

[Programowe tunelowanie połączeń TCP.]



Autor: Krystian Kloskowski (h07) <h07@interia.pl>
-<http://milw0rm.com/author/668>
-<http://www.h07.int.pl>

0x00 [INTRO]

Art ten jak zwykle kierowany jest do osób, które mają dość chałtury prezentowanej przez niektóre "hack" serwisy i chcą nauczyć się czegoś nowego, ciekawego, czegoś co poszerza horyzony i rozwija wyobraźnię.

"Tunelowanie połączenia" jest procesem przesyłania danych między klientem a serwerem poprzez inny serwer lub grupę serwerów. Jedną z korzyści takiego połączenia jest zmiana adresu IP, dzięki czemu przykładowo możemy wejść na IRC pod "fałszywym" adresem IP. Inne korzyści, płynące z takiego rozwiązania przydają się przede wszystkim audytorom oprogramowania ;) ale to już inna bajka.

Niniejszy artykuł opisuje, jak skonstruować serwer tunelujący połączenie TCP w języku Python.

0x01 [TEORIA]

Obrazowo tunelowanie połączenia wygląda następująco:

```
[klient](192.168.0.1) ----> [tunel](192.168.0.2) ----> [serwer](192.168.0.3)
[klient](192.168.0.1) <---- [tunel](192.168.0.2) <---- [serwer](192.168.0.3)
```

Z powyższego przykładu łatwo można wywnioskować dwie zależności.

Z punktu widzenia klienta adres IP serwera docelowego jest adresem serwera tunelującego.

Z punktu widzenia serwera docelowego adres IP klienta jest adresem serwera tunelującego.

Reasumując "tunel" jest wirtualnym serwerem docelowym dla klienta oraz wirtualnym klientem dla serwera docelowego.

Można wyobrazić sobie, że serwer tunelujący składa się z dwóch "wtyczek".

Jedna wtyczka zapewnia obsługę połączenia z klientem a druga z serwerem docelowym.

Wtyczki te nawzajem przekazują sobie dane, co daje efekt "tunelu".

0x02 [ARCHITEKTURA]

Projektowanie architektury jest najważniejszą częścią całego etapu tworzenia serwera.

To właśnie w tym momencie programista musi wyobrazić sobie w jaki sposób to wszystko ma działać i jak to "zlepić" w jedną, spójną, bezawaryjną całość. Czynność ta nazywana jest tworzeniem algorytmu elektronicznego. Na szczęście Wy nie musicie się tym trudzić a jedynie zrozumieć moją myśl konstrukcyjną.

Serwer przede wszystkim musi opierać się na architekturze wielowątkowej.

Oznacza to, że każde połączenie będzie obsługiwane przez osobny wątek co zapewni niezależną obsługę kilku połączeń jednocześnie.

Wtyczki o których wspomniałem w rzeczywistości będą reprezentowane przez obiekty opisane klasą. Ów obiekty będą automatycznie tworzone w momencie odebrania nowego połączenia przez serwer. Ponadto każdy z tych obiektów będzie uruchamiany z poziomu nowego wątku co zapewni niezależny przepływ danych między obiema wtyczkami. Jednak by przekazywanie danych było możliwe wtyczki uruchomione w osobnych wątkach muszą "wiedzieć" o swoim istnieniu.

Teoretycznie serwer mógłby posługiwać się globalną listą zawierającą wskaźniki obu wtyczek. W ten sposób wtyczki "widziałyby" się nawzajem pobierając dane z globalnej listy dostępnej dla wszystkich wątków. Jednak lista ta z natury byłaby jednowymiarowa co nie pozwoliłoby na obsługę kilku połączeń jednocześnie.

Zastosowanie listy wielowymiarowej znacznie skomplikowałoby działanie serwera, zatem całą konstrukcję oparłem o "wtyczki równoległe".
Wszystko to brzmi dość niezrozumiale ale w rzeczywistości jest banalnie proste w działaniu.

Idea wtyczek równoległych polega na wyzwaniu jednej wtyczki z drugiej. Dzięki takiemu mechanizmowi wtyczka wyzwalamąca może przekazać wtyczce wyzwalamącej informację o sobie w postaci listy. W ten sposób obie wtyczki pracują niezależnie w osobnych wątkach ale "widzą" się wzajemnie dzięki temu, że operują na identycznych listach wtyczek.

Obrazowy przykład działania:

```

                [Nowy wątek] Wtyczka A
                A.lista[A, B]
client ----> dane = sock.recv()
                lista[1].sock.send(dane) ----> server ---->+
                |
                [Nowy wątek] Wtyczka B
                B.lista[A, B]
                dane = sock.recv() <-----+
+-----<---- lista[0].sock.send(dane)
```

Opis działania:

- 1) Klient wysłał zapytanie.
- 2) Wtyczka A odbiera dane od klienta za pomocą funkcji `recv()` po czym wywołuje funkcję `send()` wtyczki B przekazując odebrane dane jako argument.
- 3) Serwer otrzymuje dane i zwraca odpowiedź.
- 4) Wtyczka B odbiera dane od serwera i odsyła je do klienta wywołując funkcję `send()` wtyczki A.
- 5) Klient otrzymuje odpowiedź.

Listy zawsze indeksowane są od zera. Jeśli lista zawiera wtyczki A i B to wtyczka A dostępna jest pod indeksem 0 a wtyczka B pod indeksem 1.

0x03 [PROGRAMOWANIE]

Nadszedł czas przełożyć wyżej obmyślony algorytm elektroniczny na kod w języku Python. Głównym elementem serwera będzie wyżej już opisana "wtyczka" i to właśnie nią zajmniemy się w pierwszej kolejności.

Z punktu widzenia języka Python, wtyczka jest obiektem wskazującym na jakąś klasę. Dla przypomnienia.. Klasa jest strukturą danych zawierającą zarówno zmienne jak i funkcje do których można się odwołać za pośrednictwem operatora dostępu.

Przykład:

```
#!/usr/bin/python

class new_kwadrat:
    def __init__(self):
        self.bok = 0

    def Pole(self):
        return self.bok * self.bok

kwadrat = new_kwadrat()
kwadrat.bok = 2
print kwadrat.Pole()
```

W definicjach klas używa się słowa kluczowego "self" oznaczającego, że

obiekt odwołuje się do samego siebie.

Czas na wtyczkę..

Zmienne obiektu wtyczki:

```
class new_plug_in:
    def __init__(self):
        self.sock = 0
        self.send_to = 1
        self.plugins = []
        self.description = ''
```

sock - gniazdo na którym wtyczka będzie operować.

send_to - numer wtyczki równoległej w liście plugins[].

plugins[] - lista zawierająca wskaźniki do wtyczek równoległych.

description - opis.

Funkcje obiektu wtyczki:

```
def Run(self):
    if(len(self.plugins) == 0):
        self.plugins.append(self)
    try:
        s = socket(AF_INET, SOCK_STREAM)
        s.connect((out_addr, out_port))
    except:
        s.close()
        self.sock.close()
        return
    tunnel_out = new_plug_in()
    tunnel_out.sock = s
    tunnel_out.send_to = 0
    self.plugins.append(tunnel_out)
    self.description = '[CLIENT]'
    tunnel_out.description = '[SERVER]'
    tunnel_out.plugins = self.plugins
    tunnel_out.Run()
    start_new_thread(self.Recv, ())
```

Funkcja Run() jest najważniejszą funkcją wtyczki.

Pełni ona rolę inicjacji wtyczki wyzwalającej jak i wtyczki wyzwalanej.

W pierwszej kolejności funkcja ustala, czy należy do wtyczki wyzwalającej czy wyzwalanej przez

sprawdzenie długości listy plugins[]. Jeśli długość tej listy równa jest zeru znaczy to, że

lista wtyczek równoległych jest pusta i funkcja musi zainicjować wtyczkę równoległą.

Funkcja dodaje wskaźnik "własnej" wtyczki do "własnej" listy plugins[] po czym

próbuję nawiązać połączenie z serwerem docelowym. W przypadku niepowodzenia

zamyka ona gniazda strumieniowe i kończy wątek. Jeśli połączenie zostało nawiązane, funkcja

tworzy wtyczkę równoległą tunnel_out. Następnie wtyczka tunnel_out otrzymuje deskryptor gniazda

strumieniowego (sock = s) oraz numer wtyczki równoległej w liście plugins[] (send_to = 0).

W dalszej kolejności następuje dodanie do "własnej" listy plugins[] wskaźnika do wtyczki

równoległej tunnel_out. Wtyczka wyzwalająca dodaje "sobie" opis, że obsługuje stronę

klienta a wtyczka wyzwalana stronę serwera docelowego po czym następuje przekazanie

listy plugins[] wtyczce wyzwalanej przez wtyczkę wyzwalającą. W ten sposób dwie

wtyczki operują na tej samej liście plugins[]. Na koniec uruchamiania jest wtyczka wyzwalana

tunnel_out funkcją Run(). Wtyczka wyzwolona posiada już listę plugins[] zatem

obie te wtyczki w tym samym czasie zaczną oczekiwać na dane startując z nowego wątku

"własną" funkcję Recv().

Wiem, że może wydawać się to niezrozumiałe ale każdy, kto zna podstawy programowania w jakimś języku na pewno zrozumie w jaki sposób działa ten mechanizm ;)

Dalej jest już z górki..

```
def Recv(self):
    while(1):
        try:
            data = self.sock.recv(LEN_RECV)
            if(len(data) == 0):
                self.CloseTunnel()
                return
            print self.description
```

```

        print HexDump(data)
        self.plugins[self.send_to].Send(data)
    except:
        self.CloseTunnel()
    return

```

Funkcja Recv() jest odpowiedzialna za odbieranie danych i przekazywanie ich do wtyczki równoległej a właściwie za wywołanie funkcji Send() wtyczki równoległej przekazując jej dane jako argument. W przypadku wystąpienia błędu, wywoływana jest funkcja CloseTunnel().

```

def Send(self, data):
    try:
        self.sock.send(data)
    except:
        self.CloseTunnel()

```

Funkcja Send() wysyła dane do hosta, z którym połączone jest gniazdo. W przypadku wystąpienia błędu, wywoływana jest funkcja CloseTunnel().

```

def CloseTunnel(self):
    self.plugins[self.send_to].sock.shutdown(1)
    self.plugins[self.send_to].sock.close()

```

Funkcja CloseTunnel() zamyka gniazdo wtyczki równoległej.

Funkcje serwerowe:

```

def AcceptConnect(cl, addr):
    print "Connection accepted from: %s" % (addr[0])
    tunnel_in = new_plug_in()
    tunnel_in.sock = cl
    tunnel_in.Run()

```

Funkcja AcceptConnect() przyjmuje nowe połączenie, tworząc nową wtyczkę wyzwalającą tunnel_in.

```

def InitServer(bind_addr, bind_port):
    s = socket(AF_INET, SOCK_STREAM)
    s.bind((bind_addr, bind_port))
    print "Listening on %s:%d..." % (bind_addr, bind_port)
    s.listen(1)
    while(1):
        cl, addr = s.accept()
        start_new_thread(AcceptConnect, (cl, addr,))
    s.close()

```

Funkcja InitServer() inicjuje nasłuchiwanie serwera na określonym porcie. Gdy połączenie zostanie odebrane, z poziomu nowego wątku uruchamiana jest funkcja AcceptConnect, której przekazywany jest deskryptor gniazda nowego połączenia oraz adres klienta.

```

def HexDump(data):
    a = -1
    out = '-' * 59 + '\n'
    out += '0x00000000: '
    for i in range(0, len(data)):
        a += 1
        if(a == 16):
            out += '\n0x%08x: ' % (i)
            a = 0
        out += '%02x ' % (ord(data[i]))
    out += '\n' + ('-' * 59)
    return out

```

Funkcja HexDump() odpowiada za wyświetlanie danych przesyłanych przez wtyczki w systemie 16-stkowym.

Zmienne globalne:

LEN_RECV - określa w bajtach maksymalny rozmiar pojedynczego strumienia danych.
in_addr - określa adres IP, na którym "tunel" będzie oczekiwał na połączenie.
in_port - określa port TCP, na którym "tunel" będzie oczekiwał na połączenie.
out_addr - określa adres IP serwera docelowego.
out_port - określa port TCP serwera docelowego.

Kompletny kod źródłowy:

```
#!/usr/bin/python
# TCP tunnel 1.0 coded by h07 (C) 2007
##

from thread import start_new_thread
from socket import *

LEN_RECV = 262144

in_addr = '0.0.0.0'
in_port = 80

out_addr = '64.233.183.147'
out_port = 80

class new_plug_in:
    def __init__(self):
        self.sock = 0
        self.send_to = 1
        self.plugins = []
        self.description = ''

    def CloseTunnel(self):
        self.plugins[self.send_to].sock.shutdown(1)
        self.plugins[self.send_to].sock.close()

    def Send(self, data):
        try:
            self.sock.send(data)
        except:
            self.CloseTunnel()

    def Recv(self):
        while(1):
            try:
                data = self.sock.recv(LEN_RECV)
                if(len(data) == 0):
                    self.CloseTunnel()
                    return
                print self.description
                print HexDump(data)
                self.plugins[self.send_to].Send(data)
            except:
                self.CloseTunnel()
                return

    def Run(self):
        if(len(self.plugins) == 0):
            self.plugins.append(self)
            try:
                s = socket(AF_INET, SOCK_STREAM)
                s.connect((out_addr, out_port))
            except:
                s.close()
                self.sock.close()
                return
            tunnel_out = new_plug_in()
            tunnel_out.sock = s
            tunnel_out.send_to = 0
            self.plugins.append(tunnel_out)
            self.description = '[CLIENT]'
            tunnel_out.description = '[SERVER]'
            tunnel_out.plugins = self.plugins
            tunnel_out.Run()
            start_new_thread(self.Recv, ())

def HexDump(data):
    a = -1
    out = '-' * 59 + '\n'
    out += '0x00000000: '
    for i in range(0, len(data)):
        out += '%02x' % ord(data[i])
        if i % 16 == 15:
            out += '\n'
        else:
            out += ' '
    out += '\n'
```

```

    a += 1
    if(a == 16):
        out += '\n0x%08x: ' % (i)
        a = 0
    out += '%02x ' % (ord(data[i]))
out += '\n' + ('-' * 59)
return out

def AcceptConnect(cl, addr):
    print "Connection accepted from: %s" % (addr[0])
    tunnel_in = new_plug_in()
    tunnel_in.sock = cl
    tunnel_in.Run()

def InitServer(bind_addr, bind_port):
    s = socket(AF_INET, SOCK_STREAM)
    s.bind((bind_addr, bind_port))
    print "Listening on %s:%d..." % (bind_addr, bind_port)
    s.listen(1)
    while(1):
        cl, addr = s.accept()
        start_new_thread(AcceptConnect, (cl, addr,))
    s.close()

InitServer(in_addr, in_port)

```

0x04 [TEST]

Nadszedł czas przetestować powyższy "twór", tunelując połączenie z lokalnej maszyny do serwera google.pl. W tym celu wprowadzamy adres serwera google (64.233.183.147) do zmiennej out_addr. Porty (in_port, out_port) ustawiamy na 80 (80/HTTP)..

```

in_addr = '0.0.0.0'
in_port = 80
out_addr = '64.233.183.147'
out_port = 80

```

Uruchamiamy skrypt, w konsoli otrzymujemy następujący komunikat..

```
Listening on 0.0.0.0:80...
```

Teraz uruchamiamy przeglądarkę internetową i w polu adres wpisujemy http://localhost/. W konsoli funkcja HexDump() wyświetla "płynące" dane a w przeglądarce pojawia się strona google.pl.

```

Connection accepted from: 127.0.0.1
[CLIENT]
-----
0x00000000: 47 45 54 20 2f 20 48 54 54 50 2f 31 2e 31 0d 0a
0x00000010: 48 6f 73 74 3a 20 6c 6f 63 61 6c 68 6f 73 74 0d
0x00000020: 0a 55 73 65 72 2d 41 67 65 6e 74 3a 20 4d 6f 7a
0x00000030: 69 6c 6c 61 2f 35 2e 30 20 28 57 69 6e 64 6f 77
0x00000040: 73 3b 20 55 3b 20 57 69 6e 64 6f 77 73 20 4e 54
0x00000050: 20 35 2e 31 3b 20 70 6c 3b 20 72 76 3a 31 2e 38
0x00000060: 2e 31 2e 33 29 20 47 65 63 6b 6f 2f 32 30 30 37
0x00000070: 30 33 30 39 20 46 69 72 65 66 6f 78 2f 32 2e 30
0x00000080: 2e 30 2e 33 0d 0a 41 63 63 65 70 74 3a 20 74 65
0x00000090: 78 74 2f 78 6d 6c 2c 61 70 70 .....

[SERVER]
-----
0x00000000: 48 54 54 50 2f 31 2e 31 20 32 30 30 20 4f 4b 0d
0x00000010: 0a 43 61 63 68 65 2d 43 6f 6e 74 72 6f 6c 3a 20
0x00000020: 70 72 69 76 61 74 65 0d 0a 43 6f 6e 74 65 6e 74
0x00000030: 2d 54 79 70 65 3a 20 74 65 78 74 2f 68 74 6d 6c
0x00000040: 3b 20 63 68 61 72 73 65 74 3d 55 54 46 2d 38 0d
0x00000050: 0a 53 65 74 2d 43 6f 6f 6b 69 65 3a 20 50 52 45
0x00000060: 46 3d 49 44 3d 33 63 63 31 64 31 32 33 62 34 32
0x00000070: 37 61 36 64 31 3a 54 4d 3d 31 31 37 38 39 37 37
0x00000080: 35 31 36 3a 4c 4d 3d 31 31 37 38 39 37 37 35 31
0x00000090: 36 3a 53 3d 6a 37 70 30 75 5f .....

```

Dzięki opisom, którymi dysponują wtyczki widzimy, które dane pochodzą od klienta a które od serwera. Za pomocą programu Process Explorer możemy zaobserwować, jak "tunel" zarządza nowymi wątkami oraz połączeniami TCP. Mechanizm ten jest bardzo wydajny o czym świadczy niskie zużycie mocy obliczeniowej procesora oraz ilości pamięci. Tunel możemy bez problemu uruchomić również w systemach UNIX'owych bez potrzeby modyfikacji czegokolwiek w kodzie źródłowym.

0x05 [OUTRO]

Art ten powstał od tak sobie, chciałem coś skrobnąć i skrobnałem ;]
Taka forma odpoczynku od audytu systemów operacyjnych i innego softu.
Ciężka to robota i niewdzięczna więc trzema słowami kończę, "Być nie mieć".

EoF ;