THE PUBLISHING HOUSE
OF THE ROMANIAN ACADEMY

PROCEEDINGS OF THE ROMANIAN ACADEMY, Series A,
Volume 14, Special Issue 2013, pp. 317–327

UNVEILLING THE PASSWORD ENCRYPTION PROCESS UNDER WINDOWS – A PRACTICAL ATTACK

Lucian OPREA*

*Independent researcher, Bucharest, ROMANIA
Corresponding author: Lucian OPREA, E-mail: luc.oprea@gmail.com

In this paper we present a practical attack that is intended to break the encryption process used to protect the users' passwords in some versions of the Windows operating system.

Key words: password encryption, Windows, brute-force attack, password hashes.

1. INTRODUCTION

Local and domain account information, such as user passwords, group definitions, and domain associations are stored by the Windows operating system in registry. By default, the registry keys like *HKEY_LOCAL_MACHINE\SAM*, which holds most of this information, are unreadable even for the system administrator account. *SAM* stands for the Security Accounts Manager and is essentially a database of security information, user permissions and passwords. It is sometimes referred to as the Windows local security database.

In the **2nd section** of this paper we present the structure of the *SAM* registry and the location of the values we need in order to launch the attack.

In the **3rd section** we present the cryptographic transformations applied to the user password. We grouped these transformations in three different "encryption" levels. These levels are applied sequentially, the resulted values being stored into the registry structure. We call these levels "encryption levels" due to the cryptographic processing of the initial and intermediate values and even if these levels contain hash primitives.

In order to recover the password of a Windows account, we implemented 3 different applications. A Windows service and a standalone application were implemented in order to gather all the data we need for the attack and to ease the final step of finding the password. The 3rd application, which recovers the password by brute-force, is implemented in CUDA technology in order to use the processing power of a compatible GPU card.

The **4th section** presents the description of the attack and a practical example of recovering an Administrator password.

Observation: the techniques shown here are strictly for educational purposes and should not be used against systems for which you do not have authorization for.

2. SAM STRUCTURE

The initial information we need in order to launch the attack is stored in Windows registry. The most important registry key in our attack is *HKEY_LOCAL_MACHINE\SAM\SAM\Domains*. This key contains two other subkeys:

– *Account* subkey: contains information about defined user and group accounts. *Administrator* and *Guest* information is stored here, although these accounts are defined by default;

– *BuiltIn* subkey: contains information about operating system users defined by default.

HKEY_LOCAL_MACHINE\SAM\SAM\Domains\Account\Names contains the list of all user accounts on the machine. Every user account in here has a key containing a hex value which is the *RID* (Relative Identifier) of the account or group.

In *HKEY_LOCAL_MACHINE\SAM\SAM\Domains\Account\Users* each account is defined by a different subkey which is the *RID* (Relative Identifier) of that account or group. For the *Administrator* account, that is the account against which we apply the example attack, the *RID* value is the constant value *0x000001F4* and it is not dependent on the computer.

For every user account, this key contains two *REG_BINARY* values (*F* and *V*) which contains some of the data we need [1].

The following information can be extracted from the *V* value:

Table 1

User information stored in *HKEY_LOCAL_MACHINE\SAM\SAM\Domains\Account\Users\{RID}\V*

Address	Information
0x0c	Offset of the account name
0x10	Length of the account name
0x18	Offset of the complete account name
0x1c	Length of the complete account name
0x24	Offset of the comment
0x28	Length of the comment
0x48	Offset of the homedir name
0x4c	Length of the homedir name
0x9c	*Offset of the final password hashes (SAM)*
0xc4	Number of hashes from history

In order to extract a value, we must add *0xcc* to the offset value from the table above.

For example, the final hashes offset is computed as being *V[0x9c] + 0xcc,* the first one starting at the computed offset being *LMHash*, followed by *NTHash*. If *SYSKEY* is enabled, the encrypted hashes are prefixed by *0x10000000* [2].

3. ENCRYPTION LEVELS

The process of encrypting the user password is done in three different steps, on three different levels of encryption. The figure below presents the notations used in this document for referring the output of each of these levels.

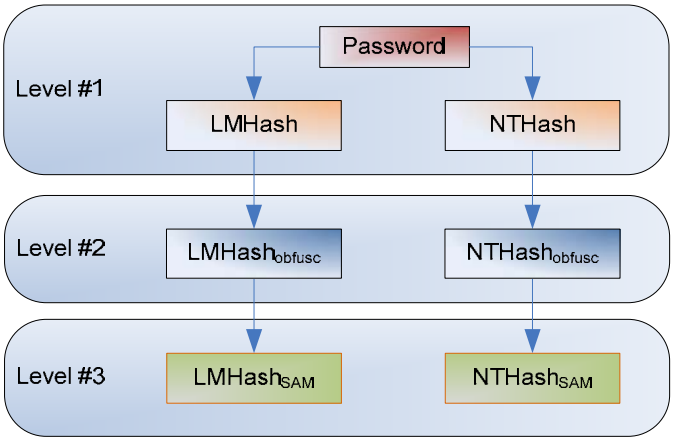

Fig. 1 – Encryption levels.

The *V* registry value presented in the previous section contains the two final hashes for the password of the corresponding user account:

– $LMHash_{SAM}$: LAN Manager hash, was the primary hash that Microsoft LAN Manager and Microsoft Windows versions prior to Windows NT (MS Client for DOS, Windows for Workgroups, Windows 95/98/Me and in some cases Windows NT or newer) used to store user passwords. Support for the legacy LAN Manager protocol continued in later versions of Windows for backward compatibility, but was recommended by Microsoft to be turned off by administrators; as of Windows Vista, the protocol is disabled by default [7];

– $NTHash_{SAM}$: used in Windows NT/2000/2003/XP.

The first encryption level is presented in the picture below.

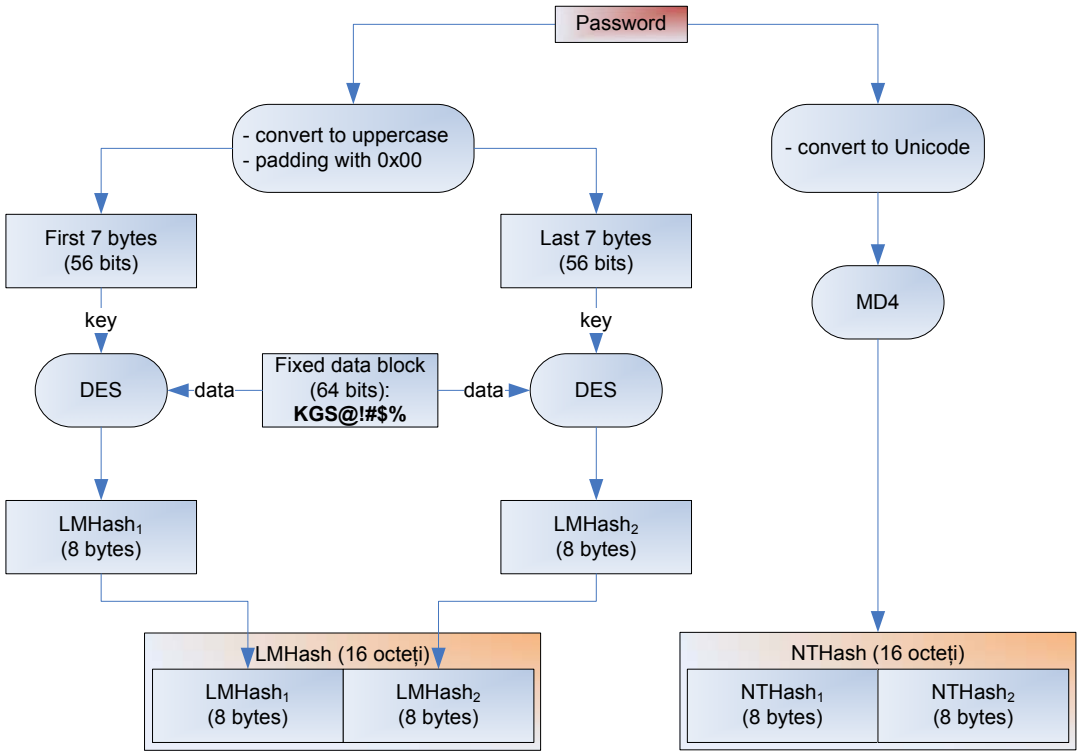

Fig. 2 – The 1st encryption level.

One could observe the following vulnerabilities in *LMHash* computation:

– the password is first converted to uppercase, thus it reduces the number of possible characters;

– the password is used to encrypt a known fixed plaintext, making it susceptible to known plaintext cryptanalysis;

– no salt, IV or any sort of randomness; two identical passwords will generate two identical hashes after in the first level;

– password padding is done with the byte *0x00* to a length of 14 characters. So, a password with at most 7 characters will have *0x00* on all its 7 MSBs; the ciphertext resulted from this fixed block is known so it is easy to find if a password is less than 8 characters;

– the password is split in two parts which then are used independently; in this way, one could attack the two parts separately.

To avoid the *LMHash* vulnerabilities, there are some actions than could be performed:

– use a password with more than 14 characters. The algorithm for *LMHash* generation supports only passwords less than this length, so it will not be generated;

– disable *LMHash* generation (Windows 2000/XP/2003):

- *HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Lsa\NoLmHash: REG_DWORD=1*, or:

- using GPO (Network Security: Do not store LAN Manager hash value on the next password change).

The drawback of disabling the generation of the *LMHash* is that if for an user account this hash is not present, that user cannot authenticate from computers which support only the LM authentication protocol (Windows 95/98 or older).

To add another layer of protection to the hashes provided by the 1st encryption level, Windows adds the 2nd encryption layer, using *DES* and keys derived from the user's *RID*, as presented in the picture below.

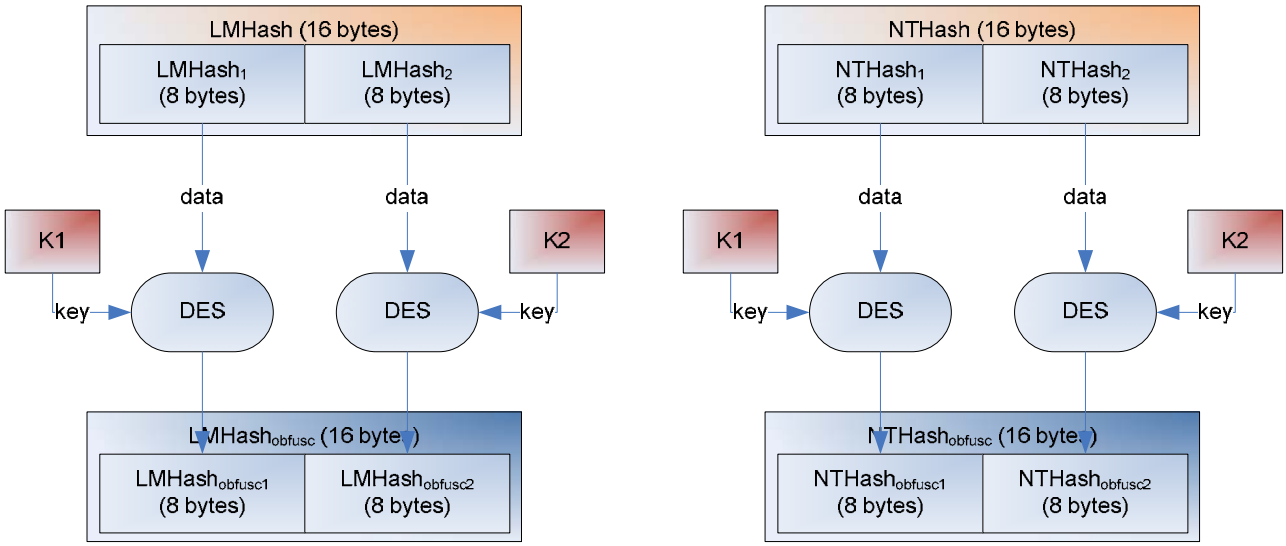

Fig. 3 – The 2nd encryption level.

The keys *K1* and *K2* used to encrypt the first and respectively the second half of the hashes resulted after applying the 1st encryption level are obtained from the user *RID* in the following manner:

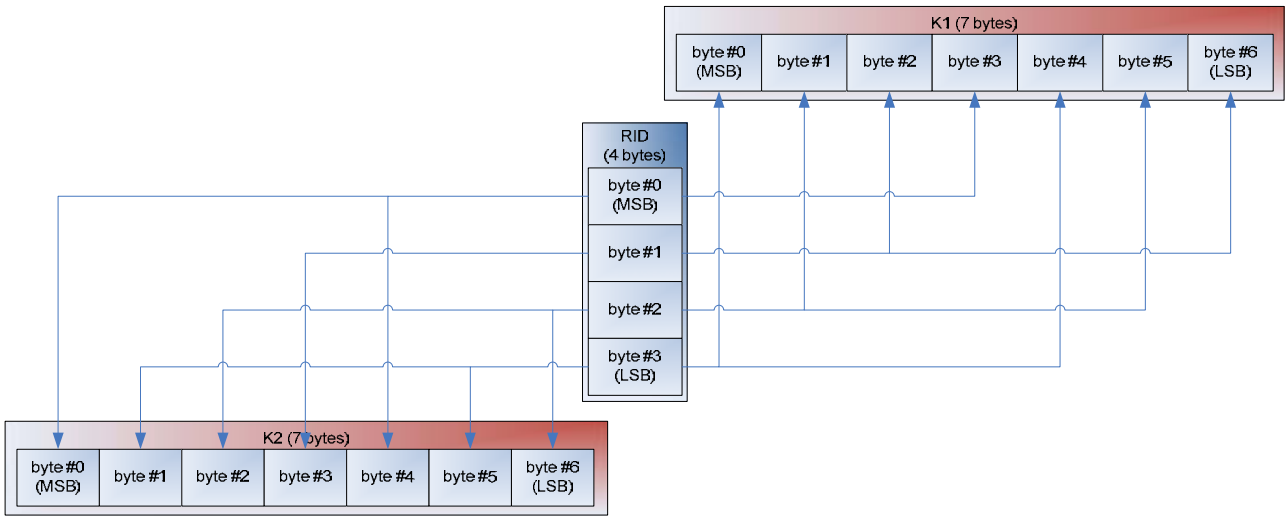

Fig. 4 – Obtaining the encryption keys for the 2nd encryption level.

By applying this level of encryption, even if two different users have the same password, the obfuscated hashes will be different because the users have different *RID* values.

The 3rd encryption level uses the most cryptographic functions, as we see in the picture below.

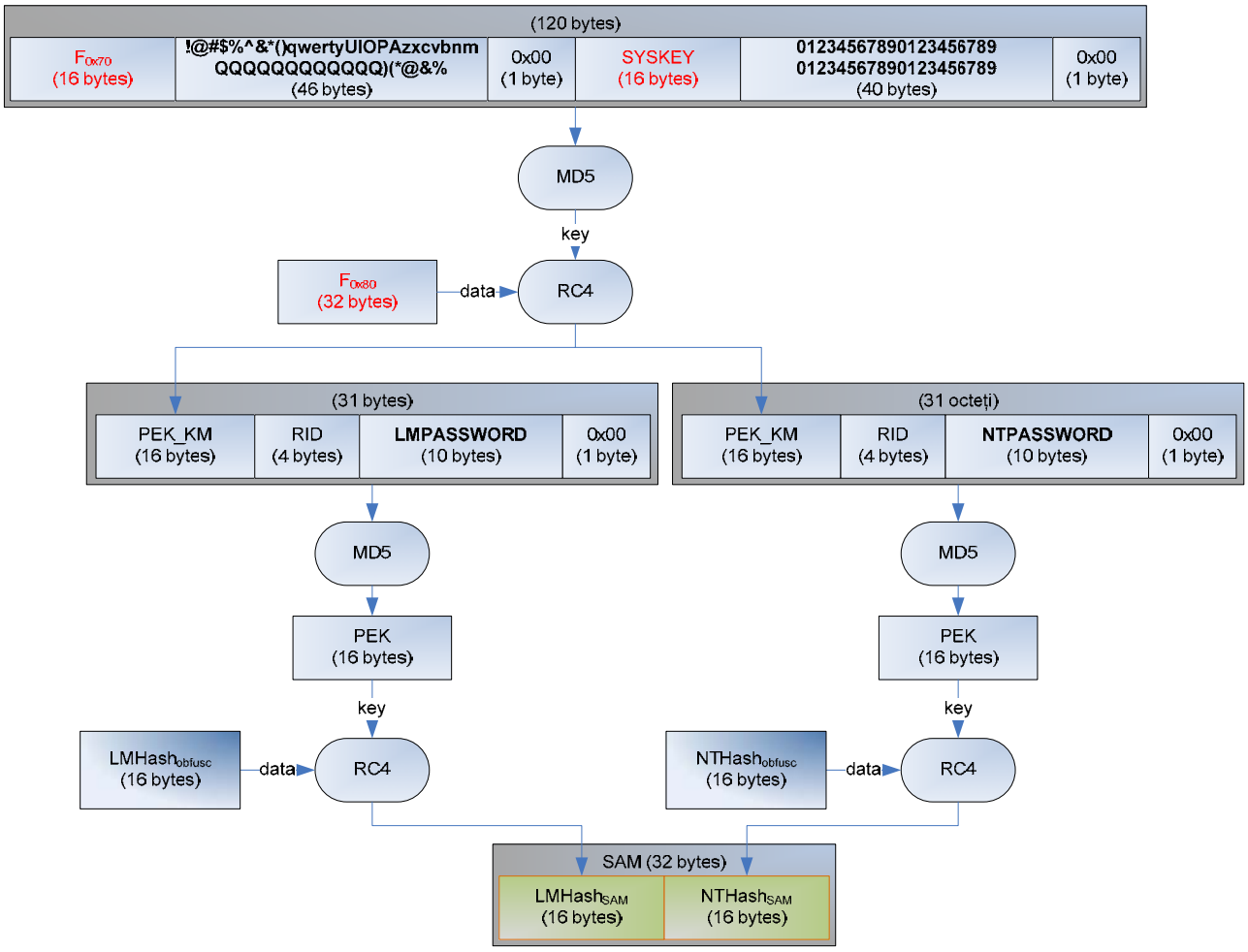

Fig. 5 – The 3rd encryption level.

Now, the only unknown values from this scheme are: *SYSKEY,* F_{0x70} and F_{0x80}.

The information we need in order to compute the *SYSKEY* value is usually stored in Windows registry, under *HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Lsa.* The *SecureBoot REG_DWORD* specifies if *SYSKEY* is stored in registry, if it is derived from a password set by administrator or if it will be stored on a Floppy Disk.

The registry key above contains four subkeys (*JD, Skew1, GBG* and *Data*), with four bytes each. In order to obtain the *SYSKEY* value, these values are concatenated and permuted by the following rules:

P[i] = {8, 10, 3, 7, 2, 1, 9, 15, 0, 5, 13, 4, 11, 6, 12, 14}
SYSKEY[P[i]] = [JD||Skew1||GBG||Data][i]

The values F_{0x70} and F_{0x80} represent 16 and respectively 32 bytes taken from the offsets *0x70* and *0x80* from the *F* value under the subkey *HKEY_LOCAL_MACHINE\SAM\SAM\Domains\Account.*

The final values, $LMHash_{SAM}$ and $NTHash_{SAM}$ are the actual hashes stored in registry.

4. THE ATTACK

Attacking the encryption process of user passwords can be done in three steps:

Step 1: extracting the necessary data from the registry:
– *JD, Skew1, GBG* and *Data*, in order to compute *SYSKEY*;
– F_{0x70} and F_{0x80};
– $LMHash_{SAM}$ or $NTHash_{SAM}$.

In order to achieve this, we have implemented a Windows service running under the System account in order to be able to access and extract these values.

As an example, we demonstrate how to break an Administrator password, so the values $LMHash_{SAM}$ and $NTHash_{SAM}$ were read from the *V* value under the registry key *HKEY_LOCAL_MACHINE\SAM\SAM\Domains\Account\Users\000001F4*. The other values were taken from *HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Lsa* and *HKEY_LOCAL_MACHINE\SAM\SAM\Domains\Account.*

Here are the actual keys extracted from the registry:

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Lsa\JD:
```
84 4E 40 71
```

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Lsa\Skew1:
```
42 E4 22 9D
```

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Lsa\GBG:
```
F4 D2 49 22
```

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Lsa\Data:
```
D5 A7 1F A2
```

HKEY_LOCAL_MACHINE\SAM\SAM\Domains\Account\F:
```
02 00 01 00 00 00 00 00 30 AE 6E 84 7B 68 C6 01
2E 00 00 00 00 00 00 00 00 00 00 00 40 DE FF FF
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 80
00 CC 1D CF FB FF FF FF 00 CC 1D CF FB FF FF FF
00 00 00 00 00 00 00 00 F0 03 00 00 00 00 00 00
00 00 00 00 00 00 00 00 01 00 00 00 03 00 00 00
01 00 00 00 01 00 01 00 01 00 00 00 38 00 00 00
F1 1B C8 92 CB 0B 0C 60 28 52 EC 58 80 2E F3 36
A7 51 FD DA 49 8C 0B CD D8 7D 52 76 56 A4 CA 72
78 7E 30 17 35 38 A7 73 C1 7D D9 D1 12 DB 62 D9
00 00 00 00 00 00 00 00 01 00 00 00 38 00 00 00
55 DC 56 3C 85 53 A0 73 D6 76 E4 9D FE F7 C5 20
0D 89 95 E0 7D 99 37 EA B4 2D E0 42 A4 61 31 53
6A 31 7A 38 F5 5C C9 22 AE 44 E1 23 06 21 4A 06
00 00 00 00 00 00 00 00 02 00 00 00 00 00 00 00
```

So, the values of F_{0x70} and F_{0x80} are:

F_{0x70} = `F1 1B C8 92 CB 0B 0C 60 28 52 EC 58 80 2E F3 36`

F_{0x80} = `A7 51 FD DA 49 8C 0B CD D8 7D 52 76 56 A4 CA 72`
`78 7E 30 17 35 38 A7 73 C1 7D D9 D1 12 DB 62 D9`

HKEY_LOCAL_MACHINE\SAM\SAM\Domains\Account\Users\000001F4\V
```
00 00 00 00 BC 00 00 00 02 00 01 00 BC 00 00 00
1A 00 00 00 00 00 00 00 D8 00 00 00 00 00 00 00
00 00 00 00 D8 00 00 00 6C 00 00 00 00 00 00 00
44 01 00 00 00 00 00 00 00 00 00 00 44 01 00 00
00 00 00 00 00 00 00 00 44 01 00 00 00 00 00 00
00 00 00 00 44 01 00 00 00 00 00 00 00 00 00 00
44 01 00 00 00 00 00 00 00 00 00 00 44 01 00 00
00 00 00 00 00 00 00 00 44 01 00 00 00 00 00 00
00 00 00 00 44 01 00 00 15 00 00 00 A8 00 00 00
5C 01 00 00 08 00 00 00 01 00 00 00 64 01 00 00
14 00 00 00 00 00 00 00 78 01 00 00 14 00 00 00
00 00 00 00 8C 01 00 00 04 00 00 00 00 00 00 00
90 01 00 00 04 00 00 00 00 00 00 00 01 00 14 80
9C 00 00 00 AC 00 00 00 14 00 00 00 44 00 00 00
02 00 30 00 02 00 00 00 02 C0 14 00 44 00 05 01
01 01 00 00 00 00 00 01 00 00 00 00 02 C0 14 00
FF FF 1F 00 01 01 00 00 00 00 00 05 07 00 00 00
```

```
02 00 58 00 03 00 00 00 00 00 14 00 5B 03 02 00
01 01 00 00 00 00 00 01 00 00 00 00 00 00 18 00
FF 07 0F 00 01 02 00 00 00 00 00 05 20 00 00 00
20 02 00 00 00 00 24 00 44 00 02 00 01 05 00 00
00 00 00 05 15 00 00 00 C2 3B F0 34 B0 BE 8D 6D
49 94 37 B1 F4 01 00 00 01 02 00 00 00 00 00 05
20 00 00 00 20 02 00 00 01 02 00 00 00 00 00 05
20 00 00 00 20 02 00 00 41 00 64 00 6D 00 69 00
6E 00 69 00 73 00 74 00 72 00 61 00 74 00 6F 00
72 00 00 00 42 00 75 00 69 00 6C 00 74 00 2D 00
69 00 6E 00 20 00 61 00 63 00 63 00 6F 00 75 00
6E 00 74 00 20 00 66 00 6F 00 72 00 20 00 61 00
64 00 6D 00 69 00 6E 00 69 00 73 00 74 00 65 00
72 00 69 00 6E 00 67 00 20 00 74 00 68 00 65 00
20 00 63 00 6F 00 6D 00 70 00 75 00 74 00 65 00
72 00 2F 00 64 00 6F 00 6D 00 61 00 69 00 6E 00
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF BA 94 D8 01 02 00 00 07 00 00 00
02 00 01 00 B2 2E F9 E6 15 F9 CC ED 53 0E 9F 5B
B5 4F D6 56 02 00 01 00 90 F2 03 0E 6B 71 A1 8F
90 62 FA C7 50 CA E5 18 02 00 01 00 02 00 01 00
```

Final hashes' offset is computed as *V[0x9c] + 0xcc = 0x0164 + 0xcc = 0x230*. So, the two hashes, named $LMHash_{SAM}$ and $NTHash_{SAM}$, are:

$LMHash_{SAM}$ = `B2 2E F9 E6 15 F9 CC ED 53 0E 9F 5B B5 4F D6 56`

$NTHash_{SAM}$ = `90 F2 03 0E 6B 71 A1 8F 90 62 FA C7 50 CA E5 18`

Step 2: "reducing" the encryption levels. This step is about inverting the 3rd and the 2nd levels in order to obtain *LMHash* and *NTHash* (the values resulted after the 1st encryption level).

This step is necessary in order to reduce the complexity and the number of transformations that must be executed in a brute-force attack.

By looking at the cryptographic transformations in the last two levels, we can observe that:

– the 3rd level can be reduced by using the data extracted in the 1st step of the attack and the constants of the 3rd level of the encryption scheme. First, we can compute the *RC4* key, by knowing F_{0x70} and the computed *SYSKEY* value. Then, knowing F_{0x80} we can apply the *RC4* function to obtain *PEK_KM*. Next, using the known *RID* of the user (*0x000001F4* for Administrator) we apply the *MD5* hash on the concatenated values as presented in the 3rd level scheme in order to compute *PEK*, the key used to encrypt $LMHash_{obfusc}$ and $NTHash_{obfusc}$. So, by applying the *RC4* algorithm on the hashes extracted from the registry in the 1st step and using the key computed above, we finally reduce the 3rd encryption level and obtain $LMHash_{obfusc}$ and $NTHash_{obfusc}$;

– the 2nd level can also be reduced, observing that the keys *K1* and *K2* used in the *DES* encryption can be easily computed, knowing the user's *RID*. After computing these keys, we apply the inverse of the *DES* algorithm on $LMHash_{obfusc}$ and $NTHash_{obfusc}$ obtained above, obtaining the *LMHash* and *NTHash*.

After reducing the 3rd and the 2nd encryption levels on the values extracted in the 1st step of the attack, we obtain the following values of *LMHash* and *NTHash*:

LMHash = 90 94 83 75 BA 1B FE 9A AA D3 B4 35 B5 14 04 EE

NTHash = 49 E4 AA 6C 65 AD DA 82 4E DB 19 21 E5 EF E7 27

Step 3: brute-force attack on the 1st encryption level in order to break the account password.

The brute-force attack is applied on *LMHash* or *NTHash* in this way:

– attacking *LMHash* consists in finding 2 *DES* encryption keys, in 2 independent steps: the 1st key determines the first 7 characters of the password and the 2nd key determines the last N-7 characters of the password, N being the length of the password;

– attacking *NTHash* consists in finding the string for which the *MD4* hash of its Unicode conversion equals *NTHash*.

In order to process large password subdomains in parallel for reducing the computational time, one may implement this step using fast parallel technologies such as FPGA, CUDA or other distributed implementations.

We implemented the 3rd step of attacking the *NTHash* using the CUDA technology on an *nVidia® Quadro FX 3700* graphic card [6]. The picture below presents the results of brute-forcing the *NTHash* resulted above after reducing the encryption levels. The password was found in less than 13 minutes, the average processing speed being 72.84 million passwords per second. The same application can be rewritten in order to brute-force DES instead of MD4, thus attacking the *LMHash* instead of *NTHash.*

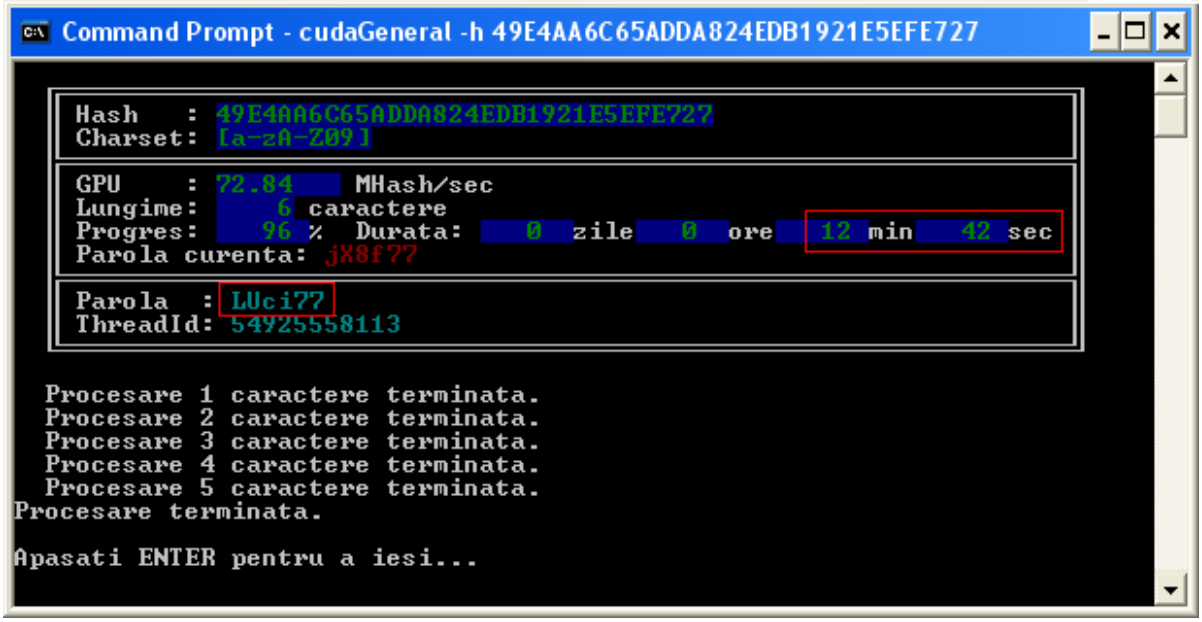

Fig. 6 – The 3rd step of the attack: brute-force example in CUDA implementation (attacking NTHash).

We have implemented a console application (*cudaGeneral*) in order to apply a brute force attack for the NTHash algorithm (that is a brute force attack against the MD4 hash of the Unicode password). The application is written in Microsoft Visual Studio, using the C++ language, a CUDA driver and library, and it is compiled using *nvcc* (NVIDIA CUDA Compiler).

The graphic card is programmed using C-CUDA, which extends the C language by allowing the definition of some C functions, called *kernels*. When called, these kernels are executed in N parallel CUDA threads, as opposite to classical functions which are executed only once [5].

A *thread ID* is automatically assigned to each thread. This ID can be accessed inside the kernel through the *threadIdx* variable (which is a 3-dimensional array). The threads are grouped into *thread blocks* which can be vectors, matrices or fields, depending on the definition of the thread.

As an example, the sum of two matrices A and B can be implemented as follows [4]:

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    // Thread block dimension
    dim3 dimBlock(N, N);

    // Calling the Kernel
    MatAdd<<<1, dimBlock>>>(A, B, C);
}
```

The number of threads in a block is limited by the memory resources of the processing cores. On current GPUs, a thread block can contain up to 512 threads.

However, a kernel can be executed on multiple blocks, so the total number of threads equals the number of blocks multiplied by the number of threads per block.

These blocks are organized into uni-directional or bi-directional block grids. The grid dimension is defined by the first parameter of the <<<...>>> syntax.

The code above transforms as follows:

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i<N && j<N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    // Thread block dimension
    dim3 dimBlock(16, 16);

// Grid dimension
    dim3 dimGrid( (N + dimBlock.x - 1)/dimBlock.x, (N + dimBlock.y - 1)/dimBlock.y );

    // Calling the Kernel
    MatAdd<<<dimGrid, dimBlock>>>(A, B, C);
}
```

In the CUDA technology, for the host (CPU) and the device (GPU) are allocated different memory spaces: global, shared, texture, local or registers. The developer must use carefully the read and write calls from/into these types of memory, as the performance of these actions will vary depending of the memory type and the overall performance will be affected.

The CUDA project contains three main files:

– *cudaGeneral.cu* – contains „device code", executed on GPU. This file will include the kernel and other device functions defined in the next file;

– *cudaGeneral_kernel.cu* – contains the kernel and other „device code", executed on GPU. The code declares the following device memory spaces:

```
__device__ __constant__ char charsetGPU[128];
__device__ __constant__ unsigned int charsetLenGPU;
__device__ __constant__ unsigned long targetHashGPU[4];
__device__ __constant__ unsigned long long idStartPassGPU;
__device__ __constant__ unsigned int passwordLenGPU;
```

charsetGPU[128]: constant memory, containing the working charset
charsetLenGPU: charset length
targetHashGPU[4]: the hash to be attacked
idStartPassGPU: the Id of the first password that will be processed in the current kernel group
passwordLenGPU: the length of the current passwords in process

Functions	Description
void init_GPU_constant_memory (char *charset, unsigned int charsetLen, unsigned char *hashBytes)	Initialization of the constant memory with: – charset – charset length – hash to attack
void init_GPU_constant_memory_id (unsigned long long idStartPassCPU)	Initialization of the constant memory with: – the Id of the first password in the kernel group
void init_GPU_constant_memory_passLen (unsigned int passwordLen)	Initialization of the constant memory with: – password length

(continued)

int print_device_info(bool bDisplayDeviceInfo)	Display CUDA info of the GPU
__device__ bool Transform_md4_GPU(unsigned char *src)	Executed on GPU. Compute the MD4 of the current Unicode password and compares with the hash to attack
__device__ bool Transform_md5_GPU(unsigned char *src)	Executed on GPU. Compute the MD5 of the current password and compares with the hash to attack
__global__ void Brute_Kernel_GPU_NTHash(unsigned long long * resultGPU)	The kernel that calls the hash processing of the current Unicode password and signals the CPU if a match is found
__global__ void Brute_Kernel_GPU_md5(unsigned long long * resultGPU)	The kernel that calls the MD5 processing of the current password and signals the CPU if a match is found

– *cudaGeneral_fnc.cpp* – contains C functions executed by CPU.

Function	Description
void hexAscii_to_bytes(char *src_ascii, unsigned char *dest_bytes)	Convert hex string to byte array
void tid_to_password(unsigned long long tid, char *charset, char *password, unsigned int passwordLen)	Convert the thread Id into the password to process

Assigning different passwords to the processing threads is performed as follows:

```
// thread Id
unsigned long long tid = idStartPassGPU + blockDim.x * blockIdx.x + threadIdx.x;

// aux thread Id
unsigned long long tid_aux = tid;

// pass length
unsigned int passwordLen = passwordLenGPU;

// charset length
unsigned int charsetLen = charsetLenGPU;

// loop Id
unsigned int i = 0;

//9 (pass) || 1 (0x80)
unsigned char curentPass[10] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

// password to be processed by the current thread
while (i < passwordLen)
{
	curentPass[i++] = charsetGPU[tid_aux % charsetLen];
	tid_aux /= charsetLen;
}

// append one bit of '1'
curentPass[i] = 0x80;

// if the computed hash equals the input one, the GPU signals the CPU
// by using the variable resultGPU from global memory (carefully use
// this variable because it drastically decreases the performance)
if (Transform_md4_GPU(curentPass))
{
	resultGPU[0] = 1;
	resultGPU[1] = tid;
}
```

Each thread creates its own password and compares the resulting hash with the target hash stored in Constant Memory. If these are equal, the device function associated with the thread will return a true value to the kernel and the kernel writes the result into the corresponding Global Memory space. In this way, the GPU signals the CPU that the password was found by the thread with the ID *tid*.

Finally, the CPU calls the conversion function (void tid_to_password(unsigned long long tid, char *charset, char *password, unsigned int passwordLen)) and displays the Built-In Administrator password found by the GPU.

5. CONCLUSIONS

Although *LMHash* has its own disadvantages over *NTHash* (reduced number of characters due to uppercase conversion, susceptible to known plaintext attacks, lacks the source of randomness, independent processing of password halves etc.), the bit level permutations in *DES* makes attacking the 1st encryption level harder and more challenging to implement than *MD4*, making the 3rd step of the attack more time consuming in the process of brute-forcing the password.

The time to finalize the attack depends with a higher degree on the 1st encryption level (the 3rd step of the attack). The other two encryption levels can be reduced in a small amount of time as long as the necessary values (F_{0x70}, F_{0x80}, *JD, Skew1, GBG* and *Data*) are extracted from the protected registry keys on the attacked machine.

The last two steps of the attack can be conducted offline, thus no interaction with the attacked computer or network is needed.

Most of the published papers related to defense techniques against password cracking recommend disabling the *LM* hashing or enabling the use of *SYSKEY*. In this paper, we presented a practical attack that works even if these recommendations are applied. One of the best security measures still remains the use of complex passwords that are frequently changed.

REFERENCES

1. Dobromir Todorov, *Mechanics of User Identification and Authentication – Fundamentals of Identity Management*, Auerbach Publications – Taylor & Francis Group, 2007.
2. Todd Sabin, *BindView Security Advisory*, http://packetstorm.foofus.com/9912-exploits/bindview.syskey.txt, 2009.
3. http://xfocus.net/articles/200306/550.html.
4. Nvidia Cuda™, *Programming Guide, Version 2.2.1*, 2009.
5. David Kirk, Wen-Mei Hwu, *CUDA (Ch. 1 to 7),* 2006-2008.
6. http://www.nvidia.com/cuda.
7. http://en.wikipedia.org/wiki/LM_hash.

Received July 25, 2013