Live Forensics – Extracting Credentials on Windows and Linux Systems

Liviu ITOAFA
Computer Science Department
Military Technical Academy
ROMANIA
liviu.itoafa@gmail.com

Abstract: ‘Post-mortem’ analysis of a system can be greatly simplified if the correct information is gathered in the live analysis stage. In this paper I’ve described Windows’ data protection APIs available for developers, some simplified versions of the API (LSA Secrets, Protected Storage), different methods used by applications to store their passwords safely and comparisons between them. As an example, I’ve built tools to dump passwords saved by browsers (Chrome, IE, Firefox) and an extractor of the login password (if available) from the registry. The basic concepts of how passwords may be stored apply to majority of applications that run on Windows and store passwords (protected or not) and understanding this makes possible recovery of other credentials also (messaging software, mail clients ...).

On the Linux side, I’ve analyzed a general method of storing passwords – keyrings, and the methods adopted by Chrome browser, and built extraction command line tools for both of them, in the form of a python script and a C++ application.

Key-Words: DPAPI, LSA, Protected Storage, registry, Chrome, Firefox, Ubuntu, gnome-keyring

1. Introduction

Extracting data from ‘live’ systems before pulling the cord supposes a set of practices to preserve memory information, registry hives, processes and network state, information that otherwise would be difficult or impossible to recreate from disk images or memory dumps. Extracting user passwords (from Protected Storage, LSA) that are protected by the current logged on user’s credentials can greatly help retrieval of further information. When the system is offline, a brute force attack of the password approach can be very slow and a dictionary attack not effective (strong passwords). Password cracking could be impossible, because internal workings of the modules are not public, are just partially or not at all reversed (EFS, DPAPI, etc).

Lots of applications give the user the possibility to save passwords (messaging software like, Skype, Yahoo Messenger, Pidgin, GTalk), browsers (Chrome, IE, Firefox, Safari), mail clients (Outlook, Thunderbird). Having these credentials can prove very useful in forensics cases where access to communications is very important.

2. Problem Formulation

If in a forensics case the investigator has access only to hard disk bit-stream images and/or memory dumps (hibernation file, crash dumps, system memory), the data protected by the above software (instant messaging, sites login credentials) may never be retrieved. In some cases password cracking attack is possible, through dictionaries or brute force (in case of FF for example) but even this approach may be impossible (if the application uses the data protection API (DPAPI) available in Windows, which is not (completely) reversed, because one simply wouldn’t know what to brute force, data being hidden in a blob with unknown structure). An approach to reverse DPAPI was done by the authors of [1], and they also made available a free tool.
2.1. LSA Secrets

"LSA secrets" is a special protected storage for important data used by the Local Security Authority (LSA) in Windows. LSA is designed for managing a system's local security policy, auditing, authenticating, logging users on to the system, storing private data. The important thing to realize about LSA Secrets is that it potentially contains credentials for services started under specific users, passwords for accounts that log on from external domains, as well as Dial-up Networking passwords. This "secret" information is stored in an encrypted format in the registry keys at HKLM\SECURITY\Policy\Secrets. Normally, these registry keys are not visible even if you run regedit as Administrator, because the permissions for this key show that only the SYSTEM account has access to it. Each secret (key) here contains the data in CurrVal sub-key. For example on systems with auto logon enabled, there is a key DefaultPassword that contains the cached logon password. For some reasons this key exists even on some systems without auto logon enabled (I had that key on a Windows XP SP3, and I have never had auto login, and other people are reporting same problem - a possible breach). A method to try to get the logon password (used by other tools also, like Cain&Abel) is to query the value from DefaultPassword key and decrypt it using functions from Windows API.

2.2. Protected Storage

It is an older implementation of data protection API, available in Windows XP. As described in MSDN at [2], the structure and content of the stored data is opaque. Access to items may be subject to confirmation, and a password could be required. For example Outlook pre-2003 versions used PStore to cache login passwords. From Windows Vista/7, it was replaced completely with DPAPI.

2.3. DPAPI

Data protection application programming interface is a stronger option for developers to protect the sensitive data, available since 2001. It is a service that provides confidentiality of data by using encryption. Because data protection is part of the operating system, every application can secure data without needing any specific cryptographic code other than the necessary function calls to DPAPI. It uses Triple-DES algorithm, strong keys and ties them to user's logon password. Detailed working specifications and security is described in [3]. Because all applications running in the context of a user would have access to data protected by this user, a ‘secret’ is introduced, to act as secondary entropy (it it's stored unprotected, other applications could use it to unprotect data). There is also an option to use a prompt which sets/asks for a password (usable only in GUI applications).

The key derivation process is as follows: DPAPI initially generates a Master Key (512 bits of random data). To protect it, a key is derived from the SHA1 hash of user's password, a salt and an iteration count, through a Password Based Key Derivation Function (PBKDF2 from PKCS #5, with 4000 iterations). A HMAC is calculated for the Maser Key (to prevent tampering). The derived key is used to encrypt (TDES, CBC mode) the Master Key and HMAC. The encrypted Master Key and HMAC, unencrypted salt and iteration count, are all stored in a Master Key file, which resides in the user's profile directory. The format of this file was reversed by some tools ([4]).

![Figure 1. Master key.](image-url)
The session key is a symmetric key used to encrypt and decrypt the data. It is not stored anywhere, it is always derived and removed from memory. It is derived by CryptoAPI calls, using as input the Master Key, 16 random bytes, an optional entropy and/or optional user password.

![Figure 2. Session key.](image)

Those random bytes are stored unprotected in the output data blob.

### 2.4. Credentials store

The “Credentials Store” is a newly introduced secret store mechanism in Windows, and it is generally used to store the network login passwords. Also the HTTP basic authentication passwords are stored in the Credentials Store. Its location is:

- C:\Documents and Settings\[username]\Application Data\Microsoft\Credentials for Windows XP, and
- C:\Users\[username]\AppData\Roaming\Microsoft\Credentials

for Windows Vista and 7. It is used by Internet Explorer (starting from version 7) to store credentials.

### 2.5. Gnome Keyring

It’s a daemon that stores different security credentials encrypted in a file in the user’s home directory. It uses the login password for encryption, and after the keyring is decrypted at login, the password is no longer necessary in the current user’s context. An attacker/forensic investigator can easily extract specific credentials from the GUI application (Applications -> Accessories -> Passwords and Encryption Keyrings), without being prompted for anything to authorize him.

To extract all security credentials programmatically, I’ve made a Python script using python-keyring-gnome module, described in section 4.2.5. The active attacks (when the attacker has access to user’s session) that gnome keyring is not protecting against are described in [16], like for example installing snooping applications, intercepting X events, etc.

Seahorse is Gnome Keyring’s replacement since Gnome 2.2. The analogous application for KDE is KWallet, working by the same principles. There is a python binding for this too.

### 2.6. Other methods

Other applications encrypt passwords without linking the encryption key to the current logged on user and DPAPI.

Another option is clear text. As insecure as it may sound, Chrome on Linux used to store passwords in clear text until recent versions, that support Gnome Keyring and KWallet.

Pidgin for Linux also does the same, and possibly many other IM applications.

For example Firefox has some encrypting/decrypting classes (Network Security Services, NSS [12]) that use 3DES in CBC mode, by default.

Opera browser also uses their own encryption functions, to encrypt credentials in Opera password file (Wand.dat). It uses 3-DES algorithm, but with a static salt. The static salt has been reversed by a reverse engineering team (sna@RETeam.org), the code is available (unwand.cpp), and decryption
is now possible for all Opera versions [15].

3. Problem Solution

Some of the passwords stored using the methods above can be retrieved, either by looking at the programs’ source (Chrome, Firefox) and learning how encryption/decryption is done, or by reverse engineering.

3.1. Dumping LSA Secrets

Because, as noted in 2.1, the keys containing LSA secrets are not by default accessible to administrator users, only to SYSTEM user, some tricks can be used to view their organization in the registry.

- Run a command prompt from task scheduler. A scheduled task will run in the context of the Task Scheduler service's account, that being SYSTEM. A new task can be programmed with the command:
  
  at 09:45 /INTERACTIVE cmd.exe

- Use PsExec tool from SysInternals, that is used to execute programs on remote systems, with the -s flag (Run the remote process in the System account):

  pseexec -s cmd

In the new command prompt, registry can be queried from command line (reg query HKEY_LOCAL_MACHINE\SECURITY\policy)

- LSA secrets can be read in the context of current user by using functions exported by Advapi32 library, LsaRetrievePrivateData being the most useful. Functions exported by a library can be viewed with tools like DLL Export Viewer [5].

3.2. Dumping Protected Storage secrets

Some applications (Internet Explorer) on Windows XP still use this method to store secrets, even without the optional password. Retrieval tools use also the same API from [2], which is a little cumbersome and little documented. Even if existing tools that extract PStore secrets are closed source, snippets of code can be found on underground (hacking) forums.

4. Tools

I’ve tested existing tools for password recovery and wrote some programs to dump passwords saved by browsers (Chrome, IE and Firefox) and the default login password. Source code is available at [8].

4.1. Existing tools

A possible concern with existing closed source tools is their behavior. For example, VirusTotal site reports an infection from almost 50% of the anti-viruses used for some tools (chromepass, iepv...). Others come with installers with lots of adware. There are also some that are not cheap at all and cannot be evaluated.

The existing tools I’ve tested are: chromepass, iepv, and passwordfox from the suite [6], Chrome Password Decryptor, IE Password Decryptor and Fire Password Viewer from the password recovery suite [7].

4.2. Open source tools

I’ve coded 4 recovery tools, for personal use and for understanding the underlying mechanisms of storing passwords. The advantages are that:

- can be extended,
- output can be exported to different formats, parsed
- being command line based, can be run as part of a forensic remote script, and their output can be sent back through
network (through psexec for example)
- having the source for some basic decryptors/dumpers, further efforts can be concentrated on reversing unknown API’s or analyzing how other closed-source applications store passwords, in order to decrypt/crack them
- Many existing closed source applications (free or for sale) advertise that they can decrypt passwords, and mislead the users, without actually offering clear explanations and protection suggestions.

After compiling LSASecretRead, LSA secrets (DefaultPassword, services accounts (_SC_*), dial-up passwords if existent) can be dumped from the registry with the exception of some that are encrypted (example NL$KM) with a key that exist in the process memory of lsass.exe file (there are techniques for dumping these also, like thread injection, grepping for the key through the process memory or creating a system service to run functions exported by lsasrv.dll).

A drill down into Chrome browser source code reveals that it stores saved login credentials using CryptProtectData() call, from crypt32.lib, without additional entropy or password. Why Google refuses to put a master password is debated in [9] (“Our decision not to implement the master password feature is based on our belief that it creates a false sense of security instead of actually providing a strong security benefit”). Chrome’s default security relies totally on the strength of the Windows logon password - and for many people that is minimal, if at all. If one has a very strong Windows password then Chrome passwords are also secure. The passwords are saved in a SQLite database (API available to read SQLite databases from C/C++ at [10]), located in the %UserProfile%\AppData\Local\Google\Chrome\User Data\Default\Login Data folder. Here the username and password for different sites are encrypted. Because the encryption is tied to the logged on user, other applications can access them invoking the same API from crypt32.lib. A good thing that some users don’t understand because of this hidden relation is that if the computer is stolen, or is accessed after the password is reset, the attacker won’t have access to their credentials, because he won’t have the correct master key (described in 2.3.1).

Internet Explorer stores two types of passwords, auto-complete (for normal sites login, email, forums) and HTTP basic authentication passwords. The versions prior and including 6.0 used the protected storage (described above in 3.2) to keep cached passwords. The location of the Protected Storage in the registry is HKEY_CURRENT_USER\Software\Microsoft\Protected Storage System Provider, [13]. Newer versions use the stronger CryptoAPI library. For the first case, the passwords from Protected Storage can be easily exposed using functions from pstorec.dll library. For the second case, the security is greatly enhanced because
- It uses CryptoAPI functions
- The hash of the website link is stored in the registry instead of the plain text link
- The website url is used as secondary entropy to CryptProtectData() calls.

In this way, IE can auto-complete the credentials only if the hash of the link corresponds, but an attacker must know the URL to unprotect the login credentials. I’ve implemented just the first approach, but there are tools that try to recover passwords for IE 7 and up also, by trying different URLs from IE cache. Another idea is to try login URLs from a list of known sites.

Figure 3. Internet Explorer
Also there is an API available to enumerate URL links from Internet Explorer’s history (UrlHist.h). We can hash (SHA1) every one of them, and if a match is found with the entry in the registry, this URL can be used as secondary entropy to decrypt (unprotect) the data associated (username and password). The original work on decrypting IE 7 passwords was done by SapporoWorks, in an article published by SecurityFocus [14].

If there are multiple accounts for the same site, multiple secrets in the form of username/password pairs will be stored. I’ve also modified the tool from 4.2.2 that dumps Chrome passwords in Windows to be cross-platform, and can be compiled and run on Ubuntu also:

```
$ make -f Makefile.lin
$ ./bin/chromePass
~/config/google-chrome/Default/Login\ Data
```

Username: admin
Password: 9c^"Y>:rDv<NZ3}$5

The browser should be closed; otherwise a 'Database locked' SQL error appears.

Firefox stores passwords using functions from their NSS library (Network Security Services, [12], [13]). The easiest way to recover passwords saved by FF, without understanding the internals of NSS, is to analyze the code that encrypts the data and use similar functions exported by NSS library (nss3.dll.) for decryption. Encrypted passwords are encoded in Base64, and then stored in a SQLite database, in the Firefox folder under the user profile directory. The decryption key is encrypted (using user master password) and stored together with a salt in key3.db file.

An NSS password decryptor is also implemented in the ‘importer’ module from the Chrome browser source tree

```
http://src.chromium.org/svn/trunk/src/google-chrome/browsing/importer/nss_decryptor.cc
```

The code that uses Firefox libraries to extract passwords (unprotected by a master key or with master key specified) is available to be build from source at [8].

```
python dump_keyrings.py
```

```
Extracting keys from "Login" keyring:
[0] http://192.168.0.1/
admin:N:rDv<NZ3}$5
```

I’ve tested with Chrome version 20.0.1132.47, which by default uses the Gnome Keyring described above to store credentials. If this is the case, the python script [17] recovers them.

There is however the possibility to start the browser with the basic password store (clear text) option:

```
/opt/google/chrome/google-chrome --password-store=basic
```

Credentials will be stored in ~/.config/google-chrome/Default/Login Data file. Failure to initialize the encrypted store, or to determine the default encrypted store, will fall back on this basic option. In this case, or in the case slightly older
versions of Chrome are used, there are 3 possibilities to extract passwords. The quickest way is the sqlitebrowser GUI tool. If a command line tool or a script is needed sqlite3 shell comes in handy:

```
$ sqlite3 ~/.config/google-chrome/Default/Login\Data
sqlite> .output pass.log
sqlite> select origin_url, username_value, password_value from logins;
sqlite> .exit
$ cat pass.log
```

The same could be achieved totally automatic by piping the input:

```
$ echo 'SELECT origin_url, username_value, password_value FROM logins;' | sqlite3 ~/.config/google-chrome/Default/Login\Data
```

5. Conclusion

This code may help penetration testers in building audit tools, and forensic examiners in live system analyses, but will be useless to an attacker unless he has Administrator privileges (for dumping LSA secrets), or has the users credentials, or runs these in the user's context (in these last 2 cases he would have easier attack methods available, than this).

- IE security seems the most secure as it is tied to the logged on user and also uses the hash of the site as the secondary entropy for CryptProtectData()
- Because Firefox passwords are not related to the logged on user, can be dumped offline, and key3.db keys can be brute forced in case if a master password is used
- Google will not implement a master password feature for Chrome browser, so 3rd party plugins must be used instead, to keep protected the login credentials.
- A better option is to give up the auto-complete feature (at least for important credentials) and save them in a password management software such as KeePass.
- Last version of KeePass has very important security features that, if used correctly, can make the sniffing/cracking/dumping passwords impossible (erase clipboard, block clipboard monitors, auto-type, auto-type obfuscations)
- For the tools that recover Internet Explorer (later versions, from 7 on) passwords to work, URL history is needed, to generate and compare hashes. If history is cleared, they are no longer able to unprotect passwords.

Acknowledgement

Parts of this research have been published in the Proceedings of the 11th International Conference on Informatics in Economy, IE 2012

References

[14] SapporoWorks - HTTP Basic

Authentication and auto-complete passwords in Internet Explorer 7, http://www.securityfocus.com/archive/1/458115/30/0/threaded