LAN Manager (LM) dan NT LAN Manager (NTLM)

Arie Minandar Anggiat – NIM : 13503074

Program Studi Teknik Informatika, Institut Teknologi Bandung
Jl. Ganesha 10, Bandung
E-mail : if13074@students.if.itb.ac.id

Abstraksi

LM, atau dikenal juga dengan *LAN Manager* dan LANMAN, merupakan protokol otentikasi yang dikembangkan oleh IBM dan digunakan juga oleh Microsoft untuk sistem operasi Windows dengan versi di bawah NT 4.0. Protokol LM menggunakan *LM hash* sebagai format untuk menyimpan password yang panjangnya kurang dari 15 karakter. Pada perkembangannya, protokol ini kemudian diketahui memiliki kelemahan yang berakibat pada ketidakhandalan proses otentikasi. Untuk menangani hal tersebut, Microsoft kemudian menciptakan protokol baru, yaitu NTLM.

Protokol otentikasi NTLM atau *NT LAN Manage*r banyak digunakan pada sistem operasi Windows saat ini. Salah satu penggunaannya antara lain pada protokol SMB. SMB (*Server Message Block*), atau sekarang dikenal dengan nama CIFS (*Common Internet File System*), merupakan protokol level aplikasi yang menangani *sharing* file, printer, *serial port*, dan bermacam-macam komunikasi dalam jaringan komputer, terutama pada komputer-komputer yang menggunakan sistem operasi Windows.

Protokol NTLM ini menggunakan rangkaian *challenge-response* yang melibatkan pertukaran tiga buah pesan antara *client* (pihak yang akan diotentikasi) dan *server* (pihak yang melakukan *request* otentikasi). Algoritma yang digunakan untuk mengalkulasi *response* pada protokol ini berbeda-beda tergantung pada negosiasi awal antara server dan client. Namun pada umumnya, algoritma yang digunakan adalah MD4/MD5 dan DES. Dengan menggunakan mekanisme *challenge-response*, *client* dimungkinkan untuk mengotentikasi dirinya tanpa mengirimkan password kepada server.

Makalah ini akan membahas lebih dalam mengenai mekanisme kerja LM dan NTLM, pemanfaatannya, serta kelemahan yang dimiliki oleh kedua protokol tersebut, berikut dengan proses eksploitasinya. Untuk mendukung pembuatan makalah ini, akan dibangun beberapa aplikasi sederhana dan dilakukan pengujian dengan aplikasi yang sudah ada.

***Kata kunci**:* LM, *LAN Manager*, NTLM, *NT LAN Manager*, *challenge-response*, MD4, MD5, DES, SMB, *Server Message Block*, CIFS, *Common Internet File System*

1. Pendahuluan

Proses otentikasi merupakan hal yang penting dilakukan untuk menjaga kerahasiaan data dan akses terhadap sumber daya tertentu. Dengan proses otentikasi, suatu informasi ataupun sumber daya dapat diatur sedemikian rupa sehingga hanya dapat dimanfaatkan oleh orang atau kelompok tertentu saja. Berbagai macam metode dapat dimanfaatkan untuk otentikasi, namun salah satu metode yang umum dipakai adalah penggunaan *password*.

Pada awal penggunaan *password*, informasi mengenai *password* disimpan dalam bentuk plainteks ke dalam sebuah *file*. Kerahasiaan *password-password* tersebut didasari pada kerahasiaan lokasi penyimpanan *file*. Namun seiring dengan berkembangnya waktu, metode yang digunakan tersebut dirasakan tidak lagi aman. Penyerang dapat saja dengan sabar mencari-cari *file* yang berisi *password-password* tersebut walaupun mungkin membutuhkan waktu yang lama. Selain itu, metode penyimpanan tersebut hanya mungkin berhasil apabila sistem yang digunakan merupakan sistem yang unik. Namun pada kenyataannya, pada saat ini sangat jarang terdapat sebuah sistem yang unik, yang dibangun hanya untuk keperluan satu orang atau satu perusahaan saja. Pembangunan sistem seperti itu tentu saja akan menghabiskan biaya yang sangat besar dan juga kurang menguntungkan. Oleh karena itu, cara menjaga kerahasiaan *password* yang disimpan kemudian menjadi suatu permasalahan yang tidak kalah penting.

Pada saat ini, metode yang sering digunakan untuk menyimpan *password* adalah dengan menggunakan algoritma *hashing*. Dengan metode ini, enkripsi satu arah dilakukan terhadap *password* sebelum *password* tersebut disimpan. Dengan demikian, sekalipun *file* yang menyimpan *password* dapat diperoleh oleh penyerang, nilai *password* sesungguhnya tidak dapat diperoleh.

Selanjutnya, dengan berkembangnya teknologi jaringan, proses otentikasi seringkali perlu dilakukan antara dua atau lebih komputer yang berbeda. Berbagai protokol untuk otentikasi melalui jaringan kemudian diciptakan.

2. *LAN Manager*

LM, atau dikenal juga dengan *LAN Manager* dan LANMAN, merupakan protokol otentikasi yang dikembangkan oleh IBM dan digunakan juga oleh Microsoft untuk sistem operasi Windows dengan versi di bawah NT 4.0. Protokol LM menggunakan *LM hash* sebagai format untuk menyimpan password yang panjangnya kurang dari 15 karakter. Panjang *LM hash* sendiri adalah 16 *byte*. *LM hash* diperoleh dengan proses komputasi sebagai berikut:

a. *Password* yang dimasukkan oleh pengguna dikonversi ke dalam bentuk *uppercase*.
b. *Password* ini kemudian di-*padding* dengan *null byte* hingga menjadi 14 *byte*.
c. Selanjutnya, *password* dengan panjang 14 *byte* ini dipecah menjadi 2 bagian, masing-masing sepanjang 7 *byte*.
d. Masing-masing dari bagian ini kemudian digunakan untuk membangkitkan sebuah kunci DES (*8 byte odd parity*). Dengan demikian terdapat dua buah kunci DES masing-masing sepanjang 8 *byte*.
e. Kemudian, masing-masing dari kunci DES ini digunakan untuk mengenkripsi *string* ASCII konstan “KGS!@#$%”. Hasilnya berupa dua buah *cipher* sepanjang 8 *byte*.
f. Kedua buah *cipher* ini dikonkatenasi dan membentuk 16 *byte LM hash*.

Berikut ini adalah kode program dalam bahasa C untuk membangkitkan *LM hash*. Program ini menerima masukkan *string* dengan ketentuan panjang lebih kecil dari 15 karakter dan menampilkan *LM hash* hasil komputasi. Pada program ini terdapat fungsi ‘deshash’ yang berfungsi untuk mengenkripsi *string* ASCII “KGS!@#$%”, namun implementasi fungsi tersebut tidak ditampilkan dalam makalah ini.

```
/*nama file: lmhash.h*/

#ifndef LMHASH_H
#define LMHASH_H

#include <stdlib.h>
#include <string.h>
#include "des.h"

#define PWDSIZE 14
#define LEFTSIZE 7
#define RIGHTSIZE 7
#define DESKEYSIZE 8
#define HASHSIZE 8
#define LMHASHSIZE 16

char *lmhash(char passwd[]);

#endif
```

```
/*nama file: lmhash.c*/

#ifndef LMHASH_C
#define LMHASH_C

#include "lmhash.h"

const char text[]="KGS!@#$%";
const unsigned char odd_parity[256]={
  1,  1,  2,  2,  4,  4,  7,  7,  8,  8, 11, 11, 13, 13, 14, 14,
 16, 16, 19, 19, 21, 21, 22, 22, 25, 25, 26, 26, 28, 28, 31, 31,
```

```
 32, 32, 35, 35, 37, 37, 38, 38, 41, 41, 42, 42, 44, 44, 47, 47,
 49, 49, 50, 50, 52, 52, 55, 55, 56, 56, 59, 59, 61, 61, 62, 62,
 64, 64, 67, 67, 69, 69, 70, 70, 73, 73, 74, 74, 76, 76, 79, 79,
 81, 81, 82, 82, 84, 84, 87, 87, 88, 88, 91, 91, 93, 93, 94, 94,
 97, 97, 98, 98,100,100,103,103,104,104,107,107,109,109,110,110,
112,112,115,115,117,117,118,118,121,121,122,122,124,124,127,127,
128,128,131,131,133,133,134,134,137,137,138,138,140,140,143,143,
145,145,146,146,148,148,151,151,152,152,155,155,157,157,158,158,
161,161,162,162,164,164,167,167,168,168,171,171,173,173,174,174,
176,176,179,179,181,181,182,182,185,185,186,186,188,188,191,191,
193,193,194,194,196,196,199,199,200,200,203,203,205,205,206,206,
208,208,211,211,213,213,214,214,217,217,218,218,220,220,223,223,
224,224,227,227,229,229,230,230,233,233,234,234,236,236,239,239,
241,241,242,242,244,244,247,247,248,248,251,251,253,253,254,254};

char *lmhash(char passwd[]){
       int i;
       char buffer[PWDSIZE+1];
       char bufleft[LEFTSIZE], bufright[RIGHTSIZE];
       char deskey1[DESKEYSIZE], deskey2[DESKEYSIZE];
       char hash1[HASHSIZE], hash2[HASHSIZE];
       char *result;

       //padding dengan null byte
       memset(buffer,0,sizeof(buffer));
       strncpy(buffer,passwd,sizeof(buffer));

       //konversi ke uppercase
       strupr(buffer);

       //split
       memset(bufleft,0,sizeof(bufleft));
       memset(bufright,0,sizeof(bufright));
       memcpy(bufleft,buffer,LEFTSIZE);
       memcpy(bufright,buffer+LEFTSIZE,RIGHTSIZE);

       //enkripsi
       deshash(hash1, bufleft, text);
       deshash(hash2, bufright, text);

       //konkatenasi
       result = (char *)malloc(sizeof(char)*LMHASHSIZE);
       memcpy(result,hash1,HASHSIZE);
       memcpy(result+HASHSIZE,hash2,HASHSIZE);

       return result;
}

#endif
```

```
/*nama file: main.c*/

#include <stdio.h>
#include "lmhash.h"

int main(int argc, char* argv[]){
       int i;
       char *hash;

       if(argc!=2){
              printf("penggunaan: %s <password>\n", argv[0]);
              return 0;
       }

       if(strlen(argv[1])>PWDSIZE){
              printf("panjang password harus lebih kecil dari "
                                                    "15 karakter\n");
```

```
                return 0;
        }

        hash=lmhash(argv[1]);

        //output hasil
        for(i=0;i<LMHASHSIZE;i++){
                printf("%0.2X",(unsigned char)hash[i]);
        }
        printf("\n");
        free(hash);

        return 0;
}
```

Sebagai contoh, apabila *password* yang dimasukkan oleh pengguna adalah "Welcome", maka *LM hash* yang dihasilkan dalam representasi heksadesimal adalah "C23413A8A1E7665fAAD3B435B51404EE".

Pada protokol *LAN Manager*, *LM hash* inilah yang kemudian dipertukarkan antar-komputer untuk proses otentikasi.

Kelemahan *LAN Manager*

Dengan meninjau kembali prinsip kerja protokol *LAN Manager* dan algoritma *hashing* yang digunakan, dengan mudah kita dapat menemukan beberapa kelemahan pada protokol tersebut.

Pertama-tama, Windows menerima *password* dan memastikan panjang dari *password* tersebut tepat 14 *byte*, dan apabila panjangnya lebih kecil dari 14, maka *password* tersebut di-*padding* dengan *null byte*. *Brute force* terhadap *password* sepanjang 14 karakter tentu akan menghabiskan waktu yang sangat panjang. Karakter-karakter yang mungkin antara lain: A-Z, a-z, 0-9, <spasi> , dan karakter-karakter "%!@\#$%^&*()_-=+‘~[]\{}|\:;"'<>,.?/", yaitu sebanyak 97 karakter. Kombinasi karakter yang mungkin mencapai 98^{14} = 7,536 x 10^{27}. Kalaupun dalam satu milidetik dapat dibangkitkan sejuta kunci, maka waktu yang diperlukan untuk memecahkan kunci dapat mencapai 238,978 miliar tahun. Walaupun demikian terdapat beberapa faktor yang menyebabkan *brute force* menjadi sangat mudah.

Faktor yang pertama adalah jumlah karakter kunci yang sudah cukup kecil ini ternyata dikurangi lagi oleh Microsoft. Microsoft mereduksi jumlah kunci dengan memastikan *password* disimpan dalam bentuk *uppercase*. Hal ini berarti bahwa kata "test" akan menghasilkan *hash* yang sama dengan "Test", begitu juga dengan "tesT" dan "TEST".

Faktor yang kedua, panjang *password* ternyata tidak benar-benar 14 *byte*. Windows memecah password tersebut menjadi dua bagian masing-masing berukuran 7 *byte*. Dengan demikian, penyerang tidak perlu lagi melakukan *brute force* terhadap password dengan panjang 14 *byte*, melainkan cukup memecahkan kunci sepanjang 7 *byte* sebanyak dua kali. Perbandingan antara (kunci yang mungkin)^{14} dan (kunci yang mungkin)^{7} x 2 tentu sangat jauh sekali.

Permasalahan selanjutnya adalah kurangnya proses *salting* atau *cipher block chaining* dalam proses *hashing*. Untuk menghasilkan hash dari *password*, 7 *byte* pertama dari *password* ditransformasikan menjadi 8 *byte* kunci DES (*odd parity*). Kunci ini kemudian digunakan untuk mengenkripsi string "KGS!@#$%". Hal yang sama dilakukan terhadap 7 *byte* bagian kedua dari *password*. Kekurangan proses *salting* menimbulkan dua buah konsekuensi yang menarik. Konsekuensi yang pertama yaitu dengan jelas dapat terlihat bahwa *password* selalu disimpan dengan cara yang sama dan serangan *lookup table* sederhana dapat berhasil dengan mudah. Konsekuensi selanjutnya adalah sangat mudah untuk mengetahui apabila panjang *password* lebih dari 7 *byte* atau sebaliknya. Apabila panjang *password* lebih kecil dari 7 *byte*, maka 7 *byte* terakhir akan bernilai *null*, dengan nilai *hash* DES yang konstan, yaitu "AAD3B435B51404EE".

Proses *brute force* terhadap *LM hash* dapat diimplementasikan sebagai berikut:

a. Memecah *hash* menjadi dua bagian, masing-masing sepanjang 8 *byte*. Apabila kedua bagian tersebut bernilai "AAD3B435B51404EE" maka nilai *password* kosong dan proses dihentikan, sedangkan bila hanya bagian kedua saja yang bernilai "AAD3B435B51404EE" maka panjang password lebih kecil dari 8 karakter.

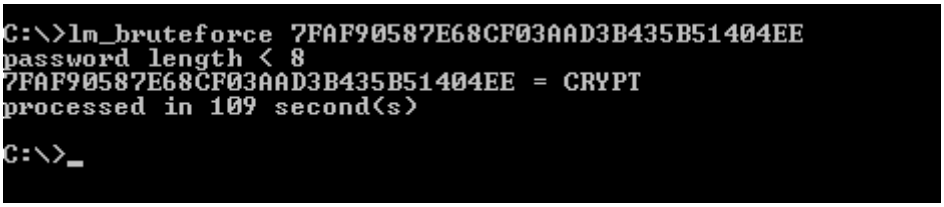
b. Membangkitkan *password* dengan panjang maksimal 7 karakter. Apabila panjang *password* yang dibangkitkan kurang dari 7 karakter, lakukan *padding* dengan *null byte*.
c. *Password* yang dibangkitkan ini kemudian digunakan untuk membangkitkan kunci DES (*odd parity*).
d. Selanjutnya, kunci DES ini digunakan untuk mengenkripsi *string* ASCII konstan "KGS!@#$%" dan menghasilkan sebuah *cipher* sepanjang 8 *byte*.
e. *Cipher* ini kemudian dibandingkan terhadap 8 *byte* bagian pertama dari *password* dan juga terhadap 8 *byte* bagian kedua apabila panjang password lebih besar dari 7.

Dengan menggunakan algoritma di atas, maka proses pembangkitan kunci hanya dilakukan satu kali saja. Dengan demikian diharapkan proses *brute force* menjadi lebih efisien dan cepat.

```
/*nama file: lm_bruteforce.c*/

#include <stdio.h>
#include <string.h>
#include <time.h>
#include "lmhash.h"

char nullhash[]="\xAA\xD3\xB4\x35\xB5\x14\x04\xEE";
char charsequence[]={'A','B','C','D','E','F','G','H','I','J','K',
       'L','M','N','0','P','Q','R','S','T','U','V','W','X','Y','Z',
       '0','1','2','3','4','5','6','7','8','9'};

char currentpwd[7+1], currentseq[7];
int seqlen;
extern const char text[];

int chartoint(char c){
       if(c>='0'&&c<='9') return c-'0';
       else if(c>='a'&&c<='f') return c-'a'+10;
       else if(c>='A'&&c<='F') return c-'A'+10;
       else return 0;
}

char isEquals(char hash1[],char hash2[]){
       int i; char equals=1;
       for(i=0;i<8;i++){
              if(hash1[i]!=hash2[i]){
                     equals=0; break; }
       }
       return equals;
}

int init(){
       seqlen=1;
       memset(currentseq,0,sizeof(currentseq));
       memset(currentpwd,0,sizeof(currentpwd));
       currentpwd[0]=charsequence[0];
}

int inc(){
       //mengembalikan 0 bila semua kemungkinan password
       //sudah dibangkitkan
       int i; char all=0;

       for(i=seqlen-1;i>=0;i--){
              if(currentseq[i]==sizeof(charsequence)-1){
                     currentseq[i]=0;
                     currentpwd[i]=charsequence[0];
                     if(i==0) all=1;
              }else{
                     currentseq[i]++;
                     currentpwd[i]=charsequence[currentseq[i]];
                     break;
```

```
                }
        }

        if(all==1){
                if(seqlen==sizeof(currentseq)) return 0;
                currentpwd[seqlen]=charsequence[0];
                seqlen++;
        }

        return 1;
}


int main(int argc, char *argv[]){
        char *hashstring, lt8=0, found1,found2;
        char hash[16], dhash[8], left[7+1], right[7+1];
        int i;
        time_t start,stop;

        if(argc!=2){
                printf("penggunaan: %s <lmhash>\n", argv[0]); return 0;}

        if(strlen(argv[1])!=32){
                printf("hash lm tidak valid\n"); return 0;}

        //konversi menjadi uppercase
        hashstring=strupr(argv[1]);

        for(i=0;i<16;i++)
                hash[i]=(chartoint(hashstring[i*2])<<4) +
                                        chartoint(hashstring[i*2+1]);

        if(isEquals(hash, nullhash)){
                printf("%s = <empty>\n",hashstring); return 0;}

        if(isEquals(hash+8, nullhash)){
                lt8=1;  printf("password length < 8\n");}

        found1=0; found2=0;
        if(lt8) found2=1;

        memset(left,0,sizeof(left));
        memset(right,0,sizeof(right));
        init();
        time(&start);

        do{
                deshash(dhash, currentpwd, text);
                if(!found1 && isEquals(dhash,hash)){
                        found1=1; memcpy(left,currentpwd,8);}
                if(!found2 && isEquals(dhash,hash+8)){
                        found2=1; memcpy(right,currentpwd,8);}
        }while(inc() && (!found1 || !found2));

        time(&stop);

        printf("%s = %s%s\n",hashstring,left,right);
        printf("processed in %d second(s)\n",stop-start);
        return 0;
}
```

Pada program di atas, *brute force* terhadap *hash* dilakukan hanya dengan menggunakan *string-string* yang terbentuk dari karakter A-Z dan 0-9 saja. *Brute force* dilakukan mulai dari string “A” hingga “Z”, dilanjutkan dengan string “0” sampai “9”, dan kemudian “AA”. Panjang *string* bertambah hingga maksimalnya tercapai, yaitu tujuh. Dengan demikian, proses *brute force* akan mencoba semua kombinasi *string* dari “A” hingga “9999999”.



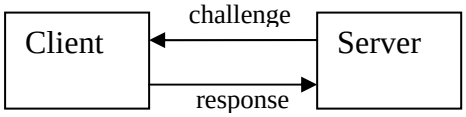
Sebagai contoh, program tersebut dijalankan untuk melakukan brute force terhadap *hash* "7FAF90587E68CF03AAD3B435B51404EE" yang merupakan *LM hash* dari *string* "CrYpT". Proses *brute force* menghasilkan *string* "CRYPT" yang nilai *hash*-nya sama dengan *string* "CrYpT". Panjang *string* dapat diketahui lebih kecil dari 8 karena 8 *byte* terakhir bernilai "AAD3B435B51404EE", yang merupakan nilai *hash* untuk string *null*. Waktu yang dibutuhkan untuk proses *brute force* adalah sekitar 109 detik, seperti yang terlihat pada gambar di atas.

Berbagai optimasi dapat dilakukan terhadap proses *brute force*. Salah satu contohnya adalah dengan melakukan pengubah-ubahan penempatan karakter yang digunakan untuk membangkitkan *string*. Karakter yang paling sering muncul dapat diletakkan di awal, sedangkan karakter yang jarang atau tidak pernah muncul dapat diletakkan di akhir atau bahkan tidak digunakan. Metode optimasi lainnya adalah dengan memulai proses *brute force* pada panjang *string* tertentu, misalnya proses *brute force* dilakukan membangkitkan *string* yang panjangnya 5, dilanjutkan dengan 7, 6, dan sebagainya. Sebagian besar optimasi yang efektif umumnya dilakukan dengan memanfaatkan data-data statistik ataupun dengan metode heurestik.

3. *NT LAN Manager*

Kelemahan protokol otentikasi *LAN Manager* yang diakibatkan oleh kesalahan desain memang tidak dapat dipungkiri lagi. Microsoft kemudian menciptakan *NT LAN Manager* (NTLM), yang diperkenalkan dalam sistem operasi Windows NT, untuk menyediakan metode otentikasi yang lebih kuat. Protokol NTLM yang pertama kali dirilis adalah NTLM versi 1.0. Kemudian, beberapa perubahan dan penguatan dilakukan dalam NTLMv2. Dalam pertukaran *file* antara komputer di dalam *local area network* (LAN), mencetak dokumen pada printer yang berada pada jaringan, atau mengirimkan perintah pada sistem *remote*, Windows menggunakan sebuah protokol yang dinamakan *Server Message Block* (SMB) atau sekarang dikenal dengan *Common Internet File System* (CIFS). Protokol SMB atau CIFS ini menggunakan NTLM untuk proses otentikasi.

Tahapan-tahapan proses otentikasi dengan protokol *NT LAN Manager* adalah sebagai berikut. Pertama-tama, ketika *client* melakukan koneksi ke *server* dan mengirimkan *request session* baru, *server* mengembalikan respon positif. Selanjutnya, *client* mengirimkan *request* untuk melakukan negosiasi protokol yang akan digunakan (salah satu dari sekian banyak dialek yang terdapat dalam keluarga SMB/ CIFS) dengan menyertakan daftar dialek-dialek yang dimengerti oleh *client* pada *request*. *Server* kemudian memilih dialek yang terbaik dan mengirimkan respon yang memuat informasi protokol yang akan digunakan beserta dengan 8 *byte challenge* yang dibangkitkan secara random. Untuk menyelesaikan proses otentikasi, *client* kemudian mengirimkan respon yang berisi informasi *username* dan *domain* dalam bentuk plainteks serta hasil kalkulasi *challenge* dalam format *NTLM hash*, yang diturunkan dari *password*. Respon terakhir ini merupakan bagian yang paling kritis, karena bagian ini berperan dalam membuktikan kepada *server* bahwa *client* mengetahui *password* untuk *account* terkait.



NTLM hash hasil kalkulasi *callenge* yang dikirimkan kepada *server* dibangkitkan melalui tahapan-tahapan sebagai berikut:

a. *Client* mengambil 16 *byte LM hash* ataupun menurunkannya dari *password*, dan menambahkan 16 *byte hash* tersebut dengan 5 *null byte* sehingga membentuk sebuah *string* dengan panjang 21 *byte*.
b. *String* sepanjang 21 *byte* tersebut kemudian dipecah menjadi 3 bagian, masing-masing dengan ukuran 7 *byte*.
c. Selanjutnya, masing-masing bagian sebesar 7 *byte* ini digunakan untuk membentuk kunci DES (*odd parity*) sepanjang 8 *byte*. Dengan demikian, terdapat tiga bagian ini menghasilkan 3 buah kunci DES.
d. Masing-masing kunci DES digunakan untuk mengenkripsi *challenge* dan

masing-masing menghasilkan *hash* sepanjang 8 *byte*.

e. Ketiga *hash* ini kemudian dikonkatenasi dan menghasilkan sebuah *hash* yang panjangnya 24 *byte*. *Hash* inilah yang kemudian dikirimkan kepada *server* sebagai jawaban dari *challenge* yang diberikan.

Untuk lebih jelasnya, akan digunakan contoh sebagai berikut. Asumsikan *challenge* yang dikirimkan oleh *server* adalah "0001020304050607" dan *LM hash* yang diambil atau dibangkitkan oleh client adalah "C23413A8A1E7665fAAD3B435B51404EE", yang merupakan *hash* dari string "welcome".

5 *null byte* ditambahkan kepada *LM hash* sehingga *hash* tersebut menjadi "C23413A8A1E7665fAAD3B435B51404EE 0000000000". *String* tersebut kemudian dipecah menjadi 3 bagian masing-masing berukuran 7 *byte*, yaitu "C23413A8A1E766", "5fAAD3B435B514", dan "04EE0000000000". Ketiga bagian tersebut digunakan untuk membangkitkan 3 buah kunci DES dengan ukuran 8 *byte*, sebut saja 8byteDESKey1, 8byteDESKey2, dan 8byteDESKey3.

8byteDESKey1 digunakan untuk mengenkripsi *challenge* "0001020304050607" dan asumsikan hasilnya adalah "AAAAAAAAAAAAAAAA". Selanjutnya, 8byteDESKey2 digunakan untuk mengenkripsi *challenge* "0001020304050607" dan menghasilkan "BBBBBBBBBBBBBBBB". Begitu juga dengan 8byteDESKey2 yang menghasilkan "CCCCCCCCCCCCCCCC". Ketiga *hash* hasil enkripsi ini kemudian digabungkan dan menghasilkan *hash* 24 *byte* "AAAAAAAAAAAAAAAABBBBBBBBBB BBBBBBCCCCCCCCCCCCCCCC".

Berikut ini adalah aplikasi untuk membangkitkan *NTLM hash*, dengan menerima masukkan berupa *password* dan *challenge*.

```
/*nama file: ntlmhash.c*/

#include <stdio.h>
#include "lmhash.h"
#define CHALLSIZE 8

char *ntlmhash(char *lmhash, char chall[]){
	char buff1[7], buff2[7], buff3[7], hash1[8], hash2[8], hash3[8];
	char *result;

	//memecahkan lm hash ke dalam 3 bagian, masing-masing sebesar 7 byte
	//bagian ketiga dipadding dengan null byte
	memcpy(buff1,lmhash,7);
	memcpy(buff2,lmhash+7,7);
	memset(buff3,0,sizeof(buff3));
	memcpy(buff3,lmhash+14,2);

	//enkripsi challenge dengan menggunakan masing-masing bagian
	deshash(hash1, buff1, chall);
	deshash(hash2, buff2, chall);
	deshash(hash3, buff3, chall);

	//konkatenasi
	result=(char *)malloc(sizeof(char)*24);
	memcpy(result,hash1,8);
	memcpy(result+8,hash2,8);
	memcpy(result+16,hash3,8);

	return result;
}

int main(int argc, char* argv[]){
	int i;
	char *nthash, *hash;

	if(argc!=3){
		printf("penggunaan: %s <password> <challenge>\n", argv[0]);
		return 0;
	}
```

```
        if(strlen(argv[1])>PWDSIZE){
                printf("panjang password harus lebih kecil dari "
                                                        "15 karakter\n");
                return 0;
        }else if(strlen(argv[2])!=CHALLSIZE){
                printf("panjang challenge harus sama dengan 8\n");
                return 0;
        }

        hash=lmhash(argv[1]); nthash=ntlmhash(hash,argv[2]);

        //output hasil
        for(i=0;i<24;i++){ printf("%0.2X",(unsigned char)nthash[i]); }

        printf("\n");
        free(hash); free(nthash);

        return 0;
}
```

Kelemahan *NT LAN Manager*

Berikut ini akan dijelaskan alasan mengapa protokol *NT LAN Manager* seperti yang telah dijelaskan di atas tidak aman. Pertama-tama, seperti yang telah dijelaskan sebelumnya pada penjelasan mengenai *LAN Manager*, apabila panjang *password* kurang dari 8 *byte*, maka setengah bagian kedua dari *LM hash* akan selalu bernilai "AAD3B435B51404EE". Sebagai contoh, asumsikan setengah bagian pertama bernilai "1122AABBCCDDEEFF", maka keseluruhan hash akan tampak sebagai berikut.

1122AABBCCDDEEFF	AAD3B435B51404EE

Ketika transformasi menjadi *NTLM hash* dilakukan, 8 *byte* pertama dari *NTLM hash* yang dihasilkan hanya bergantung pada 7 *byte* pertama *LM hash*. 8 *byte* selanjutnya diperoleh dari 1 *byte* terakhir dari setengah bagian pertama *LM hash* dan 6 *byte* pertama dari setengah bagian kedua *LM hash*. Selanjutnya terdapat 2 *byte* yang tersisa dari bagian kedua *LM hash*. Kedua *byte* ini di-*padding* dengan 5 *null byte* dan digunakan untuk mengenkripsi *challenge* serta menghasilkan 8 *byte* bagian terakhir dari *NTLM hash*.

1122AABB CCDDEE	FFAAD3B4 35B514	04EE0000 000000

Apabila *password* lebih kecil dari 8, maka bagian ketiga sebelum enkripsi *challenge* dilakukan akan selalu tampak seperti gambar di atas. Dengan demikian, pengujian panjang *password* terhadap suatu *NTLM hash* dapat dilakukan dengan membandingkan bagian ketiga dari *NTLM hash* dengan hasil enkripsi *challenge* (yang diperoleh dari proses *sniffing* jaringan) dengan "04EE0000000000". Apabila hasil enkripsi *challenge* sama dengan bagian ketiga dari *NTLM hash*, maka cukup aman untuk mengatakan bahwa panjang *password* dapat tidak lebih dari 7 karakter. Untuk memastikannya ternyata juga tidak sulit. Karena bagian kedua dari *LM hash* untuk *password* yang panjangnya kurang dari 8, maka bagian kedua *NTLM hash* akan memiliki bentuk ??AAD3B435B514. Bagian yang tidak diketahui dengan pasti adalah *byte* pertama, yang berasal dari *byte* terakhir bagian pertama *LM hash*. Dengan melakukan *brute force* sebanyak 256 kemungkinan nilai *byte* pertama, maka panjang *password* dapat dipastikan.

Bahkan apabila panjang *password* ternyata lebih besar dari 7 *byte* dan 2 *byte* terakhir *LM hash* bukanlah "04EE", proses *brute force* terhadap 2 karakter terakhir *password*, yang dilakukan dengan dengan membangkitkan kombinasi nilai-nilai yang mungkin untuk 2 *byte* dan melakukan *padding* dengan 5 *null byte*, dapat dilakukan tidak lebih dari 65536 permutasi.

Selanjutnya, proses *brute force* pada *NTLM hash* kurang lebih sama dengan *brute force* pada *LM hash*. Proses *brute force* dilakukan terhadap bagian pertama *NTLM hash* dengan menghasilkan kombinasi 7 karakter yang kemudian diubah menjadi *LM hash* dan selanjutnya *NTLM hash*. Proses ini tidak hanya memberikan informasi bagian pertama dari *NTLM hash* melainkan juga *byte* pertama dari bagian kedua. Informasi bagian kedua dari *NTLM hash* juga dapat diperoleh dengan cara yang tidak jauh berbeda. Perlu diingat juga bahwa 2 *byte* terakhir *password* yang turut membentuk bagian kedua *hash* ini sudah dapat kita ketahui dengan metode yang dijelaskan sebelumnya.

Memanfaatkan Tabel Prakomputasi

Tabel prakomputasi adalah tabel yang memuat nilai-nilai *password* yang mungkin. Tabel prakomputasi dibangkitkan sebelum proses *brute force* sesungguhnya terhadap suatu nilai *hash* dilakukan. Dengan demikian, proses *brute force* menjadi lebih mudah dilakukan dan cepat, yaitu dengan melakukan pencarian nilai password untuk *hash* tertentu dalam tabel.

Melakukan prakomputasi tabel untuk *hash NTLM* belum lama ini dideklarasikan mustahil untuk dilakukan dengan sumber daya komputer pada saat ini. Permasalahannya adalah prakomputasi dilakukan terhadap semua nilai *hash* yang mungkin, yang tentu saja akan memakan waktu yang sangat lama, dan kalaupun prakomputasi dapat dilakukan, permasalahan lain yang muncul adalah bagaimana menyimpan hasil prakomputasi yang sangat banyak tersebut.

Perlu diperhatikan juga bahwa perbedaan terbesar antara cara kerja mekanisme *hashing* LM dan NTLM terletak pada penggunaan *challenge*. *Challenge* pada NTLM berperan seperti *salt* pada implementasi kriptografi lainnya. Adanya *challenge* ini membuat proses prakomputasi menjadi sulit dilakukan. Untuk *challenge* sepanjang 8 *byte* terdapat 2^{64} kemungkinan nilai *challenge* yang mungkin.

Apabila nilai *challenge* dapat diabaikan dari proses prakomputasi, maka prakomputasi *hash NTLM* menjadi mungkin dilakukan dan bahkan semudah membangkitkan tabel *hash LM*. Berbagai macam trik dapat dilakukan untuk mengeliminasi *challenge* dari proses prakomputasi, salah satunya adalah dengan membangun *server* CIFS yang mengirimkan *challenge* statik untuk setiap *client* yang berusaha untuk berkomunikasi dengan *server* tersebut. *Client* akan mengirimkan informasi *username*, *domain*, dan *NTLM hash* untuk *challenge* statik tersebut. Dengan trik ini, prekomputasi tabel *hash* dilakukan dengan menggunakan nilai *challenge* yang tetap.

Salah satu tabel prakomputasi yang sering digunakan adalah tabel *rainbow*. Pembangkitan tabel *rainbow*, merupakan pendekatan yang baik untuk memecahkan *hash* dengan mudah pada saat ini. Namun seiring dengan bertambahnya kapasitas *harddisk* yang juga disertai dengan turunnya *cost*, maka suatu pendekatan baru dapat dibuat. Pendekatan ini akan membutuhkan kapasitas penyimpanan yang lebih besar dibandingkan dengan tabel *rainbow*. Dengan pendekatan ini, diharapkan probabilitas *password* yang tepat ditemukan dapat mencapai 100% bila dibandingkan dengan menggunakan tabel *rainbow* yang probabilitasnya tidak mencapai 100%.

Pertanyaan terbesar dalam menciptakan tabel adalah bagaimana menyimpan semua data secara efisien. Untuk memenuhi batasan tersebut, pada pembahasan ini akan digunakan tabel alfanumerik. Alfanumerik terdiri dari a-z, A-Z, dan 0-9. Total nilai yang mungkin untuk sebuah karakter alfanumerik adalah sebanyak 62. Dengan demikian permutasi yang harus dilakukan adalah sebanyak 62^7. Namun karena *NTLM hash* menggunakan *LM hash* sebagai input dan algoritma *hashing* LM tidak membedakan nilai antara huruf kapital dan huruf kecil, maka ruang kunci yang mungkin menyusut menjadi 36. Karena itu, jumlah permutasi yang mungkin tinggal 36^7. Kemudian, nilai input lain yang perlu diperhitungkan adalah *null byte*. Akan tetapi, hal ini hanya akan sedikit berpengaruh pada jumlah permutasi yang harus dilakukan.

Pada pendekatan baru yang diambil di sini, untuk memungkinkan kemudahan dalam penyimpanan dan *recovery hash* dan plainteks, maka lokasi penyimpanan dibagi-bagi menjadi 2048 bagian. Program pembuat tabel secara sederhana membangkitkan setiap *password* alfanumerik yang mungkin, menurunkan *hash* dari *password* tersebut, serta kemudian menyimpan *password* tersebut pada salah satu dari 2048 lokasi penyimpanan (yang diimplementasikan sebagai *file*) berdasarkan pada nilai 11 bit pertama dari *hash*. Dengan kondisi seperti ini, apabila terdapat suatu *hash* yang akan dicari nilai *password*-nya, maka pemeriksaan terhadap 11 bit pertama dari *hash* akan menentukan lokasi penyimpanan untuk proses pencarian lebih lanjut. Selanjutnya, proses yang dilakukan adalah melakukan *hashing* terhadap semua *password* dalam lokasi penyimpanan sampai ditemukan *hash* yang sesuai. Proses ini akan membutuhkan operasi *hash* rata-rata sebanyak $(36^7/2048)/2$, atau sekitar 19.131.876. Waktu yang dibutuhkan adalah sekitar 3 menit pada komputer dengan prosesor Pentium IV 2,8 GHz. Adapun proses pembangkitan tabel sendiri akan memakan waktu 94 jam dan untungnya proses ini hanya perlu dilakukan satu kali saja.

Pertanyaan selanjutnya adalah bagaimana cara menyimpan lebih dari 37^7 *password* plainteks dengan ukuran yang bervariasi antara 0 (*password* kosong) hingga 7 *byte*.

Pendekatan pertama adalah dengan menyimpan *password* dengan menggunakan *newline* sebagai pemisah. Karena sebagian besar *password* berukuran 7 *byte* dan *newline* menghabiskan 1 *byte*, maka panjang setiap entri menjadi sekitar 8 *byte*. Keseluruhan kapasitas penyimpanan yang dibutuhkan adalah 36^7 x 8 *byte*, yaitu secara kasar sekitar 548 *gigabyte*, untuk ruang kunci alfanumerik.

Pendekatan kedua adalah dengan menyimpan setiap *password* dalam format 7 *byte*, baik *password* tersebut tepat berukuran 7 *byte* ataupun lebih kecil. Rata-rata panjang setiap entri berkurang dari 8 menjadi 7, karena *newline* tidak lagi diperlukan untuk memisahkan *password* satu dengan yang lainnya. Dengan pendekatan ini, kapasitas penyimpanan yang dibutuhkan adalah sebesar 36^7 x 7 byte.

Pendekatan ketiga adalah sebagai berikut. *Password* plainteks dibangkitkan dengan menggunakan 7 buah *nested loop*. Karakter pertama selalu berubah. Karakter kedua berubah setiap kali ruang kunci karakter pertama habis. Begitu juga dengan karakter ketiga yang bertambah setiap kali ruang kunci karakter kedua berulang dari awal. Hal yang menarik adalah 3 *byte* terakhir tampak jarang berubah. Dengan menyimpan 3 *byte* tersebut hanya pada saat mereka berubah, maka memungkinkan untuk hanya menyimpan 4 *byte* pertama dari setiap *password* dan kadang-kadang sebuah penanda yang menginformasikan perubahan pada 3 *byte* terakhir yang diikuti dengan 3 *byte* terakhir untuk *password* plainteks selanjutnya. Secara kasar, kapasitas yang dibutuhkan adalah sebesar 36^7 x 4 *byte*, atau sekitar 292 *gigabyte*.

Pendekatan keempat adalah sebagai berikut. Untuk setiap karakter terdapat 37 nilai yang mungkin, yaitu A-Z, 0-9, dan *null byte*. 37 nilai yang berbeda dapat direpresentasikan dengan menggunakan 6 bit. Jadi, 4 karakter dapat disimpan dalam 4 x 6 bit = 24 bit, yaitu 3 *byte*. Dengan pendekatan ini kapasitas yang dibutuhkan menjadi 37^7 x 3, yaitu sekitar 265 *gigabyte*.

Pendekatan kelima adalah dengan melakukan penyimpanan *offset* sebagai pengganti *password*. *Hash* memang menentukan lokasi tempat suatu *password* akan disimpan, namun *password-password* pada satu lokasi selalu membesar. Kemudian, sebuah percobaan yang dilakukan dengan menggunakan 2048 lokasi penyimpanan menunjukkan bahwa dalam satu *file*, *offset* antara *password* yang disimpan dengan *password* yang disimpan selanjutnya tidak pernah melebihi 55.000. Dengan menyimpan *offset* terhadap *password* sebelumnya, setiap entri dapat disimpan dalam 2 *byte*.

Sebagai contoh, asumsikan *password* yang pertama disimpan ke dalam *file* adalah “A”. “A” adalah indeks ke 10 dari ruang kunci karakter yang dimulai dari 0-9. Program pembangkitan tabel akan menyimpan 10 ke dalam *file*, karena *password* tersebut selisihnya 10 dari 0, nilai awal untuk setiap *file*. Kemudian bila password selanjutnya adalah “C”, maka yang akan disimpan adalah 2 sebab “C” memiliki *offset* sebesar 2 dari *password* sebelumnya. Lebih jauh lagi, apabila *password* selanjutnya adalah “JH6”, maka *offset*-nya adalah 31337.

Perlu diperhatikan juga bahwa penyimpanan *offset* menggunakan bilangan desimal. Oleh karena itu, untuk memperoleh *password*, transformasi dari desimal menjadi indeks ruang kunci dengan basis 36 (36 kemungkinan karakter) perlu dilakukan.

Ukuran tabel yang dihasilkan dengan pendekatan ini adalah 36^7 x 2, atau sekitar 146 *gigabyte*. Ukuran ini masih cukup besar, namun cukup kecil untuk disimpan ke dalam *harddisk* saat ini. Namun, sebenarnya ukuran aktualnya sedikit lebih besar, karena banyak *password* yang berakhiran dengan *null byte* seperti yang telah disebutkan di awal. Jadi, ukuran yang dibutuhkan bukan 146 *gigabyte*, melainkan 151 *gigabyte*.

Kemudian, terdapat permasalah besar terkait dengan pembangkitan tabel *lookup* NTLM. 8 *byte* pertama dari *hash* final diturunkan dari 7 *byte* pertama *LM hash*, dan 7 *byte* ini diturunkan dari 7 *byte password* plainteks. Dengan demikian, menciptakan tabel untuk mencocokan 8 *byte* pertama dari *NTLM hash* memang mungkin dilakukan, namun tabel yang sama tidak dapat digunakan untuk blok yang kedua dan ketiga dari 24 *byte NTLM hash*.

8 *byte* bagian kedua dari *NTLM hash* diturunkan dari *byte* terakhir blok pertama *LM hash* dan 6 *byte* pertama blok kedua *LM hash*. *Byte* pertama ini menyumbangkan 256 kemungkinan terhadap blok kedua *LM hash*. Bila 8 *byte* bagian pertama *NTLM hash* murni berasal dari *LM hash*, maka bagian kedua terdiri dari 1 byte yang tidak pasti dan tambahan 6 *byte* yang berasal dari *LM hash*.

Dapat melakukan pencarian tabel terhadap 7 *byte* pertama *password* merupakan suatu keuntungan tersendiri. Bagian kedua dari *password*, apabila panjang *password* tersebut lebih besar dari 7, biasanya dapat dengan mudah ditebak atau di-*bruteforce*. Misalnya, sebuah *password* yang dimulai dengan "ILLUSTR", paling sering akan berakhir dengan "ATION" atau "ATOR". Pada sisi lain, apabila pendekatan *brute force* dilakukan, hanya dibutuhkan 4 sampai 5 karakter hingga keseluruhan *password* diketahui. Pada kasus ini, *brute force* dapat dilakukan dalam hitungan detik. Namun apabila bagian *password* terakhir ternyata panjangnya sekitar 6-7 karakter, *brute force* akan sangat melelahkan.

Seperti yang telah dijelaskan sebelumnya, bagian kedua dari *password*, sama seperti bagian pertama, digunakan untuk mengenkripsi string konstan "KGS!@#$%" dan membentuk 8 byte *LM hash*. Selanjutnya, dengan mengetahui *challenge* yang dikirimkan oleh *server* kepada *client*, deduksi 2 *byte* terakhir *LM hash* dari blok ketiga *NTLM hash* dapat dilakukan. Jadi, 2 *byte* terakhir dari blok kedua *LM hash* yang dihasilkan oleh bagian kedua *password* dapat diketahui. Selanjutnya, apabila pendekatan untuk memecahkan bagian pertama *password* kembali diterapkan, melakukan pencarian terhadap bagian kedua *password* menjadi mungkin dilakukan

Kunci utamanya terletak pada sekumpulan set dari tabel prakomputasi LM yang diurutkan berdasarkan pada 2 *byte* terakhir *LM hash*. Jadi, ketika 2 *byte* terakhir dari *LM hash* diketahui, sebuah *file* kemudian diidentifikasi yang ketika *hashing* dilakukan menghasilkan rangkaian 2 *byte* yang cocok.

Bagian kedua *NTLM hash* dapat diturunkan dari 6 *byte* pertama *hash* dari salah satu *password* plainteks pada file yang diidentifikasi sebelumnya dan *byte* tunggal yang merupakan *byte* terakhir dari bagian pertama *LM hash*.

Dengan mempertimbangkan bahwa bagian pertama *password* sudah berhasil dipecahkan, *byte* tunggal tersebut sudah diketahui. Jadi, hal yang tersisa yang harus dilakukan adalah melakukan *hashing* semua *password* yang mungkin yang terdapat dalam *file*. *Byte* tunggal yang diketahui kemudian dikonkatenasi dengan 6 *byte* pertama dari hash yang baru dibangun tersebut. Selanjutnya proses *hashing* terhadap 7 *byte* gabungan tersebut dilakukan, dan hasilnya dibandingkan dengan bagian kedua *NTLM hash*. Apabila perbandingan cocok, maka *password* sudah berhasil dipecahkan.

Bahkan apabila bagian pertama *password* belum berhasil ditemukan, metode ini masih dapat diterapkan. Perubahan yang terjadi hanya pada penambahan 256 kemungkinan nilai *byte* pertama yang harus dikomputasi dan dibandingkan.

Satu hal yang menarik untuk diperhatikan di sini adalah kumpulan tabel yang kedua ini, tidak seperti tabel yang pertama, tidak bergantung pada *challenge* tertentu. Tabel ini dapat digunakan untuk *challenge* apapun, selama *challenge* dapat diketahui, yang biasanya diperoleh dari jaringan bersama dengan *NTLM hash* dan informasi *username*.

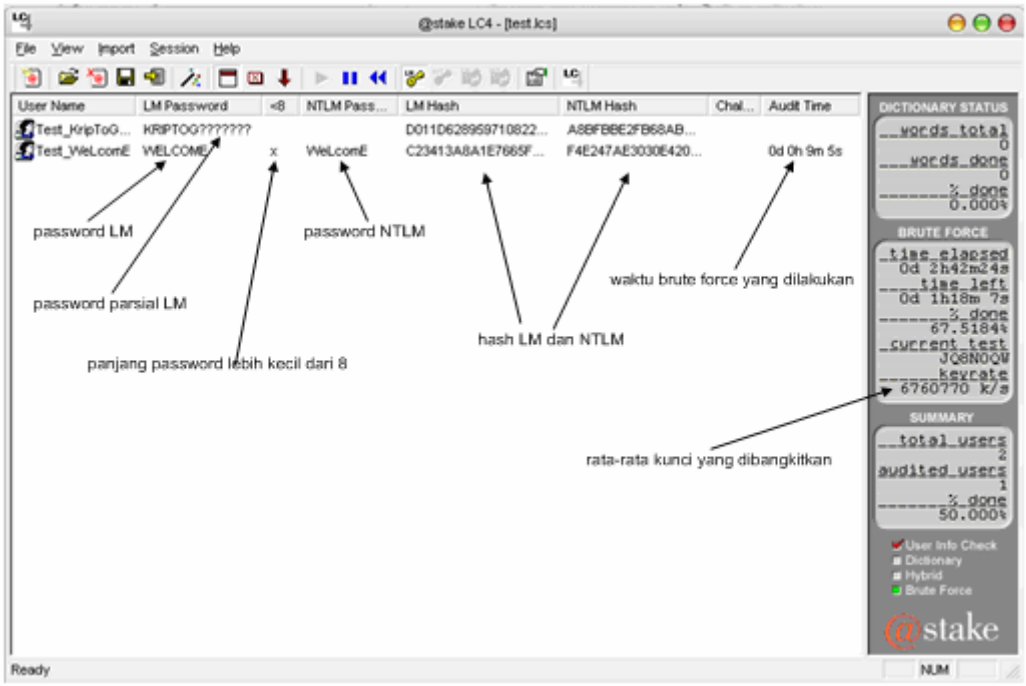
Sebelumnya telah dijelaskan mengenai pembangunan *server* CIFS yang mengirimkan *challenge* statik kepada semua *client* yang berusaha berkomunikasi dengan *server*. Permasalahannya adalah bagaimana membuat *client* menghubungi *server*, sehingga informasi *username* dan *hash NTLM* dapat diperoleh. Terdapat beberapa pendekatan untuk masalah ini.

Pendekatan yang pertama adalah dengan mengirimkan *email* dalam format html yang menyertakan *link* dalam bentuk *path* UNC. *Path* UNC biasanya memiliki bentuk seperti berikut ini: \\192.168.0.1\share, di mana alamat IP 192.168.0.1 menspesifikasikan *host* dan "share" merupakan sumber daya yang di-*share* pada *host* tersebut. Selanjutnya, ketika *client* menekan *link* tersebut, Windows akan secara otomatis berupaya *login* ke dalam *host* tersebut dan apabila *username* dan *password* dibutuhkan, maka Windows akan mengirimkan *username* dan *hash password* dari pengguna.

Pendekatan selanjutnya adalah dengan membuat *client* mengunjungi situs yang memiliki *path* UNC, misalnya dalam bentuk tag gambar seperti berikut: "<img src='\\\\192.168.0.1\gambar.jpg'>". Apabila *client* menggunakan *browser* Internet Explorer, maka informasi *username* dan *hash password* akan secara otomatis dikirimkan bila diminta.

4. Pengujian

Berikut ini adalah *screenshot* aplikasi L0phtCrack 4 yang dikembangkan oleh @stake.



L0phtCrack merupakan aplikasi untuk *auditing* dan *recovery password*. Secara spesifik, aplikasi ini dapat digunakan untuk memecahkan *hash-hash* LM dan NTLM. Aplikasi ini memanfaatkan kelemahan *LM hash* dan *NTLM hash* seperti yang telah dijelaskan sebelumnya. Metode yang digunakan adalah *brute force*, baik itu dengan melakukan pembangkitan *password* ataupun dengan menggunakan daftar *password* (*dictionary*). Selain itu, aplikasi ini mampu melakukan optimasi-optimasi proses *brute force*, salah satunya adalah dengan metode *hybrid* yang menggabungkan penyerangan dengan *dictionary* dan pembangkitan *password*.

Pengujian dilakukan dengan *string password* "KripToGrafi" dan "WeLcomE". Sedangkan metode yang digunakan adalah *brute force* dengan pembangkitan *password*. Dengan menggunakan komputer berprosesor 3 GHz, jumlah kunci yang dapat dibangkitkan setiap detik mencapai 6,8 juta kunci. Apabila kedua nilai *hash* LM dan NTLM diketahui, *brute force* dilakukan terlebih dahulu terhadap *LM hash*. Apabila *password* untuk *hash LM*, yang nilainya dalam huruf kapital semua, sudah diketahui, maka *brute force* terhadap *NTLM hash* baru dilakukan (dengan mengkombinasikan huruf besar dan kecil dari *password* untuk *LM hash*).

Dengan diketahui nilai kedua *hash*, waktu maksimal yang diperlukan untuk memecahkan *hash* adalah 4 jam untuk ruang kunci alfanumerik. Berdasarkan pada pengujian, waktu *hash* dari *string* "WeLcomE" adalah sekitar 9 menit, sedangkan waktu yang diperlukan untuk memecahkan *hash* dari *string* "KripToGrafi" adalah lebih dari 3,5 jam. (*Password* parsial "KRIPTOG" sendiri ditemukan dalam 2,5 jam).

5. Implementasi Protokol Otentikasi *NT LAN Manager*

Protokol *NT LAN Manager*, seperti yang telah dijelaskan sebelumnya, banyak diaplikasikan pada protokol-protokol level aplikasi yang membutuhkan proses otentikasi. Aplikasi-aplikasi yang memanfaatkan protokol *NT LAN Manager* ini kebanyakan merupakan aplikasi yang dibangun di atas sistem operasi Windows. Pada sistem operasi Windows sendiri, protokol *NT LAN Manager* ini banyak digunakan sebagai standar, terutama untuk komunikasi antar-komputer dalam jaringan. Salah satu protokol yang menggunakan *NT LAN Manager* adalah *Server Message Block* (SMB).

SMB, atau sekarang dikenal dengan nama *Common Internet File System* (CIFS), merupakan protokol level aplikasi yang menangani *sharing* file, printer, *serial port*, dan bermacam-macam komunikasi dalam jaringan komputer. SMB pada mulanya dibangun untuk berjalan pada sistem operasi Windows. Namun karena penggunaannya yang cukup meluas, protokol ini diaplikasikan juga pada sistem operasi lainnya, Linux misalnya. Implementasi protokol pada sistem operasi lain ini dilakukan dengan tujuan untuk memungkinkan dilakukannya komunikasi

dengan komputer yang menggunakan sistem operasi Windows (yang merupakan sistem operasi paling banyak digunakan di dunia saat ini). SMB kemudian dikembangkan lebih lanjut dan beberapa fitur ditambahkan. Pengembangan dari SMB ini kemudian diberi label dengan *Common Internet File System* (CIFS).

Selain diimplementasikan pada SMB/ CIFS, protokol *NT LAN Manager* juga dapat diimplementasikan protokol SMTP, POP3, IMAP, dan HTTP. Berikut ini akan dijelaskan lebih lanjut pada protokol HTTP.

Hypertext Transport Protocol (HTTP) merupakan protokol yang digunakan untuk pertukaran informasi dalan *world wide web* (WWW). Seiring dengan berkembangnya Internet, maka protokol ini sering sekali digunakan. Otentikasi dengan menggunakan *NT LAN Manager* biasanya digunakan pada IIS (server HTTP yang dikembangkan oleh Microsoft). Perlu diperhatikan juga bahwa secara *default* HTTP merupakan protokol yang bersifat *connectionless* dan skema yang akan dibahas berikut ini bukanlah skema otentikasi HTTP, melainkan skema otentikasi koneksi pada HTTP. Skema otentikasi dengan menggunakan *NT LAN Manager* akan lebih aman dibandingkan otentikasi *Basic*.

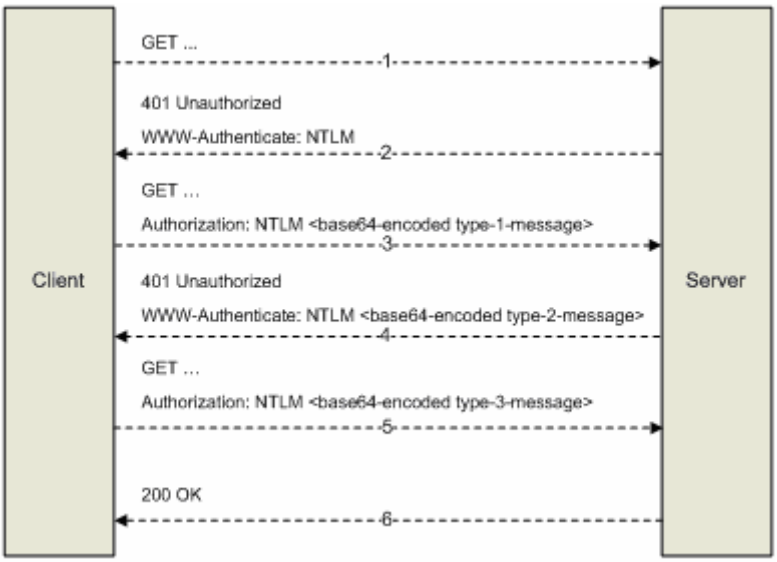
Ketika *client* harus mengotentikasikan dirinya kepada *proxy* atau *server*, maka mekanisme yang tampak pada gambar di bawah ini harus dilakukan. Skema yang digunakan adalah merupakan skema *4-way handshake*, di mana dibutuhkan 4 buah pesan yang dipertukarkan dalam proses otentikasi. Adapun setiap pesan yang dikirimkan menggunakan *header-header* standar HTTP.

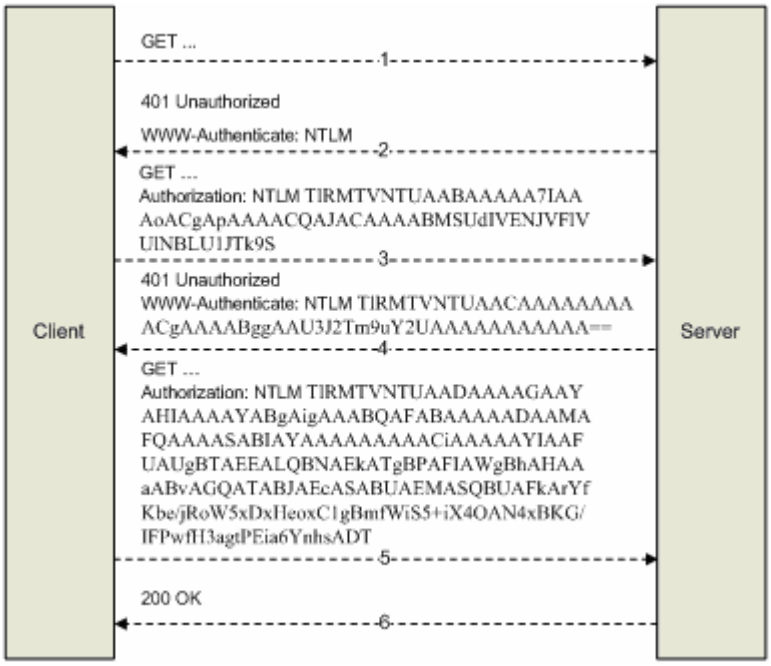
Pertama-tama, *client* menghubungi *server* HTTP untuk melakukan *request* suatu halaman *website*, sama seperti halnya yang biasa dilakukan oleh *web browser*, dengan menggunakan metode GET. Untuk *request* tersebut, *server* kemudian mengirimkan respon yang menyatakan bahwa proses otentikasi dibutuhkan, berikut dengan jenis otentikasinya, yaitu NTLM.

Selanjutnya, *client* akan kembali mencoba menghubungi *server* dengan menggunakan skema otentikasi NTLM. Tahapan-tahapan otentikasi dimulai pada saat ini. Proses otentikasi dilakukan sama seperti mekanisme otentikasi NTLM umumnya, hanya saja *message-message* untuk proses otentikasi disisipkan ke dalam melalui *header-header* protokol HTTP.

Seperti yang terlihat pada gambar, *message-message* yang dikirimkan oleh *client* kepada *server* dijadikan sebagai parameter *header* "Authorization", sebaliknya *message-message* yang dikirimkan oleh *server* kepada *client* disisipkan ke dalam *header* "WWW-Authenticate". *Message-message* tersebut, baik yang dikirimkan oleh *client* kepada *server* ataupun sebaliknya di-*encode* dengan menggunakan sistem *encoding* base64.

Sebagai contoh, apabila nama *host* adalah "LightCity", domain NT adalah "Ursa-Minor", *username* adalah "Zaphod", *password* adalah "Beeblebrox", dan *challenge* yang dikirimkan oleh server adalah "SrvNonce", maka komunikasi antara *server* dan *client* akan tampak seperti pada gambar berikutnya.





Selanjutnya, seperti yang telah dijelaskan sebelumnya, skema otentikasi ini adalah untuk mengotentikasi koneksi, bukan *request*. Oleh karena itu, koneksi harus dijaga terutama pada tahapan kedua proses *handshake*, misalnya antara penerimaan *message* tipe-2 (tahap 4) dan pengiriman *message* tipe-3 (tahap 5). Setiap penutupan koneksi pada saat tahapan ini dilakukan (dari tahap 3 sampai dengan tahap 6) mengakibatkan tahapan ini harus diulang. Selanjutnya, apabila koneksi sudah diotentikasi, *header* "Authorization" tidak perlu dikirimkan kembali selama koneksi masih terbuka.

Dalam implementasinya, karena koneksi harus dijaga, maka penggunaan parameter "Keep-Alive" pada protokol HTTP/1.0 atau penggunaan HTTP/1.1 yang koneksinya persisten harus dipastikan baik oleh aplikasi pada sisi *client* maupun *server*. Selain itu, pada implementasi *server* harus dijamin bahwa respon protokol HTTP/1.0 mengandung *header* "Content-Length" atapun bila protokol yang digunakan adalah HTTP/1.1, respon mengandung *header* "Content-Length" atau menggunakan *encoding* transfer yang terpecah-pecah.

6. Kesimpulan

Tidak dapat dipungkiri protokol otentikasi *LAN Manager* merupakan protokol yang tidak aman digunakan. Apabila ditinjau lebih mendalam, panjang *hash* yang digunakan pada protokol ini sebenarnya cukup panjang, yaitu sebesar 16 *byte* atau 128 bit dan panjang *password* yang dapat diterima juga mencapai 14 karakter. Faktor utama penyebab kegagalan protokol ini terletak pada kesalahan desain. Terdapat banyak sekali tahapan-tahapan dalam protokol ini yang membuat panjang *password* menjadi tidak efektif. Salah satunya adalah pengurangan ruang kunci dengan tidak membedakan huruf besar dengan huruf kecil. Namun yang paling fatal adalah kurangnya *cipher block chaining*. Pada kasus ini, *password* yang panjang 14 karakter dibagi menjadi 2 bagian dan masing-masing dienkripsi, lalu digabungkan kembali. Proses tersebut menjadikan keefektifan password berkurang jauh. Walaupun demikian, kesalahan tersebut masih dapat dimaklumi karena protokol ini memang dirancang pada saat komputer masih menggunakan disket.

NT LAN Manager sebagai pengganti *LAN Manager* juga tidak luput dari kelemahan-kelemahan. Namun sebenarnya kelemahan yang terdapat pada protokol ini tidak lepas dari kelemahan yang terdapat pada protokol LM. Protokol *NT LAN Manager* pada kerjanya menggunakan *LM hash* dan melakukan penguatan terhadap *hash* tersebut. Kemudian, kesalahan ini diperparah dengan proses penguatan yang sederhana dan tidak jauh berbeda dengan LM.

DAFTAR PUSTAKA

[1] *Attacking NTLM with Precomputed Hashtables.* http://www.uninformed.org/?v=3&a=2&t=pdf. Diakses tanggal 12 Desember 2006 pukul 12.30 WIB.

[2] *The NTLM Authentication Protocol and Security Support Provider*. http://davenport.sourceforge.net/ntlm.html Diakses tanggal 12 Desember 2006 pukul 12.00 WIB.
[3] Hertel, C.R. 2003. *Implementing CIFS: The Common Internet File System*. Prentice Hall.
[4] *Windows NT Rantings from the L0pht*. http://www.packetstormsecurity.org/Crackers/NT/l0phtcrack/l0phtcrack.rant.nt.passwd.txt. Diakses tanggal 29 Desember 2006 pukul 11.45 WIB.
[5] *NTLM Authentication Scheme for HTTP*. http://www.innovation.ch/personal/ronald/ntlm.html. Diakses tanggal 31 Desember 2006 pukul 18.00 WIB.