Recovering Windows Secrets and EFS Certificates Offline

Elie Burzstein
Stanford University

Jean Michel Picod
EADS

Abstract

In this paper we present the result of our reverse-engineering of DPAPI, the Windows API for safe data storage on disk. Understanding DPAPI was the major roadblock preventing alternative systems such as Linux from reading Windows Encrypting File System (EFS) files. Our analysis of DPAPI reveals how an attacker can leverage DPAPI design choices to gain a nearly silent backdoor. We also found a way to recover all previous passwords used by any user on a system. We implement DPAPI data decryption and previous password extraction in a free tool called DPAPIck. Finally, we propose a backwards compatible scheme that addresses the issue of previous password recovery.

1 Introduction

DPAPI (Data Protection Application Programming Interface) is the cryptographic programming interface offered by Microsoft since Windows 2000 for safe storage of sensitive data on disk. In a nutshell, DPAPI is a cryptographic scheme that provides a transparent way to encrypt data with a key derived in a secure manner from the user password. Many popular applications such as GTalk, Windows Mail, Internet Explorer and system components such as Encrypting File System (EFS) rely on DPAPI to securely store their data on disk. Surprisingly, this key component of Windows security has very sparse documentation, covering only its public interface and giving no details of its internal structure. The documentation instead states that the DPAPI "blob" that holds the encrypted data is "*an opaque structure*" [8]. Because of this lack of documentation and the extreme complexity of DPAPI's internal structure, reverse-engineering DPAPI and building a portable implementation has been a long lasting challenge, on-going for almost 10 years. Many such attempts were made by various open-source teams hoping to provide full Windows emulation and full NTFS support. However, no team has been able to complete a fully working implementation. For example, in 2005, Kees Cook from the Wine project wrote about the DPAPI blob : "*The encryption is symmetric, but the method is unknown. However, since it is keyed to the machine and the user, it is unlikely that the values would be portable*" in one of the Wine header files.

There are at least three main reasons why DPAPI needs to be reverse-engineered and re-implemented: First DPAPI is used to encrypt sensitive information. Therefore, until DPAPI is completely understood, there is no hope for a full implementation of the NTFS file system because EFS private keys cannot be decrypted. Without a re-implementation, emulators like Wine will not be able to fully support Windows core applications such as Internet Explorer, because they cannot access protected data such as stored passwords. Secondly, without a proper re-implementation, it is impossible to migrate offline data stored under EFS from one disk to another, since the files cannot be decrypted. Accordingly, it is impossible to build efficient Windows offline forensic tools because they do not have access to EFS files and sensitive information. Finally, DPAPI is extensively used by many popular applications as a cryptographic blackbox for data-protection, thus making its security a legitimate concern. Without being fully reverse engineered, and in the absence of source code, it is impossible to vouch for DPAPI's security. As this paper shows, auditing DPAPI yields surprising results.

In this paper, we present what is to the best of our knowledge the result of the first complete reverse engineering and audit of DPAPI [1]. We also provide the first tool that is able to decrypt DPAPI secrets offline in a generic manner [5]. Other researchers, such as Nir Sofer, offer tools that only decrypt application specific secrets [11]. Moreover, we confirmed during our tests that these tools do not fully handle offline recovery because they do not have a complete understanding of how DPAPI works. Therefore, our results will also make these tools better.

[1] We did present the preliminary results of this work, without the improved DPAPI scheme, at BlackHat DC 2010

More precisely, by reverse-engineering DPAPI, we were able to accomplish three breakthroughs: First, we were able to decrypt DPAPI data offline. This allows us to create the long-awaited tool that can perform forensics on DPAPI data and provide a way to migrate EFS encrypted files. This breakthrough will also positively impact the open source community, as being able to decrypt and access EFS certificates will open the door to a full implementation of NTFS on non-Windows systems. Second, we found a way to exploit a lack of verification in DPAPI design that allows attackers to replace the master key with a key of their choice and put a process in place that prevents this key from expiring. This allows attackers to backdoor DPAPI in an nearly silent way that guarantees their ability to decrypt EFS files and DPAPI secrets, even if the user changes passwords or patches the system. The method only requires tampering with DPAPI timestamps. Finally, we were able to exploit DPAPI design choices to recover the hashes of all previous passwords used by any users on the system. To demonstrate the feasibility of our recovery method, we implemented the recovery of the hashes in our tool, DPAPIck, and the hash cracking in our password cracker, called Nightingale.

To address the recovery of previous user passwords issue, we also propose a backwards compatible improved password encryption scheme that does not rely on previous passwords.

2 Background

Microsoft has added numerous mechanisms to Windows to protect user data over the years. Figure 1 depicts a high level overview of the relationships between the key security mechanisms that interact with DPAPI.

This diagram emphasizes the complex inter-relation between these mechanisms, with DPAPI playing a central role as the API responsible for tying the encryption key to the user password. The following three mechanisms are linked to DPAPI:

- **Crypto API**: The Windows crypto API provides an implementation of the main cryptographic algorithms, including SHA1, 3DES and AES. DPAPI uses this API as its default cryptographic provider.
- **EFS**:The Encrypting File System (EFS) is a "filter" that provides filesystem-level encryption on Windows that was introduced in version 3.0 of NTFS. It is used to encrypt specific files, not the entire volume like BitLocker (see below). The private key needed to decrypt

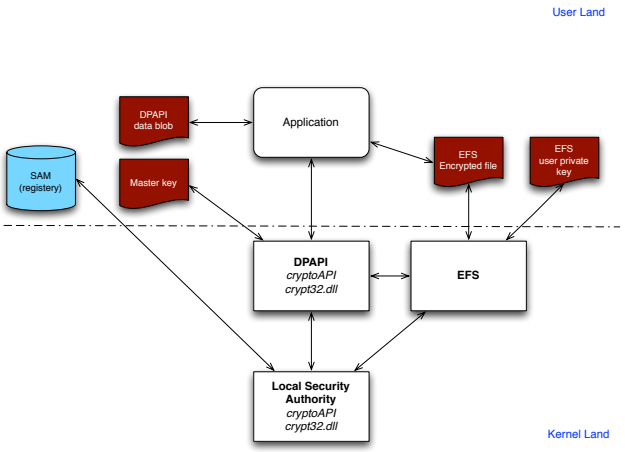

Figure 1: Relationships between Windows security mechanisms

EFS files is encrypted on the disk through DPAPI. Being able to decrypt DPAPI data offline will therefore allow alternative OSes such as Linux and also forensic tools to read encrypted files.

- **Security Accounts Manager**: The Security Accounts Manager (SAM) is the Windows password database. It is stored as a registry file and contains LM or NTLM hashes of user passwords.

For completeness, we also describe the two following data security mechanisms even though they do not interact with DPAPI, since they are also used to protect files:

- **BitLocker**: BitLocker Drive Encryption is a full-disk encryption feature included in some editions of Windows Vista, Windows 7, and Windows Server 2008. By default, it uses 128-bit AES encryption in CBC mode. It relies on a Trusted Platform Module (TPM) to store its keys.
- **CardSpace**: Windows CardSpace, also known at InfoCard, is Microsoft's client software for the Identity Metasystem. It aims at securely storing users' digital identities with a consistent UI.

3 DPAPI overview

In this section we provide an overview of how DPAPI works. We detail the main DPAPI structures and their uses. We also explore the relations that exist between the three different types of keys used by DPAPI. In addition,

we show how to decrypt DPAPI-protected data and also the changes to DPAPI across versions of Windows.

3.1 DPAPI functions

DPAPI exposes two main functions [8] to applications, **CryptProtectData()** and **CryptUnprotectData()**. **CryptProtectData()** takes the supplied data, encrypts them and returns a **DPAPI blob**. Conversely, **CryptUnprotectData()** decrypts a **DPAPI blob** and return the data in the clear. The documentation refers to the **DPAPI blob** as an "*opaque protected data blob*" and therefore does not explain its structure or how the data are encrypted. This lack of information has persisted since the release of Windows 2000/XP and prevented the development of offline forensic tools and the recovery of EFS encrypted files.

To get a sense of what DPAPI does, we will examine the parameters supplied to the functions exposed by DPAPI. Since these two functions are very similar, we will only discuss the parameters for CryptUnprotectData(), as it is enough to illustrate what is supplied by the user:

```
BOOL WINAPI CryptUnprotectData (
DATA_BLOB *pDataIn,
LPCWSTR *ppszDataDescr,
DATA_BLOB *pOptionalEntropy,
PVOID pvReserved,
CRYPTPROTECT_PROMPTSTRUCT
*pPromptStruct,
DWORD dwFlags,
DATA_BLOB *pDataOut)
```

- **pDataIn**: is a pointer to a **data blob** that contains the encrypted data.
- **ppszDataDescr**: is an optional description that will be stored along with the encrypted data in the data blob, as we will see below.
- **pOptionalEntropy**: is optional entropy provided by the application that will be added to the key derivation as explained in Sec 4. By default, DPAPI already uses different entropy for each blob, so in practice adding additional entropy does not improve encryption security. According to the documentation, its purpose is to allow applications relying on DPAPI to mitigate the risk of having their secrets stolen by another application. In our test, we found that only GTalk used the conditional entropy, with its value is stored in a registry key and therefore not a real hurdle for the attacker.
- **pvReserved**: Unused currently.
- **pPromptStruct**: Used to prompt a window to the user that requests an additional password for the user. This password will be used to derive the DPAPI blob key if it exists. To the best of our knowledge, no application use this feature.
- **dwFlags**: Various flags that provide the ability to test the validity of the key and reset it.
- **pDataOut**: is a pointer to the data_blob that contains the data in the clear.

From this function call, it is very difficult to tell how DPAPI operates, aside from the implication that DPAPI must use the user login password to encrypt the data because DPAPI function calls do not require a key. We now examine the complex derivation scheme that DPAPI uses to tie the user login password to the data blob and then take a look at DPAPI data structures and the decryption process.

3.2 Derivation scheme

Figure 2 depicts how DPAPI uses a derivation scheme with three types of keys to tie the user password to the encrypted data.

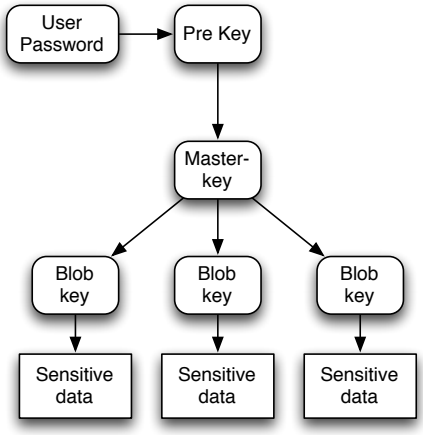

Figure 2: Derivation scheme overview

As depicted in the figure 2, DPAPI uses the following three kinds of keys to tie the user password to the user sensitive data:

- The **Pre key**: This key is used to decrypt the master-key and is derived from the user password. This level of encryption serves two purposes: first it allows users to change their passwords without having to change all the

DPAPI keys. Second, it allows Windows to renew the master key without forcing users to change their passwords.

- The **Master key**: This is a 512-bit random key used to derive all of the DPAPI Blob keys. This key is renewed every three months by Windows. This renewal process is passive and only occurs when DPAPI is called. This passive process means that every master-key ever created needs to be kept, because DPAPI has no way to tell if a master-key is still used by a blob. Accordingly, all the master keys are stored in the user keyring and a way to select the current master key has been implemented by Microsoft. We discuss how this master key selection process can be exploited to backdoor DPAPI in section 5.
- The **Blob key** : This key is directly used to encrypt and decrypt the data. This level of encryption exists to allow each program to add an additional password and/or a second salt, like Google does for GTalk. As explained previously, this information is passed as a parameter to DPAPI functions and therefore is not stored on disk. Instead, it must be stored in the data blob itself.

As one may observe, the entire cryptographic scheme depends on the Pre key, derived from the user password. This makes it challenging to migrate user access to encrypted files across a password change. To address this issue, the straightforward solution would be to re-encrypt all the master keys with the new password. However, Microsoft chose a different approach that consists of re-encrypting only the last key(s). We believe that the rationale behind this decision is to ensure that users do not wait too long after changeing their passwords, as re-encrypting all master key may result in unacceptable wait times. Our benchmark with DPAPIck, presented in Sec. 7, supports this hypothesis. The decision to re-encrypt only a few master keys requires that all previous password hashes be stored. Otherwise, users will not be able to access their data encrypted with a master key that was not re-encrypted with their new password. Storing all the previous password hashes is the only option because, as explained earlier, Windows has no way to tell if a given master key is still used by a DPAPI blob. As undesirable as this scheme seems, we observed it in action in Windows: all the previous password hashes are stored in a file called CREDHIST which is located in the user key-ring directory. We discuss how we leveraged this design to recover all of the previous password hashes belonging to a user in section 5 and describe our implementation in section 7.

3.3 DPAPI Key structures

DPAPI uses two main data structures: the **data blob** and the **master key blob**. In this section we take a closer look at the key fields present in these structures. The reader interested in a more complete description of DPAPI data-structures and algorithms can find it in our technical report [2].

3.3.1 The Data Blob Structure

The **Data Blob** is the opaque structure used by DPAPI to store the encrypted version of sensitive user-data and the meta-data required to decrypt them. The main elements contained in this structure are :

```
struct dpapi_blob_t {
  DWORD cbProviders;
  GUID  *arrProviders;
  DWORD cbKeys;
  GUID  *arrKeys;
  DWORD ppszDataDescrSize;
  WCHAR *ppszDataDescr;
  DWORD idCipherAlgo;
  DWORD idHashAlgo;
  BYTE  *pbSalt;
  BYTE  *pbEncData;
  BYTE  *pbHMAC;
};
```

- **cbProviders** is the number of cryptographic providers. During our tests, we only found blobs with a single cryptographic provider.
- **arrProviders** is the array of the cryptographic providers' GUID. This is used to tell DPAPI which cryptographic provider to use for the cryptographic functions. We believe this mechanism is used to provide a way for organization to supply their own cryptographic primitive and be able to use alternative ciphers, e.g. "Blowfish" instead of AES, to encrypt data. It might also be useful to deal with cryptographic regulation.
- **arrKeys** is the array of master key ID used to encrypt the data. In theory it seems possible to have multiple master keys that can encrypt the data. This mechanism is likely used either for the active domain backup key system or for the compatibility key. We were unfortunately unable to test these hypotheses, because, as explained below, we could not force the compatibility mode.
- **ppszDataDescr** contains the optional description string that can be supplied by the developer when calling

CryptProtectData() (Sec. 3.1). If the developer supplied `NULL` then DPAPI stores an empty UTF-16LE encoded string, ie. `ppszDataDescr = L""` and `ppszDataDescrSize = 2`.

- **idCipherAlgo** is the ID of the algorithm used to encrypt the data. The complete list of IDs is available from Microsoft MSDN here [7]. IDs starting with `0x88` are for hash functions. For example the ID `0x880e` is used to denote the SHA512 algorithm (CALG_SHA_512). IDs starting with `0x66` are for block ciphers. For example the ID `0x6610` is used for AES 256bit (CALG_AES_256).
- **idHashAlgo** is the ID of the hash algorithm used to derive the blob key.
- **pbSalt** is the salt used to derive the key.
- **pbEncData** is the encrypted data.
- **pbHMAC** is the HMAC used to ensure the integrity of the entire blob. Note that there is a second HMAC encrypted with the key to ensure that the key was not tampered with.

3.3.2 The Master Key Structure Blob

The **Master Key Blob** is the opaque structure used by DPAPI to store the user's long term master-key along with the meta-data required to decrypt it. Windows renews it every three months. We were unable to find where this limit is stored, as explained in Sec. 5, but we can confirm that this limit is enforced. The master key file contains five distinct structures.

The master key structure starts with a header used to identify the master-key. This header contains the following two fields:

```
struct masterkeyfile_header_s {
  DWORD dwMagic;
  WCHAR szKeyGUID[36];
}
```

In our testing, we only found **dwMagic** DWORD containing the value 2, which led us to develop two possible explanations. The first is that dwMagic is a type of DPAPI version, with Windows XP, Vista and 7 using the value of 2 and Windows 2000, which was the first edition of Windows implementing DPAPI, using the value 1. Alternatively, dwMagic may be used to differentiate the usage of particular the Master Key blobs by DPAPI versus other mechanisms, e.g. Protect Storage, which also use the identical data-structure. Under the second hypothesis, the dwMagic value of 2 denotes DPAPI usage, while the value of 1 denotes the Protect Storage system.

szKeyGUID is a UTF-16LE string representing the master key ID.

The **Keys Info** structure contains a set of four fields that allow the file parser to determine how big subsequent structures are.

```
struct masterkeyfile_infos_s {
  DWORD    dwUnknown;
  DWORD64  cbMasterKey;
  DWORD64  cbMysteryKey;
  DWORD64  dwHMACLen;

};
```

- **dwUnknown**: We currently have not determined the purpose of this field.
- **cbMasterKey**: This field contains the size of the Master key structure.
- **cbMysteryKey**: This field contains the size of the Mystery key structure.
- **dwHMACLen**: This field is the HMAC length. When the HMAC-SHA1 algorithm is used, its value is 0x014.

After these fixed length structures, there are two successive variable length structures used to store an encrypted key along with the parameters required to decrypt it. The first structure is used to store the master key itself, and the second is used to store what we call the "**Mystery key**". This mystery key is the most intriguing structure that we came across while reversing DPAPI. In regular encryption/decryption operation, this key plays no role; we did not observe a case where it was used. A possible explanation behind the existence of this key is backward compatibility with Windows 2000, as the key size (256 bits) is consistent with a RC4 key size.

These two keys are stored in the following structure:

```
struct masterkey_block_s {
  DWORD dwMagic;
  BYTE  pbSalt[16];
  DWORD cbIteration;
  DWORD idMACAlgo;
  DWORD idCipherAlgo;
  BYTE  pbCipheredKey[];
};
```

- **dwMagic**: This is once again a field that contains an apparently fixed value. Again, it may be a versioning or usage-determination mechanism.

- **pbSalt**: This is the salt used to encrypt the key.
- **cbIteration**: This is the number of hash rounds needed to derive the key. See table 1 for the value specific to each Windows version. For the second key, the **Mystery key**, the number of rounds is 1 on XP and Vista.
- **idMACAlgo**: This is the ID of the HMAC algorithm used.
- **idCipherAlgo**: This is the ID of the algorithm used to encrypt the key.
- **pbCipheredKey**: This is the encrypted key along with its HMAC.

The master key file ends with the following footer structure, used to identify the user-login password with which the master key is encrypted:

```
struct masterkeyfile_footer_s {
  DWORD dwMagic;
  BYTE      credHist[16];
};
```

- **dwMagic**: This fixed value field appears again.
- **credHist**: This is the GUID of the password used when this blob was encrypted. This GUID corresponds to the GUID found in the CREDHIST file (See section 6). When a blob is encrypted by the SYSTEM account, the GUID value is 0x00 because the SYSTEM account does not have a password.

3.4 Implementation By Windows Versions

At each revision of Windows, Microsoft made some substantial changes to how DPAPI works. These changes are summarized in table 1. Windows 7 brought a lot of changes in terms of algorithms. Moreover, according to our tests, the number of rounds required to derive the Pre key on Windows 7 seems to change from one computer to another. However, this does not create a security issue, because it is faster to brute force the user password hash encrypted in NTLM or LANMAN than to brute force the master key.

Windows XP and Vista use the strongest version of 3DES that requires three keys. This choice has a direct impact on encryption/decryption performance, because it forces Windows to execute the PBKDF2 derivation twice, since the PBKDF2 function outputs 20 bytes of data per call and Windows needs 32 bytes (8 bytes for each 3DES key plus 8 bytes for the initialization vector).

4 Decrypting a DPAPI blob

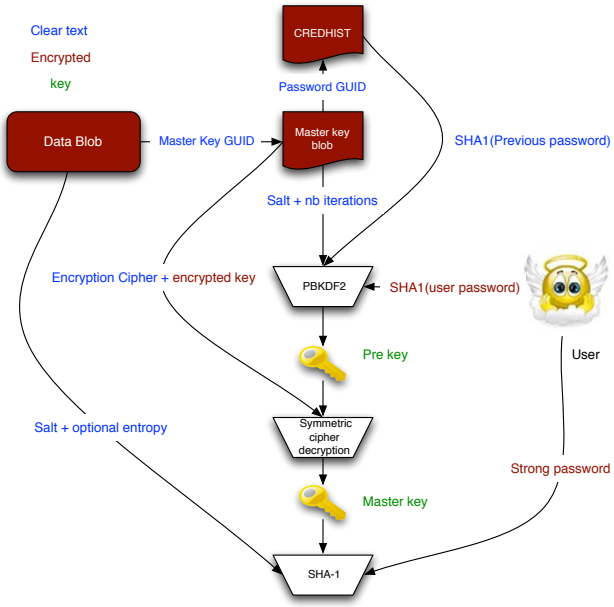

Figure 3: DPAPI blob decryption overview

Now that we have laid out how DPAPI works, we will describe step-by-step how Windows extracts/computes every piece of information that is needed to decrypt a DPAPI blob from the user password SHA1 and the DPAPI blob content itself. The decryption of *DPAPI Blob* content takes five steps, as shown in figure 3.

1. Extract the **Master key GUID** from the Data Blob structure.

2. Use the **Master key GUID** to find the correct master key file and extract the **Salt** and the **number of iterations** used by the PBKDF2 function. If the blob was encrypted with one of the user's previous passwords, then CREDHIST needs to be decrypted and the correct SHA1 needs to be extracted from it.

3. Use the correct SHA1 with the **Salt** and the **number of iterations** to compute the **Pre key** that has been used to encrypt the master key.

4. Use the **Pre key** to decrypt the **Master key**. The pre-key is used as a seed for the PBKDF2 function to compute the 40 bytes need by 3DES. To achieve this, the PBKDF2 function is called twice.

	XP	Vista	7
PKCS#5 PBKDF2 rounds	4000	24000	*Variable*
Symmetric algorithm	3DES-CBC	3DES-CBC	AES256-CBC
HMAC algorithm	HMAC-SHA1	HMAC-SHA1	HMAC-SHA512

Table 1: Implementation Changes in DPAPI by Windows version

5. Compute the **blob-key** using PBKDF2 once again. Here, Microsoft uses a slightly modified version of HMAC, as the conditional entropy and the optional password are not part of the message hashed with the inner padding. Instead, they are appended after the inner hash when the outer hash is performed.

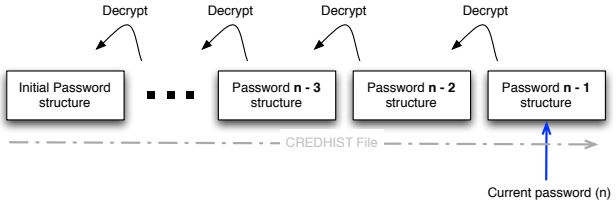

Figure 4: CREDHIST file structure overview

5 Backdooring DPAPI

Every time DPAPI encrypts data, it looks for the current master key GUID in a file named *Preferred* and checks if the key need to be renewed. If the key does not need to be renewed, then DPAPI simply uses it. Otherwise, DPAPI creates a new key, updates the preferred file, and uses the newly created key. The *Preferred* file contains only two fields: the GUID of the master key and a timestamp that DPAPI uses to determine when to renew the key.

This timestamp field is not protected by any HMAC mechanism and therefore can be changed arbitrarily. Attacker can leverage this lack of verification to extend the life of a key indefinitely and therefore ensure that they will be able to decrypt DPAPI blob indefinitely once they decrypt the current key. While this technique does not give the attackers more privileges than they already have, it allows them to sustain access to the computer secret in a stealthy way. As mentioned earlier, Windows enforces that the maximum lifetime of a key at three months. We were unable to determine whether this parameter is stored somewhere or if it is a hard-coded constant. Therefore, to extend the lifetime of the key, the attacker must periodically update the timestamp. This can be achieved in at least three ways that do not trigger anti-virus software: the attacker can add a program that will be launched by one of the Windows startup mechanisms, add a service, or simply use the task scheduler.

6 Recovering Previous Passwords

As explained in Sec 3, DPAPI needs to store the hashes of all the users previous passwords to guarantee that a user will be able to access all the data ever encrypted with DPAPI. These previous hashes are stored in the CREDHIST file. Figure 4 presents an overview of the CREDHIST file structure. Every time the user changes his password the previous SHA1 hash is encrypted with the new one and added at the end of the CREDHIST file. The encryption algorithm used for the CREDHIST is similar to the one used to encrypt data blob. While the CREHIST entry structure is very similar to the blob structure (See Appendix A), two things are worth mentioning about this structure. First, the user SID, computer SID and account SID are present in the CREDHIST file because they are required when decrypting a blob. These fields are present to enable a global CREDHIST file on the domain controller. Also, the passwordID field is present here and also in the master key structure, which allows the master key to be linked a given password. From an attacker's perspective, it worth noting that all the users previous SHA1 hashes are available at the same time and that they are not salted. This allows an attacker to crack them in parallel and use rainbow tables to speed up the recovery.

7 DPAPIck

In order to validate that our theoretical understanding of DPAPI was correct, we implemented a free off-line decryption tool called DPAPIck in C++/C# that is available for download from `www.dpapick.com`. DPAPIck allows users to decrypt the master key and the data blob off-line and to recover the hashes of previous passwords from the CREDHIST files. In addition to DPAPIck, we also implemented the cracking of the previous passwords

hashes extracted from the CREDHIST in our open source GPU based password cracker Nightingale. We needed to provide such an implementation, as standard SHA1 crackers do not work on the CREDHIST since the password are encoded in UTF-16LE before being hashed. Currently, Nightingale is able to do about 99M computation per Tesla 1070C GPU

8 Improving DPAPI scheme

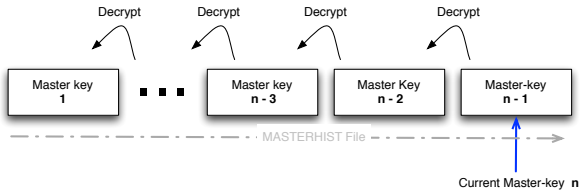

Figure 5: DPAPI improved scheme

While the CREDHIST mechanism should be eliminated to stop its password leaks, we believe that the straightforward solution of re-encrypting all master keys at every password change is not a good option. Therefore we developed a new scheme to accomplish this goal, with the goal of backwards compatibility so Microsoft can deploy it as a patch. We also wanted to have a re-encryption process that requires a constant time without decreasing DPAPI security. These requirements rule out numerous options such as changing the key less often, keeping track of which keys are still used, or assigning a fixed location to store DPAPI data.

While these requirements seem very strong, we came up the simple scheme depicted in figure 5 that provides an interesting trade-off. The key idea behind our scheme is that instead of using the current password hash to encrypt previous hashes, DPAPI should use the current key to encrypt the previous master keys. This scheme provides a constant re-encryption time as it only requires the re-encryption of the last key with the user password hash. It is also backwards compatible. However, the backwards compatibility and constant re-encryption time come with a price: if one application wants to decrypt a blob that was encrypted with a very old master key, the decryption process will be very long. However, this latency can be made to occur only once, since the blob may be re-encrypted with the new master-key.

We believe that this tradeoff is acceptable because the more an application is used, the more its blobs are updated with a recent key. When we presented our finding to Microsoft along with this new scheme, they told us that it was interesting and they might consider it, subject to other constraints they must take into account. We speculate that part of the reason why they may not adopt our solution lies in the fact that Windows security policy contains an option to prevent users from reusing a previous password. This means, of course, that Windows must still store previous user passwords. To deal with this issue, we believe that Microsoft should remove the CREDHIST file by using our scheme, modify the security policy exclude the last n passwords, and encrypt the storage of these n passwords in a way that make them very hard to crack, for instance by by storing a hash that uses PBKDF2.

9 Related Work

Slowing down hash functions, like DPAPI does, is a standard defense. Many password management proposals discuss how to slow hash functions for slowing down dictionary attacks [3, 4]. These methods are based on the assumption that the attacker has limited computing power. Several recent papers propose models for how humans generate passwords [12]. These results apply their models to speeding dictionary attacks. These methods can be used to crack user previous passwords faster. Rainbow tables [9], implemented in standard tools [10], can be also used to improve the cracking speed. PBKDF2 (Password-Based Key Derivation Function) is a key derivation function that is part of RSA Laboratories' Public-Key Cryptography Standards (PKCS) series, specifically PKCS #5 v2.0 [6]. Finally, others have proposed methods to migrate EFS files offline [1], which we feel are tedious and error-prone. We feel that DPAPIck is in improvement upon previous methods in this regard.

10 Conclusion

In this paper, we have presented what is, to the best of our knowledge, the result of the first complete reverse-engineering and audit of DPAPI. We also presented the first tool, DPAPIck, that is able to decrypt DPAPI secrets offline in a generic manner. We were able to leverage our knowledge of DPAPI design to find a way to backdoor DPAPI and demonstrate a very significant attack: recovering all previous passwords for any user.

References

[1] Recoverying efs files offlines. http://www.beginningtoseethelight.org/

efsrecovery/. 8

[2] Elie Bursztein and Jean-Michel Picod. Dpapi: Inner working technical report http://www.dpapick.com. Technical report, Nightingale team, 2019. 4

[3] David Feldmeier and Philip Karn. UNIX password security – 10 years later. In *Proceedings of Crypto 1989*, pages 44–63, 1989. 8

[4] J. Alex Halderman, Brent Waters, and Edward W. Felten. A convenient method for securely managing passwords. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 471–479. ACM, 2005. 8

[5] Elie Bursztein Jean-Michel Picod. Dpapick: Windows offline forensic tool. http://www.dpapick.com/, 2010. 1

[6] B. Kaliski. Pkcs #5: Password-based cryptography specification. RFC: http://tools.ietf.org/html/rfc2898, 2000. 8

[7] Microsoft. Algorithm id table. http://msdn.microsoft.com/en-us/library/aa375549%28VS.85%29.aspx. 5

[8] Microsoft. Windows data protection. MSDN http://msdn.microsoft.com/en-us/library/ms995355.aspx, 2001. 1, 3

[9] P. Oechslin. Making a faster cryptanalytic time-memory trade-off. In *Proceedings of CRYPTO 2003*, pages 617–630, 2003. 8

[10] Openwall Project. John the ripper password cracker, 2005. http://www.openwall.com/john. 8

[11] Nir Sofer. Nir sofer password recovery tools. http://www.nirsoft.net/, 2010. 1

[12] Matt Weir, Sudhir Aggarwal, Bill Glodek, and Breno de Medeiros. Password cracking using probabilistic context-free grammars. In *proceedings of IEEE Security and Privacy*, 2009. 8

A CREDHIST Structure

The CREDHIST structure looks like this :

```
struct credhist_entry_s {
DWORD dwMagic1; // 0x00000001
DWORD idHashAlgo;
DWORD dwRounds; // 0x00000AF0
DWORD dwCipherAlgo; // 0x00006603
BYTE bSID[12];
DWORD dwComputerSID[3];
DWORD dwAccountID;
BYTE bData[28];
BYTE bPasswordID[16]
};
```

Where

- **idHashAlgo**: is hash function ID.
- **dwRounds**: similarly to the master key structure, this is the number of PBKDF2 rounds needed to derive the key.
- **dwCipherAlgo**: is the cipher algorithm ID.
- **dwDataLength**: is the length of the encrypted data.
- **dwMACLength**: is the length of the HMAC.
- **bSID**: it seems to be the user SID in a strange format.
- **dwComputerSID**: is the computer SID.
- **dwAccountID**: is the account ID.
- **bData**: contains the encrypted password and the hmac.
- **bPasswordID**: is the password ID that is also founded in the master key.