Attacking NTLM with Precomputed Hashtables

warlord
warlord@nologin.org

Contents

Chapter 1

Introduction

Breaking encrypted passwords has been of interest to hackers for a long time, and protecting them has always been one of the biggest security problems operating systems have faced, with Microsoft's Windows being no exception. Due to errors in the design of the password encryption scheme, especially in the LanMan(LM) scheme, Windows has a bad track in this field of information security. Especially in the last couple of years, where the outdated DES encryption algorithm that LanMan is based on faced more and more processing power in the average household, combined with ever increasing harddisk size, made it crystal clear that LanMan nowadays is not just outdated, but even antiquated.

Until now, breaking the LanMan hashed password required somehow accessing the machine first of all, and grabbing the password file, which didn't render remote password breaking impossible, but as a remote attacker had to break into the system first to get the required data, it didn't matter much. This paper will try to change this point of view.

Chapter 2

The design of LM and NTLM

2.1 The LanMan disaster

By default Windows stores all users passwords with two different hashing algorithms. The historically weak LanMan hash and the more robust MD4. The LanMan hash is based on DES and has been described in Mudge's rant[1] on the topic. A brief recap of the LM hash is below, though those unfamilliar with LM will probably want to read[1].

First of all, Windows takes a password and makes sure it's 14 bytes long. If it's shorter than 14 bytes, the password is padded with null bytes. Brute forcing up to 14 characters can take a very long time, but two factors make this task way more easy. First, not only is the set of possible characters rather small, Microsoft further reduces it by making sure a password is stored all uppercase. That means "test" is the same as "Test" is the same as "tesT" is the same as...well...you get the idea. Second, the password is not really 14 bytes in size. Windows splits it up into two times 7 bytes. So instead of having to brute force up to 14 bytes, an attacker only has to break 7 bytes, twice. The difference is $(keyspace^{14})$ versus $(keyspace^{7}) * 2$. That's a huge difference.

Concerning the keyspace, this paper focuses on the alphanumerical set of characters only, but the entire possible set of valid characters is:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789 %!@\#$%^&*()_-=+‘~[]\{}|\:;"’<>,.?/
```

The next problem with LM stems from the total lack of salting or cipher block chaining in the hashing process. To hash a password the first 7 bytes of it are transformed into an 8 byte odd parity DES key. This key is used to encrypt the 8 byte string "KGS!@#$%". Same thing happens with the second part of the password.

This lack of salting creates two interesting consequences. Obviously this means the password is always stored in the same way, and just begs for a typical lookup table attack. The other consequence is that it is easy to tell if a password is bigger than 7 bytes in size. If not, the last 7 bytes will all be null and will result in a constant DES hash of 0xAAD3B435B51404EE.

As I already pointed out, LM has been extensively documented. "L0phtcrack" and "John the Ripper" are both able brute force tools to break these hashes, and Philippe Oechslin of the ETH Zuerich was the first to precompute LM lookup tables that allow breaking these hashes in seconds[2].

2.2 NTLM

Microsoft attempted to address the shortcomings of LM with NTLM. Windows NT introduced the NTLM(NT LanManager) authentication method to provide stronger authentication. The NTLM protocol was originally released in version 1.0(NTLM), and was changed and fortified in NT SP6 as NTLMv2. When exchanging files between hosts in a local area network, printing documents on a networked printer or sending commands to a remote system, Windows uses a protocol called CIFS - the Common Internet File System. CIFS uses NTLM for authentication.

In NTLM, the protocol covered in this document, the authentication works in the following manner. When the client connects to the server and requests a new session, the server replies with a positive session response. Next, the client sends a request to negotiate a protocol for one of the many dialects of the SMB/CIFS family by providing a list of dialects that it understands. The server picks the best out of those and sends the client a response that names the protocol to use, and includes a randomly generated 8 byte challenge.

In order to log in now, the client sends the username in plaintext(!), and also the password, hashed NTLM style. The NTLM hash is generated in the following manner:

```
[UsersPassword]->[LMHASH]->[NTLM Hash]
```

The NTLM hash is produced by the following algorithm. The client takes the 16 byte LM hash, and appends 5 null bytes, so that the result is a string of 21 bytes length. Then it splits those 21 bytes into 3 groups of 7 bytes. Each 7 byte string is turned into an 8 byte odd parity DES key once again. Now the first key is used to encrypt the challenge with the DES algorithm, producing an 8 byte hash. The same is done with keys 2 and 3, so that there are two additional

8 byte hashes. These 3 hashes are simply concatenated, resulting in a single 24 byte hash, which is the one being sent by the client as the encrypted password.

Mudge already pointed out why this is really stupid, and I'll just recapitulate his reasons here. An attacker capable of sniffing traffic can see the username, the challenge and the 24 byte hash.

First of all, as stated earlier, if the password is less than 8 bytes, the second half of the LM hash always is 0xAAD3B435B51404EE. For the purpose of illustration, let's assume the first part of the hash is 0x1122AABBCCDDEEFF. So the entire LM hash looks like:

```
-------------------------------------------
| 0x1122AABBCCDDEEFF | 0xAAD3B435B51404EE |
-------------------------------------------
```

When transforming this into an NTLM hash, the first 8 bytes of the new hash are based solely on the first 7(!) bytes of the LM hash. The second 8 byte chunk of the NTLM hash is based on the last byte of the first LM hash, and first 6 bytes of the second LM hash. Now there are 2 bytes of the second LM hash left. Those two, padded with 5 null bytes and used to encrypt the challenge, form the third 8 byte chunk of the NTLM hash. That means in the example this padded LM hash

```
------------------------------------------------------
| 0x1122AABBCCDDEE | FFAAD3B435B514 | 04EE0000000000 |
------------------------------------------------------
```

is being turned into the 24 byte NTLM hash. If the password is smaller than 8 characters in size, the third part, before being hashed with the challenge to form the NTLM hash, will always look like this. So in order to test wether the password is smaller than 8 bytes, it's enough to take this value, the 0x04EE0000000000, and use it to encrypt the challenge that got sniffed from the wire. If the result equals the third part of the NTLM hash which the client sent to the server, it's a pretty safe bet to say the password is no longer than 7 chars. It's even possible to make sure it is. Assuming from the previous result that the second LM hash looks like 0xAAD3B435B51404EE, the second chunk of the 24 byte NTLM hash is based on 0x??AAD3B435B514. The only part unknown is the first byte, as this one is based on the first LM hash. One byte, thats 256 permutations. By brute forcing those up to 256 possibilities as the value of the first byte, and using the resulting key to encrypt the known challenge once again, one should eventually stumble over a result that's the same as the second 8 bytes of the NTLM hash. Now one can rest assured, that the password really is smaller than 8 bytes.

Even if the password is bigger than 7 bytes, and the second LM hash does not end with 0x04EE thus, creating all possible 2 byte combinations, padding them with 5 null bytes and hashing those with the challenge until the final 8 byte chunk of the NTLM hash matches will easily reveal the final 2 byte of the LM hash, with no more than up to 64k permutations.

2.3 The NTLM challenge

The biggest difference between the way the LM and the NTLM hashing mechanism works is the challenge. In NTLM the challenge acts like a a salt in other cryptographic implementations. This throws a major wrench in our precomputing table designs, adding 2^{64} permutations to the equation.

Chapter 3

Breaking NTLM with precomputed tables

3.1 Attacking the first part

Precomputing tables for NTLM has just been declared pretty much impossible with todays computing resources. The problem is pre-computing every possible hash value (and then, of course storing those values even if computation was possible). By applying a trick to remove the challenge from the equation however, precomputing NTLM hashes becomes almost as easy as the creation of LM tables. By writing a rogue CIFS server that hands out the same static challenge to every client that tries to connect to it, the problem has static values all over the place once again, and hashtable precomputation becomes possible.

The following screenshot depicts a proof of concept implementation that accepts an incoming CIFS connection, goes through the protocol negotiation phase with the connecting client, sends out the static challenge, and disconnects the client after receiving username and NTLM hash from it. The server also logs some more information that the client conveniently sends along.

```
IceDragon wincatch # bin/wincatch
This is Alpha stage code from nologin.org
Distribution in any form is denied


Src Name: BARRIERICE
IP: 192.168.7.13
Username: Testuser
Primary Domain: BARRIERICE
```

Native OS: Windows 2002 Service Pack 2 2600
Long Password Hash:
3c19dcbdb400159002d8d5f8626e814564f3649f0f918666

That's a Windows XP machine connecting to the rogue server running on Linux. The client is connecting from IP address 192.168.7.13. The username is "Testuser", the name of the host is "BarrierIce", and the password hash got captured too of course.

3.2 Table creation

The creation of rainbow tables to precompute the hashes is a good approach to easily breaking the hashes now, but as harddisks grow bigger and bigger while costing ever less, I decided to roll my own table layout instead. As the reader will see, my approach requires way more harddisk space than rainbow tables do since they are computationally less expensive to create and contain a determined set of data, unlike rainbow tables with their less than 100% probability approach to contain a certain password.

In order to create those tables, the big question is how to efficiently store all the data. In order to stay within certain bounds, I decided to stick to alphanumeric tables only. Alphanumeric, that's 26 chars from a-z, 26 chars from A-Z, and additional 10 for 0-9. Thats 62 possible values for each character, so thats 62^7 permutations, right? Wrong. NTLM hashes use the LM hash as input. The LM hashing algorithm upper-cases its input. Therefore the possible keyspace shrinks to 36 characters, and the number of possible permutations goes down to 36^7. The only other input that needs accounting is the NULL padding bytes used, bringing the total permutations to a bit more than 36^7.

The approach taken here to allow for easy storage and recovery of hashes and plain text is essentially to place every possible plaintext password into one of 2048 buckets. It could easily be expanded to more. The table creation tool simply generates every valid alphanumeric password, hashes it and checks the first 11 bits of the hash. These bits determine which of the 2048 buckets (implemented as files in this case) the plaintext password belongs to. The plaintext password is then added to the bucket. Now whenever a hash is captured, looking at the first 11 bits of the hash determines the correct bucket to look into for the password. All that's left to do now is hashing all the passwords in the bucket until a match is found. This will take on average case $((36^7)/2048))/2$, or 19131876 hash operations. This takes approximately three minutes on my Pentium 4 2.8 Ghz machine. It takes the NTLM table generation tool 94 hours to run on my machine. Fortunately, I only had to do that once :)

The question is how to store more than 36^7 plaintext passwords, ranging in size from 0(empty password) to 7 bytes.

Approach #1: Store each password separated by newlines. As most passwords are 7 byte in size and an additional newline extends that to 8 byte, the outcome would be somewhere around $(36^7) * 8$ bytes. That's roughly 584 gigabytes, for the alphanumeric keyspace. There has to be a better way.

Approach #2: By storing each password with 7 bytes, be it shorter than 7 or not, the average space required for each password goes down from 8 to 7, as it's possible to get rid of the newlines. There's no need to separate passwords by newlines if they're all the same size. $(36^7) * 7$ is still way too much though.

Approach #3: The plaintext passwords are generated by 7 nested loops. The first character changes all the time. The second character changes every time the first has exhausted the entire keyspace. The third increments each time the second has exhausted the keyspace and so on. What's interesting is that the final 3 bytes rarely change. By storing them only when they change, it's possible to store only the first 4 bytes of each password, and once in a while a marker that signals a change in the final 3 bytes, and is followed by the 3 byte that now form the end of each plaintext password up to the next marker. That's roughly $(36^7) * 4$ bytes = 292 gigabytes. Much better. Still too much.

Approach #4: For each character, there's 37 possible values. A-Z, 0-9 and the 0 byte. 37 different values can be expressed by 6 bits. So we can stuff 4 characters into 4*6 = 24 bits, which is 3 byte. How convenient! $(37^7)*3 ==$ 265 gigabytes. Still too much.

Approach #5: The passwords are being generated and stored in a consecutive way. The hash determines which bucket to place each new plaintext password into, but it's always 'bigger' than the previous one. Using 2048 buckets, a test showed that, within any one file, no offset between a password being stored and the next one stored into this bucket exceeded ˜55000. By storing offsets to the previous password instead of the full word, each password can be stored as a 2 byte value.

For example, say the first password stored into one bucket is the one char word "A". That's index 10 in the list of possible characters, as it starts with 0-9. The table creation tool would now save 10 into the bucket, as it's the first index from the start of the new bucket, and it's 10 bigger than zero, the start value for each bucket. Now if by chance the one character password "C" was to be stored into the same bucket next, the number 2 would be stored, as "C" has an offset of 2 to the previous password. If the next password for this bucket was "JH6", the offset might be 31337.

Basically each password is being stored in a base36 system, so the first 2 byte password, being "00", has an index of 37, and all the previous password offsets and the offset for "00" itself of the bucket that "00" is being stored in add up to 37. To retrieve a password saved in this way requires a transformation of the decimal index back into the base36 system, and using the resulting individual numbers as indexes into the char keyspace[].

The resulting table size is $(36^7) * 2 == 146$ gigabytes. Still pretty big, but small enough to easily fit on today's harddisks. As I mentioned earlier the actual resulting size is a bit bigger in fact, as a bunch of passwords that end with null bytes have to be stored too. In the end it's not 146 gigabytes, but 151 instead.

3.3 The big problem

Now there's a big problem concerning the creation of the NTLM lookup tables. The first 8 byte of the final hash are derived from the first 7 byte of the LM hash, which are derived from the first 7 byte of the plaintext password. Creating tables to match the first 8 byte of the NTLM hash to the first 7 bytes of the password is thus possible, but the same tables do not work for the second or even third block of the 24 byte NTLM hash.

The second 8 byte chunk of the hash is derived from the last byte of the first LM hash, and the first 6 byte of the second LM hash. This first byte adds 256 possible values to the second LM hash. While the first 8 byte chunk of the 24 byte LM hash stems purely from a LM hash of a plaintext password, the second 8 byte chunk stems from an undetermined byte and additional 6 byte of a LM hash.

Being able to look up the first up to 7 bytes of the password is a big advantage already though. The second part of the password, if it's longer than 7 bytes at all, can now usually be easily guessed or brute forced. Having determined that the password starts with "ILLUSTR" for example, most often it may end with "ATION" or "ATOR". On the other hand, when applying the brute force approach to this example after looking up the first 7 bytes, it'd require to brute force 4-5 characters until the final password is revealed. Even off-the-shelf hardware does this in seconds. While taking a bit longer, even brute forcing 6 bytes is nothing one couldn't sit out. 7 bytes, however, requires an inconvenient amount of time. That's where being able to look that part up as well would really come in handy. Well, guess what. There is a way.

3.4 Breaking the second part of the password

As described earlier in this paper, the second part of the password, just as the first one, is used to encrypt a known string to form an 8 byte LM hash. Knowing the challenge sent from the server to the client, it is possible to deduce the final 2 bytes of that LM hash out of the third chunk of the NTLM hash. Doing so was explained in section 2.2.

So the final 2 byte of the LM hash of the second half of the original password are known. If a similar approach to breaking the first half of the password is being applied now, looking up the second part of the password as well becomes quite possible.

The key here is to create a set of precomputed LanMan tables that are sorted by the final 2 bytes of the LM hash. So once the final 2 byte of the LM hash are known, a file is thus identified that contains plaintext passwords that when hashed result in a matching 2 byte sequence at the end.

The second chunk of the NTLM hash is derived from 6 bytes that are the start of the hash of one of the plaintext passwords out of the file that just got identified, and a single byte, the first one, which is the final byte of the first LM hash.

Considering the first part of the password broken, that byte is known. So all that's left to do is hash all the possible passwords in the file, fit the single known byte into the first position of a string and concatenate this one with 6 bytes from the just created hash, hashing those 7 bytes again and comparing the result to the second chunk of the NTLM hash. If it matches, the second part of the password has been broken too.

Even if looking up the first part of the password didn't prove successful, the method may still be applied. The only change would be that up to 256 possible values for the first byte would have to be computed and tested as well.

What's really interesting to note here, is that the second set of tables, the sorted LM tables, unlike the first set of NTLM tables, does NOT depend on a certain challenge. It will work with just any challenge, which is usually sniffed or aquired from the wire when the password hash and the username are being taken.

Chapter 4

How to get the victim to log into the rogue server?

The big question to answer is how one can get the victim to log into the rogue server, thus exposing his username and password hash for the attacker to break.

Approach #1: Sending a html mail that includes a link in the form of a UNC path should do the trick, depending primarily on the sender's rhetoric ability in getting his victim to click the link, and the mail client to understand what it's expected to do. A UNC path is usually in the form of \\192.168.7.6\share, where the IP address obviously specifies the host to connect to, and "share" is a shared resource on that host. Due to Microsoft always being concerned about comfort first, the following will happen once the victim clicks the link on a Windows machine. The OS will try to log into the specified resource. When asked for a username and password, the client happily provides the current user's username and his hashed password to the server in an effort to try to log in with these credentials. No user interaction required. No joke.

Approach #2: Getting the victim to visit a site that includes a UNC path with Internet Explorer has the same result.
An image tag like `<img src="\\\\192.168.7.6\ble.jpg">` will do the trick. IE will make Windows try to log into the resource in order to get the image. Again, no user interaction is required. This trick does not work with Mozilla Firefox by the way.

Approach #3: If the rogue server is part of the LAN, advertising it in the network neighbourhood as "warez, porn, mp3, movie" - server should result in users trying to log into it sooner or later. There's no way anyone can withstand the power of the 4 elements!

There's plenty of other ways that the author leaves to the readers imagination.

Chapter 5

Things to remember

Once a hash has been received and successfully broken, it may still not be the correct password, and accordingly not allow the attacker to log into his victims machine. That's due to the password being hashed all uppercase for LM, while the MD4 based second hash actually is case sensitive. So a hash that's been deciphered as being "WELCOME" may originally have been "Welcome" or "welcome" or even "wELCOME" or "WeLcOme" or .. well, you get the idea. Then again, how many users actually apply uncommon spelling schemes?

Chapter 6

Covering it up

Having read this paper the reader should by now realize that NTLM, an authentication mechanism that probably most computers on this planet support, is actually a big threat to hosts and entire networks. Especially with the recently discovered remote Windows exploits that require valid accounts on the victim machines for the attacker to log into first, a worm that makes people visit a website, which in turn makes them log into a rogue server that breaks the hash and automatically exploits the victim is a frightening threat scenario.

Bibliography

[1] Windows NT rantings from the L0pht
http://www.packetstormsecurity.org/Crackers/NT/l0phtcrack/l0phtcrack.rant.nt.passwd.txt

[2] Making a Faster Cryptanalytic Time-Memory Trade-Off
http://lasecwww.epfl.ch/~oechslin/publications/crypto03.pdf