Case of a wireless hack

Written by Siim Pader
Thursday, 26 May 2005

This is a short story about using a couple of computers, some interesting tools, an operating system and a bit of thinking to solve a not-entirely-artificial problem of getting wireless internet access where measureas are in place to stop it. Both the technical side as well as some of the reasoning behind the actions are explained.

1. Intro

Well, I had just got my first laptop and really wanted to try and do something cool with it (even tried to do some work but grew tired of it). Wardriving was exciting at first, but it became a bit dull when I realised that WEP enabled networks are out of my reach (as they didn't have any traffic to analyze - almost dead home networks) and unencrypted ones aren't a challenge at all. Luckily, the wireless network on my university's campus proved to be a bit more interesting.

They do provide free wireless internet, but require you to register your MAC address to your name before allowing access - unregisterd users get redirected to a web page explaining the situation. Registering would have involved 2 minutes with an administrator, but I thought "maybe there is a way to get access without 'all the hassle'". Surely enough, there was.

2. First attempt - MAC spoofing

As the dance was all around MAC addresses, my first idea was to find a MAC address that had already been registered and spoof it. Sure, it's easy enough to say but it still took quite a bit of effort, as it was all so new.

First thing I did was run kismet ('kismet -c orinoco,eth1,wifi') and sniff the network. Kismet dumps all the sniffed packets into a file ('/var/log/kismet/*.dump' in my case) that can be later opened by ethereal. It was possible to look through the packets and write suitable MAC addresses down, which I did.

The commands to change the MAC address of a NIC:

```
ifconfig eth1 down
killall dhclient
```

```
ifconfig hw ether 00: 11:22:33:44:55
ifconfig eth1 up
dhclient eth1
```

All the commands aren't neccessary, but they are useful when trying several MAC addresses in a row - which in turn was neccessary because the addresses I tried didn't work straight away. I had strange lockups (that ended when I pulled the card out), wireless going down and not coming up again and many other interesting and challenging annoyances to deal with. They were propably because of faulty hardware/firmware/driver and possibly because there already was a NIC with the same MAC address on the network.

Not many stations were very active, and using kismet/ethereal was inefficient (lots of cruft to look through), so I tried another approach. As the MAC-based filtering was done at a pretty high level (they wanted to display the explanitory web page to unregistered MACs) I could associate with the network and even get an IP address without a problem. Naturally, thinking about finding active hosts, nmap came to mind. So I ran a ping scan on the IP range active stations seemed to be in:

```
marktwain:~# nmap -sP 10.0.0.1/22
Starting nmap 3.81 ( http://www.insecure.org/nmap/ ) at 2005-05-23 12:54 EEST
Host 10.1.0.14 appears to be up.
MAC Address: 00:0E:35:97:8C:A7 (Intel)
Host 10.1.0.95 appears to be up.
MAC Address: 00:02:2D:4D:1C:43 (Agere Systems)
Host 10.1.0.109 appears to be up.
MAC Address: 00:0D:54:A0:81:39 (3Com Europe)
... snip ...
Host 10.1.2.92 appears to be up.
MAC Address: 00:02:2D:4D:1C:CE (Agere Systems)
Host 10.1.2.187 appears to be up.
MAC Address: 00:02:2D:4D:1C:43 (Agere Systems)
Nmap finished: 1024 IP addresses (20 hosts up) scanned in 53.980 seconds
```

A lot of MACs. The best bit is that it also populated my arp tables with (what I presume is) the MAC addresses of all the stations that had visited the network in recent days. The table had 245 distinct MAC addresses. I don't know if it's normal behaviour for an AP, but it would make sense if it had something to do with the MAC-blocking scheme they used (propably bridged the wireless and wired networks somehow?).

Whatever the reason, I now had enough MAC addresses from stations thad had visited the network but were most likely long gone. A couple of tries at the spoofing thing and I was already surfing to neworder.box.sk and revealing my cookie to anyone who cared to listen. I walked home at a speedy pace and changed the password.

3. Planning the second attempt - ICMP tunneling

I had accomplished what I had set out to do, but there was still some marrow in the bone of this network (figuratively speaking). What had I done, if there hadn't been any stations on the network with me? If nmap'ing hadn't revealed all those MACs? If I had been born a mighty dragon? Well, whatever the

reasons, I wanted to try yet another way of getting access.

It hasn't been mentioned yet, but aside from allowing association and DHCP the network also allowed ICMP messages to pass through. Pinging any internet site worked fine (can't really understand why they haven't blocked it - unless they forgot) and the pings even showed up on a sniffer I ran on my server (so the responses weren't just faked by some box at the university).

My the plan was to try to create an ICMP tunnel between my laptop at the university and a server at home. And then pass all the connections through it.

I had looked for a ICMP tunneling applications on the net before but none had really worked as I would have liked (namely, I wanted it to be transparent - so that I could fire up my favourite browser or any other program and it would "just work ©" with the tunnel), or at least none that I had tried. If you know a piece of software that does this, feel free to embaress me by letting everybody know.

4. A little bit of coding

I didn't plan to write any code at first. Just tried some icmp tunnel applications from packetstorm, but suddenly found myself reading the source for one of them and realising how marevelously straightforward it is and how easy it would be to build on it. The programs name was itunnel - a general purpose ICMP tunneling tool.

Itunnel is great. But it is just a tunnel. You run it on one machine and on another machine and get pretty much the sort of thing you'd get when connecting two netcats together. That was not enought for what I wanted to do.

I had also heard about this kernel module TUN/TAP that allows user proccesses to send and recieve packets straight to/from the kernel. It creates a network interface that you can use like a normal interface - like define your default gateway to be on the interface. The interesting bit is, that the a userspace application must act as the physical layer for the tunnel interface. It must read the packets from a device file (for example '/dev/net/tun0') and send them by whatever means and write response packets back to the file. That could provide for transparency.

I wasn't able to find any good resources on TUN/TAP as such, but there is an application - vtun - that uses TUN/TAP for it's tunnels, so I grabbed vtun's sources. A little exploring revealed, that it had a nice tiny library of functions to create, read from and write to tun* devices.

Why would you write a program yourself if it can be patched together from bits and pieces of other, more capable programmers? I know a couple of reasons but I'm keeping them to myself as not to undermine my actions.

All the code I actually wrote is this:

```
#include "driver.h" /* the vtun's library declarations */
#include
#include

/* slightly modified main() from itunnel */
```

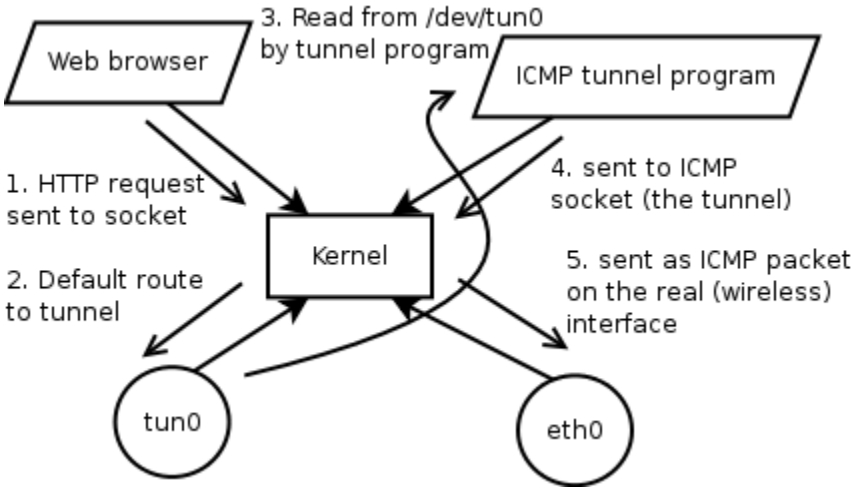
```
int run_icmp_tunnel (int id, int packetsize, char *desthost, int tun_fd);

/* max transfer unit - max capsuled packet size */
const int mtu = 1400;

int main(int argc, char **argv) {
char *dev;
int tun_fd = 0;

/* create the tunnel device */
dev = (char *) malloc(16);
if (dev == NULL) {
printf( "If you have never had problems allocating 16 bytes\"
"of memory, then now is your first time. Fatal.n");
return 1;
}
dev[0] = 0;
/* nice library function from vtun - takes an empty string as an *
* argument, creates a tunX device and fills *dev with it's name */
tun_fd = tun_open(dev);

if (tun_fd < 1) {
printf("Could not create tunnel device. Fatal.n");
return 1;
}
else {
printf("Created tunnel device: %sn", dev);
}

/* 7530 is the ICMP id field used for tunneling packets */
run_icmp_tunnel(7530, mtu, argv[1], tun_fd);

tun_close(tun_fd, dev);
}
```

That's it. And most of it is comments and error checking.

As I already mentioned, itunnel is perfect for building upon. It has a main function that sets up file descriptors for input, output and the socket; also gets some parameters for command line (this was not useful for my purposes). Then it calls a quite abstract function that basically just transports "packets" between the three. ICMP header is very loosely attached to the code and could be easilly subtituted by any other header and the input/output/socket passed to the function could be set up with some other logic - the function would work with minimal adjustments.

The biggest change I made was removing all the command line handling - basically a deleting a few blocks of code. More importanly to the tunnel logic, I removed the distinction between input and output as they both happend on the same descriptor (the tunX device) - the effect of which is that instead behaving like netcat and forwarding stdin to the icmp socket and icmp socket output to stdout it would forward the

output from the tunX device (like HTTP requests from the browser) to the icmp socket and the icmp socket output (like HTTP results from a web server sent back over the tunnel) to the tunX device (to be transported back to the browser). As the last sentence is long and winding, I'll provide a tiny scheme to illustrate it.



Response packets go through the same path but the other way around.

At one point I went completely crazy and actually wrote an absolutely new line of code. It looks like this:

```
memcpy(&(target->sin_addr.s_addr), &(from.sin_addr.s_addr), 4);
```

It's function is to start sending all packets to wherever the last packet came from. So I can basically run it on my server and when I connect from wherever I happen to be and send a packet through the tunnel, it recalibrates it's destination to my laptop's current IP. I can't do it though, as some of you may be able to maybe somehow find the server and maybe use the tunnel without my knowledge!

You can get the resulting hack from here: http://p6drad-teel.net/~windo/release/icmp_tunnel.tar.gz

5. Setting up the tunnel

I tried the tunnel out at home and it worked fine, but there was no firewall at home. And I knew how stuff worked at home.

The next day at the university I was ready to try it in a "real-world" situation. Casually sitting at a table in the cafeteria I spoofed the MAC address and set the tunnel up. Server first.

It's very hard to remember all the stupid things I tried and all the tiny experiments I made so I'll make this part look like I was very systematical about it. I really wasn't actually. It took about 3 hours to get to work and I tried all sorts of things while sniffing all over the place and modifying the code to print "Packet!" when it received a packet and to print something equally disturbing when it sent one. And so on.

I ran these commands on the server:

```
tsee-diees:~# ./create_tun wifi.ttu.ee
Created tunnel device: tun0

[1]+ Stopped ./create_tun wifi.ttu.ee
tsee-diees:~# ifconfig tun0 mtu 1400 192.168.3.1 up
tsee-diees:~# route add 192.168.3.2 tun0
tsee-diees:~# tethereal -i eth0 -f 'icmp'
```

Or at least, this is what I could have done. In reality - as noted above - I ran all sorts of slight variations of the above, over and over again. The sniffer is an absolute must to have any kind of idea how far the packets get (eth0 is my internet interface).

At first I kind of thought it would work straight away and just run this on the laptop:

```
marktwain:~# ./create_tun 194.204.48.52
Created tunnel device: tun0

[1]+ Stopped ./create_tun 194.204.48.52
marktwain:~# ifconfig tun0 mtu 1400 192.168.3.2 up
marktwain:~# route add 192.168.3.1 tun0
marktwain:~# tethereal -i eth0 -f 'icmp'
```

What it should have done is create a LAN with two hosts on it - the server as 192.168.3.1 and the laptop as 192.168.3.2. Just a normal LAN, only that it's physical layer would be the ICMP tunnel. As you can propably deduce from both the scrollbar position on your browser and almost every murphy law in existance, it didn't (work). I ran a ping ('ping 192.168.3.1') on my laptop but the packets wouldn't go through.

Luckily the sniffer on the laptop gave a good clue - the ICMP packets were echo replies. Of course they wouldn't be forwarded. So, kill the tunnel, make itunnel on the laptop use icmp echo requests (change 'icmp->type = 0;' to 'icmp->type = 8;'), rerun the tunnel. It still fails, but this time the packets also show up on the sinffer on the server.

What could be wrong? I try one of the modifications that exclaim "Packet!" when one arrives but exclamation does not take place. "Why should it?" I wonder "when the firewall is set to block all ICMP packets from the internet?" shortly after I realise that it is (blocking all ICMP packets from the internet).

As a side notice, the last paragraph was a tribute to Douglas Adams.

It's starting to get better. The tunnel is exclamating "Packet!" and responses can be seen on the sniffer on the server. In fact, there are two responses for every request, only one of which can be seen on the sniffer on the laptop. And the ping still doesn't work.

Since one of the two packets is redundant, I tell the kernel not to answer icmp echo requests:

```
tsee-diees:~# echo "1" > /proc/sys/net/ipv4/icmp_echo_ignore_all
```

And pings are coming through! At this point I was still relying on the spoofed MAC to access the server. As the idea was to get packets moving as an unregistered wireless user, I stopped spoofing the MAC address. As a pleasant suprise, the tunnel still continued to work.

General packet flow established, it was now time to get the "internet" to work. Some modifications had to be made to the IP routes on the laptop. The default route through the university's wireless router had to be removed and be replaced by a route through the server's tunnel address (192.168.3.1 in this case). Yet there still had to be a route to the server, so that the tunnel itself would work (the tunnel ICMP packets also need a route to follow). The following had good results:

```
marktwain:~# route add 194.204.48.52 gw 10.0.0.111 # through the wireless
router
marktwain:~# route del default
marktwain:~# route add default gw 192.168.3.1 # everything else through the
tunnel
```

And since I'm kind of clever, I thought that there might be an imbalance between the number of packets sent to and from the server (namely, more packets from the server). I ran a ping in background to mend the situation - there would always be a couple of outstanding echo request and the replys from the server would be forwarded to the laptop.

```
marktwain:~# ping 194.204.48.52 -i 0.2
PING 194.204.48.52 (194.204.48.52) 56(84) bytes of data.
```

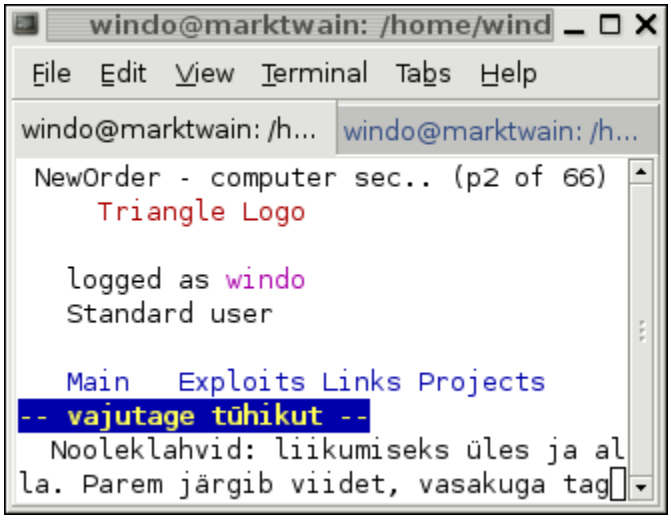
There weren't any replys of course, as these werent the "tunnel pings" and the replys from the kernel were disabled.

Since my server is already set up to share the internet connection among a few computers, all I had to do on the server was to add two rules to the FORWARD chain with iptables to accept packets from and to tun0. When I routinely checked the current rules ('iptables -vL FORWARD'), the connection suddenly died.

I reconnected and investigated the matter and shortly it died again. I was really amazed - why would the connection be so instable? I would have imagined that the TCP would handle all the retransmissions and create reliability above the tunnel. As the connection seemed to die whenever there was a lot of output from a command, I set the rules up very quietly.

The setup was tolerable, but would not stand any real transfers. The connections would all die. As I had

started a thread on neworder about the little project and was generally happy with the the thing (in all ways that really mattered, in a proof-of-concept kind of way). I wanted to post a new message. I kind of sensed it might be a MTU problem and as the site wouldn't load I ended up running a really tiny terminal (38x10) connecting to the server and running lynx on it. It looked kind of funny but I was able to make the post (the bits you don't understand are in estonian).



As it often happens writing a post helped organize my thoughts and I realised, that whenever the server sent a big ICMP reply, it was discarded by the uni's equipment (why would you forward 1400+header of ping?). As the tunnel was the physical layer for IP, TCP of course tried to send the packets again, but it was still the same size and still got discarded. So I changed the MTU for the tunnel to 300 (pretty randomly).

```
marktwain:~# ifconfig tun0 mtu 300
tsee-diees:~# ifconfig tun0 mtu 300
```

All of a sudden it worked like a charm. I could use firefox to make another post and did exacly that (no screenshot this time, you all know how it looks like).

6. Conclusion

Make up your own mind.

Article written by Siim Põder