Cheating CHAP

Sebastian Krahmer *krahmer@cs.uni-potsdam.de*

February 2, 2002

Abstract

The Challenge Handshake Authentication Protocol (CHAP) is used to verify the identity of a peer in a 3-way handshake and is usually embedded in other protocols, commonly PPP. Several extensions (MS-CHAP) exist to allow the encryption of link layer packets via CHAP authenticated connections. In this paper I will describe how CHAP may be attacked, gaining unauthorized access to CHAP protected dialins or VPN's and show that CHAP is not the right protocol to authenticate clients in IP networks.

Disclaimer

This material [1] is for educational purposes only. It demonstrates weaknesses within authentication schemes often used in wireless networks for example. All described tests and captures have been done in my own private LAN. It is strongly recommended that you do not test these programs without prior written permission of the local authorities.

1 Introduction

CHAP is most frequently encountered in conjunction with PPP, the Point-to-Point Protocol as used to connect to your ISP with a modem for instance. While I am not aware of any non-PPP protocol that uses CHAP, there is no reason why it should not work on top of other protocols despite the tight binding to PPP in the CHAP RFC 1994.

I am not going to explain the 100th insecurity of PPTP or that the hashing mechanism some companies are using are weak. The weaknesses of PPTP have been described in [PPTP] and others papers.

The insecurities of CHAP do not depend on the underlying layer used, i.e. CHAP is vulnerable regardless of wether PPP+CHAP is used for ISP dialins or with PPTP to protect wireless networks. However, I will only focus on PPP+CHAP over IP (PPTP) because IP network spoofing and sniffing techniques are widely known. As will be demonstrated later, we will need these techniques.

2 How things work

RFC 1994 describes the CHAP authentication process. This process is summarized below with a server authenticating a client, the authenticator and the peer:

- After the Link Establishment phase is complete, the authenticator sends a ”challenge” message to the peer.
- The peer responds with a value calculated using a ”one-way hash” function.

[1] this paper and provided programs

- The authenticator checks the response against its own calculation of the expected hash value. If the values match, the authentication is acknowledged; otherwise the connection SHOULD be terminated.
- At random intervals, the authenticator sends a new challenge to the peer, and repeats steps 1 to 3.

The first three steps gave it the name challenge/response and are ok provided the following conditions are true.

1. The used hash function is cryptographicaly strong. Common algorithms are MD5 [2] or SHA1.
2. The secret used to calculate the response is strong, i.e. it does not appear in a dictionary.
3. The challenge that is sent to the peer is truely random and not predictable by attackers. It is also assumed that the same challenge is never sent twice in a certain timeframe especially after reboots etc. That implies that the rand-pool is large enough.

The fourth RFC requirement for CHAP exists to protect channels from take-overs i.e. once an attacker took over your communication-channel he may not know the answer to the next challenge and is disconnected. This is meant to be a special protection within the CHAP protocol but we will see that this fourth point will break CHAPs neck.

Challenge/response itself is a quite weak authentication method. It is similar to giving the attacker your /etc/shadow file which should be avoided. This is because the second point is often handled too lax, e.g. commonly students choose their password themself when getting WLAN access.

2.1 CHAP packets found ”in the cage”

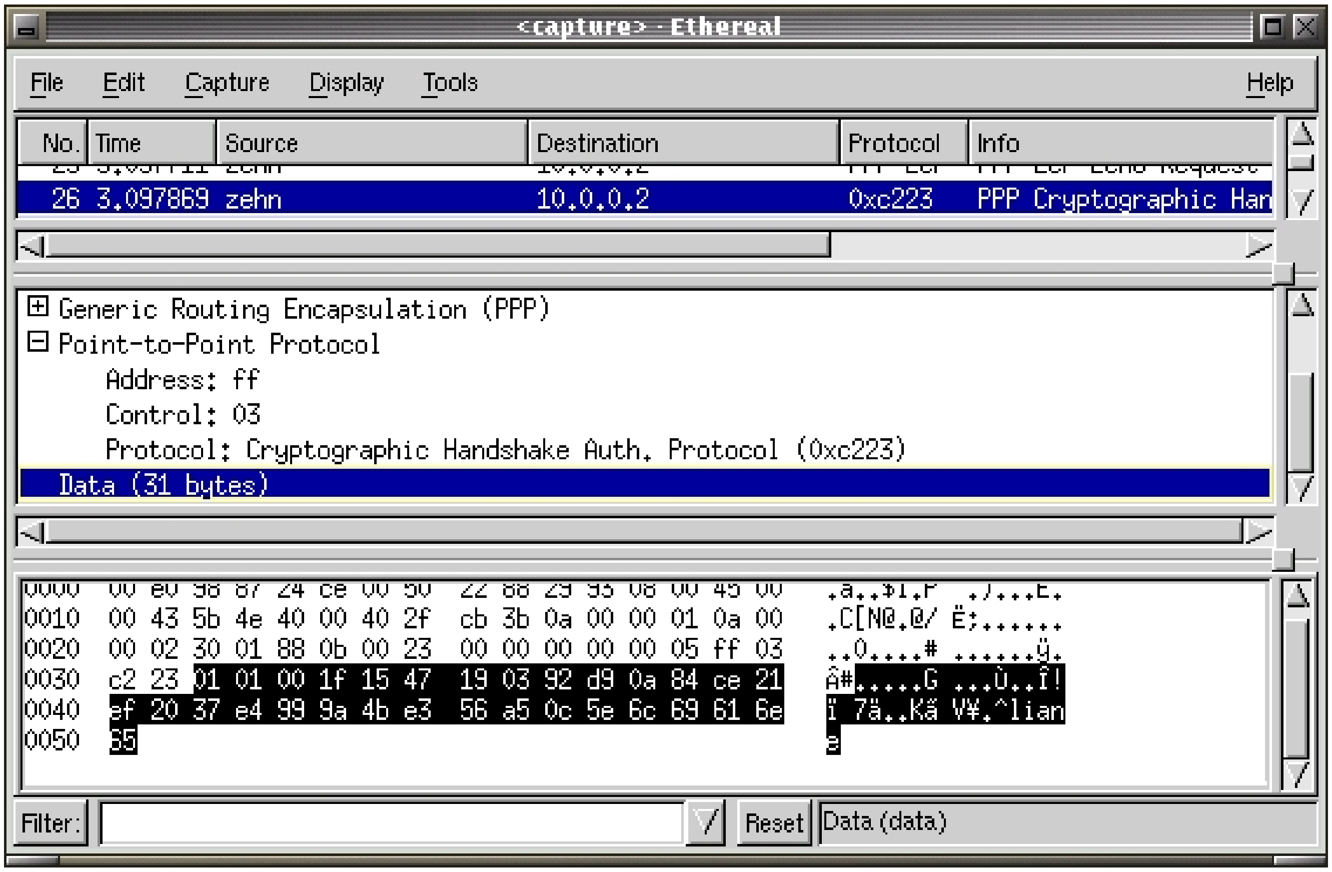
Figure 1 shows a captured challenge. The marked data part consists per RFC of:

- 1 Byte Code; 01 for Challenge 02 for Response
- 1 Byte Identifier
- 2 Byte Length of CHAP portion
- 1 Byte Value-Size to tell the peer the length of the Challenge or Response payload
- the Challenge/Response payload
- In challenge-case (code 01) the name of the server, in response-case (code 02) the username

In this sample-session we see a challenge packet from host ’zehn’ to 10.0.0.2. The servername is ’liane’ (last 5 byte of packet), the challenge is 16 byte long. The first packet from 10.0.0.2 to zehn [3] initiated the talk and requested to use MD5 to hash the challenge. Both sides now compute the hash as follows: They concatenate the identifier [4] with the secret and the challenge and compute the MD5-hash. The secret is determined by looking up in a certain file which holds the secrets (’passwords’).

Once the client computed the response he sends it to the server: A packet containing 16 bytes of response and the username the server should use to lookup the secret. If the client sent the correct response the server now sends a success-packet. The captured response-packet is seen in Figure 2.

[2]CHAP is using MD5.
[3]This packet was not captured because it did not contain useful information.
[4]The second byte of the packet.



```
No. Time       Source  Destination  Protocol  Info
 26 3.097869   zehn    10.0.0.2     0xc223    PPP Cryptographic Han

⊞ Generic Routing Encapsulation (PPP)
⊟ Point-to-Point Protocol
    Address: ff
    Control: 03
    Protocol: Cryptographic Handshake Auth. Protocol (0xc223)
  Data (31 bytes)

0010  00 43 5b 4e 40 00 40 2f  cb 3b 0a 00 00 01 0a 00   .C[N@.@/ Ë;......
0020  00 02 30 01 88 0b 00 23  00 00 00 00 00 05 ff 03   ..0....# ......ÿ.
0030  c2 23 01 01 00 1f 15 47  19 03 92 d9 0a 84 ce 21   Â#.....G ...Ù..Î!
0040  ef 20 37 e4 99 9a 4b e3  56 a5 0c 5e 6c 69 61 6e   ï 7ä..Kã V¥.^lian
0050  65                                                 e

Data (data)
```

Figure 1: A captured CHAP Challenge packet.

3 Open the gates!

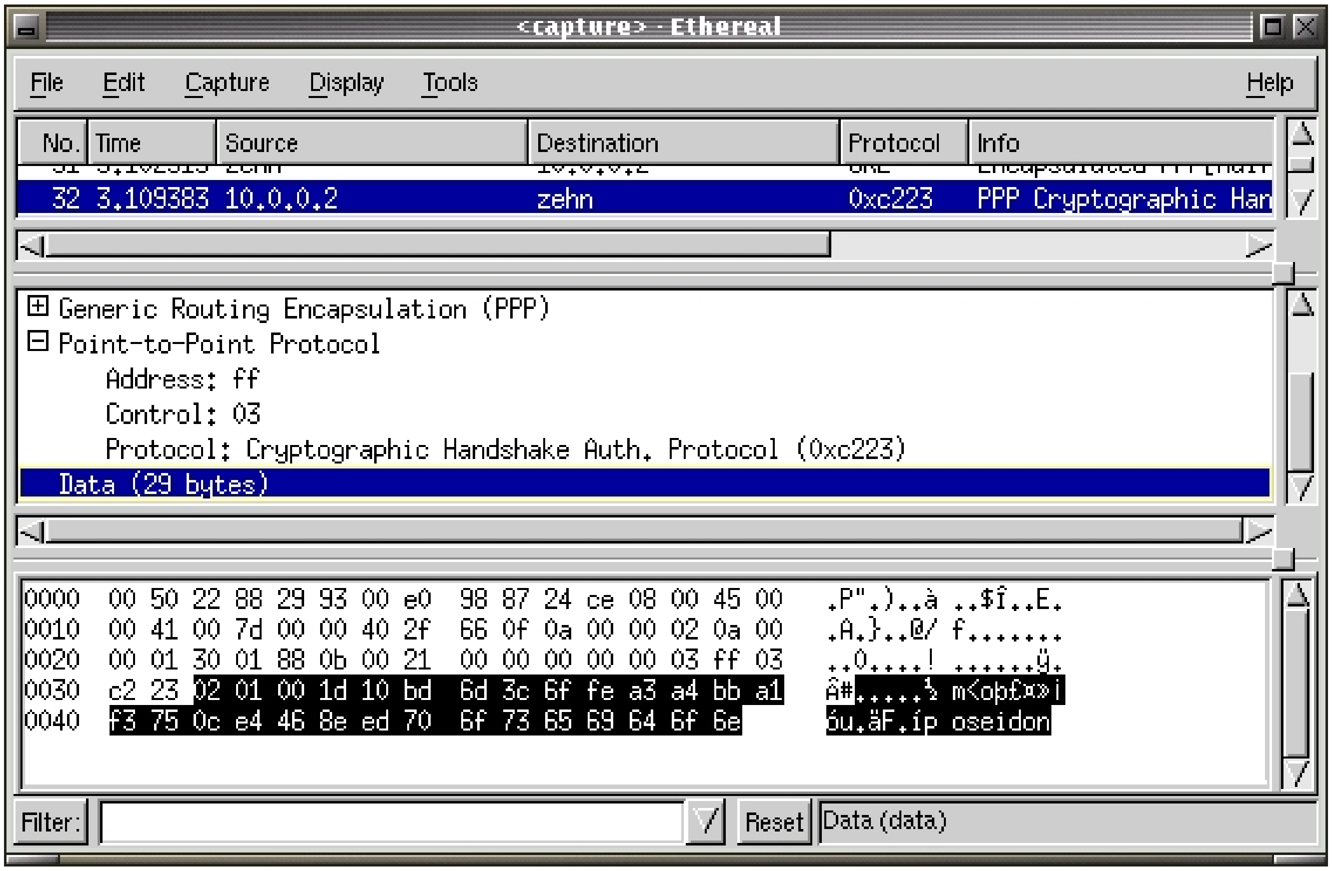
Despite the fact that CHAP was designed to send as little information as possible across the link we still gather:

- username
- servername
- client and server IP
- the ID used to compute response
- challenge and associated response

The most important thing is missing: the secret. Yes, challenge response was designed to keep the secret secret.

The easiest thing we may now do is try a dictionary attack because we have all the necessary information to calculate the response. Only the secret is unknown which allows us to try a dictionary-attack. We assume the secret is strong [5]. Further, the challenge is probably unique and the randomness is good enough, so we won't succeed with precomputing or replaying attacks either.

[5]this is probably not the case, such an attack is always worth a try, but theres a better way, so lets continue



```
No. Time     Source    Destination  Protocol  Info
32  3.109383 10.0.0.2  zehn         0xc223    PPP Cryptographic Han
```

```
⊞ Generic Routing Encapsulation (PPP)
⊟ Point-to-Point Protocol
    Address: ff
    Control: 03
    Protocol: Cryptographic Handshake Auth. Protocol (0xc223)
  Data (29 bytes)
```

```
0000  00 50 22 88 29 93 00 e0  98 87 24 ce 08 00 45 00   .P".)..à ..$Î..E.
0010  00 41 00 7d 00 00 40 2f  66 0f 0a 00 00 02 0a 00   .A.}..@/ f.......
0020  00 01 30 01 88 0b 00 21  00 00 00 00 00 03 ff 03   ..0....! ......ÿ.
0030  c2 23 02 01 00 1d 10 bd  6d 3c 6f fe a3 a4 bb a1   Â#.....½ m<oþ£¤»¡
0040  f3 75 0c e4 46 8e ed 70  6f 73 65 69 64 6f 6e      óu.äF.íp oseidon
```

Data (data)

Figure 2: A captured CHAP response packet.

However, looking again over the data we already gathered and thinking about the protocol, immediately the 4th requirement in the RFC comes in mind: The client MUST answer *any* challenge it receives. Aha! What if we send a request to the server, await the challenge and let an already authenticated client where the handshake has just been captured compute the response?

Analyzing the sourcecode of *pppd* [6] shows that this could work. The server does not send repeated challenges, but the client will answer any challenge it receives.

Since PPP (the protocol itself) does not know about IP addresses there is no way for the *pppd* to determine who was asking and it happily sends a response. Thus, in theory the *pppd* is answering any question. In practise things are a bit more hairy since the PPP packets are wrapped in PPTP frames because we use PPTP to tunnel PPP over IP. PPTP however uses sequence numbers [7] and source/destination addresses to match incoming packets to the daemons that should handle them. That means the 10.0.0.2 client in the example will not respond to challenges from 10.0.73.50 but only to challenges from 10.0.0.1 which is the legitimate PPTP server in this configuration. Now you know why we use IP. It is trivial to send a spoofed challenge to the client which appears to originate from the original server. The client computes the response and sends it to the server. Two things happen then. First we sniff the response and use it to authenticate ourself, second the server also obtains the challenge from the legit client which he just answers with a success packet as required by the RFC.

As already stated the PPTP tunnel also uses 32 Bit sequencenumbers to keep track of already sent packets. This is not a big obstacle since we may sniff the last valid sequencenumber. But things are even simpler because the pptp program accepts any sequencenumber which is larger than the last one, i.e. we may safely send a PPTP packet with sequencenumber of 0x00ffffff. Using this sequencenumber we can be pretty sure that it is greater than the last valid sequencenumber the server has seen from this client.

3.1 Finetuning

Figure 3 shows successfull exploitation of the CHAP weakness inside a test-LAN (hubbed). The first step is to gather valid client and server IP addresses and the login-name/server-name pair. The special *pppd* will use these informations to log into the VPN without knowing the secret accociated with the login-name it is using. Additionally the challenge and the response have been saved which allows dictionary attacks when apropriate.

When using the patched *pppd* from team TESO [7350pppd] to use other clients to authenticate ourself, as a sideeffect this victim-client will lose its connection. The *pptpd* running on the server drops all packets from the legit client because the sequencenumber the client is using are too small to be accepted by the server. This might be avoided by keeping track of sent sequence numbers and only choosing last_seen+1 to ask the victim-client for the response. That way, the client will only miss one packet from the server. A second way is to let the sequence-number wrap-around, i.e. sending MAXSEQ-1 for the fake-request and after obtaining the response, sending a MAXSEQ packet so the sequence number wraps around. Any following packets from the client will have bigger sequence numbers and are passed through.

4 Conclusion

The take home message of this paper should be that CHAP is not the right protocol for authentication to be used within IP based networks. I am not sure how the authentication-

[6] the program responsible for handling CHAP on both ends

[7] The sequencenumbers in its own header, do not mix this with TCP seqence-numbers. TCP never plays any role in our discussion.

bypassing could be done at dialins because it lacks sniffing/spoofing capabilities. However, this does not mean its secure.

Never ever use CHAP for authentication in IP networks, wireless LANs in particular. Think about strong authentication schemes such as RSA authentication or Kerberos.

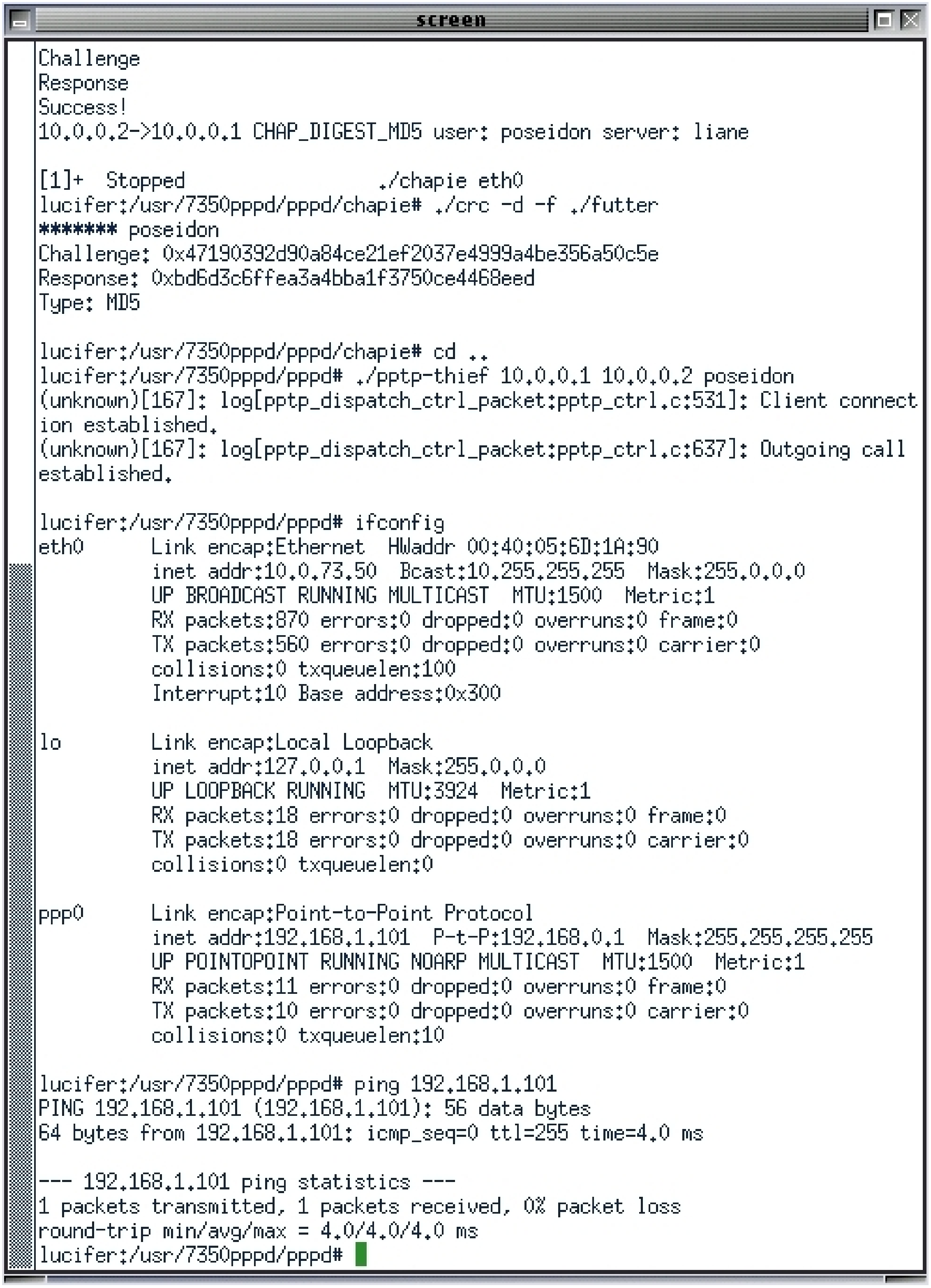
Acknowledgments

I would like to thank grugq for proof-reading this paper. It was hard to understand all the grammatics but I think I got it right at the end. Same to scut for kicking my ass to make new screenshots. The old ones really sucked. And finally again thanks to segfault.net consortium for giving valuable discussion about crypto and why NOC people should watch their cables :-).

References

[PPTP] Bruce Schneier and Mudge. *Cryptoanalysis of Microsoft's Point-to-Point Tunneling Protocol (PPTP)* .

[7350pppd] TESO *An implementation of the attack described in this paper.* `http://stealth.7350.org/7350pppd.tgz` .



```
Challenge
Response
Success!
10.0.0.2->10.0.0.1 CHAP_DIGEST_MD5 user: poseidon server: liane

[1]+  Stopped                 ./chapie eth0
lucifer:/usr/7350pppd/pppd/chapie# ./crc -d -f ./futter
******* poseidon
Challenge: 0x47190392d90a84ce21ef2037e4999a4be356a50c5e
Response: 0xbd6d3c6ffea3a4bba1f3750ce4468eed
Type: MD5

lucifer:/usr/7350pppd/pppd/chapie# cd ..
lucifer:/usr/7350pppd/pppd# ./pptp-thief 10.0.0.1 10.0.0.2 poseidon
(unknown)[167]: log[pptp_dispatch_ctrl_packet:pptp_ctrl.c:531]: Client connect
ion established.
(unknown)[167]: log[pptp_dispatch_ctrl_packet:pptp_ctrl.c:637]: Outgoing call
established.

lucifer:/usr/7350pppd/pppd# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:40:05:6D:1A:90
          inet addr:10.0.73.50  Bcast:10.255.255.255  Mask:255.0.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:870 errors:0 dropped:0 overruns:0 frame:0
          TX packets:560 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          Interrupt:10 Base address:0x300

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:3924  Metric:1
          RX packets:18 errors:0 dropped:0 overruns:0 frame:0
          TX packets:18 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0

ppp0      Link encap:Point-to-Point Protocol
          inet addr:192.168.1.101  P-t-P:192.168.0.1  Mask:255.255.255.255
          UP POINTOPOINT RUNNING NOARP MULTICAST  MTU:1500  Metric:1
          RX packets:11 errors:0 dropped:0 overruns:0 frame:0
          TX packets:10 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:10

lucifer:/usr/7350pppd/pppd# ping 192.168.1.101
PING 192.168.1.101 (192.168.1.101): 56 data bytes
64 bytes from 192.168.1.101: icmp_seq=0 ttl=255 time=4.0 ms

--- 192.168.1.101 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 4.0/4.0/4.0 ms
lucifer:/usr/7350pppd/pppd# 
```

Figure 3: A successfull login without knowing the secret.