Exploiting known security holes in Microsoft's PPTP Authentication Extensions (MS-CHAPv2)

Jochen Eisinger
University of Freiburg
`eisinger@informatik.uni-freiburg.de`

July 23, 2001

Abstract

The implementation of the Point to Point Tunneling Protocol (PPTP) from Microsoft using MS-CHAPv2 and Microsoft Point to Point Encryption (mppe) is widely used to secure and control access to wireless networks. We show why the MS-CHAPv2 protocol is not suitable for user authentication in a heterogenous Unix network context.

1 Introduction

With the appearance of wireless networks based on the IEEE 802.11b standard [IEE99] and in the light of the insecure Wired Equivalent Privacy (WEP) protocol, solutions to integrate wireless components into existing networks are needed.

A solution based on Microsoft's PPTP [Gro99] authentication extensions is widely used, e.g. at German universities [fBuF00], to authenticate the users and to secure the wireless transmission of sensible data.

Easy client setup and long experience with the protocol are arguments for this solution. In the following, we will show that the use of an authentication based on MS-CHAPv2 [Gro00] will compromise password security.

A possible implementation of an attack exploiting well-known security holes in Microsoft's protocol MS-CHAPv2 [SMW99] will be given and discussed to demonstrate how easy passwords and logins can be gained.

2 MS-CHAPv2

We will concentrate on the peer authentication using MS-CHAPv2. This stage takes place after a PPTP tunnel is established and the setup for the PPP connection has started.

1. The client requests an authenticator challenge from the server.

2. The server sends back a 16-bytes random authenticator challenge.
3. The client generates the response:
   (a) The client generates 16-bytes random peer challenge.
   (b) The client generates the challenge by hashing the authenticator challenge, the peer challenge, and the user's login using SHA [oST93].
   (c) The client generates the NT password hash from the user's password.
   (d) The 16-byte NT password hash from step (c) is padded with 5 bytes of zero. From these 21 bytes three 7-byte DES keys are derived.
   (e) The first 8 bytes of the hash generated in step (b) (these 8 bytes are later reffered to as the challenge) are encrypted using DES with each of the three keys generated in step (d).
   (f) The 24 bytes resulting from step (e), the 16-byte random peer challenge, and the user's login are sent back to the server as response.
4. The server decrypts the response with the hashed password of the client that is stored in a database.
5. If the decrypted response matches the challenge, the server sends a positive authenticator response:
   (a) The server hashes the NT password hash using MD4 [Gro90] to generate a password-hash-hash.
   (b) The server generates a hash using SHA from the clients response, the password-hash-hash, and the literal constant "Magic server to client signing constant".
   (c) The server generates another hash using SHA from the 20-byte output of step (c), the 8-byte challenge (see step 3 (b)), and the literal constant "Pad to make it do more than one iteration".
   (d) The resulting 20 bytes are send back to the client in the form "S= ⟨upper-case ASCII representation of the byte values⟩".
6. The client uses the same procedure to generate the 20 bytes and compares them to the servers authenticator response. If they match, both the client and the server are authenticated.

2.1 Analysis

This protocol has some major weaknesses [SMW99] which we will point out now. In section 3 we will show how to exploit these.

2.1.1 Generating the 8-byte Challenge

It is neither obvious nor documented why such an effort is made to generate the 8-byte challenge. Since this 8-byte challenge is derived from information transmitted in plain-text, it can be computed by any eavesdropper, so it does not provide any additional security.

Fact is that the 8-byte challenge can be computed by a potential attacker without problems (see section 3.2). Once the challenge is known, the authenticator challenge, the peer challenge, and the user's login are superseded by this information.

2.1.2 Deriving 3 DES Keys from the NT Password Hash

The derivation of the third DES Key is the major flaw in the MS-CHAPv2 protocol. Since the last five bytes of the third key are all zero, the key has an effective length of 16 bits.

With a brute force attack trying at most 65536 different DES Keys, the last 16 bits of the NT password hash can be determined. Such an exhaustive search takes no more than some seconds, even on old hardware.

One property of a hash function is that the hash values are evenly distributed over the whole hash space. With the information of the last 16 bits, we can reduce the hash space of possible hash values from the search password by the factor 2^{16}. Since the hash values are evenly distributed, the password space also collapses by this factor.

With this we can speed up both a dictionary attack and an exhaustive search by the factor 2^{16} if it is possible to determine which part of the password space has to be searched.

The remaining question is how large the password space is. We assume that normal Unix passwords are used. As we stated before, we will analyse MS-CHAPv2 when used for authentication to a heterogenous Unix network, so only 8 character long passwords are allowed. Another restriction is that they need to be typeable on different kinds of keyboards. we will assume that passwords consist of both upper and lower case letters, numbers, and 33 common special characters (`()[]{},;.:-_ #'+*~^ °%!\/"§$&=?<>|`), all in all 95 different characters. The number of possible passwords with up to eight characters is

$$\sum_{i=1}^{8} 95^i = 6704780954517120 \approx 2^{52}$$

With the information gained from the third DES key, we can reduce this to

$$2^{52} \cdot 2^{-16} = 2^{36}$$

On a system equipped with a 550MHz Intel Celeron, we can test around 2^{16} passwords per second. At this speed, the remaining password space is searched in

$$2^{36} \cdot 2^{-16}\text{secs} = 2^{20}\text{secs} \approx 2^{14}\text{mins} \approx 291\text{h} \approx 12\text{days}.$$

When we restrict the allowed characters in passwords further to alpha-numeric characters, all possibilities can be checked in about 16 hours.

Note however that it is still required to determine which passwords should be tested, i.e. which passwords have a hash value ending with the known 16 bits. This problem is addressed in section 3.1

3 Exploiting

We will now show how to exploit these security holes. To implement such a password attack neither special cryptographical knowledge nor experience is required [Eis01].

We will make use of cryptographic libraries taken from the pppd-2.3.11 source tree [MCL$^+$99]. The copyright holders of these are

SHA-1 library Copyright (C) 1995-1997 Eric Young (eay@mincom.oz.au)

MD4 library (C) 1990 RSA Data Security, Inc.

extra crypto routines Copyright (c) Tim Hockin, Cobalt Networks Inc. and others

MS-CHAPv2 implementation Copyright (c) 1995 Eric Rosenquist, Strata Software Limited. http://www.strataware.com

3.1 Generating a Dictionary

To be able to conduct such an attack a fast way to determine passwords that should be tested is needed.

Basically, there are two possible solutions to do this: A dictionary could be generated with precomputed MD4 hashes assigned to each word, possible passwords can be extracted quite fast then.

Another solution is to calculate such a dictionary at run-time. Since only every 2^{16}th password is a candidate for testing, this has to be done distributed over many computers. Luckily, MD4 was designed to run very fast, so given the ressources, e.g. provided in a standard workstation pool of an university, this is not impossible to achieve. Furthermore, there are other advantages of this solution. One could try to crack more than one password at a time, which requires passwords with different hash endings. Another improvement can be achieved by restricting ourself to a more realistic password space ("normal" user passwords are of little entropy, i.e. not containing special characters at all).

We will not provide any dictionary nor an exact description how to generate such a list at run-time. However, we will provide a program that converts a word list into a word list with hash values.

However, good dictionaries for password checking are widely available on the internet [Muf96, Ley92].

The program reads passwords from `stdin`, truncates them to 8 characters, converts the ASCII passwords to UNICODE and calculates the MD4 hash (i.e. the NT password hash). It then outputs the truncated password and the hash value seperated by a tabulator character.

Assuming you have a dictionary you want to search, you would have to prepare it like this:

```
$ cat dict | genkeys > hash-dict
```

3.2 Auditing a valid Authentication

The next step is to audit a valid authentication. To do this, you need an IEEE 802.11b compatible wireless device, available in the computer store of your choice. Equipped with this, you can immediately audit all wireless network traffic if WEP encryption isn't used. Otherwise, you have to take this barrier first. How this is done is described in [NB01] in detail.

Packet sniffers like ethereal [GC] extract the information needed from network traffic. For this purpose the first two packets of the PPP authentication phase are needed, the authenticator challenge from the server and the response from the client (we only show the actuall PPP packet without Ethernet data, Internet Protocol data, and Generic Routing Encapsulation data):

Data/Length	Description
`c2 23`	Challenge Handshake Authentication Protocol
`01`	CHAP code: CHAP challenge
1 byte	unique ID
2 bytes	CHAP packet length in bytes (21 + server name length)
`10`	challenge length in bytes
16 bytes	authenticator challenge
x bytes	server name

From this packet we keep the authenticator challenge. The next packet looks like this

Data/Length	Description
`c2 23`	Challenge Handshake Authentication Protocol
`02`	CHAP code: CHAP response
1 byte	unique ID (same as in the previous packet)
2 bytes	CHAP packet length in bytes (54 + login name length)
`31`	response length in bytes
16 bytes	peer challenge
8 bytes	zero
24 bytes	response
1 byte	additional state information
x bytes	client login name

Here we need the peer challenge, the response, and the client login name.

For ease of use, the binary challenges are converted into an ASCII representation of theier hexadecimal values.

3.3 Deriving the 8-byte Hash

Given these three informations, the authenticator challenge, the peer challenge, and the login, the actuall challenge is easily derived: the three byte sequences are hashed with the SHA hash function. The first eight byte of the hash value are the challenge:

```
$ genhash auth-challenge peer-challenge login
```

The output of this program is the 8-byte challenge. The challenges and the login will not be used further.

Although the SHA hash is relatively slow to generate, it does not affect the password search itself since it is only used once, before the actual search is started.

3.4 Extracting 16 Bits from the Password

Next the weakness of the third DES key is exploited. The third key consists of 16 unknown bits and 120 bits of zero, so there are 65536 possible keys. The program nthash encrypts the challenge from the previous step with every possible key and compares the result to the last 8 bytes of the response. The 16 unknown bits of the DES key that produces the correct cipher text are returned.

```
$ nthash challenge response
```

3.5 Searching the remaining Password Space

Now that the last 16 bits of the password hash are known, we can search the remaining password space.

The program crack reads passwords with their NT password hash values from `stdin`, encrypts the challenge with the password hash using DES, and compares the cipher text to the response. As soon as the cipher text matches the response, the password will be printed out and the program terminates.

The program itself does not use the known 16 bits from the password hash, we have to take care of this.

Continuing the dictionary attack, the program should be invoked like this:

```
$ cat hash-dict | grep 16bits$ | crack challenge response
```

4 Conclusions

While testing this software, we used a dictionary of about three gigabytes containing about 74 million words. Equipped with this, we were able to derive all passwords used in our test network in about four hours.

It is true that dictionary attacks tend to fail on good passwords, but it is enough to have one password to break into a system. The step to gaining root access (or doing any other kind of abuse) from there is small.

Furthermore, there are other known security holes that could also be exploited without much effort.

Normally, the file /etc/shadow (or /etc/password on old systems) is regarded one of the most vulnerable points of an unix system [Uni99]. If an attacker can obtain the information in this file, the system is nearly hacked. Using Microsoft's PPTP protocol, information about your passwords is not only publicly available, you also provide additional hints about the passwords, which allow to speed-up the attack by a factor of up to 2^{16}.

With this said, it is clear why we believe Microsoft's PPTP implementation isn't suitable for securing wireless networks.

References

[Eis01] Jochen Eisinger. *Exploiting known security holes in Microsoft's PPTP Authentication Extensions (MS-CHAPv2).* http://mopo.informatik.uni-freiburg.de/pptp_mschapv2/, 2001.

[fBuF00] Bundesministerium für Bildung und Forschung. *BMBF — Förderung von Demonstrationsprojekten für die Funkvernetzung (WLAN) von Hochschulen.* http://wiss.informatik.uni-rostock.de/bmbf/gmd/, 2000.

[GC] Gilbert Ramirez Gerald Combs. *Ethereal — A graphical network traffic analyser.* http://www.ethereal.com.

[Gro90] Network Working Group. *The MD4 Message Digest Algorithm.* http://www.ietf.org/rfc/rfc1186.txt, 1990.

[Gro99] Network Working Group. *Point-to-Point Tunneling Protocol (PPTP).* http://www.ietf.org/rfc/rfc2637.txt, 1999.

[Gro00] Network Working Group. *Microsoft PPP CHAP Extensions, Version 2.* http://www.ietf.org/rfc/rfc2759.txt, 2000.

[IEE99] IEEE. *ANSI/IEEE Std 802.11, Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specification, 1999 Edition.* http://standards.ieee.org/getieee802/, 1999.

[Ley92] Paul Leyland. *Various wordlists and dictionaries.* ftp://coast.cs.purdue.edu/pub/dict/wordlists/, 1992.

[MCL+99] Paul Mackerras, Michael Callahan, Al Longyear, Brad Parker, and Greg Christy. *ppp-2.3.11.* ftp://ftp.samba.org/pub/ppp/, 1999.

[Muf96] Alec Muffett. *Crack 5.0a (Password Cracker).* http://www.users.dircon.co.uk/ crypto/, 1996.

[NB01] David Wagner Nikita Borisov, Ian Goldberg. *Intercepting Mobile Communications: The Insecurity of 802.11 (Draft).* http://www.isaac.cs.berkeley.edu/isaac/wep-draft.pdf, 2001.

[oST93] National Institute of Standards and Technology. *Secure Hash Standard.* U.S. Department of Commerce, 1993.

[SMW99] Bruce Schneier, Mudge, and David Wagner. *Cryptanalysis of Microsoft's PPTP Authentication Extensions (MS-CHAPv2).* http://www.counterpane.com/pptp.html, 1999.

[Uni99] Carnegie Mellon University. *Protecting Yourself from Password File Attacks.* http://www.cert.org/tech_tips/passwd_file_protection.html, 1999.