OpenVPN Secure Bridge Implementation

Introduction/Prerequisites

Dave Andrews

This explanation outlines the procedures by which OpenVPN server and client setup was implemented, with a bridge to the local network of an OpenVPN server, so that local network resources can be made available to "road warriors", and offsite employees. We will start with a brief summary of the testing environment, and systems and software involved.

The OpenVPN Server and Bridge was implemented on a Debian Linux Pentium-III computer, with as little as 288MB of RAM. The operating system itself was a bootstrapped installation of Debian (i.e. barebones), for limited memory usage and ultimate control over the software installed. On my internal network, I assigned this server an IP of 192.168.5.178.

Typically in corporate VPN installations, the VPN server is given an external/WAN IP and has two NIC's installed; one for internal network access, and one for external network access – which helps improve network load and associated security. However, for the sake of this test (and because I am dealing mostly with home-networking components), I only have one internal NIC card installed on this server, and I will be port-forwarding the VPN server's listening port through the network gateway, 192.168.5.1 – so all the VPN connections associated traffic will physically reside on the same subnet as the Local Network. Keep in mind that in practice, this is highly discouraged, but as I found out, the actual configuration of this one-NIC setup is hardly different than a two-NIC setup for OpenVPN.

OpenVPN 2.1_rc20 was the OpenVPN version of choice for the server, available as a precompiled package from the Debian unstable tree at the time of this writing. Later on, bridge-utils was used for the actual adapter bridging. According to OpenVPN, this does not run as fast as using a routed solution for connecting two networks for a similar outcome, but bridging allows for Broadcast and ARP packet transmission, which was an objective of my network – and is required for convenient client communications without getting overcomplicated with routing.

(To rephrase this in technical terms, we are building a layer-2 bridge between the a VPN Server implementation, and our local network)

The Client system was running Microsoft Windows XP, with the latest update as of Oct 15, 2009. The precompiled version of OpenVPN 2.1_rc20 was installed on this system. (It should be noted that everything described in this document was also tested successfully with a Vista client in another instance).

Server Configuration

The server was configured according to the official OpenVPN howto, available at http://www.openvpn.net/index.php/open-source/documentation/howto.html . After installation of OpenVPN, the guide instructs administrators to generate a master certificate authority and associated

key, which uses 1024-bit public-key RSA encryption (I believe this is the default recommendation by OpenSSL). Luckily, OpenVPN simplifies the process of keeping uniformity in the keys and certificates, by providing some bash scripts available in /usr/share/doc/openvpn/examples/easy-rsa/2.0 (Debian). Most users copy the easy-rsa folder to /etc/openvpn / folder, so that all generated keys are placed in a convenient path later on. All an administrator has to do is to configure the values of his country, province, city, and various other certificate-specific fields in the easy-rsa's "vars" shell script, and then runs:

```
#. ./vars
#./clean-all
#./build-ca
# ./build-key-server server
```

OpenSSL generates the associated server certificates and keys.

To generate new client key files (Note: This can be done while openvpn is already serving active clients):

```
#./build-key myclientname
```

After the client keys are generated, the OpenVPN administrator must generate Diffie Hellman parameters. From my understanding of the OpenVPN connection process, this is used to provide a means of exchanging secure information (i.e. keys, random data for later security) without having any prior-established security in a new client-server connection.

```
#./build-dh
```

After executing the above scripts, all of the needed keys are placed in the easy-rsa/2.0/keys folder. Who gets what keys/certificates? The following chart amasses a solid answer:

Filename	Needed By	Purpose	Secret
ca.crt	server + all clients	Root CA certificate	NO
ca.key	key signing machine only	Root CA key	YES
dh{n}.pem	server only	Diffie Hellman parameters	NO
server.crt	server only	Server Certificate	NO
server.key	server only	Server Key	YES
client1.crt	client1 only	Client1 Certificate	NO
client1.key	client1 only	Client1 Key	YES

(OpenVPN Technologies)

Debian Server, /etc/openvpn/server.conf

The following **diff-file** outlines the critical differences between the default sample configuration of the OpenVPN server, available on Debian at /usr/share/doc/openvpn/examples/sample-config-files/server.conf.gz . I have omitted the configuration file commenting, and have substituted my own explanations for the server configuration settings:

```
# Use port 443 – hopefully we will be able to use this VPN network through various firewalls, so that company employees that travel to various hotels and institutions will not have firewalled access.
-port 1194
+port 443

# Use TCP instead of default UDP – better odds of having our clients' access appear as though it looks like SSL.
-;proto tcp
-proto udp
+proto tcp
+;proto udp

# "dev tap" We need a layer-2 tunnel. This is because we are using a bridge, and not using routing.
-;dev tap
-dev tun
+dev tap0
+;dev tun

#Certificate Paths. Remember, ours were generated in /etc/openvpn/easy-rsa/2.0/keys
-ca ca.crt
-cert server.crt
-key server.key # This file should be kept secret
+ca /etc/openvpn/easy-rsa/2.0/keys/ca.crt
+cert /etc/openvpn/easy-rsa/2.0/keys/server.crt
+key /etc/openvpn/easy-rsa/2.0/keys/server.key # This file should be kept secret

# Same location for the diffie-hellman keys
-dh dh1024.pem
+dh /etc/openvpn/easy-rsa/2.0/keys/dh1024.pem

# Disable server-mode. Note: this means that we are not creating a non-existent subnet with the openvpn process – but instead, we are routing the packets through a bridge (more on that later).
-server 10.8.0.0 255.255.255.0
+;server 10.8.0.0 255.255.255.0

# Provide the bridged network's gateway, subnet, and the range of IP addresses that the OpenVPN process will be assigning to the VPN clients.
;server-bridge 10.8.0.4 255.255.255.0 10.8.0.50 10.8.0.100
+server-bridge 192.168.5.1 255.255.255.0 192.168.5.200 192.168.5.254
```

Instruct connecting client that we are replacing their default gateway (note: this takes place via client-side routing tables).
+push "redirect-gateway"

Instruct the client that we are offering a DNS Server to our client adapter.
;push "dhcp-option DNS 208.67.222.222"
;push "dhcp-option DNS 208.67.220.220"
+push "dhcp-option DNS 192.168.5.1"

Allow client-to-client communications (i.e. client 192.168.5.200 pings client 192.168.5.201)
;client-to-client
+client-to-client

Lower max-clients – this is just because our test server has only an Intel P-III processor.
-;max-clients 100
+max-clients 10

Enable Logging (Good for debugging)
+log openvpn.log
;log-append openvpn.log

Enable Usernames and passwords after certificate exchange. This is just a bash script of my own making – and is NOT required for this setup to work. All it does it checks the contents of the $username and $password environment variables, and ensures that they are acceptable for the $common-name environment variable. (i.e. if certificate is “john” and supplied username is “johnanderson” and supplied password is “12345”, return 0, else return 1)
+auth-user-pass-verify /etc/openvpn/openvpnsignin via-env

Allow transmission of passwords as environment variables to the above auth-user-pass-verify script.
+script-security 3

#Learn-Address script. We will use this later in ARP-poison prevention, but for the moment, this is just a blank bash script. This gets called whenever a new client MAC address is acquired. Returning a ‘0’ from this script indicates that the client address can be learned, while a ‘1’ indicates that the client MAC address should not be learned. (More on this later)
+learn-address /etc/openvpn/test.sh

Debian Server, Bridge Configuration:

Now that the Debian VPN Server is configured, there is now a means by which the OpenVPN process can communicate with a kernel device called a "tap", which is essentially a layer-II device that exchanges Ethernet frames. This tap, at the moment, does not go anywhere, and does not do anything except create a dead-end for anything that comes out of it from OpenVPN clients. By our configuration above, OpenVPN is capable of directing packets between clients, but they do not have a means by which they can reach the physical LAN.

Reaching the VPN Server's physical LAN is accomplished with a **bridge**, between eth0 (our physical network adapter) and tap0 (our OpenVPN process' tap).

Luckily, rather than having to configure the bridge-utils package separately, OpenVPN has once again created some example scripts for constructing a bridge between the above two adapters, in /usr/share/doc/openvpn/examples/sample-scripts/ . The two that we will be using, are bridge-start and bridge-stop.

Again, all that has to be configured are some variables within each script:

```
#bridge-start
#--------------
        # Define Bridge Interface
        br="br0"

        # Define list of TAP interfaces to be bridged,
        # for example tap="tap0 tap1 tap2".
        tap="tap0"

        # Define physical ethernet interface to be bridged
        # with TAP interface(s) above.
        eth="eth0"
        #The IP Address of the VPN Server on the internal network that we are bridging to.
        eth_ip="192.168.5.178"
        eth_netmask="255.255.255.0"
        eth_broadcast="192.168.5.255"

#bridge-stop
#---------------
        # Define Bridge Interface
        br="br0"

        # Define list of TAP interfaces to be bridged together
        tap="tap0"
```

Debian Server, Turning on the OpenVPN Server and Process:

```
./bridge-start
# (Note: you may want to make sure that your new bridge has an IP and appropriate routes here).
/etc/init.d/openvpn start
```

Client Configuration

The OpenVPN Windows Client configuration is less cumbersome than the server configuration. All configuration takes place in a configuration file name of your choosing, located in:

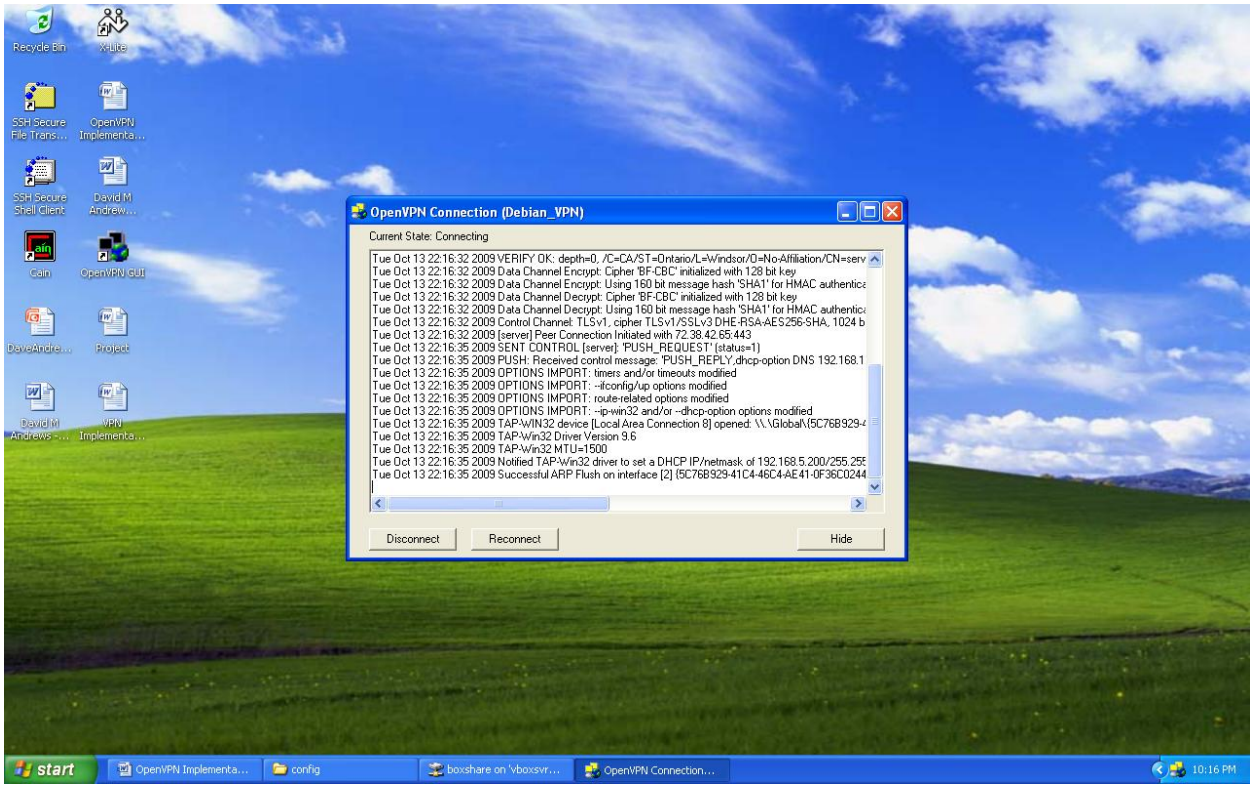
C:\Program Files\OpenVPN\config*.ovpn

The following diff indicates the critical differences between the default sample client configuration script, and our implementation:

```
# Instruct our client to use tap, rather than tun (for layer-2 frames, and bridging, not just layer-3).
-;dev tap
-dev tun
+dev tap0
+;dev tun

#User TCP - many firewalls (including the University of Windsor's) are much kinder to TCP Sessions.
-;proto tcp
-proto udp
+proto tcp
+;proto udp

 # Configure our server's address and port.
-remote my-server-1 1194
 ;remote my-server-2 1194
+remote hackers.servegame.org 443

#We called our client "client1", so the key and certificate was named accordingly.
ca ca.crt
-cert client.crt
-key client.key
+cert client1.crt
+key client1.key

#Tell the login script to ask for a username and password, in accordance to our server's configured expectation (Our server earlier was instructed to validate login/passwords, after the key exchange is accepted).
+auth-user-pass
```

Connecting a Client

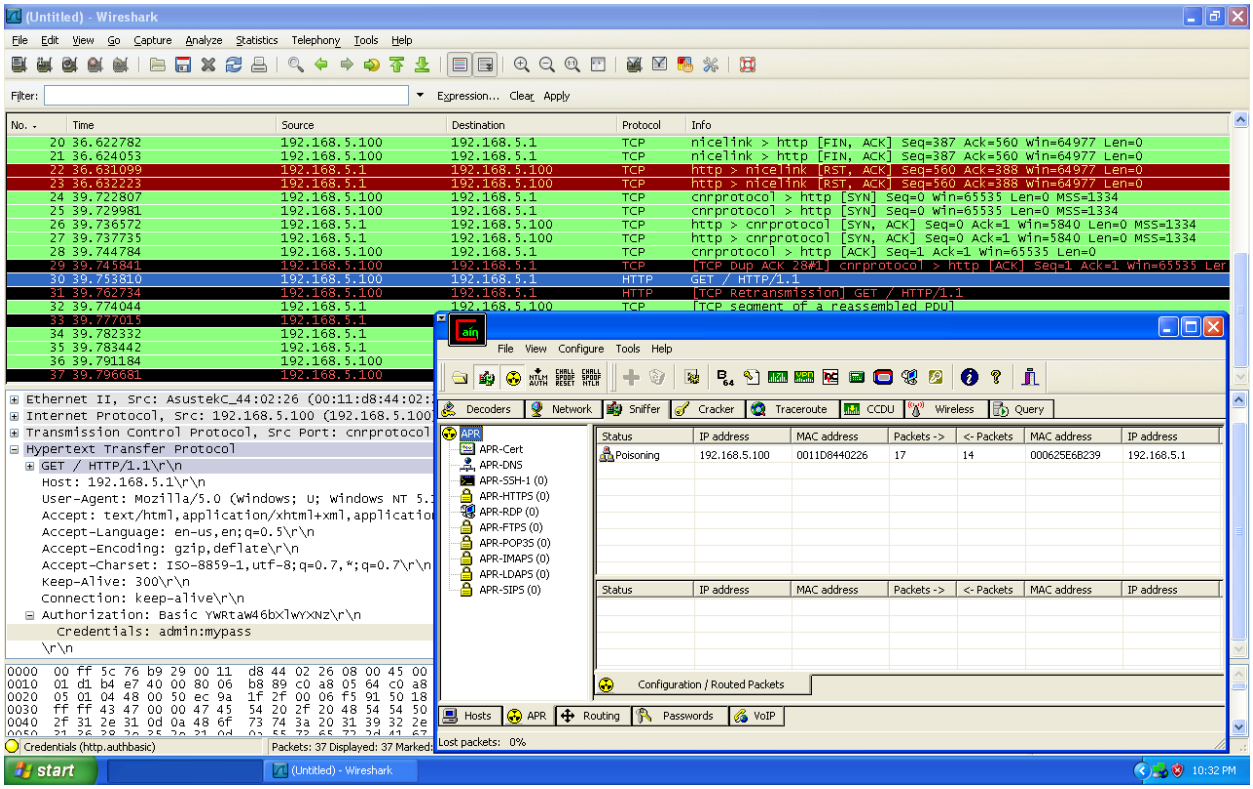
Without further ado, the above is enough to begin a VPN bridge to a LAN server. Please note that the IP addresses that get assignment to clients are actually from the OpenVPN server process – and not from the physical LAN's DHCP server. As a result, it would make sense to allocate a region of the IP addresses on the physical LAN's DHCP server to the OpenVPN process, so that no IP address collisions occur.



ARP Poison – Problem Affirmation

ARP Poisoning is a very common security concern on many networks, and is definitely reason for concern in corporate networks. It is especially a problem when forming connections that do not provide any guarantee that the destination is the expected one – such as in many router configuration pages, sign-in systems, forums, plaintext email servers, and many SIP VoIP phone conversations.

The following image (Client) shows that such an attack is possible in a default OpenVPN implementation:



Traffic was redirected from a physical LAN IP, 192.168.5.100, which attempted to perform a sign-in on the router's configuration page, 192.168.5.1. However, because VPN client 192.168.5.200 was ARP Poisoning 192.168.5.100, saying that 192.168.5.1 was at its' own MAC address, then any traffic destined for 192.168.5.1 from 192.168.5.100 will be redirected to 192.168.5.200 across the VPN connection. The same applies in reverse. You can see that the sample http authorization packet made its' way all the way to the VPN client's computer (above), even through it was destined for the MAC address of the gateway.

As such, ARP poisoning is a very valid concern for corporations implementing layer-II bridged VPN connections on their networks.

Preventing ARP Poisoning

The first thing that needed exploring was how to identify ARP poisoning. This actually depends on the ARP Packets that are being faked.

Typically, a faked ARP packet will be coming from an attacker trying to tell one (or all) machines that the target's IP address is at the attacker's MAC address. Consequentially a simple rule of thumb for preventing ARP poisoning would be to prevent machines identifying themselves as owners of IP-addresses that are not supposed to belong to them.

Because we know that our VPN network would only exist on the 192.168.5.200-192.168.5.254, we can immediately conclude that ARP Packets that make claims for machines outside of this range that are coming from OpenVPN, are not acceptable. However, the problem doesn't stop there.

Supposedly, a client (Alice) within the OpenVPN network; could start talking to an http authentication page, through the bridge, accessible via the gateway. However, a MITM attacker named Mallory, within the OpenVPN network, starts to make an ARP claim for the IP owned by Alice, and sends this claim to the gateway. Now the gateway, thinking it is talking to Alice, will instead be transmitting traffic to Mallory, and Mallory will just redirect the traffic to Alice. Mallory does the same to Alice (about the gateway's IP), and suddenly Mallory has full control over the session.

1. Mallory can pretend to be Alice, for any communications to the bridged LAN.
2. Mallory can pretend to be any communications to the bridged LAN, to Mallory.

While we may have stopped our LAN network from getting ARP poisoned, this doesn't stop clients from poisoning one-another, and performing MITM attacks on each other while talking to the LAN.

Strict Firewall Rules for OpenVPN Clients

A theoretical solution for preventing invalid ARP packets from OpenVPN from reaching the local LAN would involve having a map between client MAC addresses and client IP addresses. If this information were available, we could tell our firewall to accept only ARP messages that make claims for IP/MAC combinations that fit this map.

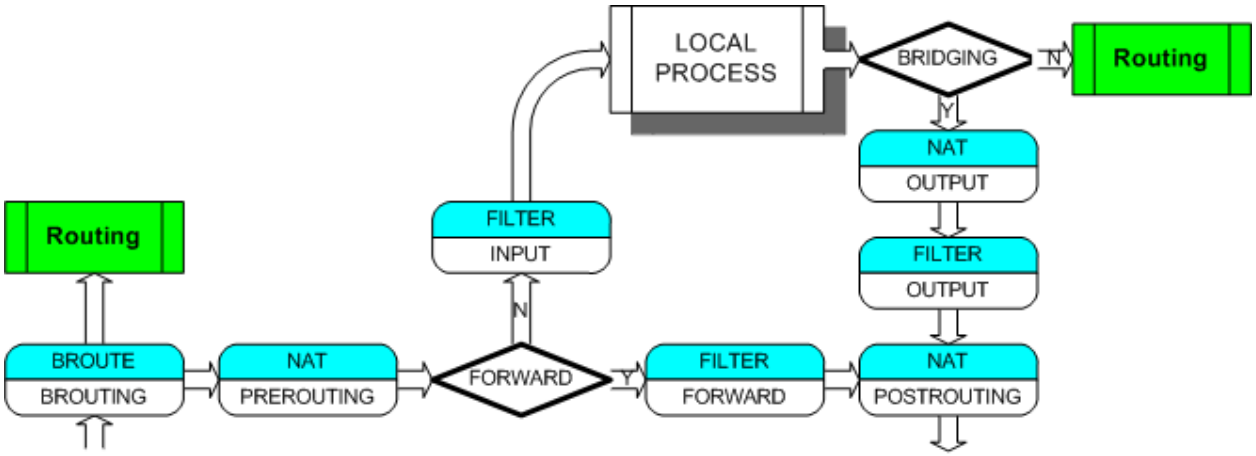
Luckily for us, OpenVPN already maintains a list of MAC address/IP address combinations, and actually passes the MAC/IP combinations to a script of our choosing at the time a client's connection. Recall the line in server.conf: **learn-address /etc/openvpn/test.sh.**

OpenVPN passes the desired action "add/del/modify" as the first argument of this script, along with the MAC address as the second. The IP address resides within an environment variable named $ifconfig_pool_remote_ip. If we add rules to the firewall, while running this script, to automate the process of allowing and forgetting specific ARP packets with IP/MAC claims, then this would be a sound solution to prevent MAC Poisoning of the local network.

The Linux Firewall & Linux BR-NF (Bridge/Netfilter)

One of the first things that Linux administrators think of when they wish to firewall unwanted traffic, is a program named iptables. Iptables is a layer-III firewall, and unfortunately ARP is a type of packet that exists only in Layer-II. This means that layer-II firewall rules will have to be created. For this task, we employ a program named ebtables.

Ebtables is similar to iptables, in that it offers all the same possible chains as well as the logical sequencing of all manipulations. Its operations are actually highly intermixed with the layer-III filters, but typically we can observe ebtables rules occurring logically before any corresponding iptables rule. (I.e. ebtables "FORWARD FILTER" takes place before iptables "FORWARD FILTER"). This allows us to be confident that any use of ebtables will be the first line of defense.



(Bart De Schuymer, 2003)

We can see from this diagram that any ARP prevention will be best placed in the FORWARD filter. Forwarding takes place in this map when the bridge determines that the MAC address destination exists on the other side – and that the MAC destination does not belong to the local machine. We also need to protect the local machine from ARP poisoning, on the INPUT filter.

The following rules can now be added to the ebtables configuration, for ARP Poison prevention:

```
ebtables -A INPUT -i tap0 -p arp --arp-opcode 1 -j ACCEPT
ebtables -A INPUT -i tap0 -p arp -j DROP
ebtables -A FORWARD -i tap0 -p arp --arp-opcode 1 -j ACCEPT
ebtables -A FORWARD -i tap0 -p arp -j DROP
```

These rules will stop any machine from sending any ARP packets, such as “192.168.5.1 is at AA:BB:CC:DD:EE:FF”. Please note that these rules actually stop all ARP traffic.
Now we can modify the test.sh script to add ARP-allow rules to the ebtables configuration for new MAC/IP combinations:

```
echo LEARN SCRIPT
if [ $1 = "add" ]
then
ebtables -I FORWARD 1 -i tap0 -p arp --arp-ip-src $ifconfig_pool_remote_ip --arp-mac-src $2 -j ACCEPT
exit 0
fi
```

This will result in new client-specific ARP rules being added to the firewall, so that clients may only make valid ARP claims that do not poison any LAN caches. They are forced to provide valid IP/MAC combinations that are informed by OpenVPN and enforced by the Linux firewall.

```
Bridge chain: INPUT, entries: 3, policy: ACCEPT
-p ARP -i tap0 --arp-ip-src 192.168.5.200 --arp-mac-src 0:ff:5c:76:b9:29 -j ACCEPT
-p ARP -i tap0 --arp-op Request -j ACCEPT
-p ARP -i tap0 -j DROP
Bridge chain: FORWARD, entries: 3, policy: ACCEPT
-p ARP -i tap0 --arp-ip-src 192.168.5.200 --arp-mac-src 0:ff:5c:76:b9:29 -j ACCEPT
-p ARP -i tap0 --arp-op Request -j ACCEPT
-p ARP -i tap0 -j DROP
```

Learn-address deletions are actually supposed to be just as convenient, however, when I investigated further I discovered that OpenVPN actually doesn't provide the IP address of the remote client at the time of "learn-address" disconnection events. This is actually because OpenVPN has long past eliminated the IP address of a client, and actually delays the learn-address script (for disconnection) until some other internal code has actually established that a client is no longer reachable.

And unfortunately for us, ebtables requires specific rule attributes upon deletion and does not allow generalizations (i.e. delete all ARP rules that apply to this MAC). Without an IP address to delete, ebtables ARP-prevention rules will remain after disconnection!

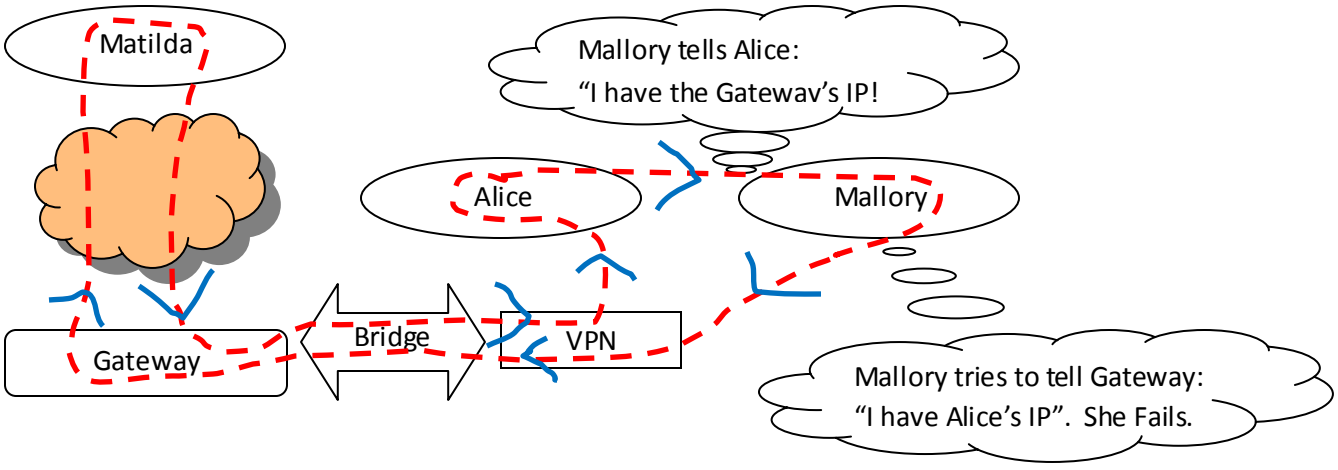
To fix this, I wrote a script that can add, delete, and fetch IP addresses as a function of MAC addresses. It stores the combinations in a file, so that the learn-address disconnection script can fetch an IP associated with a MAC address that is being unlearned (after previously learned/memorized), so that it can tell ebtables the specific attributes of the line to be unlearned. It worked flawlessly!

No More ARP Poisoning the LAN! And…

After testing my configuration, I can confirm that absolutely no fake ARP packets could be transmitted to the physical LAN being bridged to. This means that the LAN is protected from the bridged VPN network. However, there is one **FATAL** flaw that I found. Client-to-client interactions cannot be firewalled! This is because they never leave the OpenVPN process, so the Linux Kernel Firewall never even sees client-to-client communications!

After consulting the OpenVPN mailing lists, I have discovered that they offer no known solution to client-client firewalls in bridged networks. While this means that complete ARP MITM attacks can take place with clients attacking other client-to-client interactions (assuming that our configuration allows client-to-client interactions in server.conf), all hope is not lost for protecting client-<LAN Host> attacks.

For example: Suppose Alice is a VPN client. She attempts to contact a self-signed server on the internet, Matilda. Mallory (VPN) attempts to perform a full-fledged ARP and SSL attack with an application named Cain and Able, against Alice's connection with Matilda self-signed cert. In order to do this, Mallory would attempt to so that she can substitute her own self-signed certificate in the place of Matilda's (She thinks she can!). Because any ARP packets that Mallory would send to the gateway to disguise her as Alice would actually get blocked by our firewall rule, Mallory would not be able to gain full MITM on the Alice-Gateway-Matilda connection. Mallory would only have one-way access.



Potentially, this means that Mallory could try to get away with making small changes, for instance, changing payloads under TCP connections, provided that she knew that the content was still acceptable, that she stayed within the same length (so that TCP does not error on Alice's length count!), and that she set the checksums according to her changes. This means that theoretically, it would be extremely difficult if not impossible to learn or change anything from Alice-Mallory's self-signed connection.

Mallory could also snoop all of Alice's plaintext connections – which is somewhat of a concern.

In all, protecting OpenVPN's internal client-client relations is very much impossible without modifying the whole OpenVPN application code, as it is not currently designed to protect or firewall client-to-client communications. Furthermore, doing so in the OpenVPN user-space process would have a terrible efficiency cost on network performance, and would not be viable. In all, using a layer-3 tunnel for OpenVPN would eliminate the problem of layer-2's ARP attacks, and it would be possible to enforce client-to-client security.

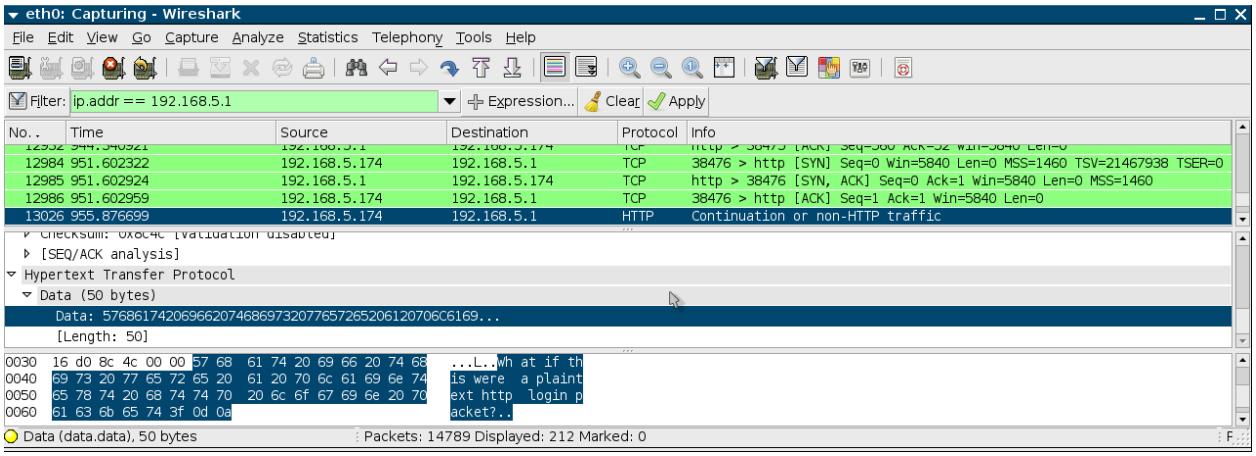
Client MAC Address Cloning…

Because this is a layer-II bridge, suddenly OpenVPN clients can create all the problems that a layer-II identity can create for a network. This includes transmitting packets that trick the bridge into thinking that certain identities lie on the VPN side of the bridge, including the gateway, or other LAN machines.

While this itself is just as much a security concern (for masquerading) as it can be an inconvenience (i.e. stopping the flow of traffic), it can actually be used as a form of snooping information. For instance, suppose that a LAN client, Charlie, is an administrator. Charlie opens up a login page on the gateway, and keys in his login/password. However, just at this moment, Eve, a VPN client, connects to the VPN with a client MAC address that is equal to the Gateway's MAC address. Eve pings the Charlie (or someone else on a switch between Charlie and the gateway), which results in the bridge and any switches that this ICMP request passes through, to await a possible return value (switches/bridges think that they might learn a new MAC address – so if they get a response they change the known location of the source's MAC address). If Charlie or somebody on his switch pings back, Charlie's gateway login packets can get falsely redirected to Eve.

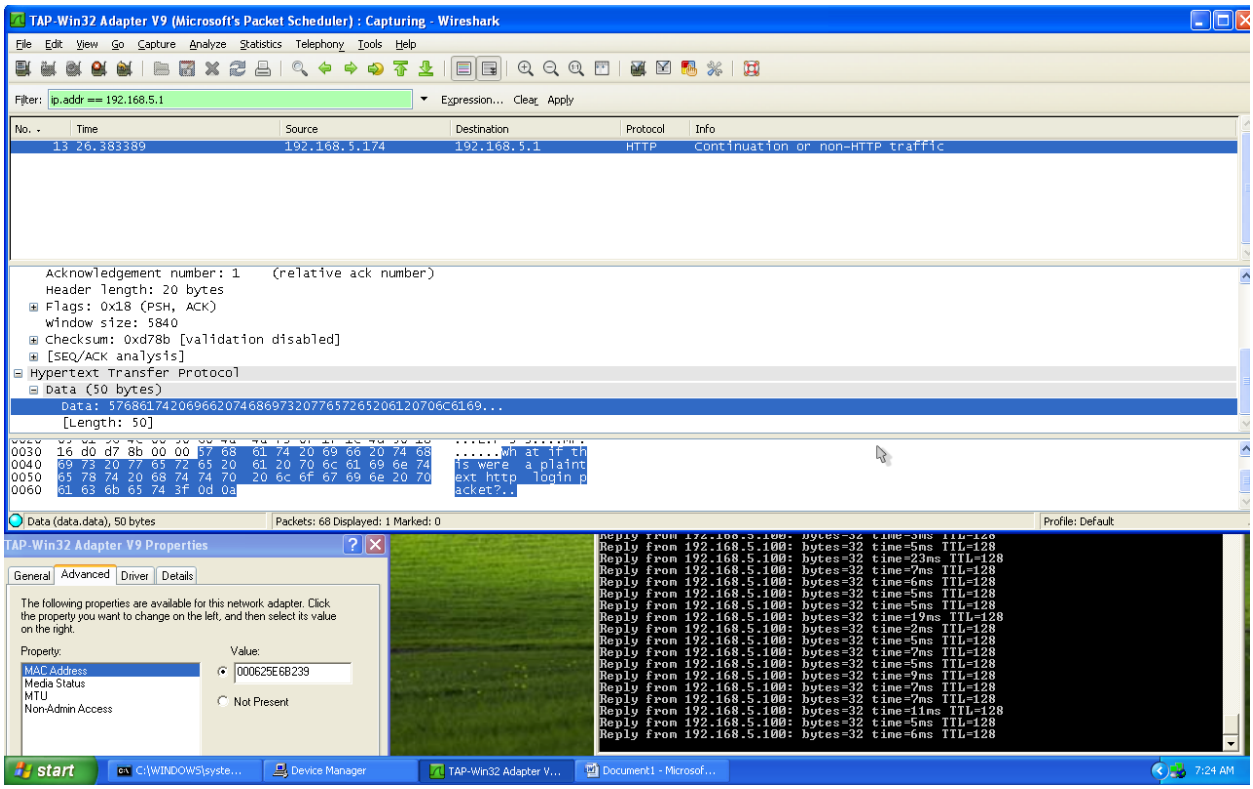
When testing this attack, I found that it is infinitesimally impossible to time a redirect of a "GET with authorization packet" from an browser HTTP session that includes: syn/ack/syn,ack/get-with-authorization/etc that lasts about 0.2 us. So I slowed down the process with a practical demonstration, which forced a longer delay between the Syn,Ack and the transmission of a "GET-with-authorization".

The following was accomplished with telnet (Alice, on LAN, at 192.168.5.174):



Notice that this is not a proper http packet, since it only contains plaintext accomplished with telnet. However, while it is not valid HTTP, it is meant to convey the point that this packet can land in an impersonator's hands, if the imposter were to convince the switches/bridges between it and a switch along Alice's connection, that the gateway's MAC is in its direction…

The following is a view of the Attacker's screen, on the VPN, at 192.168.5.200:



By timing a switch/packet redirect with the cloned MAC address of a gateway, an attacker can stimulate the network into thinking that the gateway is in its direction, and immediately receive any packets that would shortly follow, including the middle of active TCP connections. Theoretically, this illustrates that plaintext login packets can be captured, although only for short periods before TCP connections die out.

So Cloning Exploits are Layer-II-Specific Problems?

It is important to note that these short fallings are not specific to OpenVPN itself, but are instead a problem of the flexible Layer-II technology that is being allotted to any trusted VPN clients.

Hypothetically, one could allow only non-LAN MAC addresses into the network, so that no cloning can take place. I found this to be less than implementable, given that Linux dropped support for the protocol RARP, which would allow responses to queries of "who owns this MAC address?" on a network - to verify that it is not already in use.

Another alternative would be if the VPN server monitored every single MAC address which is valid on the LAN (similar to what a bridge/switch does), though all traffic wouldn't necessarily pass by the VPN server to actually allow the verification of this information.

As an immediate fix, I found that blacklisting misdirected traffic to the VPN Bridge as a result of MAC cloning is actually sufficient enough to keep those packets from reaching the VPN (i.e. stopping any traffic that gets sent to tap0, if the destination IP addresses are not within the VPN IP range).

The following iptables rule implements the above mentioned restriction:

```
iptables -A FORWARD -i br0 -m physdev --physdev-out tap0 -m iprange \! --dst-range 192.168.5.200-192.168.5.255 -j DROP
```

Also, while I was dealing with iptables, it also makes sense to blacklist outgoing packets that are not in the range of the VPN client address range – just to prevent race attacks and/or impersonated hosts. In Layer-3.

```
iptables -A FORWARD -i br0 -m physdev --physdev-in tap0 -m iprange \! --src-range 192.168.5.200-192.168.5.254 -j DROP

iptables -A INPUT -i br0 -m physdev --physdev-in tap0 -m iprange \! --src-range 192.168.5.200-192.168.5.254 -j DROP
```

While this doesn't stop the network from redirecting packets as a result of MAC cloning, it does stop those packets from reaching any snooping VPN clients. This means that VPN clients actively cloning LAN targets can still temporarily disrupt streams; but they cannot actually intercept any of the redirected data.

Conclusion

I still feel that there is more I can do to actually protect a layer-II bridged VPN network. Ultimately the above mentioned problem of being able to alter the bridged/switched state of the LAN network that is being connected to via the layer-II VPN tap driver still allows an attacker to disrupt service. While my solution above identifies a possible means of ensuring that security isn't compromised (via ARP or cloning), the ultimate solution to preventing denial of service from a VPN client that is given layer-II access to a physical LAN **requires** verification or hard-coded MAC addresses, to stop cloning/poisoning.

While ARP poisoning can be stopped with proper dynamic firewalls, client-to-client ARP attacks currently have no means of protection if layer-2 access is part of the VPN network requirements. On the other hand, MAC address cloning prevention requires hard-coded MAC address restrictions from VPN clients, such as blacklisting MAC addresses already on the LAN. Another solution would be employing the use of advanced switches that detect frequent switching of MAC address client positioning in network topology, or that can have known client/MAC locations digitally locked.

For the sake of integrity and security/efficiency, I would **strongly advise all administrators of OpenVPN networks to try and find ways to route the information they are using with proper layer-III tunnel drivers**, as opposed to identity-free layer-II adapters – so that they do not have to face the above problems. My solution in this document may stop the security exploits on a bridged network, but it allows clients to deny service to your network, as well as poison one-another if they are allowed client-to-client interactions.

Perhaps all network devices should be sold with a little warning on the bottom:

"Caution: Layer-II may be hazardous to your mental sanity"

Sources/Bibliography

A method to prevent source address spoofing in TCP/IP based networks so as to reduce the risk of Denial of Service (DoS) attacks on any host in the network: Background. (n.d.). Retrieved October 16, 2009, from IP.com Prior Art Database: http://www.priorartdatabase.com/IPCOM/000021778/

Arpi. (2004, August 5). *Linux Kernel Mailing Lists.* Retrieved October 16, 2009, from how to read /proc/net/arp properly: http://linux.derkeiler.com/Mailing-Lists/Kernel/2004-08/1302.html

Bart De Schuymer, N. F. (2003, November 9). *ebtables/iptables interaction on a Linux-based bridge.* Retrieved October 17, 2009, from Ebtables: http://ebtables.sourceforge.net/br_fw_ia/br_fw_ia.html

darkness. (2006, March 3). *Selectively firewalling OpenVPN users.* Retrieved October 14, 2009, from darkness: http://darkness.codefu.org/wordpress/2006/03/03/228

Difference in Hub, Switch, Bridge, & Router. (2004, November 20). Retrieved October 16, 2009, from Nutt.net: http://www.nutt.net/2004/11/20/difference-in-hub-switch-bridge-router/comment-page-1/

Feilner, M. (2006). *OpenVPN: Building and Integrating Virtual Private Networks.* Birmingham: PACKT Publishing.

Luk, P. (2009, August 19). *Using linux ethernet bridge to counter arp posioning.* Retrieved October 9, 2009, from Peter Luk's Blog: http://staff.ie.cuhk.edu.hk/~sfluk/wordpress/?p=535

OpenVPN on Sourceforge. (n.d.). Retrieved October 14, 2009, from Sourceforge: http://sourceforge.net/projects/openvpn/

OpenVPN Technologies. (n.d.). *Howto.* Retrieved October 5, 2009, from OpenVPN: http://www.openvpn.net/index.php/open-source/documentation/howto.html

OpenVPN Technologies. (n.d.). *OpenVPN.* Retrieved October 5, 2009, from Ethernet Bridging: http://www.openvpn.net/index.php/open-source/documentation/miscellaneous/76-ethernet-bridging.html

RTNETLINK (Linux Manual Pages). (1999, 04 30).

Snyder, J. (n.d.). *BR-NF Packet Flow*. Retrieved October 17, 2009, from Ebtables: http://ebtables.sourceforge.net/br_fw_ia/PacketFlow.png

Various. (2004, September 21). *RARP support disapeard in kernel 2.6.x ?* Retrieved October 16, 2009, from Linux Kernel Mailing Lists: http://linux.derkeiler.com/Mailing-Lists/Kernel/2004-09/6619.html

And many, many hours of testing.