2
Capturing von Anwendungsverkehr

Überraschenderweise kann das Capturing, also das Erfassen nützlichen Verkehrs bei der Protokollanalyse, eine Herausforderung darstellen. Dieses Kapitel beschreibt zwei Aufzeichnungstechniken: *passives* und *aktives* Capturing. Passives Capturing interagiert nicht direkt mit dem Netzwerkverkehr. Stattdessen extrahiert es die Daten, während sie *über die Leitung laufen*, was Ihnen aus Tools wie Wireshark vertraut sein dürfte.

Sie werden sehen, dass unterschiedliche Anwendungen unterschiedliche Mechanismen (mit ihren jeweiligen Vor- und Nachteilen) verwenden, um Verkehr umzuleiten. Aktives Capturing greift in den Verkehr zwischen einer Clientanwendung und dem Server ein, was zwar sehr leistungsfähig ist, aber auch zu Komplikationen führen kann. Sie können sich aktives Capturing als eine Art Proxy, oder auch als Man-in-the-Middle-Angriff vorstellen. Sehen wir uns diese aktiven und passiven Techniken etwas genauer an.

2.1 Passives Capturing von Netzwerkverkehr

Passives Capturing ist eine relativ einfache Technik: Sie verlangt üblicherweise keine spezielle Hardware und Sie müssen auch keinen eigenen Code entwickeln. Abbildung 2–1 zeigt ein gängiges Szenario: Ein Client und ein Server kommunizieren per Ethernet über ein Netzwerk.

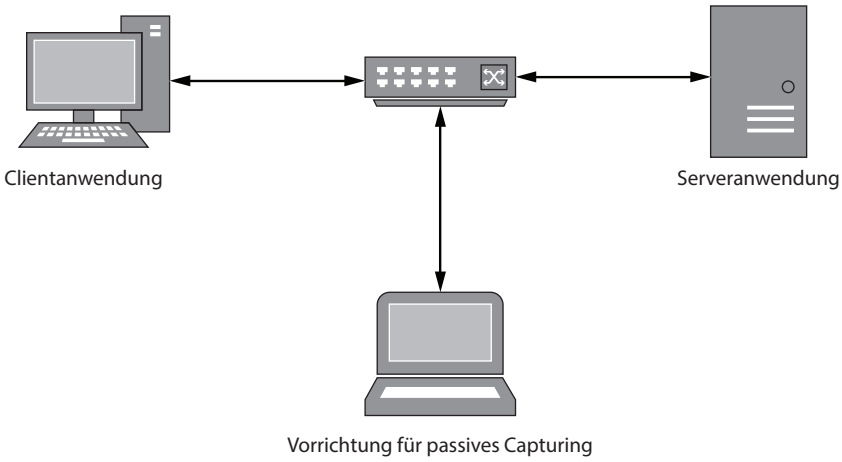

Abb. 2–1 *Passives Netzwerk-Capturing*

Passives Capturing kann entweder im Netzwerk erfolgen, indem man den laufenden Verkehr abhört, oder durch direktes Sniffing auf dem Client oder Server.

2.2 Eine kurze Einführung in Wireshark

Wireshark ist der wohl beliebteste Paket-Sniffer. Er läuft auf vielen Plattformen, ist einfach zu verwenden und hat viele Features für die Protokollanalyse an Bord. In Kapitel 5 werden Sie lernen, wie man einen sogenannten Dissector entwickelt, der Sie bei der Protokollanalyse unterstützt. Doch für den Moment wollen wir Wireshark nur einrichten und IP-Verkehr aus dem Netzwerk aufzeichnen.

Um Verkehr von einer Ethernet-Schnittstelle (kabelgebunden oder drahtlos) zu erfassen, muss sich die Capturing-Vorrichtung im »Promiskuitätsmodus« (engl. *Promiscuous Mode*) befinden. In diesem Modus empfängt und verarbeitet eine Schnittstelle jeden Ethernet-Frame, den sie sieht, selbst wenn dieser Frame nicht für diese Schnittstelle gedacht ist. Das Erfassen einer Anwendung, die auf dem gleichen Rechner läuft, ist einfach: Sie brauchen nur die ausgehende Netzwerkschnittstelle oder das lokale Loopback-Interface (besser bekannt als localhost) zu überwachen. Anderenfalls müssen Sie Netzwerk-Hardware wie einen Hub oder einen konfigurierten Switch verwenden, um sicherzustellen, dass der Verkehr an Ihre Netzwerkschnittstelle geht.

Abbildung 2–2 zeigt die Standardansicht beim Erfassen von Verkehr über eine Ethernet-Schnittstelle.

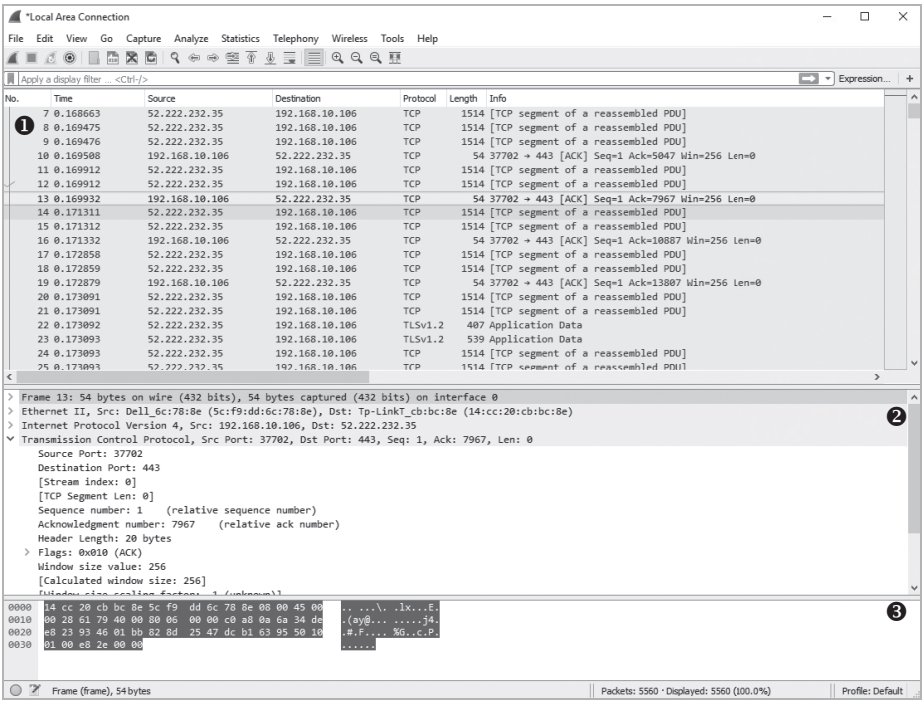

Abb. 2–2 *Standardansicht von Wireshark*

Die Hauptansicht ist in drei wichtige Bereiche unterteilt. Bereich ❶ stellt eine Zeitachse der Rohpakete dar, die im Netzwerk erfasst wurden. Sie enthält eine Liste der IP-Quell- und -Zieladressen sowie eine Zusammenfassung decodierter Protokollinformationen. Bereich ❷ enthält eine analysierte Ansicht des Pakets, untergliedert in verschiedene Protokollschichten, die dem OSI-Modell entsprechen. Bereich ❸ zeigt das abgegriffene Paket in Rohform.

Das TCP-Protokoll ist Stream-basiert und kann verlorene Pakete und beschädigte Daten wiederherstellen. Bedingt durch die Natur von Netzwerken und des IP-Protokolls gibt es keine Garantie, dass Pakete in einer bestimmten Reihenfolge empfangen werden. Die Interpretation des Zeitleistenbereichs kann daher beim Erfassen von Paketen recht schwierig sein. Glücklicherweise bietet Wireshark »Sezierer« für bekannte Protokolle an, die den gesamten Stream wiederherstellen und alle Informationen an einem Ort bündeln. Markieren Sie beispielsweise eine TCP-Verbindung in der Zeitleisten-Ansicht und wählen Sie dann **Analyze ▸ Follow TCP Stream** aus dem Hauptmenü, so erscheint ein Dialog wie in Abbildung 2–3. Für Protokolle ohne eigenen Sezierer kann Wireshark den Stream decodieren und in einer einfachen Ansicht darstellen.

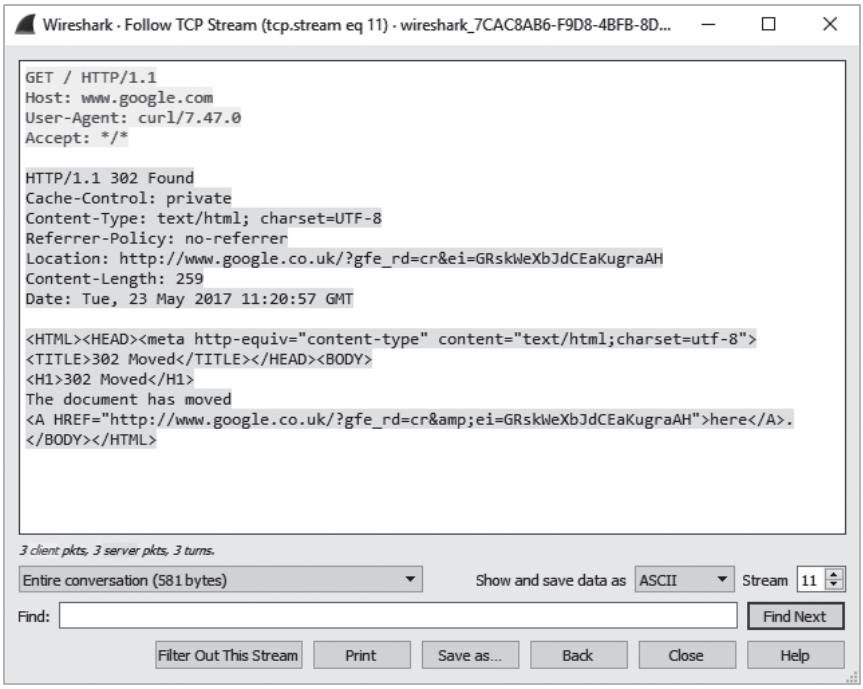

Abb. 2–3 *Einem TCP-Stream folgen*

Wireshark ist ein sehr umfangreiches Werkzeug. Alle verfügbaren Features zu behandeln geht weit über den Rahmen dieses Buches hinaus. Wenn Sie nicht mit ihm vertraut sind, sollten Sie sich eine gute Referenz besorgen, z.B. *Wireshark® 101: Einführung in die Protokollanalyse* (mitp, 2018), und die vielen nützlichen Features kennenlernen. Wireshark ist für die Analyse von anwendungsbezogenem Netzwerkverkehr unverzichtbar und unter der General Public License (GPL) kostenlos verfügbar.

2.3 Alternative passive Capturing-Techniken

Manchmal ist die Nutzung eines Paket-Sniffers nicht möglich, z.B. in den Fällen, in denen man nicht das Recht hat, Netzwerkverkehr zu erfassen. Sie könnten etwa Penetrationstests auf einem System durchführen, für das Sie keine administrativen Rechte besitzen, oder Sie könnten auf einem mobilen Gerät mit einer Shell mit nur eingeschränkten Rechten arbeiten müssen. Sie könnten auch nur sicherstellen wollen, dass nur der für die zu testende Anwendung notwendige Verkehr untersucht wird. Das ist per Paket-Sniffing nicht immer einfach, solange man Netzwerkverkehr und Zeit nicht in Beziehung setzt. In diesem Abschnitt wollen wir einige Techniken beschreiben, mit denen Netzwerkverkehr einer lokalen Anwendung ohne Paket-Sniffing-Tool extrahiert werden kann.

2.3.1 Tracing von Systemaufrufen

Viele moderne Betriebssysteme bieten zwei Ausführungsmodi an. Der *Kernel-Modus* läuft mit hohen Privilegien und enthält Code, der die Kernfunktionalität des Betriebssystems implementiert. Die alltäglichen Prozesse laufen hingegen im *User-Modus*. Der Kernel stellt dem User-Modus seine Dienste über den Export einer Reihe spezieller Systemaufrufe (siehe Abb. 2–4) zur Verfügung, die es den Nutzern erlauben, auf Dateien zuzugreifen, Prozesse zu erzeugen und – für unsere Zwecke das Wichtigste – die Verbindung mit Netzwerken herzustellen.

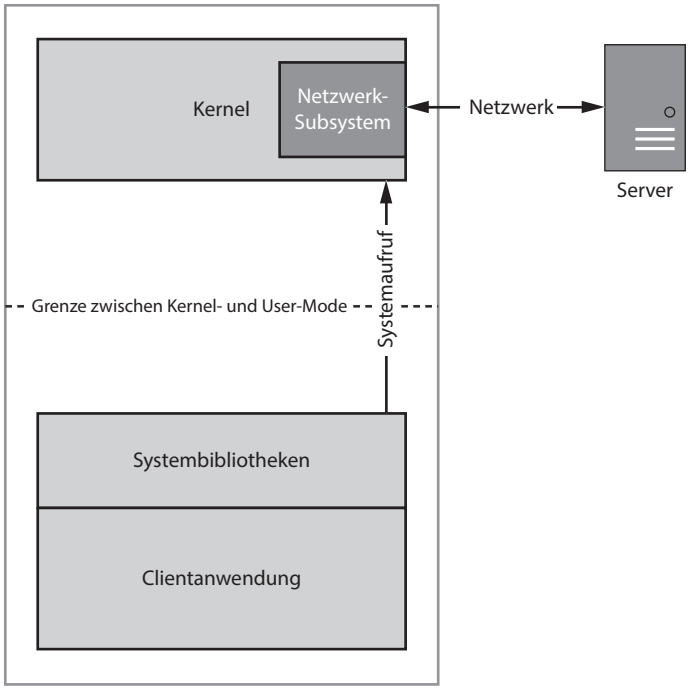

Abb. 2–4 *Nutzer-Kernel-Netzwerkkommunikation über Systemaufrufe*

Möchte eine Anwendung sich mit einem entfernten Server verbinden, stellt sie einen speziellen Systemaufruf an den Betriebssystemkern, der die Verbindung aufbaut. Die Anwendung kann dann die Netzwerkdaten lesen und schreiben. Je nachdem, auf welchem Betriebssystem Ihre Netzwerkanwendung läuft, können Sie diese Aufrufe direkt überwachen, um passiv Daten aus der Anwendung zu extrahieren.

Die meisten unixoiden Systeme implementieren Systemaufrufe basierend auf dem Berkeley-Sockets-Modell. Das ist nicht weiter überraschend, da das IP-Protokoll ursprünglich in der Berkeley Software Distribution (BSD) 4.2 eingeführt wurde. Die Socket-Implementierung ist Teil von POSIX und damit ein De-facto-

Standard. Tabelle 2–1 führt einige der wichtigsten Systemaufrufe der Berkeley-Sockets-API auf.

Name	Beschreibung
`socket`	Erzeugt einen neuen Socket-Dateideskriptor
`connect`	Verbindet einen Socket mit einer bekannten IP-Adresse und einem Port
`bind`	Bindet den Socket an eine lokal bekannte IP-Adresse und einen Port
`recv, read, recvfrom`	Empfängt Daten aus dem Netzwerk über den Socket. Die generische Funktion `read` liest aus einem Dateideskriptor, während `recv` und `recvfrom` Aufrufe der Socket-API sind.
`send, write, sendfrom`	Sendet Daten über den Socket an das Netzwerk

Tab. 2–1 *Gängige Netzwerk-Systemaufrufe unter Unix*

Wer wissen möchte, wie diese Systemaufrufe funktionieren, findet in *The TCP/IP Guide* (No Starch Press, 2005) eine ausgezeichnete Quelle. Viele Ressourcen sind auch online verfügbar und die meisten unixoiden Betriebssysteme beinhalten Handbücher, die man sich im Terminal mit einem Befehl wie `man 2 syscall_name` ansehen kann. Schauen wir uns nun an, wie man Systemaufrufe überwacht.

2.3.2 Das strace-Utility unter Linux

Unter Linux können Sie die Systemaufrufe eines Benutzerprogramms ohne besondere Rechte direkt verfolgen, solange die zu überwachende Anwendung nicht unter einem privilegierten Nutzer ausgeführt wird. Viele Linux-Distributionen liefern ein nützliches Utility namens `strace` mit, das den Großteil der Arbeit für Sie erledigt. Ist es standardmäßig nicht installiert, können Sie es über den Paketmanager Ihrer Distribution nachinstallieren oder es aus dem Quellcode kompilieren.

Um die Netzwerk-Systemaufrufe der Anwendung festzuhalten, führen Sie den folgenden Befehl aus und ersetzen Sie dabei */pfad/auf/app* durch die zu testende Anwendung und *args* durch die benötigten Parameter:

```
$ strace –e trace=network,read,write /pfad/auf/app args
```

Wir wollen beispielhaft eine Netzwerkanwendung überwachen, die ein paar Strings über das Netzwerk liest und schreibt, und uns die Ausgabe von `strace` ansehen. Listing 2–1 zeigt vier Logeinträge (wobei wir die irrelevanten Einträge der Kürze halber weggelassen haben).

```
$ strace -e trace=network,read,write customapp
--schnipp--
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
```

Listing 2–1 *Ausgabe von `strace`-Utility*

Der erste Eintrag ❶ erzeugt einen neuen TCP-Socket, der dem Handle 3 zugewiesen wird. Der nächste Eintrag ❷ zeigt den Systemaufruf `connect`, der eine TCP-Verbindung mit der IP-Adresse 192.168.10.1 an Port 5555 herstellt. Die Anwendung schreibt dann den String `Hello World!` ❸, bevor sie den String `Boo!` ❹ einliest. Die Ausgabe zeigt, dass man eine recht gute Vorstellung davon bekommt, was die Anwendung auf Ebene der Systemaufrufe macht, selbst wenn man nicht über besonders hohe Privilegien verfügt.

2.3.3 Netzwerkverbindungen mit DTrace verfolgen

DTrace ist ein leistungsfähiges Tool, das für viele unixoide Systeme verfügbar ist, darunter Solaris (wo es ursprünglich entwickelt wurde), macOS und FreeBSD. Es erlaubt das Setzen systemweiter Messpunkte für spezielle Trace-Provider, darunter auch Systemaufrufe. Sie konfigurieren DTrace über Skripte in einer Sprache mit C-ähnlicher Syntax. Weitere Details zu diesem Tool finden Sie online im DTrace-Handbuch unter *http://www.dtracebook.com/index.php/DTrace_Guide*.

Listing 2–2 zeigt ein Beispielskript, das ausgehende IP-Verbindungen mit DTrace überwacht.

traceconnect.d

```
/* traceconnect.d – Einfaches DTrace-Skript zur Überwachung eines
/* connect-Systemaufrufs */
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("process:'%s' %s:%d", execname, inet_ntop(2, &addr->sin_addr),
      ntohs(addr->sin_port));
}
```

Listing 2–2 *Einfaches DTrace-Skript zur Überwachung eines connect-Systemaufrufs*

Dieses einfache Skript überwacht den Systemaufruf `connect` und gibt IPv4-TCP- und -UDP-Verbindungen aus. Der Systemaufruf erwartet drei Parameter, die im DTrace-Skript durch `arg0`, `arg1` und `arg2` repräsentiert werden und für uns im Kernel initialisiert werden. Der Parameter `arg0` ist der Socket-Dateideskriptor (den wir nicht brauchen), `arg1` ist die Adresse des Sockets, zu dem wir die Verbindung herstellen, und `arg2` ist die Länge dieser Adresse. Parameter `0` ist das Socket-Handle, das in diesem Fall nicht benötigt wird. Der nächste Parameter ist die Speicheradresse einer Socket-Adressstruktur innerhalb des Benutzerprozesses, d.h. die Adresse, mit der die Verbindung hergestellt werden soll. Die Größe dieser Adresse kann variieren und ist vom Socket-Typ abhängig (z.B. sind IPv4-Adressen kleiner als IPv6-Adressen). Der letzte Parameter ist die Länge der Socket-Adressstruktur in Bytes.

Das Skript definiert eine `sockaddr_in`-Struktur für IPv4-Verbindungen bei ❶. In vielen Fällen kann diese Struktur direkt aus den C-Header-Dateien des Systems kopiert werden. Der zu überwachende Systemaufruf steht bei ❷. Bei ❸ wird ein DTrace-spezifischer Filter genutzt, um sicherzustellen, dass wir nur connect-Aufrufe untersuchen, bei denen die Socket-Adresse die gleiche Größe hat wie `sockaddr_in`. Bei ❹ wird die `sockaddr_in`-Struktur von Ihrem lokalen Prozess in eine lokale Struktur kopiert, damit DTrace sie untersuchen kann. Bei ❺ wird der Name des Prozesses, die IP-Zieladresse und der Port über die Konsole ausgegeben.

Um dieses Skript auszuführen, kopieren Sie es in eine Datei namens *traceconnect.d* und führen dann den Befehl **`dtrace -s traceconnect.d`** als root aus. Wenn Sie eine mit dem Netzwerk verbundene Anwendung nutzen, sollte die Ausgabe in etwa so aussehen wie in Listing 2–3.

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

Listing 2–3 *Ausgabe des `traceconnect.d`-Skripts*

Die Ausgabe zeigt einzelne Verbindungen zu IP-Adressen. Ausgegeben werden der Name des Prozesses (z.B. `'Google Chrome'`), die IP-Adresse und der Port. Leider ist die Ausgabe nicht immer so hilfreich, wie die Ausgabe von `strace` unter Linux, doch DTrace ist mit Sicherheit ein nützliches Werkzeug. Dieses Beispiel kratzt nur an der Oberfläche dessen, was DTrace kann.

2.3.4 Process Monitor unter Windows

Im Gegensatz zu unixoiden Systemen implementiert Windows die Netzwerkfunktionen für den User-Modus ohne direkte Systemaufrufe. Der Netzwerkstack wird über einen Treiber bereitgestellt und der Aufbau einer Verbindung erfolgt über die Systemaufrufe `open`, `read` und `write`. Selbst wenn Windows so etwas wie `strace` unterstützen würde, macht diese Implementierung das Monitoring von Netzwerkverkehr im Gegensatz zu anderen Plattformen wesentlich schwieriger.

Seit Vista unterstützt Windows ein Ereignisse generierendes Framework, das es Anwendungen erlaubt, die Netzwerkaktivität zu überwachen. Das alles selbst zu implementieren wäre recht komplex, doch glücklicherweise hat jemand bereits ein Tool geschrieben, das das für Sie erledigt: Microsofts Process Monitor. Abbildung 2–5 zeigt die Hauptschnittstelle beim Filtern von Netzwerkereignissen.

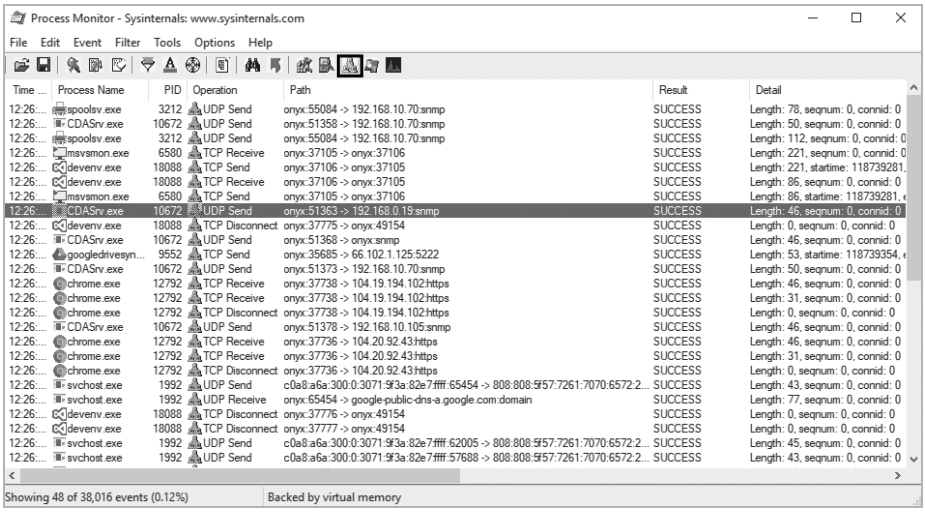

Abb. 2–5 *Capturing mit dem Process Monitor*

Die Wahl des Filters in Abbildung 2–5 gibt nur Ereignisse aus, die etwas mit den Netzwerkverbindungen des überwachten Prozesses zu tun haben. Zu den Details gehören die beteiligten Hosts sowie das Protokoll und der verwendete Port. Obwohl das Capturing keine mit den Verbindungen verknüpften Daten liefert, bietet es doch wertvolle Einblicke in die von der Anwendung aufgebaute Netzwerkkommunikation. Der Process Monitor kann auch den Zustand des aktuellen Aufrufstacks festhalten, was dabei zu ermitteln hilft, wo in einer Anwendung Netzwerkverbindungen hergestellt werden. Das wird in Kapitel 6 wichtig, wenn wir mit dem Reverse Engineering von Binaries beginnen, um das Netzwerkprotokoll auszuloten. Abbildung 2–6 zeigt eine einzelne HTTP-Verbindung mit einem entfernten Server im Detail.

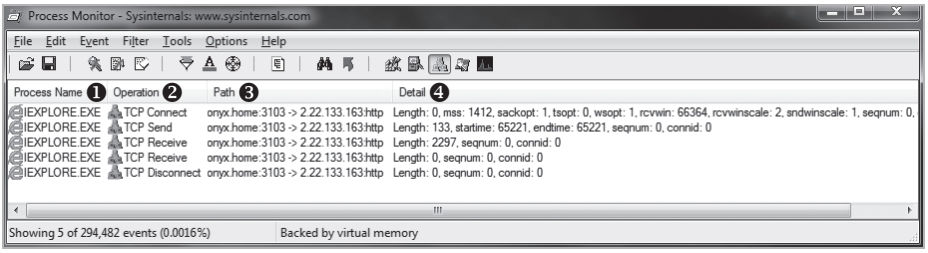

Abb. 2–6 *Capturing einer einzelnen Verbindung*

Spalte ❶ zeigt den Namen des Prozesses, der die Verbindung hergestellt hat. Spalte ❷ zeigt die durchgeführte Operation, in diesem Fall den Verbindungsaufbau mit dem entfernten Server, das Senden des einleitenden HTTP-Requests und den Empfang der Response. Spalte ❸ enthält die Quell- und Zieladressen und Spalte ❹ zeigt detaillierte Informationen zum festgehaltenen Ereignis.

Zwar ist diese Lösung nicht so hilfreich wie das Monitoring von Systemaufrufen bei anderen Plattformen, sie ist unter Windows aber dennoch nützlich, wenn man die Netzwerkprotokolle feststellen möchte, die eine bestimmte Anwendung nutzt. Ein Erfassen von Daten ist mit dieser Technik nicht möglich, doch sobald Sie die verwendeten Protokolle kennen, können Sie diese Daten mit aktiven Capturing-Techniken abrufen.

2.4 Vor- und Nachteile passiven Capturings

Der größte Vorteil des passiven Capturings besteht darin, dass es die Kommunikation zwischen Client- und Serveranwendung nicht stört. Die Quell- oder Zieladressen des Datenverkehrs werden nicht geändert und eine Modifikation oder Rekonfiguration der Anwendungen ist nicht nötig.

Passives Capturing könnte auch die einzige zur Verfügung stehende Technik sein, wenn Sie keine direkte Kontrolle über den Client oder den Server haben. Üblicherweise findet sich ein Weg, den Netzwerkverkehr abzuhören und mit relativ wenig Aufwand zu erfassen. Nachdem Sie Ihre Daten gesammelt haben, können Sie ermitteln, welche aktiven Capturing-Techniken Sie nutzen wollen und welcher Weg des beste ist, das zu analysierende Protokoll anzugreifen.

Ein wesentlicher Nachteil des passiven Capturings besteht darin, dass Capturing-Techniken wie Paket-Sniffing auf niedriger Ebene laufen und die von der Anwendung empfangenen Daten daher schwer zu interpretieren sein können. Tools wie Wireshark sind natürlich hilfreich, doch wenn Sie ein benutzerdefiniertes Protokoll analysieren, ist es eventuell nicht möglich, es in seine Bestandteile zu zerlegen, ohne direkt mit ihm zu interagieren.

Beim passiven Capturing ist es auch nicht immer einfach, den von einer Anwendung erzeugten Verkehr zu modifizieren. Eine Modifikation ist nicht immer nötig, aber nützlich, wenn Sie verschlüsselte Protokolle entdecken, die

Komprimierung deaktivieren wollen oder die Daten verändern wollen, um Sicherheitslücken auszunutzen.

Wenn die Analyse des Verkehrs und das Einspeisen neuer Pakete nicht zu den gewünschten Ergebnissen führen, sollten Sie die Taktik ändern und auf aktive Capturing-Techniken zurückgreifen.

2.5 Aktives Capturing von Netzwerkverkehr

Aktives Capturing unterscheidet sich vom passiven Capturing darin, dass man versucht, den Fluss der Daten zu beeinflussen. Das geschieht üblicherweise durch einen Man-in-the-Middle-Angriff auf die Netzwerkkommunikation. Wie in Abbildung 2–7 zu sehen, sitzt die Einrichtung, die den Verkehr erfasst, üblicherweise zwischen der Client- und der Serveranwendung und agiert als Bridge. Dieser Ansatz hat verschiedene Vorteile. So ist es beispielsweise möglich, den Verkehr zu modifizieren und Features wie Verschlüsselung und Komprimierung zu deaktivieren, was die Analyse des Protokolls und die Ausnutzung von Sicherheitslücken vereinfacht.

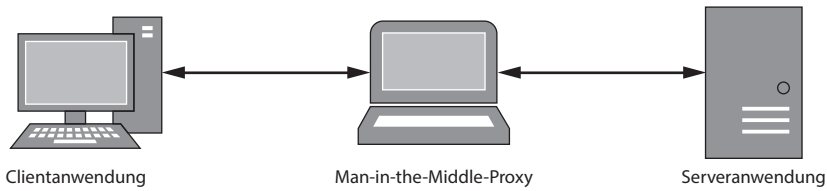

Abb. 2–7 *Man-in-the-Middle-Proxy*

Ein Nachteil dieses Ansatzes besteht darin, dass er üblicherweise schwieriger zu realisieren ist, weil Sie den Netzwerkverkehr der Anwendung über Ihr Capture-System leiten müssen. Aktives Capturing kann darüber hinaus zu unerwarteten und unerwünschten Effekten führen. Wenn Sie zum Beispiel die Netzwerkadresse des Servers oder Clients auf den Proxy setzen, kann das Verwirrung stiften und die Anwendung Daten an die falsche Stelle senden lassen. Trotz dieser Probleme ist das aktive Capturing die nützlichste Technik zur Analyse von Anwendungsprotokollen und zum Ausnutzen von Sicherheitslücken.

2.6 Netzwerk-Proxys

Um einen Man-in-the-Middle-Angriff auf Netzwerkverkehr durchzuführen, zwingt man die Anwendung üblicherweise, über einen Proxy-Service zu kommunizieren. In diesem Abschnitt möchte ich die Vor- und Nachteile gängiger Proxys erläutern, die Sie zum Capturing, zur Datenanalyse und zur Ausnutzung von Sicherheitslücken verwenden können. Ich werde Ihnen auch zeigen, wie Sie den Verkehr typischer Clientanwendungen an einen Proxy umleiten.

2.6.1 Port-Forwarding-Proxy

Port-Forwarding, also die Weiterleitung von Ports, ist die einfachste Form der Proxy-Verbindung. Richten Sie einfach einen Server (für TCP oder UDP) ein, der eingehende Verbindungen verarbeitet, und warten Sie auf eine neue Verbindung. Wird eine neue Verbindung mit dem Proxyserver hergestellt, öffnet er eine Forwarding-Verbindung mit dem eigentlichen Dienst und verbindet die beiden logisch miteinander, so wie in Abbildung 2–8 zu sehen.

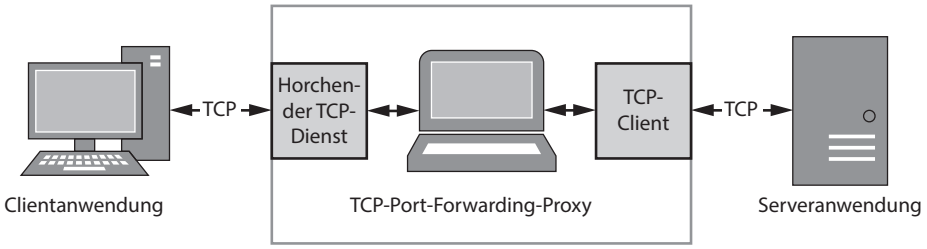

Abb. 2–8 *Übersicht eines TCP-Port-Forwarding-Proxys*

Einfache Implementierung

Für unseren Proxy nutzen wir den in den Canape-Core-Bibliotheken enthaltenen TCP-Port-Forwarder. Kopieren Sie den Code aus Listing 2–4 in eine C#-Skriptdatei und passen Sie *LOCALPORT* ❷, *REMOTEHOST* ❸ und *REMOTEPORT* ❹ an die für Ihr Netzwerk passenden Werte an.

PortFormat Proxy.csx

```
// PortFormatProxy.csx – Einfacher TCP-Port-Forwarding-Proxy
// Methoden wie WriteLine und WritePackets bereitstellen
using static System.Console;
using static CANAPE.Cli.ConsoleUtils;

// Proxy-Template erzeugen
var template = new ❶FixedProxyTemplate();
template.LocalPort = ❷LOCALPORT;
template.Host = ❸"REMOTEHOST";
template.Port = ❹REMOTEPORT;

// Proxy-Instanz erzeugen und starten
❺ var service = template.Create();
service.Start();

WriteLine("Created {0}", service);
WriteLine("Press Enter to exit...");
ReadLine();
❻ service.Stop();
```

```
    // Pakete ausgeben
    var packets = service.Packets;
    WriteLine("Captured {0} packets:",
        packets.Count);
❼   WritePackets(packets);
```

Listing 2–4 *Einfacher TCP-Port-Forwarding-Proxy*

Dieses sehr einfache Skript erzeugt eine Instanz von `FixedProxyTemplate` ❶. Canape Core arbeitet mit einem Templatemodell, bei Bedarf kann man sich aber auch die Finger mit der Low-Level-Konfiguration des Netzwerks schmutzig machen. Das Skript konfiguriert das Template mit den für das lokale und entfernte Netzwerk benötigten Informationen. Das Template wird verwendet, um eine Service-Instanz bei ❺ zu erzeugen. Sie können sich das so vorstellen, als würden die Dokumente im Framework als Templates für Dienste fungieren. Der neu erzeugte Dienst wird dann gestartet. An diesem Punkt sind die Netzwerkverbindungen konfiguriert. Nachdem auf einen Tastendruck gewartet wurde, wird der Service bei ❻ beendet. Alle erfassten Pakete werden dann über die Methode `WritePackets()` ❼ an der Konsole ausgegeben.

Wird das Skript ausgeführt, wird eine Instanz unseres Forwarding-Proxys an den *`LOCALPORT`* der localhost-Schnittstelle gebunden. Geht eine neue TCP-Verbindung an diesem Port ein, stellt der Proxycode eine neue Verbindung mit dem *`REMOTEHOST`* und dem TCP-Port *`REMOTEPORT`* her und verbindet die beiden miteinander.

WARNUNG

Die Bindung eines Proxys an alle Netzwerkadressen ist unter Sicherheitsgesichtspunkten riskant, weil die zum Testen von Protokollen geschriebenen Proxys nur selten robuste Sicherheitsmechanismen implementieren. Solange Sie nicht die vollständige Kontrolle über das Netzwerk oder keine andere Wahl haben, sollten Sie Ihren Proxy nur an die lokale Loopback-Schnittstelle binden. In Listing 2–4 ist *`LOCALHOST`* voreingestellt. Sollen alle Schnittstellen gebunden werden, setzen Sie die *`AnyBind`*-Property auf *`true`*.

Verkehr auf Proxys umleiten

Nachdem unsere einfache Proxy-Anwendung läuft, müssen wir den Anwendungsverkehr darauf umleiten.

Bei einem Webbrowser ist das sehr einfach: Um einen bestimmten Request zu untersuchen, ersetzen Sie die URL der Form *http://www.domain.com/ressource* durch *http://**localhost:localport**/ressource*, was den Request durch den Port-Forwarding-Proxy leitet.

Bei anderen Anwendungen ist das nicht ganz so leicht. Möglicherweise müssen Sie in die Konfigurationseinstellungen der Anwendung eintauchen. Manchmal können Sie in einer Anwendung nur die IP-Zieladresse konfigurieren. Das kann aber zu einem Henne-Ei-Problem ausarten, weil Sie nicht wissen, welche

TCP- oder UDP-Ports die Anwendung bei dieser Adresse nutzt. Das gilt besonders für Anwendungen mit komplexen Funktionen, die mit mehreren verschiedenen Serviceverbindungen arbeiten. Das ist bei RPC-Protokollen (*Remote Procedure Calls)* wie CORBA (Common Object Request Broker Architecture) der Fall. Dieses Protokoll erzeugt eine initiale Netzwerkverbindung, die als Verzeichnis der verfügbaren Dienste fungiert. Eine zweite Verbindung wird dann mit dem gewünschten Dienst über einen instanzspezifischen TCP-Port hergestellt.

In diesem Fall besteht ein guter Ansatz darin, so viele netzwerkgebundene Features der Anwendung wie möglich zu nutzen und gleichzeitig mit passivem Capturing Daten zu sammeln. Auf diese Weise sollten Sie die Verbindung aufspüren, die die Anwendung üblicherweise nutzt. Diese können Sie dann recht einfach mit Forwarding-Proxys replizieren.

Erlaubt es die Anwendung nicht, die Zieladresse zu ändern, müssen Sie etwas kreativer vorgehen. Löst die Anwendung die Adresse des Zielservers über den Hostnamen auf, haben Sie mehrere Möglichkeiten. Sie könnten einen eigenen DNS-Server einrichten, der auf Requests mit der IP-Adresse des Proxys antwortet. Oder Sie könnten die *hosts*-Datei nutzen, die bei den meisten Betriebssystemen (inklusive Windows) zur Verfügung steht. Das setzt natürlich voraus, dass Sie auf dem Gerät, auf dem die Anwendung läuft, die Kontrolle über das Dateisystem haben.

Während der Auflösung von Hostnamen schaut das Betriebssystem (oder die Resolver-Bibliotheken) zuerst in der *hosts*-Datei nach, ob ein Eintrag für diesen Namen existiert. Ein DNS-Request wird erst ausgeführt, wenn kein Eintrag gefunden wird. Die *hosts*-Datei in Listing 2–5 leitet zum Beispiel die Hostnamen *www.badgers.com* und *www.domain.com* auf *localhost* um.

```
# Standard-Localhost-Adressen
127.0.0.1       localhost
::1             localhost

# Dummy-Einträge, um den Verkehr durch den Proxy zu leiten
127.0.0.1       www.badgers.com
127.0.0.1       www.domain.com
```

Listing 2–5 *Beispiel einer `hosts`-Datei*

Die *hosts*-Datei findet man bei unixoiden Betriebssystemen in */etc/hosts*, während sie unter Windows in *C:\Windows\System32\Drivers\etc\hosts* angelegt ist. Den Pfad auf den Windows-Ordner müssen Sie natürlich entsprechend Ihrer Umgebung anpassen.

Hinweis
Einige Antiviren- und Sicherheitsprodukte rückverfolgen Änderungen an der hosts-Datei des Systems, da das ein Zeichen für Malware sein kann. Eventuell müssen Sie diesen Schutz deaktivieren, wenn Sie Änderungen an der hosts-Datei vornehmen wollen.

Vorteile eines Port-Forwarding-Proxys

Der Hauptvorteil eines Port-Forwarding-Proxys ist seine Einfachheit: Sie warten auf eine Verbindung, öffnen eine neue Verbindung zum eigentlichen Ziel und leiten den Verkehr zwischen den beiden hin und her. Mit dem Proxy ist kein besonderes Protokoll verknüpft, an das Sie sich zu halten hätten, und es ist auch kein besonderer Support durch die Anwendung nötig, deren Verkehr Sie erfassen wollen.

Ein Port-Forwarding-Proxy ist auch der Hauptansatz für das Umleiten von UDP-Verkehr. Da UDP-Verkehr nicht verbindungsorientiert ist, lässt sich ein Forwarder für UDP wesentlich einfacher implementieren.

Nachteile eines Port-Forwarding-Proxys

Natürlich trägt die Einfachheit eines Port-Forwarding-Proxys auch zu seinen Nachteilen bei. Weil Sie den Netzwerkverkehr nur von einer horchenden Verbindung an ein einzelnes Ziel weiterleiten, sind mehrere Proxy-Instanzen nötig, wenn die Anwendung mehrere Protokolle an unterschiedlichen Ports verwendet.

Stellen Sie sich beispielsweise eine Anwendung vor, die einen einzelnen Hostnamen oder eine einzelne IP-Adresse als Ziel verwendet. Diese können Sie entweder direkt in der Konfigurationsdatei durch Manipulation des Hostnamens oder durch Spoofing des Hostnamens kontrollieren. Die Anwendung versucht dann, die Verbindung mit den TCP-Ports 443 und 1234 herzustellen. Weil Sie die Verbindung zu den Adressen kontrollieren, nicht aber die Ports, müssen Sie Forwarding-Proxys für beide Ports einrichten, auch wenn Sie nur der Verkehr an Port 1234 interessiert.

Dieser Proxy kann es auch schwierig machen, mehr als eine Verbindung zu einem bekannten Port aufzubauen. Horcht der Port-Forwarding-Proxy etwa an Port 1234 und stellt er die Verbindung mit *www.domain.com* an Port 1234 her, läuft nur der umgeleitete Verkehr der ursprünglichen Domain wie erwartet. Wenn Sie auch *www.badgers.com* umleiten wollen, wird die Sache schwieriger. Sie können das ein wenig entschärfen, wenn die Anwendung die Änderung der Zieladresse und des Ports erlaubt oder indem Sie andere Techniken wie DNAT (Destination Network Address Translation) verwenden, um die Verbindungen an separate Forwarding-Proxys umzuleiten. (Kap. 5 geht detaillierter auf DNAT und viele andere fortgeschrittene Capturing-Techniken ein.)

Darüber hinaus könnte das Protokoll die Zieladresse für eigene Zwecke verwenden. Zum Beispiel kann der Host-Header bei HTTP (HyperText Transport Protocol) für die Wahl virtueller Hosts verwendet werden, wodurch das Proto-

koll durch die Umleitung anders oder gar nicht funktionieren könnte. Zumindest für HTTP beschreibe ich aber im Abschnitt 2.6.5, wie Sie dieses Problem umgehen können.

2.6.2 SOCKS-Proxy

Ein SOCKS-Proxy ist ein Port-Forwarding-Proxy auf Steroiden. Er leitet TCP-Verbindungen nicht nur an die gewünschte Netzwerkadresse weiter, sondern alle neuen Verbindungen beginnen auch mit einem einfachen Handshake-Protokoll, das den Proxy über das eigentliche Ziel informiert. (Es muss also nicht von uns festgelegt werden.) Es unterstützt auch horchende Verbindungen, was für Protokolle wie FTP (File Transfer Protocol) wichtig ist, die neue lokale Ports für den Server öffnen müssen, um Daten senden zu können. Abbildung 2–9 zeigt eine Übersicht eines SOCKS-Proxys.

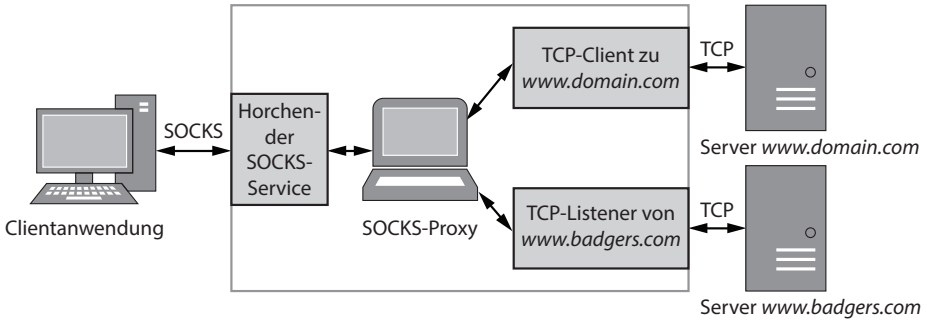

Abb. 2–9 *Übersicht eines SOCKS-Proxys*

Drei Varianten des Protokolls sind momentan verbreitet – SOCKS 4, 4a und 5 – und jede hat ihre Daseinsberechtigung. Version 4 ist die am häufigsten unterstützte Variante des Protokolls, kennt aber nur IPv4-Verbindungen und die Zieladresse muss als 32-Bit-IP-Adresse angegeben werden. Ein Update der Version 4, die Version 4a, erlaubt Verbindungen über den Hostnamen (was nützlich ist, wenn Sie keinen DNS-Server haben, der IP-Adressen auflösen kann). Mit der Version 5 wurde Support für Hostnamen, IPv6, UDP-Forwarding und verbesserte Authentifizierungsmechanismen eingeführt. Sie ist die einzige Version, die in einem RFC (1928) spezifiziert ist.

Als Beispiel soll ein Client den in Abbildung 2–10 dargestellten Request senden, um eine SOCKS-Verbindung mit der IP-Adresse 10.0.0.1 an Port 12345 herzustellen. Die `USERNAME`-Komponente ist die einzige Authentifizierungsmethode in der SOCKS-Version 4 (nicht besonders sicher, ich weiß). `VER` repräsentiert die Versionsnummer, in diesem Fall also 4. `CMD` gibt an, dass es eine ausgehende Verbindung (mit der Adresse in `CMD` 2) herstellen will. Der TCP-Port und die Adresse werden im Binärformat angegeben.

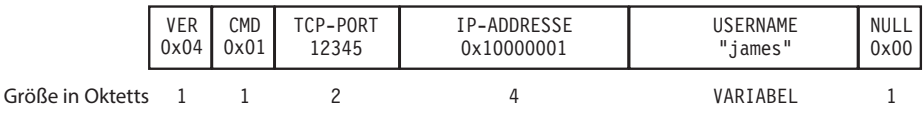

Abb. 2–10 *Request bei SOCKS 4*

Ist die Verbindung erfolgreich, sendet das Protokoll eine entsprechende Response, die in Abbildung 2–11 zu sehen ist. Das `RESP`-Feld enthält den Status der Response. Die TCP-Port- und -Adressfelder sind nur bei Bindungs-Requests relevant. Die Verbindung wird dann transparent und der Client und der Server kommunizieren direkt miteinander. Der Proxyserver leitet den Verkehr nun in beide Richtungen weiter.

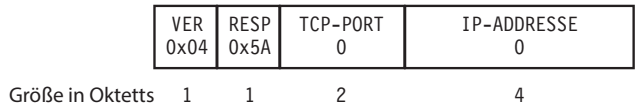

Abb. 2–11 *Erfolgreiche Response bei SOCKS 4*

Einfache Implementierung

Die Canape-Core-Bibliotheken unterstützen SOCKS 4, 4a und 5. Kopieren Sie Listing 2–6 in eine C#-Datei und ersetzen Sie *`LOCALPORT`* ❷ durch den lokalen TCP-Port, an dem der SOCKS-Proxy horchen soll.

SocksProxy.csx

```
// SocksProxy.csx – Einfacher SOCKS-Proxy
// Methoden wie WriteLine und WritePackets bereitstellen
using static System.Console;
using static CANAPE.Cli.ConsoleUtils;

// SOCKS-Proxy-Template erzeugen
❶ var template = new SocksProxyTemplate();
template.LocalPort = ❷LOCALPORT;

// Proxy-Instanz erzeugen und starten
var service = template.Create();
service.Start();

WriteLine("Created {0}", service);
WriteLine("Press Enter to exit...");
ReadLine();
service.Stop();

// Pakete ausgeben
var packets = service.Packets;
WriteLine("Captured {0} packets:",
    packets.Count);
WritePackets(packets);
```

Listing 2–6 *Einfacher SOCKS-Proxy*

Listing 2–6 folgt dem gleichen Muster wie beim TCP-Port-Forwarding-Proxy in Listing 2–4. Doch in diesem Fall erzeugt der Code bei ❶ ein SOCKS-Proxy-Template. Der Rest des Codes ist identisch.

Verkehr an den Proxy umleiten

Wenn Sie eine Möglichkeit suchen, die Netzwerkdaten einer Anwendung durch einen SOCKS-Proxy zu leiten, sehen Sie sich zuerst die Anwendung selbst an. Wenn Sie z.B. die Proxy-Einstellungen in Mozilla Firefox öffnen, erscheint der Dialog aus Abbildung 2–12. Dort können Sie Firefox so konfigurieren, dass es einen SOCKS-Proxy verwendet.

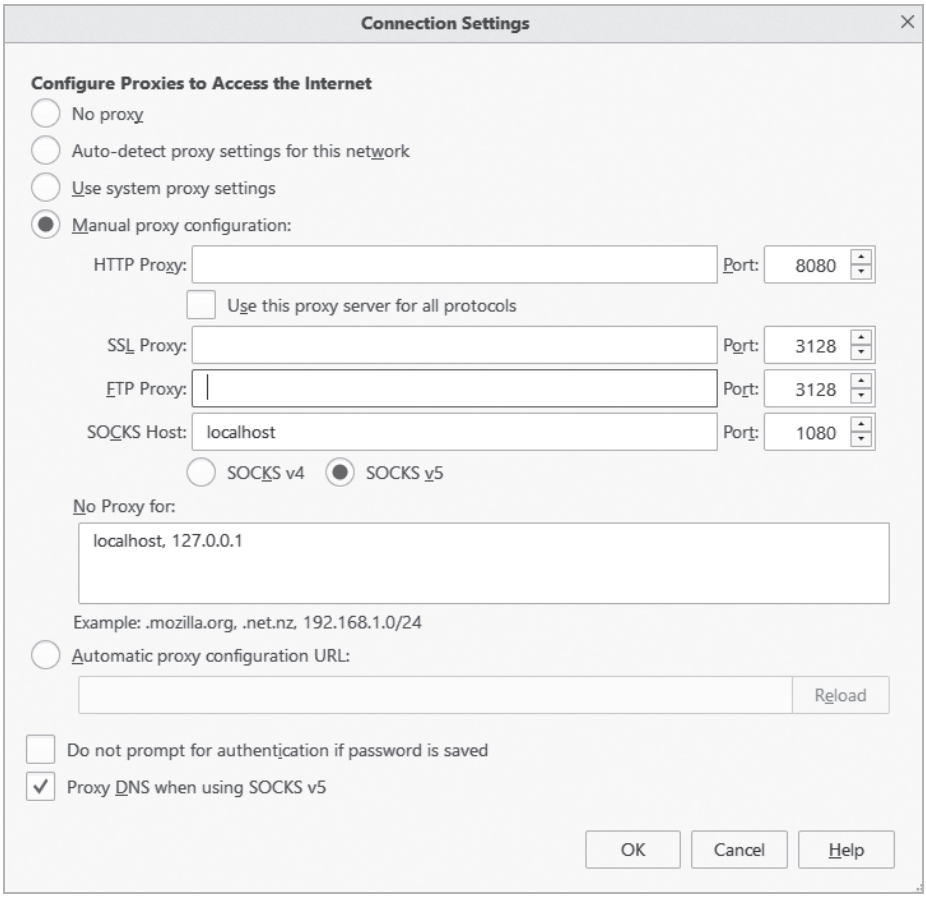

Abb. 2–12 *Proxy-Konfiguration in Firefox*

Doch manchmal ist der SOCKS-Support nicht ganz so offensichtlich. Wenn Sie eine Java-Anwendung testen, akzeptiert die Java-Runtime Kommandozeilenparameter, die den SOCKS-Support für jede ausgehende TCP-Verbindung aktivieren.

Nehmen wir zum Beispiel die sehr einfache Java-Anwendung aus Listing 2–7, die eine Verbindung mit der IP-Adresse 192.168.10.1 an Port 5555 herstellt.

SocketClient.java

```
// SocketClient.java – Einfacher Java-TCP-Socket-Client
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) {
        }
    }
}
```

Listing 2–7 *Einfacher Java-TCP-Client*

Wenn Sie dieses Programm ganz normal ausführen, funktioniert es wie erwartet. Doch wenn Sie über die Kommandozeile die System-Properties `socksProxyHost` und `socksProxyPort` übergeben, können Sie einen SOCKS-Proxy für jede TCP-Verbindung festlegen:

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

Das leitet die TCP-Verbindung über den SOCKS-Proxy am Localhost-Port 1080.

Ein anderer Ort, an dem Sie nach einer Möglichkeit suchen können, den Netzwerkverkehr einer Anwendung über einen SOCKS-Proxy zu leiten, ist der Standardproxy des Betriebssystems. Gehen Sie zum Beispiel unter macOS zu **Systemeinstellungen ▸ Netzwerk ▸ Weitere Optionen ▸ Proxies**. Der Dialog aus Abbildung 2–13 erscheint. Hier können Sie einen systemweiten SOCKS-Proxy oder Proxys für andere Protokolle einrichten. Das funktioniert nicht immer, ist aber eine einfache Möglichkeit und einen Versuch wert.

Unterstützt eine Anwendung von Haus aus keinen SOCKS-Proxy, gibt es Tools, die beliebige Anwendungen um diese Funktion erweitern. Diese Tools reichen von freien und Open-Source-Tools wie Dante (*https://www.inet.no/dante/*) unter Linux bis zu kommerziellen Werkzeugen wie Proxifier (*https://www.proxifier.com/*), das unter Windows und macOS läuft. Auf die ein oder andere Weise klinken sie sich in die Anwendung ein, ergänzen die SOCKS-Unterstützung und modifizieren den Betrieb der Socket-Funktionen.

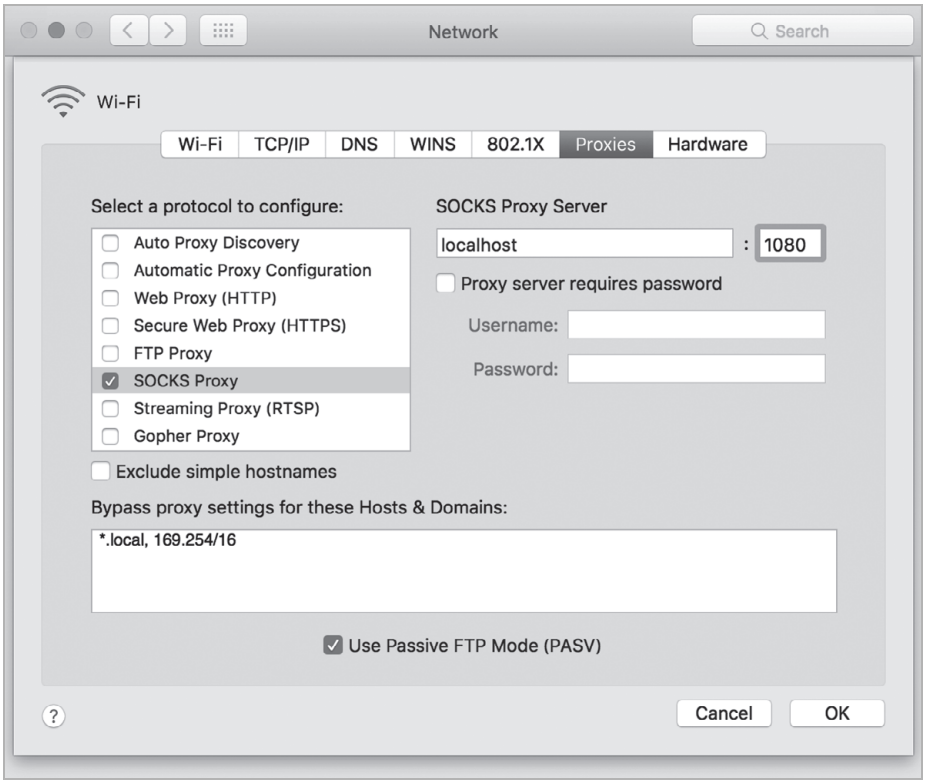

Abb. 2–13 *Proxy-Konfigurationsdialog unter macOS*

Vorteile eines SOCKS-Proxys

Der klare Vorteil eines SOCKS-Proxys anstelle eines einfachen Port-Forwarders besteht darin, dass er alle TCP-Verbindungen (und bei SOCKS 5 auch ein wenig UDP) einer Anwendung erfassen kann. Das ist ein Vorteil, solange die Socket-Schicht durch das Betriebssystem gut abgedeckt ist und alle Verbindungen durch den Proxy leitet.

Ein SOCKS-Proxy bewahrt aus Sicht der Anwendung auch das Ziel der Verbindung bei. Sendet der Client also interne Daten, die an diesen Endpunkt gerichtet sind, dann entspricht dieser Endpunkt dem, was der Server erwartet. Allerdings konserviert das nicht die Quelladresse. Einige Protokolle, wie FTP, setzen voraus, dass sie Ports auf dem Client öffnen können. Das SOCKS-Protokoll bietet die Möglichkeit, horchende Verbindungen zu binden, was aber die Komplexität der Implementierung erhöht. Das erschwert das Erfassen und die Analyse, weil Sie viele verschiedene Streams von und zum Server berücksichtigen müssen.

Nachteile eines SOCKS-Proxys

Der Hauptnachteil von SOCKS ist die recht uneinheitliche Unterstützung zwischen Anwendungen und Plattformen. Der Proxy des Windows-Systems unterstützt nur SOCKS 4, d.h., es werden nur lokale Hostnamen aufgelöst. IPv6 wird nicht unterstützt und es fehlt ein robuster Authentifizierungsmechanismus. Generell ist der Support besser, wenn Sie ein SOCKS-Tool für die Anwendung nutzen, doch das funktioniert nicht immer gut.

2.6.3 HTTP-Proxys

HTTP treibt das World Wide Web sowie eine Vielzahl von Webdiensten und REST-Protokollen an. Abbildung 2–14 zeigt eine Übersicht eines HTTP-Proxys. Das Protokoll kann auch als Transportmechanismus für Nicht-Webprotokolle wie Javas Remote Method Invocation (RMI) oder das Real Time Messaging Protocol (RTMP) verwendet werden, da es selbst durch die restriktivsten Firewalls getunnelt werden kann. Es ist wichtig zu verstehen, wie HTTP-Proxys in der Praxis funktionieren, weil das für die Protokollanalyse nahezu immer nützlich ist, selbst wenn kein Webservice untersucht wird. Die existierenden Tools zum Testen von Webanwendungen machen nur selten einen idealen Job, wenn HTTP in seiner ursprünglichen Umgebung genutzt wird. Manchmal ist die Implementierung eines eigenen HTTP-Proxys die einzige Lösung.

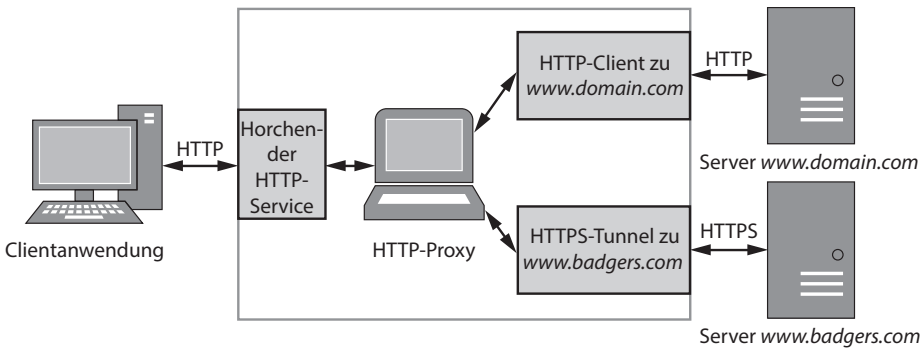

Abb. 2–14 *Übersicht eines HTTP-Proxys*

Die beiden Hauptarten eines HTTP-Proxys sind der Forwarding-Proxy und der Reverse-Proxy. Jeder hat aus Sicht der Netzwerkprotokoll-Analyse seine Vor- und Nachteile.

2.6.4 Forwarding eines HTTP-Proxys

Das HTTP-Protokoll ist in der Version 1.0 in RFC 1945 definiert und in der Version 1.1 in RFC 2616. Beide Versionen bieten einen einfachen Mechanismus zum

Proxying von HTTP-Requests. Zum Beispiel legt HTTP 1.1 fest, dass die erste Zeile eines Requests, die *Request-Zeile*, das folgende Format hat:

```
❶ GET ❷/image.jpg HTTP/1.1
```

Die Methode ❶ gibt an, was bei diesem Request zu tun ist, wobei vertraute englische Verben wie `GET`, `POST` und `HEAD` verwendet werden. Bei einem Proxy-Request unterscheidet sich das nicht von einer normalen HTTP-Verbindung. Beim Pfad ❷ wird der Proxy-Request interessant. Ein absoluter Pfad gibt die Ressource an, auf die die Methode angewandt wird. Wichtiger ist aber, dass der Pfad auch ein absoluter Uniform Request Identifier (URI) sein kann. Durch Angabe einer absoluten URI kann ein Proxyserver eine neue Verbindung mit dem Ziel herstellen, den gesamten Verkehr weiterzuleiten und Daten an den Client zurückzuliefern. Der Proxy kann sogar (in eingeschränkter Form) den Verkehr manipulieren, um eine Authentifizierung zu ergänzen, Version-1.0-Server vor Version-1.1-Clients verstecken und (neben vielen anderen Dingen) den Verkehr komprimieren. Diese Flexibilität hat aber ihren Preis: Der Proxyserver muss in der Lage sein, den HTTP-Verkehr zu verarbeiten, was eine recht komplexe Angelegenheit ist. Die folgende Request-Zeile fordert zum Beispiel ein Bild von einem entfernten Server über einen Proxy an:

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

Als aufmerksamer Leser werden Sie bei diesem Proxy-Ansatz zur HTTP-Kommunikation einen Haken gefunden haben. Da der Proxy in der Lage sein muss, auf das zugrunde liegende HTTP-Protokoll zuzugreifen, stellt sich die Frage, was bei HTTPS passiert, das HTTP über eine verschlüsselte TLS-Verbindung transportiert. Sie könnten den verschlüsselten Verkehr herausfiltern, doch in einer normalen Umgebung ist es sehr unwahrscheinlich, dass der HTTP-Client irgendeinem von ihnen bereitgestellten Zertifikat vertrauen würde. Darüber hinaus wurde TLS ganz bewusst so entwickelt, dass es einen Man-in-the-Middle-Angriff nahezu unmöglich macht. Glücklicherweise hat man das vorausgesehen und RFC 2817 bietet zwei Lösungen an: zum einen die Möglichkeit, eine HTTP-Verbindung auf eine verschlüsselte Verbindung umzustellen (weitere Details sind hier nicht nötig), und, für unsere Zwecke noch wichtiger, zum anderen die Spezifikation der HTTP-Methode `CONNECT` zum Aufbau von transparenten, getunnelten Verbindungen über HTTP-Proxys. Möchte ein Webbrowser zum Beispiel eine Proxy-Verbindung mit einer HTTPS-Site herstellen, kann er den folgenden Request an den Proxy senden:

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

Akzeptiert der Proxy diesen Request, stellt er eine neue TCP-Verbindung mit dem Server her. Ist er erfolgreich, antwortet er mit der folgenden Response:

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

Die TCP-Verbindung zum Proxy wird nun transparent und der Browser kann die TLS-Verbindung aushandeln, ohne dass der Proxy ihm dazwischenfunkt. Es ist natürlich anzumerken, dass der Proxy nicht unbedingt verifiziert, ob bei dieser Verbindung wirklich TLS verwendet wird. Tatsächlich kann es sich um jedes beliebige Protokoll handeln und diese Tatsache wird von einigen Anwendungen missbraucht, um deren eigene binäre Protokolle durch HTTP-Proxys zu tunneln. Aus diesem Grund beschränken HTTP-Proxys die Ports, die getunnelt werden dürfen, sehr stark.

Einfache Implementierung

Die Canape-Core-Bibliotheken enthalten eine einfache Implementierung eines HTTP-Proxys. Unglücklicherweise unterstützen sie die `CONNECT`-Methode nicht, um einen transparenten Tunnel zu erzeugen, doch zu Demonstrationszwecken reicht es. Kopieren Sie Listing 2–8 in eine C#-Datei und ändern Sie *LOCALPORT* ❷ in den lokalen TCP-Port, an dem Sie horchen wollen.

HttpProxy.csx

```
// HttpProxy.csx – Einfacher HTTP-Proxy
// Methoden wie WriteLine und WritePackets bereitstellen
using static System.Console;
using static CANAPE.Cli.ConsoleUtils;

// Proxy-Template erzeugen
❶ var template = new HttpProxyTemplate();
template.LocalPort = ❷LOCALPORT;

// Proxy-Instanz erzeugen und starten
var service = template.Create();
service.Start();

WriteLine("Created {0}", service);
WriteLine("Press Enter to exit...");
ReadLine();
service.Stop();

// Pakete ausgeben
var packets = service.Packets;
WriteLine("Captured {0} packets:", packets.Count);
WritePackets(packets);
```

Listing 2–8 *Einfacher HTTP-Forwarding-Proxy*

Hier haben wir einen HTTP-Forwarding-Proxy erzeugt. Der Code in Zeile ❶ erstellt ein HTTP-Proxy-Template und ist auch hier wieder nur eine leichte Abwandlung der vorherigen Beispiele.

Verkehr auf den Proxy umleiten

Wie bei SOCKS-Proxys sollten Sie zuerst einen Blick auf die Anwendung werfen. Es ist sehr selten, dass Anwendungen, die das HTTP-Protokoll verwenden, keine Form der Proxy-Konfiguration bieten. Unterstützt die Anwendung HTTP-Proxys nicht direkt, können Sie sich die Konfiguration des Betriebssystems ansehen, die sie an der gleichen Stelle finden wie die SOCKS-Konfiguration. Zum Beispiel erreichen Sie die Proxy-Einstellungen unter Windows 10 über Steuerung ▸ Windows Einstellungen ▸ Netzwerk und Internet ▸ Proxy.

Viele Kommandozeilen-Utilities bei unixoiden Systemen, wie etwa `curl`, `wget` und `apt`, unterstützen die Konfiguration von HTTP-Proxys über Umgebungsvariablen. Wenn Sie die Umgebungsvariable `http_proxy` auf die URL des HTTP-Proxys – etwa *http://localhost:3128* – setzen, nutzt die Anwendung sie. Bei verschlüsseltem Verkehr können Sie auch *https_proxy* nutzen. Einige Implementierungen erlauben spezielle URL-Schemata wie z.B. *socks4://*, um den gewünschten Proxy (hier SOCKS) festzulegen.

Vorteile eines HTTP-Forwarding-Proxys

Der Hauptvorteil eines HTTP-Forwarding-Proxys kommt zum Tragen, wenn die Anwendung ausschließlich das HTTP-Protokoll verwendet. Um Proxy-Support zu integrieren, müssen Sie dann nur den absoluten Pfad der Request-Zeile in eine absolute URI ändern, um die Daten an den horchenden Proxy weiterzuleiten. Darüber hinaus gibt es nur wenige Anwendungen, die das HTTP-Protokoll für den Transport nutzen und keine Proxys unterstützen.

Nachteile eines HTTP-Forwarding-Proxys

Ein HTTP-Forwarding-Proxy muss einen vollständigen HTTP-Parser implementieren, um alle Eigenheiten des Protokolls verarbeiten zu können, was dessen Komplexität deutlich erhöht. Diese Komplexität kann zu Problemen bei der Verarbeitung führen oder schlimmstenfalls zu Sicherheitslücken. Das Hinzufügen des Proxys innerhalb des Protokolls kann die nachträgliche Ergänzung der HTTP-Proxy-Unterstützung in eine bestehende Anwendung durch externe Techniken erschweren, wenn Sie Verbindungen nicht dazu bringen, die `CONNECT`-Methode zu nutzen (die auch bei unverschlüsseltem HTTP funktioniert).

Durch die Komplexität der Verarbeitung einer HTTP-1.1-Verbindung ist es bei Proxys üblich, die Verbindung des Clients nach einem einzelnen Request zu beenden oder die Kommunikation auf die Version 1.0 herunterzufahren (was die Response-Verbindung immer schließt, nachdem alle Daten empfangen wurden). Das könnte bei übergeordneten Protokollen zu Problemen führen, die die Version 1.1 verlangen oder *Pipelining* erfordern,wodurch mehrere Requests gleichzeitig verarbeitet werden können, um die Performance zu erhöhen oder den Zustand zu verwalten.

2.6.5 HTTP-Reverse-Proxy

Forwarding-Proxys sind in Umgebungen weit verbreitet, bei denen ein interner Client die Verbindung zu einem außerhalb liegenden Netzwerk herstellt. Sie dienen der Datensicherheit, indem sie ausgehenden Verkehr auf eine kleine Menge möglicher Protokolle beschränken. (Mögliche Sicherheitsprobleme durch `CONNECT` wollen wir im Moment ignorieren.) Doch manchmal sollen auch eingehende Verbindungen über einen Proxy laufen, etwa zum Load Balancing oder aus Sicherheitsgründen (um die Server nicht direkt der Außenwelt auszusetzen). Allerdings gibt es dabei ein Problem: Man hat keine Kontrolle über den Client. Tatsächlich erkennt der Client gar nicht, dass er die Verbindung mit einem Proxy herstellt. Hier kommt der *HTTP-Reverse-Proxy* ins Spiel.

Statt wie beim Forwarding-Proxy den Zielhost in der Request-Zeile anzugeben, können wir die Tatsache nutzen, dass alle HTTP-1.1-konformen Clients einen HTTP-Host-Header im Request senden *müssen*, der den ursprünglichen Hostnamen angibt, der in der URI des Requests verwendet wurde. (Beachten Sie, dass das bei HTTP 1.0 nicht nötig ist, doch die meisten Clients senden diesen Header auch bei dieser Version mit.) Aus dem Host-Header können Sie das eigentliche Ziel des Requests ableiten und eine Proxy-Verbindung mit diesem Server herstellen, wie in Listing 2–9 zu sehen.

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

Listing 2–9 *Beispiel für einen HTTP-Request*

Listing 2–9 zeigt einen typischen Host-Header ❶, bei dem der HTTP-Request die URL *http://www.domain.com/image.jpg* angefordert hat. Der Reverse-Proxy kann diese Informationen nehmen und daraus das ursprüngliche Ziel konstruieren. Da die HTTP-Header geparst werden müssen, ist das für HTTPS-Verkehr (der durch TLS geschützt wird) schwieriger zu realisieren. Glücklicherweise erlauben die meisten TLS-Implementierungen Wildcard-Zertifikate der Form **.domain.com*, die jede Subdomain von *domain.com* erkennt.

Einfache Implementierung

Es wird Sie kaum überraschen, dass die Canape-Core-Bibliotheken einen HTTP-Reverse-Proxy bereitstellen. Der Zugriff erfolgt über das Template-Objekt *HttpReverseProxyTemplate* anstelle von *HttpProxyTemplate*. Der Vollständigkeit halber zeigt Listing 2–10 eine einfache Implementierung. Kopieren Sie den folgenden Code in eine C#-Datei und ändern Sie *`LOCALPORT`* ❶ in den lokalen TCP-Port ab, an dem Sie horchen wollen. Wenn Sie auf einem unixoiden System arbei-

ten und *LOCALPORT* kleiner als 1024 ist, müssen Sie das Skript mit Root-Rechten ausführen.

ReverseHttp Proxy.csx

```
// ReverseHttpProxy.csx – Einfacher HTTP-Reverse-Proxy
// Methoden wie WriteLine und WritePackets bereitstellen
using static System.Console;
using static CANAPE.Cli.ConsoleUtils;

// Proxy-Template erzeugen
var template = new HttpReverseProxyTemplate();
template.LocalPort = ❶LOCALPORT;

// Proxy-Instanz erzeugen und starten
var service = template.Create();
service.Start();

WriteLine("Created {0}", service);
WriteLine("Press Enter to exit...");
ReadLine();
service.Stop();

// Pakete ausgeben
var packets = service.Packets;
WriteLine("Captured {0} packets:",
    packets.Count);
WritePackets(packets);
```

Listing 2–10 *Einfacher HTTP-Reverse-Proxy*

Verkehr auf Ihren Proxy umleiten

Der Ansatz zur Umleitung des Verkehrs an einen Reverse-Proxy ähnelt dem, der beim TCP-Port-Forwarding angewandt wurde, d.h., die Verbindung wird auf den Proxy umgeleitet. Doch es gibt einen großen Unterschied: Sie können den Hostnamen des Ziels nicht ändern. Das würde den in Listing 2–10 gezeigten Host-Header ändern. Wenn Sie nicht aufpassen, kommt es zu einer Proxy-Schleife[1]. Daher ist es besser, die mit dem Hostnamen verknüpfte IP-Adresse in der *hosts*-Datei zu ändern.

Doch möglicherweise läuft die zu testende Anwendung auf einem Gerät, das es Ihnen nicht erlaubt, die *hosts*-Datei zu ändern. Die Einrichtung eines eigenen DNS-Servers ist dann vielleicht die einfachste Lösung, zumindest wenn Sie die Konfiguration des DNS-Servers ändern können.

Sie können einen anderen Ansatz nutzen, der darin besteht, einen DNS-Server mit den entsprechenden Einstellungen zu konfigurieren. Das kann zeitaufwendig

1. Zu einer Proxy-Schleife kommt es, wenn der Proxy wiederholt die Verbindung zu sich selbst herstellt und so in einer rekursiven Schleife gefangen ist. Das Ergebnis ist eine Katastrophe oder zumindest gehen dem System schnell die verfügbaren Ressourcen aus.

und fehlerträchtig sein. Fragen Sie einfach jemanden, der schon einmal einen Bind-Server aufgesetzt hat. Glücklicherweise gibt es Tools, die machen, was wir wollen, d.h., sie liefern die IP-Adresse unseres Proxys als Antwort auf einen DNS-Request zurück. Ein solches Tool ist *dnsspoof*. Um die Installation eines weiteren Tools zu vermeiden, können Sie dazu auch Canapes DNS-Server nutzen. Der einfache DNS-Server fälscht nur eine einzelne IP-Adresse für alle DNS-Requests (siehe Listing 2–11). Ersetzen Sie *IPV4ADDRESS* ❶, *IPV6ADDRESS* ❷ und *REVERSEDNS* ❸ durch die entsprechenden Werte. Wie beim HTTP-Reverse-Proxy müssen Sie das Skript bei unixoiden Systemen mit Root-Rechten ausführen, da es die Bindung an Port 53 versucht, was normalen Nutzern nicht erlaubt ist. Unter Windows gibt es diese Einschränkung für Ports kleiner als 1024 nicht.

DnsServer.csx

```
// DnsServer.csx – Einfacher DNS-Server
// Methoden wie WriteLine global bereitstellen.
using static System.Console;

// DNS-Server-Template erzeugen
var template = new DnsServerTemplate();

// Response-Adressen festlegen
template.ResponseAddress = ❶"IPV4ADDRESS";
template.ResponseAddress6 = ❷"IPV6ADDRESS";
template.ReverseDns = ❸"REVERSEDNS";

// DNS-Server-Instanz erzeugen und starten
var service = template.Create();
service.Start();
WriteLine("Created {0}", service);
WriteLine("Press Enter to exit...");
ReadLine();
service.Stop();
```

Listing 2–11 *Einfacher DNS-Server*

Wenn Sie den DNS-Server Ihrer Anwendung jetzt so konfigurieren, dass er auf unseren falschen DNS-Server zeigt, sollte die Anwendung den Verkehr wie gewünscht durchleiten.

Vorteile eines HTTP-Reverse-Proxys

Der Vorteil eines Reverse-Proxys besteht darin, dass die Clientanwendung eine typische Forwarding-Proxy-Konfiguration nicht unterstützen muss. Das ist besonders dann nützlich, wenn die Clientanwendung nicht unter Ihrer direkten Kontrolle steht oder eine feste Konfiguration verwendet, die sich nicht einfach ändern lässt. Solange Sie die ursprünglichen TCP-Verbindungen auf den Proxy umleiten können, können Sie Requests an mehrere unterschiedliche Hosts ohne Probleme verarbeiten.

Nachteile eines HTTP-Reverse-Proxys

Die Nachteile eines Reverse-Proxys sind grundsätzlich die gleichen wie bei einem Forwarding-Proxy. Der Proxy muss in der Lage sein, den HTTP-Request zu parsen und die Eigenheiten des Protokolls zu verarbeiten.

2.7 Am Ende dieses Kapitels

Sie haben in diesem Kapitel etwas über passive und aktive Capturing-Techniken gelernt – ist nun die eine Technik besser als die andere? Das hängt von der zu testenden Anwendung ab. Wenn Sie nicht einfach nur den Netzwerkverkehr überwachen wollen, ist der aktive Ansatz besser. Wie Sie im weiteren Verlauf des Buches sehen werden, bietet aktives Capturing bei der Protokollanalyse und bei der Suche nach Exploits deutliche Vorteile. Wenn Ihnen die Anwendung eine Wahl lässt, nutzen Sie SOCKS, da es in vielen Fällen der einfachste Ansatz ist.