 *muszą* wysyłać nagłówek HTTP Host specyfikujący oryginalną nazwę hosta użytą jako URL w żądaniu (HTTP 1.0 wolny jest od tego wymogu, mimo to większość klientów używających tej wersji i tak wysyła wspomniany nagłówek). Z tego właśnie nagłówka wydedukować można oryginalną lokalizację serwera, nawet jeśli połączenie prowadzi do proxy chroniącego ów serwer, jak na listingu 2.9.

Listing 2.9. Przykładowe żądanie HTTP

```
GET /image.jpg HTTP/1.1
User-Agent: Super Funky HTTP Client v1.0
Host: ①www.domain.com
Accept: */*
```

Na listingu widzimy typowy nagłówek Host ❶, sprawiający, że docelowym zasobem do pobrania jest plik `http://www.domain.com/image.jpg`. Proxy odwrotne z łatwością wykorzystuje ten fakt do odtworzenia oryginalnej lokalizacji serwera docelowego. Ponownie, wobec wymogu przetwarzania nagłówków http, problematyczne staje się użycie połączenia szyfrowanego HTTPS zabezpieczonego przez TLS. Na szczęście większość implementacji TLS honoruje certyfikaty, w których podmiot określony jest z użyciem znaków blankietowych, na przykład `*.domain.com`, dzięki czemu dany certyfikat pasuje zarówno do domeny `domain.com`, jak i do wszystkich jej subdomen.

Prosta implementacja

Nie będzie zapewne niespodzianką, że biblioteka Canape Core zawiera wbudowaną implementację proxy odwrotnego. Jej kod różni się od tego z listingu 2.10 tylko jednym szczegółem, a mianowicie nazwą szablonu (`HttpReverseProxyTemplate` zamiast `HttpProxyTemplate`). Dla kompletności przedstawiam ją na listingu 2.10. Ponownie prezentowany kod należy umieścić w pliku skryptowym C#, zamieniając symbol `LOCALPORT` ❶ na numer lokalnego portu TCP, na którym chcemy nasłuchiwać połączeń. Jeżeli będzie to numer mniejszy niż 1024, w systemach uniksowych konieczne jest zalogowanie się jako `root`.

Listing 2.10. Prosty przykład proxy odwrotnego HTTP

```
ReverseHttp // ReverseHttpProxy.csx – proste proxy odwrotne HTTP
↳ Proxy.csx // Udostępnij metody ReadLine, WriteLine i WritePackets na poziomie globalnym
using static System.Console;
using static CANAPE.Cli.ConsoleUtils;

// Utworzenie szablonu proxy
var template = new HttpProxyTemplate();
template.LocalPort = ❶ LOCALPORT;

// Utworzenie i uruchomienie instancji proxy
var service = template.Create();
service.Start();

WriteLine("Utworzono {0}", service);
WriteLine("Naciśnij Enter, by zakończyć...");
ReadLine();
service.Stop();

// Zrzut pakietów
var packets = service.Packets;
WriteLine("Przechwycono {0} pakietów:", packets.Count);
WritePackets(packets);
```

Przekierowywanie ruchu do proxy

Podejście do przekierowywania ruchu do odwrotnego proxy jest podobne do tego, jakie stosuje się w przypadku proxy przekazującego. Jest jednak między nimi zasadnicza różnica: tym razem nie da się zmienić nazwy hosta docelowego, bo jest ona ustalona w nagłówku HTTP Host, widocznym na listingu 2.9. Beztroskie zignorowanie tego faktu może skończyć się zapętleniem proxy². Skoro nie można zmienić nazwy hosta docelowego, to najprostszym rozwiązaniem jest zmiana adresu IP odpowiadającego tej nazwie, poprzez dodanie stosownego wpisu w pliku *hosts*.

Gdy jednak aplikacja testowana jest na urządzeniu, które nie umożliwia wykonania tego prostego zabiegu (bo na przykład użytkownik nie posiada uprawnień do modyfikowania plików systemowych), wówczas pozostaje tylko użycie własnego serwera DNS, przeliczającego nazwę hosta na żądany adres IP — oczywiście jeśli rzeczona aplikacja pozwala na określenie adresu serwera DNS w swoich ustawieniach.

Jeśli nie można zmodyfikować ani pliku *hosts*, ani lokalizacji serwera DNS w ustawieniach aplikacji, można by pokusić się o edycję ustawień w aktualnie funkcjonującym serwerze DNS. Jeśli w ogóle byłoby to możliwe, to jednak byłoby po pierwsze żmudne, po drugie podatne na błędy (być może z daleko idącymi konsekwencjami). Pozostaje więc sięgnięcie po ostatnią deskę ratunku, jaką jest ingerencja w sam proces „przeliczania” konkretnej nazwy hosta na adres IP (technika zwana potocznie „podszywaniem się” — ang. *spoofing*). Istnieją narzędzia pozwalające tę ingerencję łatwo zrealizować — jednym z nich jest *dnsspoof*. Jeśli nie chcesz instalować kolejnego narzędzia, z pomocą przyjdzie Ci (jakżeby inaczej) biblioteka Canape Core ze swym szablonem *DnsServerTemplate*. Stosowny kod widoczny jest na listingu 2.11. Należy w nim podmienić symbole *IPV4ADDRESS* ❶, *IPV6ADDRESS* ❷ i *REVERSEDNS* ❸: ostatni z nich reprezentuje „przeliczaną” nazwę hosta, dwa pierwsze to adresy IPv4 i IPv6, jakie stanowią mają wynik przeliczenia. Ponieważ w procesie przeliczania używany jest port 53, w systemach uniksowych użytkownik musi być zalogowany jako *root*, podobnie jak w przypadku odwrotnego proxy (w Windows nie istnieje podobne ograniczenie w stosunku do portów o numerach mniejszych niż 1024).

Listing 2.11. Ingerencja w odwzorowanie nazwy hosta na adresy IP

```
DnsServer.csx // DnsServer.csx – prosty serwer DNS
// Udostępnij metody ReadLine i WriteLine na poziomie globalnym
using static System.Console;

// Utworzenie szablonu serwera DNS
var template = new DnsServerTemplate();
```

² Zapętleniem proxy (ang. *proxy loop*) nazywamy sytuację, w której proxy wciąż generuje nowe połączenia z samym sobą, wpadając w nieskończoną rekurencję. Taki stan rzeczy doprowadzić może tylko do awaryjnego zakończenia aplikacji spowodowanego wyczerpaniem dostępnych zasobów.

```
// Zdefiniowanie argumentu i wyniku procesu przeliczania
template.ResponseAddress = ❶ "IPV4ADDRESS";
template.ResponseAddress6 = ❷ "IPV6ADDRESS";
template.ReverseDns = ❸ "REVERSEDNS";

// Utworzenie i uruchomienie instancji serwera DNS
var service = template.Create();
service.Start();

WriteLine("Utworzono {0}", service);
WriteLine("Naciśnij Enter, by zakończyć...");
ReadLine();
service.Stop();
```

Po poprawnie zrealizowanej „podmianie” uzyskiwania adresu IP za pomocą jednej z opisanych metod ruchu sieciowy powinien być kierowany do proxy zgodnie z oczekiwaniami.

Zalety i wady proxy odwrotnego HTTP

Podstawową zaletą proxy odwrotnego HTTP jest, w przeciwieństwie do proxy przekazującego, brak konieczności konfigurowania aplikacji klienckiej pod kątem obsługi proxy. Cecha ta okazuje się szczególnie użyteczna w sytuacji, gdy niemożliwe jest sprawowanie bezpośredniej kontroli nad konfiguracją aplikacji bądź konfiguracja ta jest ustalona a priori, bądź nie daje się łatwo modyfikować. Ponadto, jeżeli przekierowywanie ruchu do proxy odwrotnego dotyczyć ma oryginalnych połączeń TCP, za cenę niewielkiego wysiłku można skonfigurować proxy do obsługi wielu hostów docelowych.

Podobnie jak w przypadku proxy przekazującego, podstawową trudnością w konstrukcji proxy odwrotnego jest konieczność zaimplementowania pełnego parsera żądań HTTP i obsługi rozmaitych osobliwości tego protokołu.

Podsumowanie

W tym rozdziale opisałem dwie kategorie technik przechwytywania ruchu sieciowego: techniki biernie i techniki czynne. Omówiłem rozmaite sytuacje, w których jedna z tych kategorii okazuje się bardziej użyteczna od drugiej: jeśli celem przechwytywania jest jedynie obserwacja i analiza przesyłanych danych, wystarczające okazują się techniki biernie, w przeciwnym razie konieczne jest sięgnięcie po techniki czynne, oznaczające ingerencję w przepływ danych. Treść następujących rozdziałów jest dowodem na to, że techniki czynne mają przewagę nad biernymi pod względem skuteczności analizy protokołów i eksploatawania ich osobliwości. Jeżeli analizowana aplikacja pozwala na określenie serwera proxy w swych ustawieniach, to najłatwiejszą w implementacji techniką czynnego przechwytywania jest proxy SOCKS.

Skorowidz

A

ABI, *Patrz:* interfejs binarny aplikacji
Address Space Layout Randomization,
Patrz: rejestr ASLR
Adleman Leonard, 204
adres

IP, 28, 79, 104

MAC, 29, 104

translacja, *Patrz:* NAT

algorytm, *Patrz też:* szyfr

CRC, 209

DES, *Patrz:* szyfr DES

DH, *Patrz:* algorytm Diffiego-Hellmana

Diffiego-Hellmana, 206, 207

DSA, 209, 213

haszowania, *Patrz:* haszowanie, algorytm

MD, algorytm SHA, algorytm CRC

koszt obliczeniowy, 279, 280

kryptograficzny, 188, 189, *Patrz też:*

kryptografia, szyfr, szyfrowanie

luka, 189

MD, 208

MD5, 173, 210, 212

podpisywania komunikatów,

Patrz: podpisywanie, haszowanie

Rijndael, 195

rozpoznawanie, 250

RSA, 204, 209, 213

dopełnianie, 206

wady, 205, 206

SHA, 208

SHA-1, 210

SHA-3, 212

sortowania bąbelkowego, 279, 280

szyfrowania, *Patrz:* szyfrowanie, szyfr

tylne drzwi, 189

zarządzania kluczami, 194

analiza częstotliwościowa, 190

aplikacja

.NET, 237

Java, 242

kliencka, 38, 110, 111

monitorowanie, 38, 39, 41, 42

sieciowa, 25

SuperFunkyChat, 109

webowa, 345, 347

Wireshark, 33, 34, 35, 36

złośliwa, 30

Application Binary Interface, *Patrz:* interfejs

binarny aplikacji

architektura x86, 150, 151, 152, 249

pamięć, 155

zestawu instrukcji, *Patrz:* ISA

assembler, 149

NASM, 315

ASLR, 331, 332, 333

ASN.1, *Patrz:* standard ASN.1

atak, 189

aktualizacji wstecz, 220

brute force, 194, 261, 272, 281

cross-site scripting, 85

DoS, 260, 278, 280

ekspansji danych, 270

atak
man-in-the-middle, 43, 44, 207, 251,
342, 344
na dopełnienie z użyciem wyroczeni, 200
red teaming, 303
siłowy, 192
wstrzykiwanie
kodu SQL, 261, 284, 285
poleceń systemowych, 283
XSS, *Patrz:* atak cross-site scripting
z człowiekiem pośrodku, *Patrz:* atak
man-in-the-middle
z rozszerzeniem długości, 210, 212, 213
z wybranym szyfrogramem, 206
authentication, *Patrz:* uwierzytelnianie
autoryzacja, 261

B

bajt, 62
najbardziej znaczący, 67
najmniej znaczący, 67
Base Class Library, *Patrz:* BCL
BCL, 178, 182
Berkeley Sockets Model, *Patrz:* model
gniazd berkeleyowski
bezpieczeństwo, 260, 261
pamięć, 262, 263, 266
błędy przydziału dynamicznego, 270
ekspansja danych, 270
indeksowanie tablic, 269
nadmiar całkowitoliczbowy, 267
wyczerpanie, 276, 277, 278
procesor wyczerpanie mocy, 278, 280
segregowanie luk, 292, 300, 301, 303,
309, 327
uwierzytelnianie, 271,
Patrz: uwierzytelnianie
wstrzykiwanie
kodu SQL, 261, 284, 285
poleceń systemowych, 283
zasoby, 273, 275, 276, 277, 278, 280
biblioteka, 150, 171, 172
Canape Core, 45, 50, 55, 58, 343
ctypes, 244, 246
Java Runtime, 50

LibPCAP, 341
ładowana dynamicznie, 244, 245
podstawowa klas, *Patrz:* BCL
Scapy, 349
struct, 121
Sulley, 350
Winsock, 160
Binary Large Object, *Patrz:* blob
bit, 62
blob, 325
blok dopełniający, 212
blokada usług, *Patrz:* atak:DoS
błąd
indeksowania tablicy, 269
stronicowania, 302
ucieczka z piaskownicy, 12
wyszukiwanie, 292, 293, 294, 295, 296,
297, 298, 299, 300, 301, 303
brama, 92
domyślna, 29, 95
bufor
niedopełnienie, 336
przepełnienie, 263, 266, 298, 299, 332
na stercie, 298, 300, 301, 303, 305, 312
na stosie, 297, 303, 304, 335
wsteczne, 336
przydzielany dynamicznie, 266, 276,
277, 278
rozmiar predefiniowany, 263

C

CA, *Patrz:* urząd certyfikacji
CBC, 197, 200
CCA, *Patrz:* atak z wybranym szyfrogramem
Certificate Authority, *Patrz:* urząd certyfikacji
certificate pinning, *Patrz:* certyfikat
przypinanie
Certificate Revocation List, *Patrz:* certyfikat
unieważniony
certyfikat
hierarchia, 214
podmiana, 252, 253, 255
przypinanie, 221, 222
samopodpisywany, 214
unieważniony, 216

- weryfikowanie, 215
- wiarygodność, 213, 214, 216, 221
- X.509, 213
 - parametr Basic Constraints, 215
 - parametr Key Usage, 216
- chosen ciphertext attack, *Patrz:* atak z wybranym szyfrogramem
- ciasteczko, 334
- CIL, 178, 179
- Cipher Blocks Chaining, *Patrz:* CBC
- cipher text, *Patrz:* szyfrogram
- CLR, 178
- Common Intermediate Language, *Patrz:* CIL
- Common Language Runtime, *Patrz:* CLR
- control flow, *Patrz:* przepływ sterowania
- cookie, *Patrz:* ciasteczko
- CRL, *Patrz:* certyfikat unieważniony
- czas, 76, 83

D

- dane
 - binarne, 62, 63
 - kodowanie, 86
 - o zmiennej długości, 72
 - dopełnianie, 75
 - enkapsulacja, 25, 27, 28
 - integralność, 187
 - mutowanie, 291, 292
 - numeryczne, 62, 82
 - o długości implikowanej, 74
 - o zmiennej długości, 83
 - ochrona, 187
 - parsowanie, 25
 - prefiksowanie, 74
 - przechwytywanie, *Patrz:* przechwytywanie
 - szybkość transferu, 64
 - szyfrowanie, *Patrz:* szyfrowanie
 - tekstowe, 68
 - trasowanie, 28, 29
 - typ, 125, 126
 - z ogranicznikiem, 72, 73, 305
- data, 76, 83
- Data Execution Prevention, *Patrz:* DEP

- deasemblacja, 150, 164, 168, 175, 295, 351, 352
 - interaktywna, 164, 165, 167, 169
- deassembler, 353
- debugger, 147, 175, 293, 294
 - CDB, 293, 294, 296
 - GDB, 293, 294, 296, 317
 - informacja symboliczna, 169, 170
 - LLDB, 293, 294, 296
- default gateway, *Patrz:* brama domyślna
- dekompilacja, 150, 352
- dekompilator, 353, 354
- DEP, 327, 328, 331
 - omijanie, 329
- deszyfracja, 188
- Diffie Whitfield, 206
- DNAT, *Patrz:* NAT odmiana docelowa
- dopełnianie, *Patrz:* szyfr blokowy dopełnianie
- downgrade attack, *Patrz:* atak aktualizacji wstecz
- dysektor, 127
 - tworzenie, 127, 128, 130, 131, 132

E

- ECB, 197
- elektroniczna książka kodowa, *Patrz:* ECB
- endianowość, 66, 67, 68
- exploit, 335, 349
 - aktywacja, 326
 - Metasploit, *Patrz:* Metasploit
 - tworzenie, 303, 304, 305, 325, 348

F

- flaga, 66, 154
- footer, 25
- fragmentacja, 79
- funkcja, 162
 - eval, 85
 - haszująca, 208
 - atak, 210, 212, 213
 - jednokierunkowa z zapadką, 204
 - parametryzowana, 246, 247
 - z parametrami strukturalnymi, 248
- fuzz testing, *Patrz:* testowanie fazyjne

fuzzer

losowy strumień danych, 290

mutacyjny, 291, 292

G

Galois Counter Mode, *Patrz:* GCM

GCM, 197

generator liczb pseudolosowych, 192

generowanie ruchu sieciowego, 113, 349

gniazdo, 40, 41

H

hasz, 208

pośredni, 212

haszowanie, 173, 208, 210, 281

kolizja, 208, 212, 213

słabe, 261

funkcja, 280

header, *Patrz:* nagłówek

Hellman Martin, 206

HMAC, *Patrz:* kod uwierzytelniania

komunikatów haszowany

hub, 34

I

Instruction Set Architecture, *Patrz:* ISA

instrukcja, 151, 152

architektura zestawu, *Patrz:* ISA

kolejność wykonywania, 156

operand, *Patrz:* parametr

skoku

bezwarunkowego, 157

warunkowego, 157

interfejs binarny aplikacji, 162

Internet Protocol Stack, *Patrz:* protokół stos

Internet Protocol Suite, *Patrz:* IPS

inżynieria wsteczna, 142, 147, 150, 177, 351, 352

dynamiczna, 147, 174

statyczna, 147, 164

Windows, 148

IPS, 23, 25, 27, 29

ISA, 151

J

jednostka danych protokołu, *Patrz:* PDU

język

asemblera, *Patrz:* asembler

bezpieczny dla pamięci, 262

C, 264, 282

C#, 148, 262

C/C++, 148, 262

interpretowany, *Patrz:* język skryptowy

Java, 148, 262

Lua, 127, 128, 129, 131

Python, 121, 149, 244, 262

Ruby, 149, 262

skryptowy, 149

SQL, 284

wspólny pośredni, *Patrz:* CIL

XML, 85, 86

zagrożający pamięci, 262, 282

JSON, 85

K

Kali Linux, 349

kanarek pamięciowy, 334, 335, 336

kernel mode, *Patrz:* tryb jądra

klienta uwierzytelnienie, *Patrz:*

uwierzytelnienie

klucz, 188

jednorazowy, 191, 192

negocjowanie, 206

prywatny, 203

publiczny, 203, 213

sesji, 205

współdzielony, 206, 207, 210

kod, 188, *Patrz też:* szyfr

bajtowy, 178

cykliczny nadmiarowy, *Patrz:* algorytm

CRC

deasemblacja, 147, *Patrz:* deasemblacja

maszynowy, 148

obfuskacja, *Patrz:* kod zaciemnianie

pośredni, 177, 178

powłoki, 314, 317, 318, 325

odwrotnej, 326

- uwierzytelniania komunikatów, 210
 - haszowany, 212
- wykonywanie zdalne, 260
- zaciemnianie, 184, 189, 236
- zarządzany, 178, 236
- źródłowy, 148
- kodowanie
 - Base64, 88, 89
 - procentowe, 87
 - szesnastkowe, 87
- kompilacja, 149, 168
 - Javy, 182, 183
- komunikatu uwierzytelnianie, 210
- konsolidacja, 150
- konwencja uzupełnienia do dwóch, 63, 64
- konwersja binarna, 121
- kryptoanaliza, 188
- kryptografia, 172, 180
 - asymetryczna, 192, 202, 209, 213
 - losowość, 191
 - podpisywanie, *Patrz:* podpisywanie, haszowanie
 - symetryczna, 192, 202
 - szyfrowanie, *Patrz:* szyfrowanie z kluczem publicznym, *Patrz:* kryptografia asymetryczna

L

- liczba
 - całkowita, 62, 66, 82
 - bez znaku, 63
 - o zmiennej długości, 64
 - ze znakiem, 63
 - dziesiętna, 62, 82
 - losowa, 191
 - nadmiar całkowitoliczbowy, 267
 - pierwsza, 204
 - pseudolosowa, 192, 290
 - zmiennopozycyjna, 65
- local loopback, *Patrz:* pętla zwrotna lokalna
- localhost, *Patrz:* pętla zwrotna lokalna
- logarytm dyskretny, 206
- LSB, *Patrz:* bit najmniej znaczący

Ł

- ładunek użyteczny, 25, 26
- łańcuch
 - poręczeń, 213
 - tekstowy, 172, 264
 - makrosymbole, 282, 283
 - wydawniczy, 215
 - zaufania, 214

M

- MAC, *Patrz:* kod uwierzytelniania komunikatów
- malware, *Patrz:* aplikacja złośliwa
- mantysa, 65
- maskarada, 97
- maszyna wirtualna, 110, 178
- metadane, 178, 236
- Metasploit, 325
- Mitigation Bypass Bounty, 12
- model gniazd berkeleyowski, 38, 159, 160, 161
- monitor procesów
 - Linux, 38, 39
 - Windows, 41, 42
- MSB, *Patrz:* bit najbardziej znaczący
- multipleksowanie, 77, 78
 - kanal, 78

N

- nadmiar stałopozycyjny, 65
- nagłówek, 25
- NAT, 97
 - odmiana
 - docelowa, 97, 99, 100
 - źródłowa, 97, 98
- network, *Patrz:* sieć komputerowa
- Network Address Translation, *Patrz:* NAT
- network protocol, *Patrz:* protokół
- No Execute, *Patrz:* NX
- node, *Patrz:* węzeł
- notacja
 - dużego O, 280
 - JSON, *Patrz:* JSON

notacja składniowa abstrakcyjna numer 1,
Patrz: standard ASN.1
NX, 327

O

OAEP, 206
odwołanie, 150
ogranicznik, *Patrz:* separator
oktet, *Patrz:* bajt
one-time pad encryption, *Patrz:* szyfrowanie
z kluczem jednorazowym

P

padding, *Patrz:* szyfr blokowy dopełnianie,
Patrz: dane dopełnianie
Padding Oracle Attack, *Patrz:* atak
na dopełnienie z użyciem wyroczeni
pakiet dSYM, 170
pamięć
adresowanie relatywne, 322
częściowe nadpisywanie, 333
kanarek, *Patrz:* kanarek pamięciowy
podział na podpule, 311
randomizacja układu, *Patrz:* rejestr:
subalokacja, 308, 309
wyczerpanie, 276, 277, 278
zwalnianie, 300, 309
parametr, 151, 162, 167
payload, *Patrz:* ładunek użyteczny
P-Box, *Patrz:* skrzynka permutacyjna
PDU, 25, *Patrz:* protokół jednostka danych
pętla zwrotna lokalna, 34
PKI, *Patrz:* klucz publiczny infrastruktura
plaintext, *Patrz:* tekst jawny
plik
.PDB, 169
wykonywalny, 157, 158, 177
sekcja pamięci, 158
uruchamianie, 159
zapis
nieuprawniony, 311
standardowy, 313
uprzywilejowany, 313

podklucz, 194
podpisywanie, 188, 208
asymetryczne, 209
podpula, 311
polecenie, 111
connect, 39, 40
port
docelowy, 26
numer, 26
przekierowywanie, 44, 45, 46, 47, 48, 56,
136, 137, 138
źródłowy, 26
port forwarding, *Patrz:* port
przekierowywanie
półbajt, *Patrz:* tetrada
procedura, 162
proces, 159
debugowanie, 293
potomny, 293
tworzenie, 159
uruchamianie, 318
procesor wyczerpanie mocy, 278, 280
program, *Patrz też:* aplikacja
Address Sanitizer, *Patrz:* program ASan
AFL, 348
ASan, 300
Burp Suite, 346, 349
Canape, 342
DNSSMasq, 350
Dotfuscator, 184
DTrace, 39
Ettercap, 102, 350
Hopper, 353
HPing, 344
IDA, 164, 165, 167, 169, 170, 172, 352
ILSpy, 353
Java Decompiler, 351
Mallory, 344
Metasploit, 349
Microsoft Message Analyzer, 340
Mitmproxy, 347
Netcat, 226, 228, 229, 344
NMap, 345, 349
objdump, 164
Page Heap, 302

- PEiD, 174
- ProGuard, 184
- Reflector, 179, 354
- strace, 38
- TCPDump, 341
- tshark, 226, 228
- Wireshark, 112, 127, 226, 341, 349
 - wtyczka, 127, 129
- Wreshark, 114, 115, 116, 117
- ZAP, 347
- programowanie zorientowane na powroty, *Patrz:* ROP
- promiscuous mode, *Patrz:* tryb nasłuchiwania
- Protocol Data Unit, *Patrz:* PDU
- protokół, 22
 - analiza, 29, 109, 119, 120, 121, 123, 124, 125, 126, 142, 344
 - przy użyciu proxy, 138, 139, 140
 - anatomizator, *Patrz:* dysektor
 - ARP, 27, 29
 - infekowanie, 104, 105, 107
 - bezpieczeństwo, 64, 76, 187
 - kategorie zagrożeń, 260, 261
 - bezpoległościowy, 101
 - binarna endianowość, 66, 67, 68
 - błąd, 188
 - DHCP, 91, 101
 - DNS, 24
 - dysekcja, 121, 127, 128, 130, 131, 132, 134
 - Ethernet, 23, 27, 28
 - FTP, 48, 273
 - HTTP, 24, 30, 53, 127, 273, 345
 - dokumentacja, 54
 - IP, 23, 26, 35, 38, 92
 - IPv4, 24, 26, 48
 - IPv6, 24, 48
 - jednostka danych, 25, 79
 - MIME, 84
 - nieoparty na WWW, 53
 - NNTP, 86
 - PPP, 23
 - RMI, 53
 - RTMP, 53
 - SMTP, 24, 86
 - SNMP, 80
 - SOCKS, 48
 - SSL, 216
 - stos, 23
 - strumieniowy, 117, 229
 - suma kontrolna, *Patrz:* suma kontrolna
 - TC, 79
 - TCP, 23, 24, 26, 35
 - flagi, 66
 - pakiet, 117, 118, 119, 120
 - TCP/IP, 23
 - tekstowy, 81
 - TFTP, 273
 - TLS, 55, 216, 250
 - bezpieczeństwo, 221
 - deszyfracja, 251
 - negocjowanie wstępne, 217
 - szyfrowanie, 220
 - uwierzelnianie punktów końcowych, 218, 220
 - uzgadnianie, 216, 217
 - wymuszanie wersji, 252
 - trasowany, 92
 - UDP, 24, 26, 79, 101, 229
 - zestaw, *Patrz:* IPS
 - proxy, 44, 136, 138, 139, 140, 251
 - HTTP, 53
 - implementacja, 55
 - odwrotne, 54, 57, 58, 60
 - przekazujące, 54
 - przechwytyjące, 44
 - SOCKS, 48, 49, 50, 52, 53, 56
 - testowe, 46
 - zapętlenie, 59
 - próbka, 39
 - próbkowanie, 192
 - przechwytywanie, 33, 91, 95
 - bierne, 33, 34, 37, 46, 135, 340, 341
 - wady, 43
 - zalety, 43
 - czynne, 33, 43, 44, 45, 46, 47, 48, 49, 50, 52, 53, 135, 137, 138, 143, 342
 - konfiguracja, 112
 - pakietów, 113, 114, 115
 - widok szesnastkowy, 117, 119
 - widok tekstowy, 115

przechwytywanie
 ruchu sieciowego, 350
 filtrowanie, 226
 Netcat, 226
przeglądarka Mozilla Firefox, 50
przełącznik, 34
przepływ sterowania, 156, 306, 312
przeskok, 93
przestrzeń nazw, 178
public key infrastructure, *Patrz:* klucz publiczny infrastruktura
pulpit zdalny, 77
punkt przerywania, 175, 177, 317

R

ramka ethernetowa, 27, 28, 34
ramka stosu, 168
RDP, *Patrz:* pulpit zdalny
refleksja, 236, 238
rejestr, 152
 EAX, 153, 156, 162, 330
 EBP, 153, 156, 162, 163
 EBX, 153, 156, 162
 ECX, 153, 156, 162
 EDI, 153, 154, 156, 162, 163
 EDP, 154
 EDX, 153, 156, 162, 163
 EFLAGS, 154
 ESI, 153, 154, 156, 162, 163
 ESP, 153, 154, 156, 162, 163, 176, 330
indeksowy, 153
 procesora, 162
 selektorowy, 155
 uniwersalny, 153, 176, 295
 znaczników, 154
Remote Desktop Protocol, *Patrz:* pulpit zdalny
Return Oriented Programming, *Patrz:* ROP
reverse engineering, *Patrz:* inżynieria wsteczna
Rivest Ron, 204
ROP, 328, 329, 330, 331
 gadżet, 330
router, *Patrz:* brama
routing, *Patrz:* dane trasowanie
rozgranicznik, *Patrz:* separator

ruch sieciowy
 generowanie, *Patrz:* generowanie ruchu sieciowego
 przechwytywanie, *Patrz:* przechwytywanie ruchu sieciowego
 reprodukcja, 226, 229, 231, 234

S

sandbox escape, *Patrz:* błąd ucieczka z piaskownicy
S-Box, *Patrz:* skrzynka podstawieniowa
seed, *Patrz:* załóżek
segment, 155
separator, 83
serwer
 DNS, 46
 TCP, 160, 161
 uwierzytelnienie, 188
sesja
 klucz, *Patrz:* klucz sesji
 token, 200
Shamir Adi, 204
sieć, 21, 22
 Feistela, 194, 195
 podstawieniowo-permutacyjna, 195
 zaufania, 213
skrzynka
 permutacyjna, 195
 podstawieniowa, 195
 s-skrzynka, 173
SNAT, *Patrz:* NAT odmiana źródłowa
s-skrzynka, 173
stała magiczna, 172
standard
 ASCII, *Patrz:* znak ASCII
 ASN.1, 80
 Unicode, *Patrz:* znak Unicode
sterta
 przepełnienie bufora, *Patrz:* bufor przepełnienie na sterce
 zarządzanie, 308, 311
stopka, 25
stos, 162, 176
 analiza, 167
 ramka, *Patrz:* ramka stosu
 wywołań, 296

strona kodowa, 69
 strumień kluczowy, 202
 substitution cipher, *Patrz:* szyfr
 podstawieniowy
 suma kontrolna, 124, 125
 system operacyjny, 157, 159
 szyfr, 188, *Patrz też:* kod
 3DES, *Patrz:* szyfr DES potrójny
 AES, 193, 195, 196
 analiza częstotliwościowa, *Patrz:* analiza
 częstotliwościowa
 asymetryczny, 192, 202
 blokowy, 193, 194, 195
 dopełnianie, 199, 200
 tryb operacyjny, 196, 197
 Blowfish, 196
 Camelia, 196
 DES, 193, 194, 196, 200
 potrójny, 194, 196
 podstawieniowy, 189
 RSA, *Patrz:* algorytm RSA
 Serpent, 196
 strumieniowy, 193, 202
 symetryczny, 192, 202, *Patrz też:* szyfr
 blokowy, szyfr strumieniowy
 tablica podstawień, 190
 TDES, *Patrz:* szyfr DES potrójny
 tryb licznikowy Galois, *Patrz:* GCM
 Twofish, 196
 wiązanie bloków szyfrogramu, *Patrz:* CBC
 XOR, 190, 191, 197
 szyfrogram, 188
 szyfrowanie, 54, 142, 143, 188
 symetryczne, *Patrz:* szyfr blokowy, szyfr
 strumieniowy
 z kluczem jednorazowym, 191, 192

Ś

środowisko
 .NET, 178, 182, *Patrz:* aplikacja .NET
 Mono Project, 178
 uruchomieniowe, 178
 wspólnego języka, *Patrz:* CLR

T

tablica
 błędy w indeksowaniu, 269
 funkcji wirtualnych, *Patrz:* V-tablica
 haszowana, 280
 trasowania, 29, 93
 konfigurowanie, 101
 tekst
 jawny, 188
 blok, 193
 konwersja, 286
 łańcuch, *Patrz:* łańcuch tekstowy
 zaszyfrowany, *Patrz:* szyfrogram
 testowanie
 fazyjne, 290, 311, *Patrz też:* fuzzer
 generowanie przypadków testowych, 292
 tetrada, 62, 87
 TLV, *Patrz:* wzorzec typ – długość – wartość
 token, 83
 sesji, *Patrz:* sesja token
 translacja adresów sieciowych, *Patrz:* NAT
 trapdoor function, *Patrz:* funkcja
 jednokierunkowa z zapadką
 trasowanie, 95
 tablica, *Patrz:* tablica trasowania
 w systemach uniksowych, 96
 w Windows, 96
 tryb
 jądra, 37
 nasłuchiwanie, 34
 użytkownika, 37

U

urząd certyfikacji, 214, 215, 221
 urządzenie trasujące, *Patrz:* brama
 user mode, *Patrz:* tryb użytkownika
 uwierzytelnianie, 188, 261
 hardkodowanie, 271, 272
 nazwa użytkownika, 272
 omijanie, 261

V

V-tablica, 299, 305, 306, 307, 312
vulnerability triaging, *Patrz:* bezpieczeństwo
segregowanie luk

W

warstwa, 25
 aplikacji, 24
 bezpiecznych gniazd, 216
 internetowa, 24
 kodowania, 29, 31
 łącza danych, 23, 24
 transportowa, 24, 29, 31, 216
 treści, 29, 31
wartość
 binarna, 87
 boolowska, *Patrz:* wartość logiczna
 logiczna, 66, 82
wątek, 159
Web of Trust, *Patrz:* sieć zaufania
wektor inicjalizacyjny, 197
węzeł, 22, 27
 proxy, 33
wielozadaniowość, 159
wiersz
 poleczeń debugger, 293
 żądania, 54
WOT, *Patrz:* sieć zaufania
wyciąg, 208
wyciek informacji, 261
wykładnik, 65
wywołanie
 systemowe, 37, 38, 40, 318
 execve, 323
 exit, 319
 Linux, 318, 319
 write, 320
wzorec typ – długość – wartość, 77, 80

X

XOR-owanie, *Patrz:* szyfr XOR

Z

załączek, 192
zasoby, 311
 postać kanoniczna nazwy, 273, 274
zdarzenie, 41
znacznik, 154
 bitowy, *Patrz:* flaga
 FILETIME, 76
 potwierdzenia, 66
 synchronizacji, 66
znak
 <, 114
 >, *Patrz:* znak zachęty
 ASCII, 68, 71, 86, 286
 biały, 83
 drukowalny, 69
 kodowanie, 69, 70, *Patrz też:* kodowanie
 UCS, 70
 UTF, 70, 71
 odwzorowanie, 70
 ukośnika, 111
 Unicode, 69, 70, 286
 kodowanie, 71
 konwersja na ASCII, 286
 wielobajtowy, 69
 zachęty, 114

Ż

żądanie
 CONNECT, 55
 HTTP, 29
 proxy, 54

PROGRAM PARTNERSKI

— GRUPY HELION —



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA
Helion 

Najlepsze zabezpieczenie: spójrz na system z perspektywy hakera!

Gwałtowny rozwój i upowszechnienie technologii sieciowych sprawiły, że dziś praktycznie każdy codziennie korzysta z sieci, i to nie tylko za pomocą komputera czy telefonu. Do internetu można podłączyć również lodówki, kamery monitoringu, pralki czy telewizory. Urządzenia w sieci komputerowej komunikują się ze sobą według ustalonych protokołów, które, choć publicznie eksponowane, nie są przez użytkowników rozpoznane tak dobrze jak komponenty sprzętowe tych urządzeń. A przecież to na oprogramowanie, a nie na sprzęt ukierunkowana jest znakomita większość ataków sieciowych.


Ta książka jest praktycznym podręcznikiem analizy protokołów powszechnie wykorzystywanych w celu wykrywania tkwiących w nich luk w zabezpieczeniach. Została napisana z punktu widzenia hakera: dzięki zawartym w niej wskazówkom można samodzielnie rozpocząć analizę ruchu sieciowego i prowadzić eksperymenty z łamaniem zabezpieczeń. W książce znalazły się również szczegółowe opisy technik przechwytywania ruchu sieciowego, analizowania protokołów sieciowych oraz wykrywania i wykorzystywania ich słabych stron. Zagadnienia teoretyczne zostały tu umiejętnie połączone z czysto praktycznym podejściem do takich działań jak dysekcja protokołów, testowanie fazyjne, debugowanie i ataki prowadzące do wyczerpywania zasobów: pamięci, przestrzeni dyskowej i mocy procesorów.

W tej książce między innymi:

- ★ podstawy działania sieci i struktura protokołów sieciowych
- ★ przechwytywanie ruchu sieciowego — techniki proste i zaawansowane
- ★ odtwarzanie kodu aplikacji w procesie inżynierii wstecznej
- ★ najczęstsze problemy bezpieczeństwa protokołów sieciowych
- ★ implementacja protokołu w kodzie aplikacji i związane z tym zagrożenia
- ★ mechanizmy destrukcyjne, w tym nadpisywanie pamięci i omijanie uwierzytelnień

James Forshaw specjalizuje się w dziedzinie bezpieczeństwa użytkownika komputerów. Od kilkunastu lat analizuje protokoły sieciowe. Jest członkiem zespołu Google Project Zero, który wykrywa i zabezpiecza luki typu zero day. Zajmuje się szerokim zakresem zagadnień bezpieczeństwa: od hakowania konsoli do gier po poszukiwanie słabych miejsc w projektach systemów operacyjnych, szczególnie Microsoft Windows. Jest twórcą Canape — narzędzia do analizy protokołów sieciowych. Bierze udział w prestiżowych konferencjach poświęconych cyberbezpieczeństwu, takich jak Black Hat, CanSecWest czy Chaos Computer Congress.

Helion 

 helion.pl

 **HELION SA**
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

Sprawdź nasze szkolenia!

SZKOLENIA



AKADEMIA IT & BUSINESS

WWW.SZKOLENIA.HELION.PL

KOD KORZYŚCI
Sięgnij po więcej! ▶



ISBN 978-83-283-5390-9



9 788328 353909

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 67,00 zł