Dylan Hutchison
CPE 490 Final Project
due 10 December 2012

PPP: Power Proxy Project

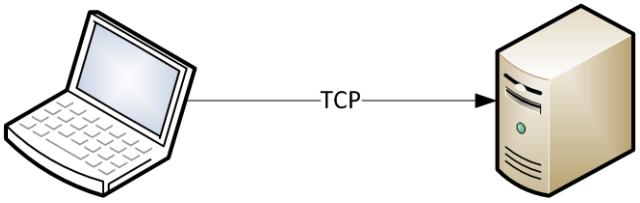
Abstract

I present an application suite that uses an Apache webserver to provide outside access to servers inside a firewalled network. The suite consists of a few PHP scripts served by the webserver, a C proxy program that will act as the middleman agent forwarding data between the client and server, and another C proxy program running on the client to automate the connection process. After implementing the suite and testing it on a few servers inside the Stevens network, it is clear that end product is effective, offering little overhead and allowing connection to any server within the Stevens network from outside the Stevens network.

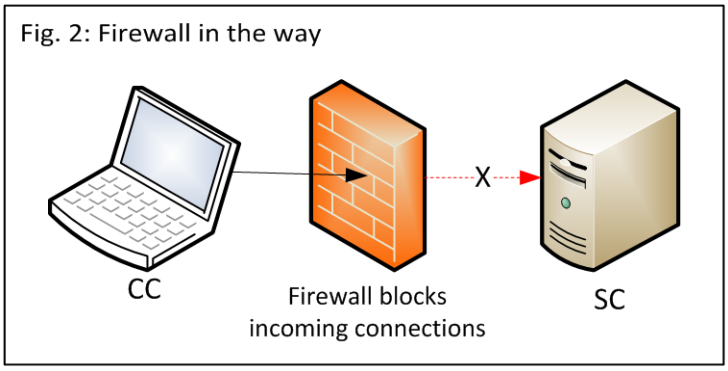
Introduction

Many students on the Stevens campus desire to host a server using their laptops or personal desktops. The range of uses for a server is vast; students may want to host a file server for easy access to files, a web server to host a personal website, a media server to stream music and videos, a game server to host multiplayer games, or simply allow remote connections to their computer so that they could work on it and use its processing power from another location. See figure 1 for the base case.

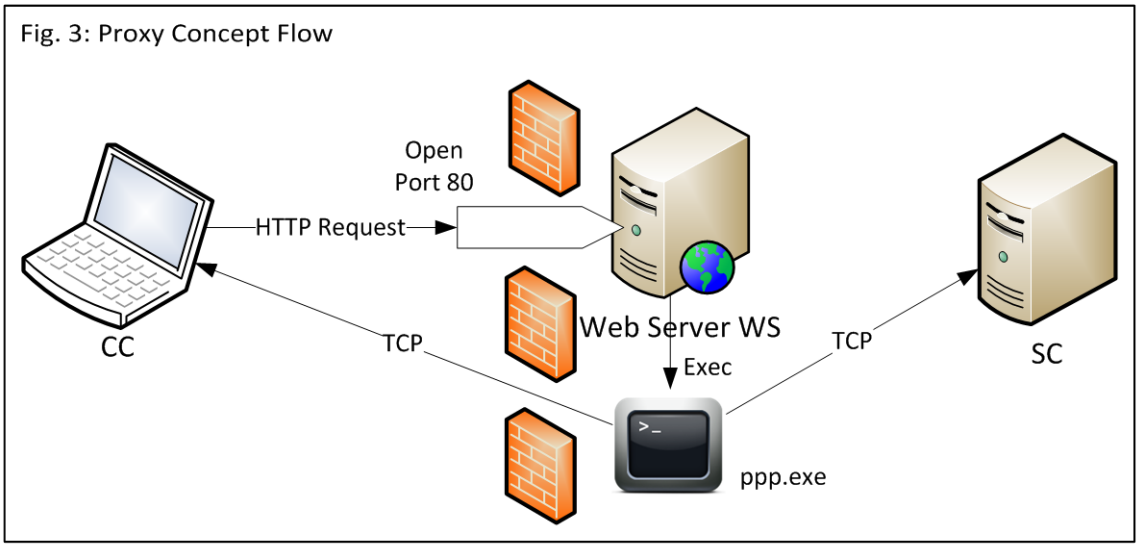
Fig. 1: Ideal Case – Direct Connection



Unfortunately, students with the desire to host a server of any type inside the Stevens network are subject to a restriction: only computers inside the Stevens network can access their server. This is due to a firewall implemented by Stevens IT, which blocks incoming connections to computers within Stevens not approved for public hosting (note that outgoing connections are never blocked). Of course, Stevens IT made a wise decision to use the firewall because it prevents a number of cybersecurity threats to Stevens resources, but some students still wish to provide access to their servers to clients outside of the Stevens network. See figure 2 for an illustration of the problem.



My solution allows clients outside the Stevens network to access servers inside the Stevens network by exploiting web hosting available to any student with a Linux Lab account. Students only need create a publicly readable `cs_html` folder in their home directory and files within it will be hosted at the address `http://www.cs.stevens.edu/~USERID`, where `USERID` is the student's Pipeline ID. The web server provides both static HTML and dynamic PHP file hosting. I make use of the latter file type as a sort of 'backdoor' into the Stevens network from outside. Note that this approach is not unique to Stevens; one can implement it on any firewalled network that hosts an Apache web server. See Figure 3 for the overall approach concept and the following section for a detailed explanation of the implementation.



Proxy Procedure

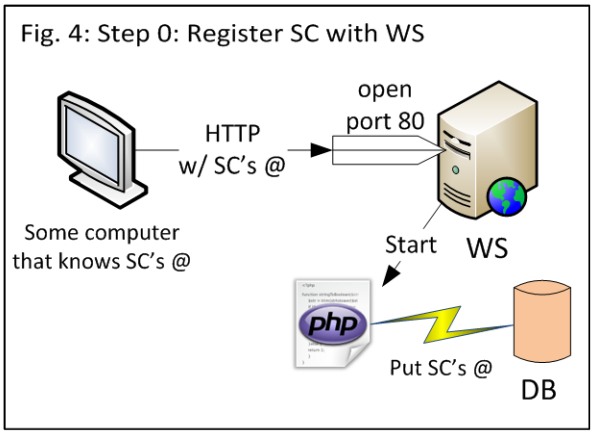
For the sake of example, let's assume you would like to ssh into a server within the Stevens network from a laptop outside the Stevens network using the Putty client. To clear any confusion of terminology, here are the different aspects of the suite:

- CC = Client Computer = your laptop outside the Stevens network
- CP = Client Program = putty running on your laptop
- WS = Webserver = the Apache Webserver running on the Stevens Linux Lab

- SC = Server Computer = the server you within the Stevens network you want to connect to. When I ran this scenario, I asked a friend of mine who has a desktop in the Stevens network to give me an account on his desktop Linux machine I could ssh into.

Portions of the networking code are adapted from the book TCP/IP Sockets in C Practical Guide for Programmers, 2nd Edition (2009) by Michael Donahoo and Kenneth Calvert.

Step 0: Registering a server



Before trying to make a connection with a SC inside the network, register the server with the WS. Do this by sending a quick HTTP GET request to

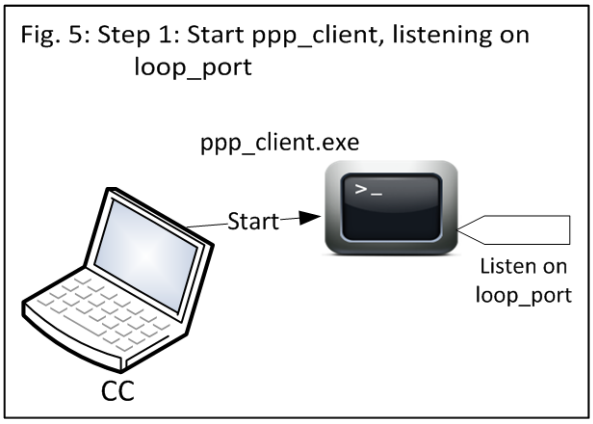
```
http://www.cs.stevens.edu/~dhutchis/reg_server.php?n=NAME&a=ADDR&p=PORT
```

where `NAME` is the name to register the SC under, `ADDR` is the internal IP address of the SC, and `PORT` is the service on the SC to make available publicly. `NAME` will be used later on by the client to identify which server he would like to connect to, allowing the web server to allow connections to multiple servers at the same time. To meet restrictions of the file format I created, `NAME` can be no longer than 15 characters. Note that making this registration need not be done from the SC; any computer knowing the internal IP address of the SC can do it.

Upon validating the information provided in the GET request, the PHP script `reg_server.php` calls the function `put_CB_info` in the included script `file_info.php`, which will update the file `info/back_info.txt` with the new server registration. One can also modify an existing registration entry by providing new info or delete a registration entry by omitting the address and port information. I use a fixed width format to lay out the IP address and port information of each registered server, with a different server registration on each line.

Think of the above process as a primitive database. With more development time and higher usage demands, one could upgrade the file system implementation to a full SQL database connection with substantially higher throughput and lower access time. For the project, using the file system to store server information is good enough. Another avenue for improvement is encrypting the back_info.txt file; currently anyone can access the file and read the server information on it.

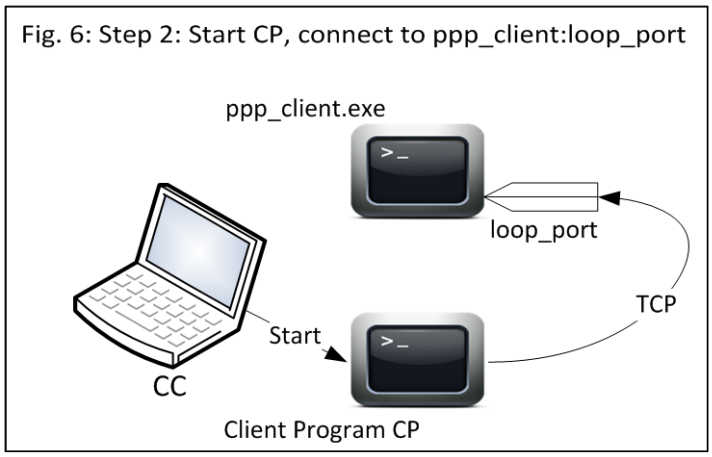
Step 1: Running ppp_client on the CC



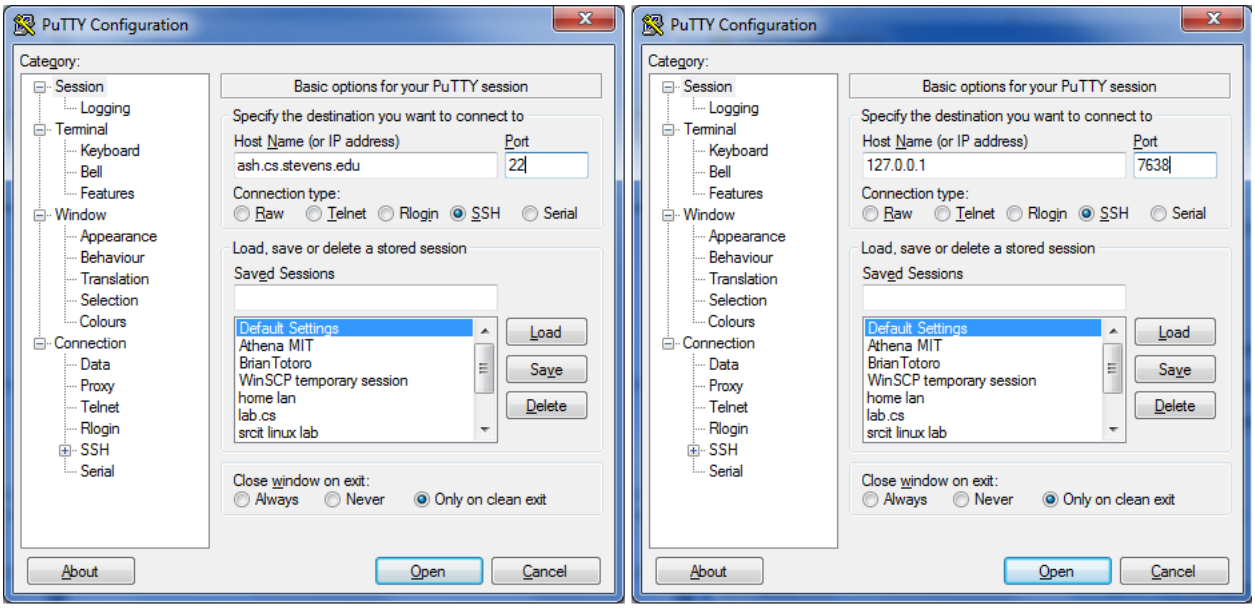
Go into the directory ppp_client on the client computer (CC). Compile the program by running `make`. The only external library referenced by the program is *libcurl* (http://curl.haxx.se/libcurl/). Luckily many popular Linux distributions include libcurl by default.

Then execute `./ppp_client.exe name_server php_port loop_port`, where `name_server` is the name of the server you want to connect to as defined in step 0, `php_port` is any available port on your system for opening that the WS can connect to, and `loop_port` is a port you need the CP to connect to locally. `ppp_client` will listen on the `loop_port` for the incoming CP connection first. The reasoning behind this step will become clear in the next:

Step 2: Start the CP and connect to localhost:loop_port

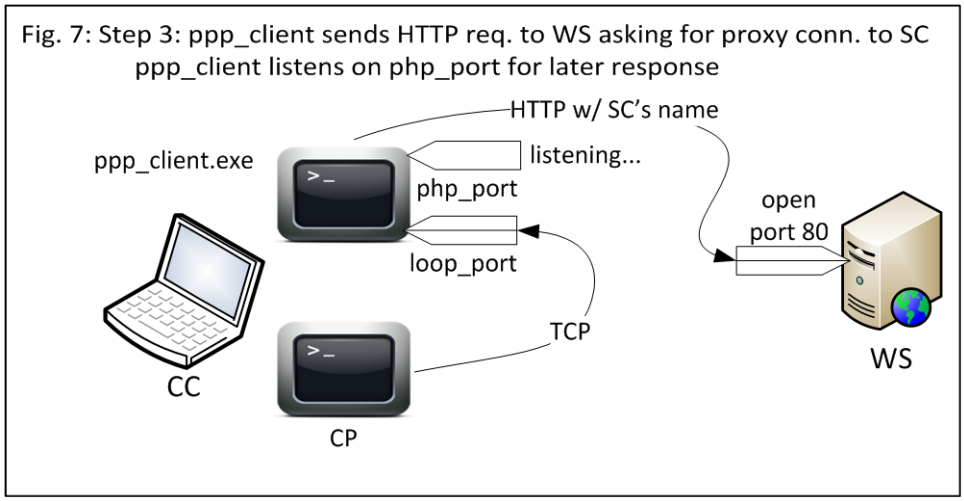


Ordinarily the CP creates an active TCP connection to the SC, which is waiting to accept the connection passively. However, we need the proxy on the WS to actively establish a TCP connection to the CP in order to take advantage of the fact that the Stevens firewall allows outbound connections. Therefore, the `ppp_client` program is necessary to act as another middleman that passively accepts a connection from the CP and the WS and forwards data from one to the other. Here are two screenshots that illustrate the change required on the CP going from a normal direct connection to one routed through `ppp_client` (note that loop_port is 7638).



Direct Connection **Connection to localhost:7638**

Step 3: ppp_client requests a proxy connection from WS



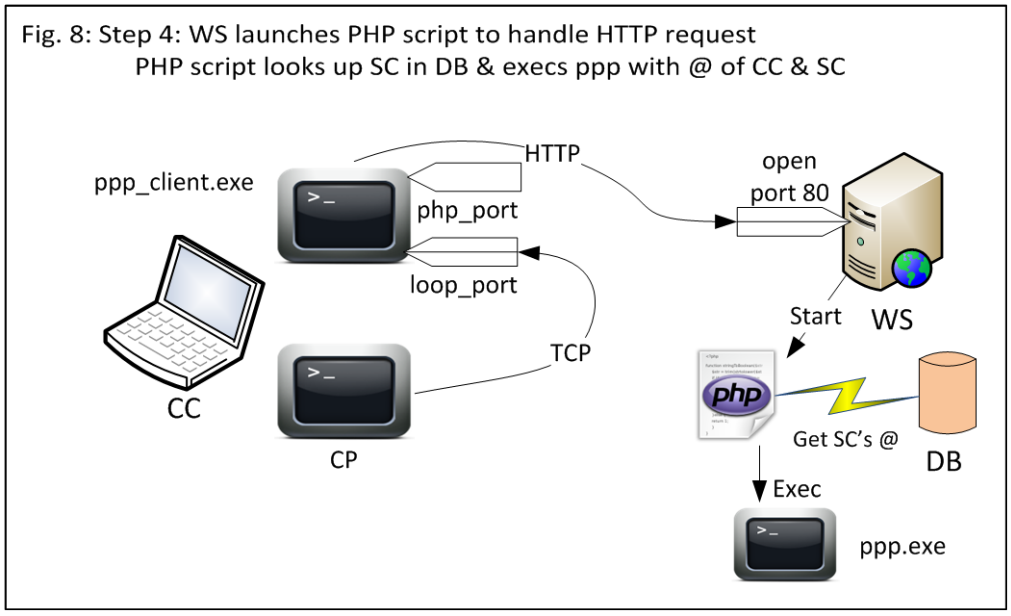
After accepting the connection from the CP on `loop_port`, `ppp_client` leverages the powerful libcurl library to send an HTTP request to the WS at the URL

`http://www.cs.stevens.edu/~dhutchis/make_conn.php?n=NAME&p=PHP_PORT`

where `NAME` and `PHP_PORT` are the inputs to `ppp_client`. While the WS handles this request, `ppp_client` will begin listening for the connection from `ppp.exe` on `php_port`.

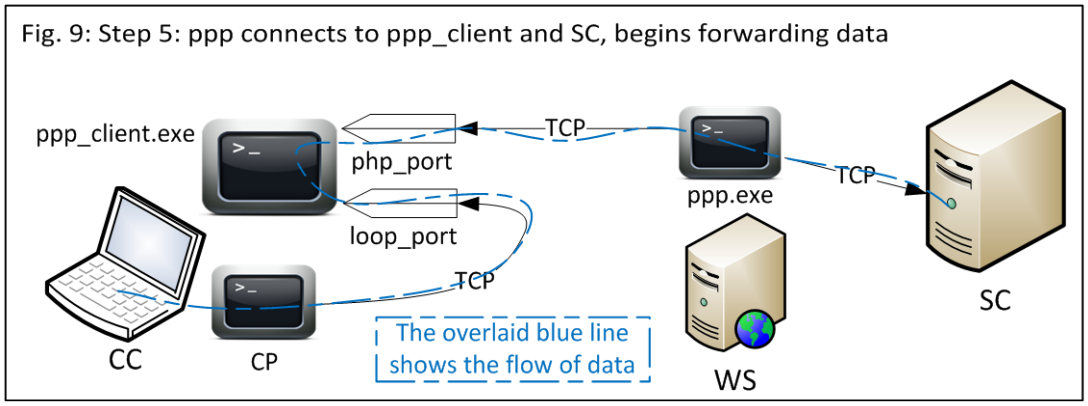
This illustrates an important limitation of the suite: the CC must be able to accept an incoming connection from the WS. This is not possible if the CC is located behind another firewall. On home network connections one can get around this by opening a port on the home router and forwarding the port to the CC. At a business location, this may not be possible. With both the SC and the CC behind a firewall, one has little hope of establishing a connection between them.

Step 4: WS responds to HTTP request and launches ppp.exe



When the WS receives the HTTP request for `make_conn.php`, it launches a thread to execute `make_conn.php`. The PHP script validates the input server name and port and assumes the IP address of the CC is the same as the IP address making the request (found under the PHP superglobal variable `$_SERVER["REMOTE_ADDR"]`). Then the PHP script queries the DB (in this case, the file `info/back_info.txt`) for the address and port information of the requested server. With all the necessary information, `make_conn.php` execs `ppp.exe`, passing the connection information of the CC and SC.

Step 5: ppp and ppp_client complete the data exchange



`ppp.exe` creates a TCP connection to the SC and `ppp_client` on the CC listening on `php_port`. It then creates an extra thread (for a total of two threads); the main thread accepts data from the SC and forwards it to the CC while the extra thread does the same thing in reverse. The program uses an 8kB buffer for temporary storage before forwarding. `ppp` will continue trafficking data between the two machines until one of them closes the connection, after which it will safely die. `ppp_client.exe` will follow the same pattern, trafficking data between the CP on the same machine and `ppp` on the WS.

With all connections created and both proxy programs running, data can freely flow between the CC and the SC! There is very little overhead too, since all the proxy programs do is forward data from one connection to another.

Extensions

Here are a few ways one could take the project for further improvement:

1. Generalize the sockets to use UDP as well as TCP.
2. Allow IPv6 addressing.
3. Use a better DB system. As noted in step 0 above, a true SQL database is most appropriate for fast information retrieval. In addition, the SQL database could store additional attributes about the registered servers, such as the type of service each server provides. This could be used in a kind of server browser, which allows people to view what servers are currently registered. At the very least, one could relax the restriction of names to 15 characters.
4. Allow for a different URL of the server. Currently my WS address is hardcoded. A more flexible design would load the WS address from a configuration file or user input.
5. Perform some kind of interesting processing at the proxy locations. A practical joke would change all ascii letters in the datastream to their uppercase equivalents, for example. A more pragmatic use would actively compress data leaving the Stevens network. The ppp_client could then uncompress the data. The result is reduced data communication load over the Internet, potentially raising response rates and bandwidth.
6. Design a Graphical User Interface to make the suite as user-friendly as possible.
7. Improve Security! In its present state, one could exploit the suite and introduce security vulnerabilities into the Stevens network by making every internal server accessible externally. Here are a few measures one could take:
    a. Encrypt the `info/back_info.txt` database file.
    b. Require a user to login before establishing a proxy. Modifications to the `ppp_client` program introducing a new encrypted HTTP POST parameter, password, can make this possible. A bold programmer could design an entire user account system with different permission levels.
    c. Restrict the IP address range that `ppp_client` will accept a connection from. For example, the `loop_port` is designed to only accept a connection from the localhost on the loopback address. It may be interesting to consider what would happen if another computer deliberately connected to it instead, potentially forming a chain of proxies among different machines.
    d. Better control the exec'ing of ppp.exe from `make_conn.php`. A malicious attacker could create his own program, name it `ppp.exe`, and place it in the web server directory. The malicious program could then masquerade as the intended `ppp.exe`.