Atak na sieć okiem hakera

Wykrywanie i eksploatacja luk w zabezpieczeniach sieci

James Forshaw

no starch press

Helion

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

Tłumaczenie: Andrzej Grażyński

ISBN: 978-83-283-5390-9

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.

Spis treści

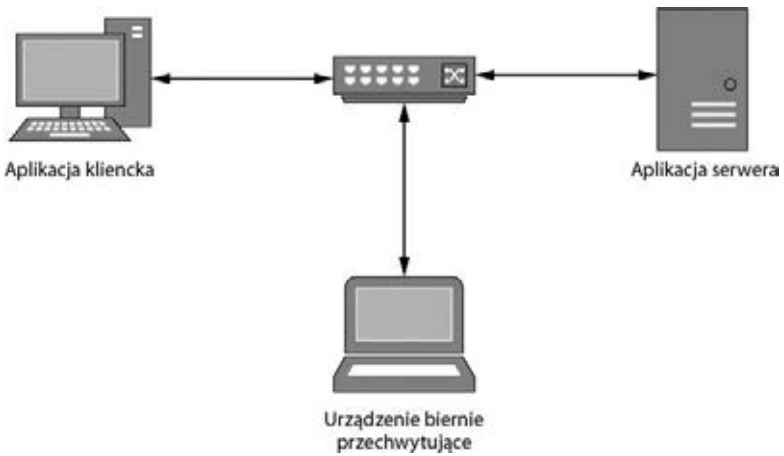
2

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* („drutu") 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.



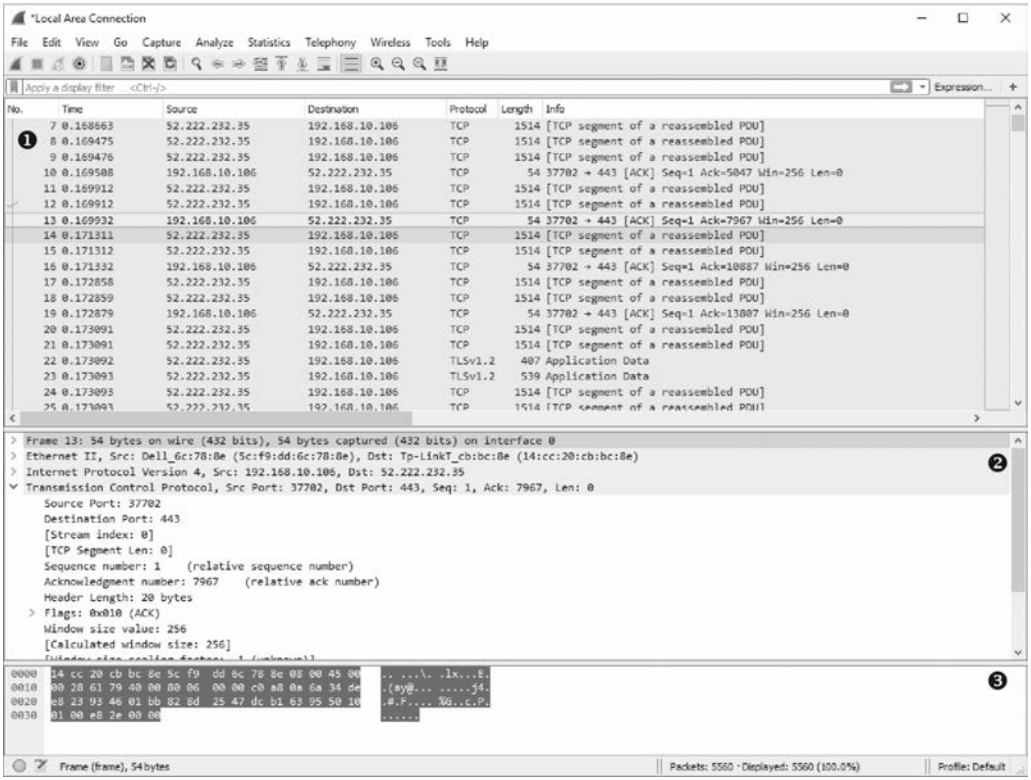
Rysunek 2.1. Przykład biernego przechwytywania ruchu sieciowego

Wireshark — podstawy

Wireshark to najpopularniejsza bodaj aplikacja do „podsłuchiwania" 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 przechwytujące 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 przechwytująca jest bardzo proste, wystarczy włączyć monitorowanie interfejsu wyjściowego lub lokalnej pętli zwrotnej (ang. *local loopback*), znanej również pod nazwą *localhost*. Podsłuchiwanie 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.

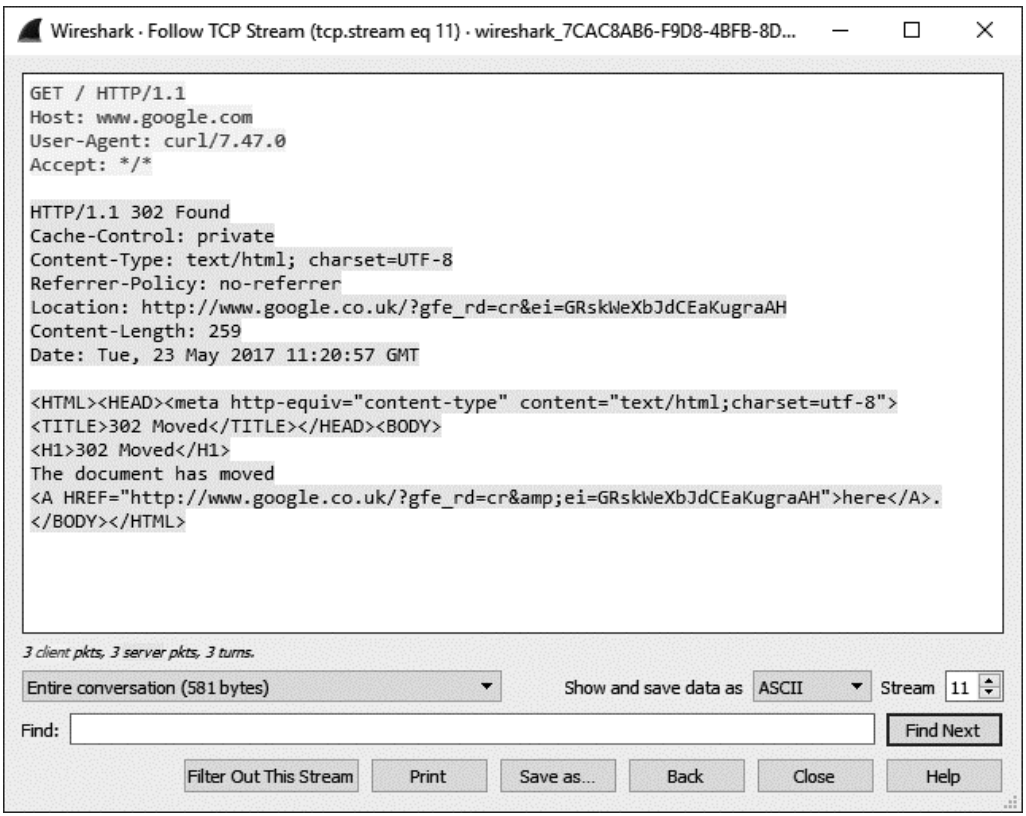


Rysunek 2.2. Domyślne okno aplikacji Wireshark

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 uszkadzania 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).

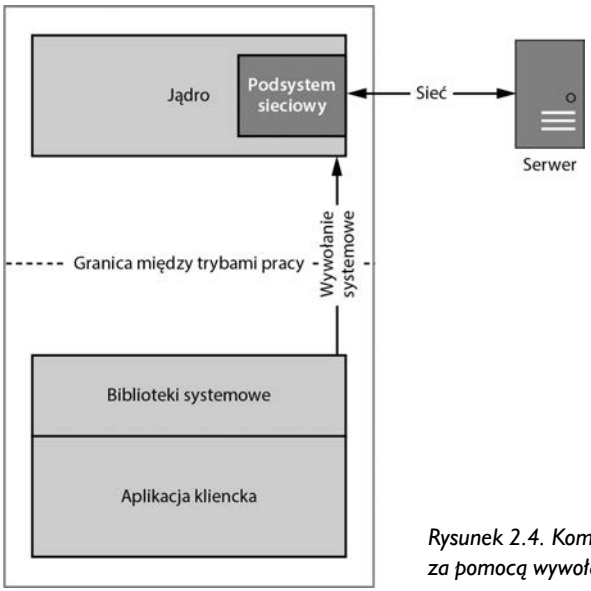
[1] Chris Sanders, *Praktyczna analiza pakietów. Wykorzystanie narzędzia Wireshark do rozwiązywania problemów związanych z siecią. Wydanie III*, Helion, Gliwice 2017.

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, obywające 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 bezpośrednie 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.

Załóżmy, 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://www.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 nagłówków 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*

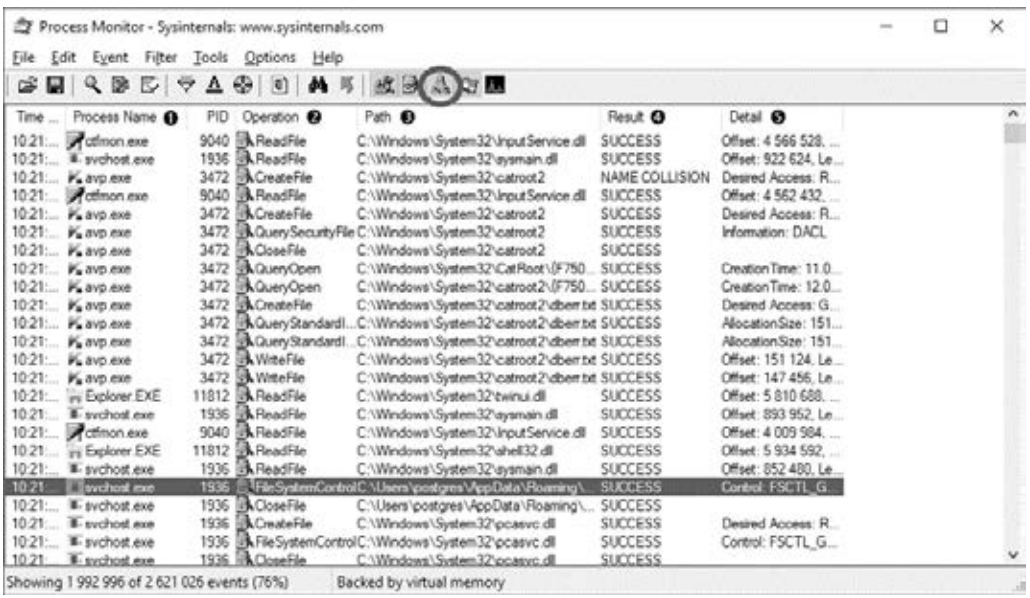
```
proces:'Google Chrome' 173.194.78.125:5222
proces:'Google Chrome' 173.194.66.95:443
proces:'Google Chrome' 217.32.28.199:80
proces:'ntpd' 17.72.148.53:123
proces:'Mail' 173.194.67.109:993
proces:'syncdefaultsd' 17.167.137.30:443
proces:'AddressBookSour' 17.172.192.30:443
```

Jak widać, dla każdego uwzględnionego wywoływania systemowego wyświetlana jest nazwa procesu wywołującego 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

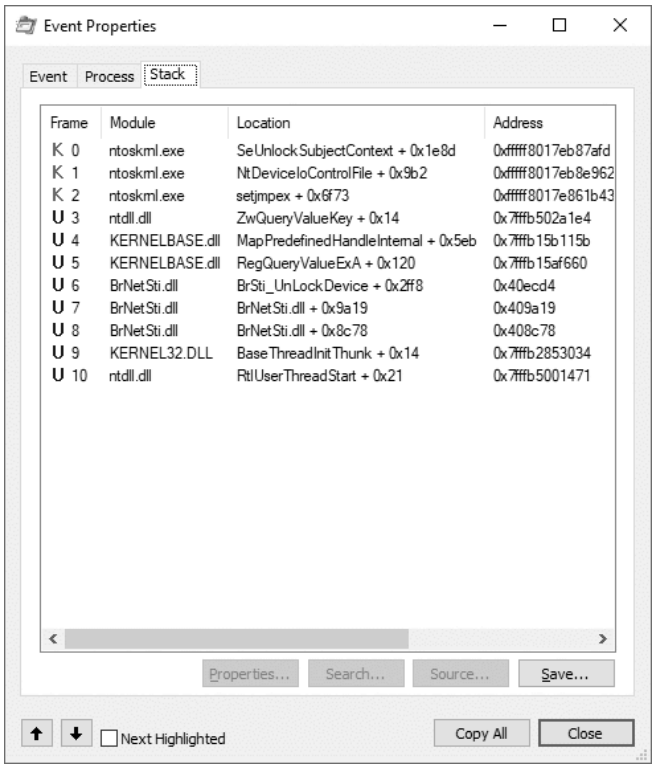
W przeciwieństwie do systemów uniksowych Microsoft Windows implementuje własny zestaw funkcji API związanych z systemem operacyjnym zamiast bezpośrednich wywołań funkcji systemowych. Dostęp do sieci odbywa się poprzez sterownik (ang. *driver*), a współpraca z gniazdami realizowana jest przez „plikowe" 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 wykonywalnego 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 obywają się one bez ingerowania w ów ruch, czyli (nazwijmy rzecz po imieniu) bez uszkadzania danych wymienianych między klientem a serwerem. Bierne przechwytywanie obywa się także bez modyfikowania adresów komunikujących się węzłów, nie wymaga też zmian w konfiguracjach komunikujących się aplikacji.

Bierne 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 podsłuchiwanych 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 bierne 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 przechwytującego. 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”

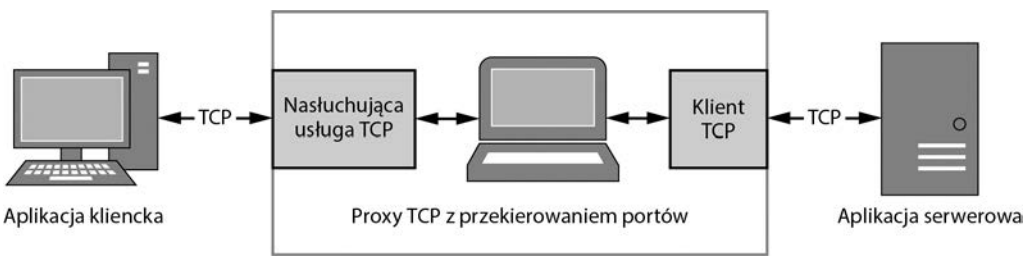
Wykonanie czynnego przechwytywania jest zadaniem znacznie trudniejszym niż bierne podsłuchiwanie pakietów, wymaga bowiem dokonania zmiany trasy przy użyciu urządzenia przechwytującego. To z kolei może powodować niezamierzone lub niepożądane efekty: jeżeli na przykład zmienimy w aplikacji adres klienta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

Proxy z przekierowaniem portów

Przekierowywanie portów (ang. *port forwarding*) to najprostszy sposób realizacji proxy przechwytującego. 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 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 skryptowym 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 ↪Proxy.csx

```
// PortFormatProxy.csx – proste proxy przekierowujące porty TCP
// 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 = ❶LOCALPORT;
template.Host = ❷"REMOTEHOST";
template.Port = ❸REMOTEPORT;

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

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

// Zrzut pakietów
var packets = service.Packets;
WriteLine("Przechwycono {0} pakietów:",
                    packets.Count);
❼ 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 ❺; 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 ❻. Raport o wszystkich przechwyconych pakietach wypisany zostaje na konsolę za pomocą metody `Write ↪Packets()` ❼.

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 rozmaitymi 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 przechwytujemy 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.

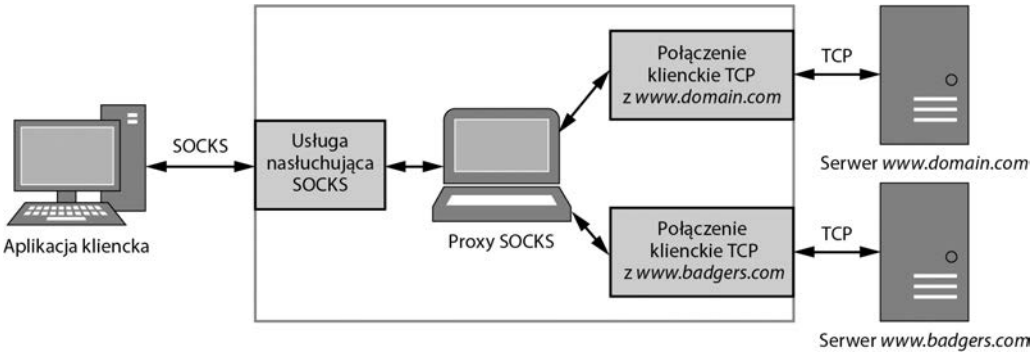
Podobnie jest w sytuacji, gdy dwa połączenia do różnych lokalizacji docelowych prowadzą do *tego samego portu*, powiedzmy 1234. Tworzymy dwie instancje proxy, po jednej dla każdej lokalizacji; obie instancje nasłuchują na porcie 1234. Gdy aplikacja kliencka zażąda połączenia z jedną z lokalizacji, jej 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 nasłuchiwania 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 przeanalizujmy 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 skryptowym 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

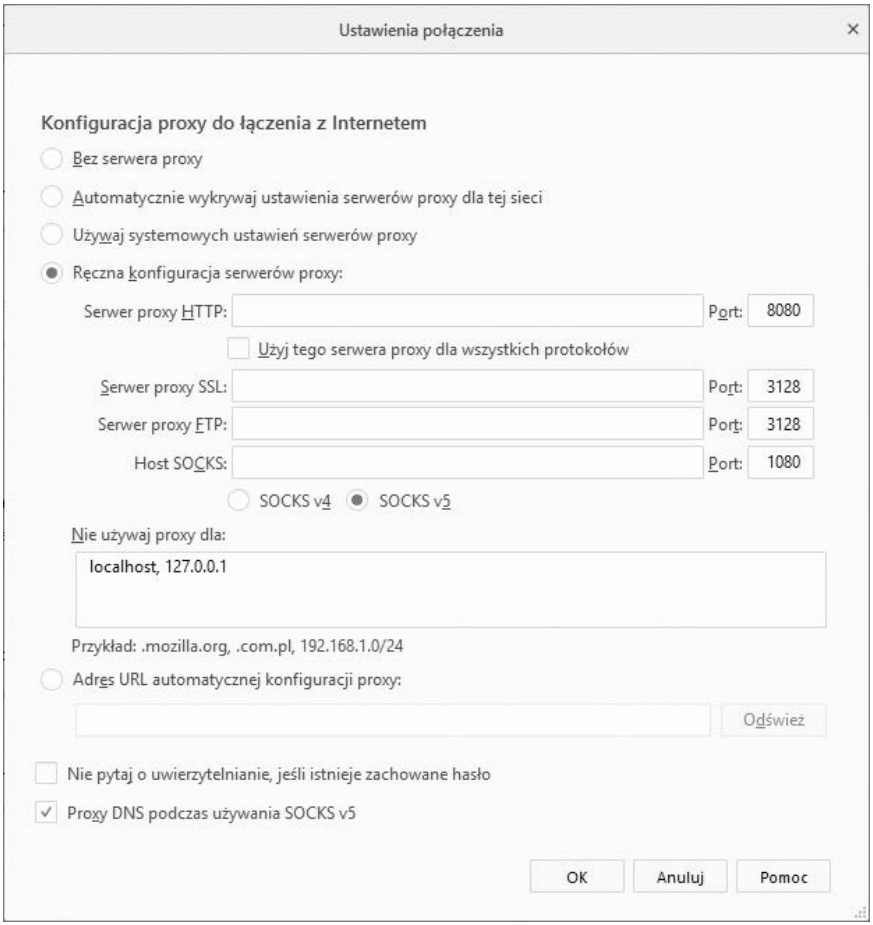
```
// SocksProxy.csx – proste proxy SOCKS
// Udostepnij metodyReadLine, WriteLine i Writepackets na poziomie globalnym
using static System.Console;
using static CANAPE.Cli.ConsoleUtils;

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

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 SOKCS, 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 ➥Client.java

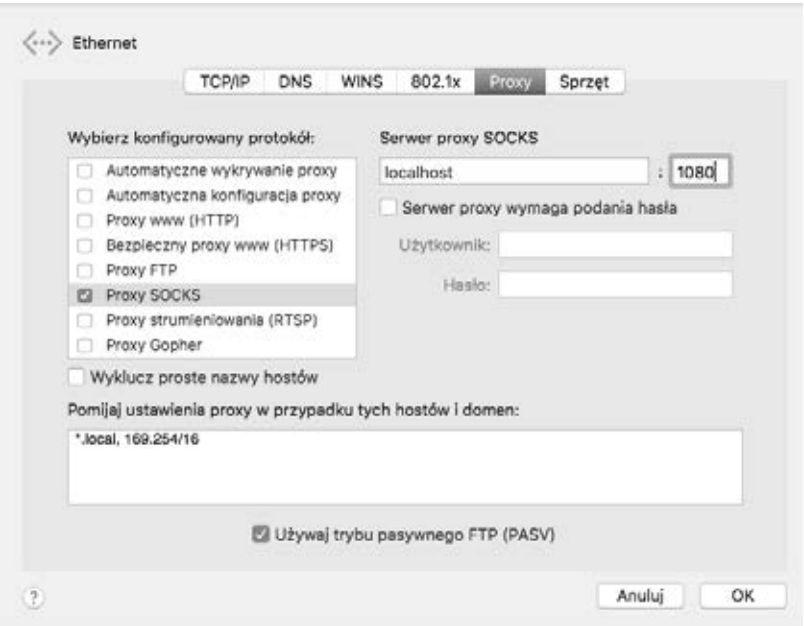
```
// SocketClient.java – prosty klient TCP socket
import java.io.PrintWriter;
import java.net.Socket;

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/Sieć/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 **Proxifier** 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.

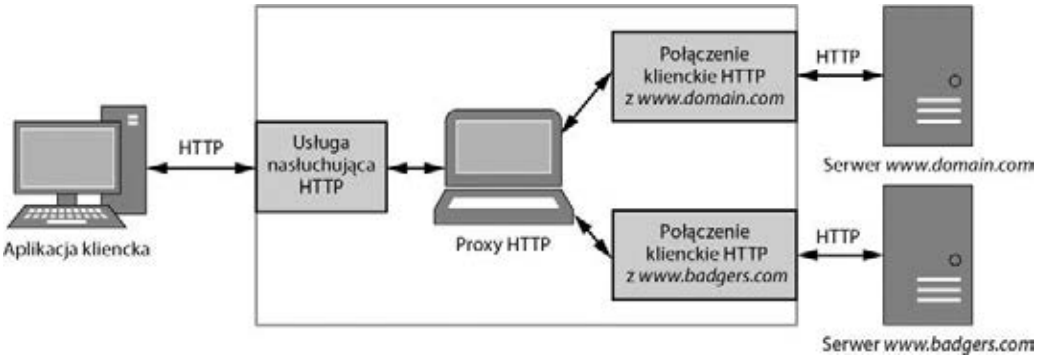
Ponadto proxy SOCKS zasadniczo zachowują tożsamość lokalizacji docelowej z punktu widzenia aplikacji klienckiej. Jeśli więc w *treści* wysyłanej przez aplikację kliencką znajduje się informacja zależna od tej tożsamości — na przykład adres IP lub numer portu — serwer otrzymuje dane, których faktycznie 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 przechwytującego 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:

```
❶GET 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 (HTTP), 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 powszechne 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 skryptowym C#, zamieniając symbol *`LOCALPORT`* ❷ 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;
```

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

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: ❶www.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↪Proxy.csx

```
// ReverseHttpProxy.csx – proste proxy odwrotne HTTP
// 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 = ❶LOCALPORT;

// 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 proxy[2]. 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 stanowić 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 ruch 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 bierne 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 bierne, w przeciwnym razie konieczne jest sięgnięcie po techniki czynne, oznaczające ingerencję w przepływ danych. Treść następnych rozdziałów jest dowodem na to, że techniki czynne mają przewagę nad biernymi pod względem skuteczności analizy protokołów i eksploatowania 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

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

V

W

X

Z

Ż

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 użytkowania 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

KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-283-5390-9
9 788328 353909
INFORMATYKA W NAJLEPSZYM WYDANIU
Cena: 67,00 zł