TBAL: an (accidental?) DPAPI Backdoor for local users

a.k.a how a convenience feature undermined a security feature

(saved from https://vztekoverflow.com/tbal on 2018-11-30)

The Data Protection API (**DPAPI**) provided by Windows is a way of protecting secrets used by a lot of popular software solutions, most famously by Google Chrome when storing passwords and cookies. A lot has been said over the years about the security of this API, but new versions of Windows 10 added a new convenience feature called **TBAL** that further undermines this protection. Basically, when you have access to the disk of a computer that is not joined to a domain and has been shut down properly, you have everything you need to decrypt all DPAPI secrets of the last logged-in user. With earlier versions of Windows 10, the data necessary to decrypt DPAPI secrets was (almost?) never stored on the disk, most certainly not on every shutdown.

I work as an ethical hacker at DCIT, a Czech company that focuses on corporate security. This research was conducted during work hours, with tremendous support from both colleagues and superiors.

Disclaimer: Most of the information found in this article was discovered through reverse engineering, either by us or by other people. Reverse engineering is a lot of guesswork and assumptions. Even though we provide all information in good faith, the information is provided "as is" without warranty of any kind, either expressed or implied, including limitation warranties of merchantability, fitness for a particular purpose, and noninfringement.

DPAPI 101

The DPAPI is the simplest API you can imagine: it's basically just a pair of functions, CryptProtectData and CryptUnprotectData. That's probably why it's so popular with developers – if you are fine with it being a black box, it's very simple to use. Just take a buffer, send it to DPAPI, and it returns a blob. You store that, and you're done. No key management; no choosing the correct cryptographic algorithms for the job; it just works™.

For this article, we will investigate only the top layer of DPAPI – if you're interested in more details, we strongly recommend the *DPAPI and DPAPI-NG: Decrypting All Users' Secrets and PFX Passwords* talk from Insomni'hack 2018 by Paula Januszkewicz.

DPAPI uses a **Master key**, and as the name suggests, this key is basically all you need to decrypt all other secrets. Obviously, this secret key can't just be stored on the disk in its raw form for anybody to grab, so it is encrypted with a hash of the user's password.

If you know anything about Windows security, you have probably heard of the NTLM hash, which is an MD4 hash of the password. This hash is used for almost everything in Windows, including authentication. When you are logging in, Windows calculates the NTLM hash of the password you typed in and compares it with the NTLM hash that's stored in something called the SAM file. So, for obvious reasons, the NTLM hash has to be stored on the disk, otherwise Windows wouldn't have a reference to compare it with. This is important, as if you only used the NTLM hash to encrypt the master key, simply grabbing it from the SAM file residing on the hard disk would be enough to decrypt every single secret that is protected by DPAPI.

According to Passcape Software's research paper, *"DPAPI Secrets. Security analysis and data recovery in DPAPI"*, the first implementation of DPAPI actually used NTLM hashes, and it was such a major issue that Microsoft quickly changed it:

> The decryption of Master Key in the first implementation of DPAPI required the NTLM hash of user's password. That was a major: No, that was the greatest blunder of DPAPI developers, which negated the security of the system. All because it technically allowed a potential malefactor to decrypt the Master Key and, consequently, any DPAPI blob by getting the NTLM hash directly from the SAM file; the actual password was not even necessary! In the second, current version of DPAPI, that error was fixed; now Master Key is encrypted using the SHA1 hash.

So, in all implementations of DPAPI from XP onward, the Master key is encrypted using `SHA-1(UTF16LE(user_password))`. This is actually a pretty clever solution to the problem: use a different hash of the same password *(domain-joined users actually still use the NTLM hash, but that's out of scope for this article).* As long as the hashing algorithms used are good enough and the SHA-1 hash never gets stored on the disk, the easiest way of getting the SHA-1 hash is through bruteforcing the original password corresponding to the NTLM hash that's stored there. This leads to a situation where (when the computer is off) your secrets are as secure as your password is strong.

This is actually a good solution to a few different attack vectors. If you have write access to a disk, it's easy to modify the SAM file and effectively reset the user's password and then log in. Windows calculates the NTLM hash of the password you typed in, compares it with your modified SAM file, and as it checks out, you get logged in. However when it calculates the SHA-1 hash of your new password and tries to decrypt your master key with that, it obviously fails, as you have changed the password *(when you change the password in a regular way, some data has to be decrypted and reencrypted to keep your access to it).*

ARSO and TBAL

Windows 8.1 quietly introduced a new feature called Automatic Restart Sign-On (**ARSO**). There's not a lot of documentation available about it, only this Microsoft docs page. Although we strongly recommend reading the linked document, it basically boils down to this:

> After a Windows Update induced reboot, the last interactive user is automatically signed on and the session is locked so the user's lock screen apps can run.

This means that user credentials have to be stored on the disk to be preserved during the reboot. And to fully restore the user session, not only the NTLM hash, but also the SHA-1 hash have to be preserved. However, this feature was implemented very cautiously: one of the requirements listed in the documentation is *Can only be enabled if BitLocker is enabled*. Therefore, when this feature was introduced, your SHA-1 hash *could* get stored on the disk, but only on a Windows Update initiated restart if BitLocker is enabled. As long as your BitLocker setup was secure, your hash was technically secure.

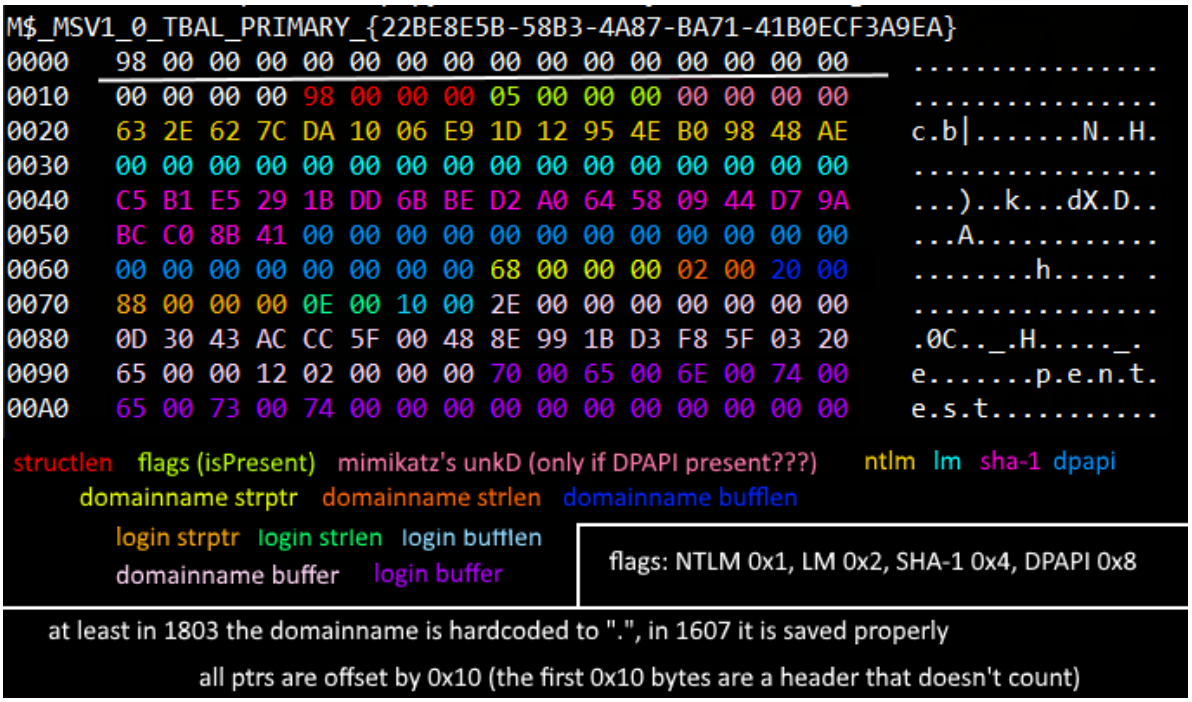
But with Windows 10, Microsoft was silently pushing this feature forward with each update, removing more and more restrictions in the name of convenient lock screen apps including Cortana. This feature was also renamed **TBAL**, the explanation of which is nowhere to be found (we guess AL stands for AutoLogon; no idea about the TB). We weren't able to find any comments on the security aspects of ARSO, and for TBAL, the internet doesn't seem to have any information whatsoever.

How does it work? When somebody asks lsasrv for the so-called *TBAL provisioning*, lsasrv checks whether the account is an offline one or an MS account, and then, based on that, asks either the msv1_0 or the cloudAP security provider to store everything it needs to restore the user session to the disk. After that, it sets up the autologon mechanism that's been in Windows since at least NT4, with the hardcoded password of `_TBAL_{68EDDCF5-0AEB-4C28-A770-AF5302ECA3C9}`.

After booting, the autologon mechanism kicks in, and tries to login to your user account with the TBAL password. The security package sees this special password, so it deserializes the stored credentials, and pretends that this password actually has worked, and then your session gets locked as if you had immediately pressed Win+L *(this is actually a simplification, there are a lot of complex hacks that make this possible, but we describe the observable behavior)*. **The stored credentials then get removed from the registry.** However, the credentials of the last logged-in user before reboot stay loaded in memory even when a different user logs in after reboot. Any (admin) user can dump them using ordinary methods like mimikatz.

Where do the security packages store their serialized credentials? In the registry, specifically in a storage commonly called LSA secrets, which is encrypted using the BOOT key (in a very obscure way). There are tools readily available for decryption of these secrets, as will be shown

in the PoC. Specifically, we'll focus on msv1_0's secret, which is called `M$_MSV1_0_TBAL_PRIMARY_{22BE8E5B-58B3-4A87-BA71-41B0ECF3A9EA}`. This LSA secret contains the NTLM and more importantly the SHA-1 hash of your password, a.k.a the secret necessary for decrypting DPAPI's Master key. The structure of the said secret is the following:



From some basic reverse-engineering we've done on the various Windows 10 versions, we believe this is how the scope of TBAL was slowly expanded:

- **1507:** TBAL provisioning on every reboot, but available only for MS accounts and enabled only when *OSVolumeProtected* (probably BitLocker)
- **1607:** TBAL added for non-MS accounts
- **1709:** TBAL doesn't require BitLocker anymore

Proof of concept attack

See this attack in action: https://www.youtube.com/watch?v=NIPKMSV-KTw

First, we start by installing a clean Windows 10 1803 VM, setting it up with a local account and opting-out of all the voluntary telemetry. After that, we just install Chrome, store a password in it, and shut down the computer. After that, all steps are taken strictly against the VM's disk. We simply mounted it to the host machine using VMware, but if it were a real computer, imagine physically connecting the HDD to your machine.

Then, we used creddump7 to dump the LSA secrets from the disk. As you can see, TBAL was provisioned, so the Default password is set to the TBAL magic string. As this is a local account, we can see the serialized credentials and grab the SHA-1 hash from it.

Afterwards, we switch to mimikatz. First, we show you that without decrypting the masterkey, we can only read out the stored username and not the password. After that, we use the dpapi::masterkey command to actually decrypt the masterkey from the VM's disk using the SHA-1 we retrieved in the previous step. As mimikatz automatically caches all masterkeys it has decrypted, just running the same dpapi::chrome command again reveals the password in plain sight.

Mitigations & remaining questions

This threat is easily mitigated by disabling ARSO and TBAL with a group policy or a registry setting (reportedly, the ARSO policies mentioned in the Microsoft docs page also work for TBAL). If you don't want to lose the convenience these features provide, full disk BitLocker should probably protect you as well.

There are still some questions we don't have answers to and would love to continue looking into them more:

- What data is serialized for Microsoft accounts?
- How does TBAL work in a multi-user scenario?
- Is TBAL completely disabled in domains?
- Can't the fact that security packages accept the fake password be somehow abused? If we managed to login with it interactively without getting the session locked, that would be very serious.

Conclusion

In this article, we have demonstrated that in some scenarios, the default Windows configuration leads to the SHA-1 hash of the user's password being stored to the disk in a way that is retrievable without any further knowledge about the password. We argue that this is an issue for DPAPI, because if the secret necessary for decrypting the master key was to be stored on the disk by design, Microsoft could have kept on using the NTLM hash it uses in domain settings (and supposedly used in the first implementation of DPAPI). We then demonstrated how this attack can be executed using readily available tools.