

How to Achieve Eternal Persistence Part 2: Outliving the Krbtgt Password Reset

& huntandhackett.com/blog/how-to-achieve-eternal-persistence-part-2

Rindert Kramer

2735	192.598296	192.168.88.1	192.168.88.129	SANR	222 GetUserPwInfo request
2736	192.598673	192.168.88.129	192.168.88.1	SANR	206 GetUserPwInfo response
2737	192.599882	192.168.88.1	192.168.88.129	SANR	1082 SetUserPwInfo request[Malformed Packet]
2739	192.615607	192.168.88.129	192.168.88.128	DRSUAPI	338 DsReplicaSync request
2740	192.616321	192.168.88.128	192.168.88.129	DRSUAPI	178 DsReplicaSync response
2741	192.616552	192.168.88.128	192.168.88.129	DRSUAPI	658 DsGetNCChanges request[Long frame (456 bytes)]
2747	192.618983	192.168.88.129	192.168.88.128	DCERPC	1510 Response: call_id: 10, Fragment: 1st, Ctx: 1 [DCE/RPC 1st fragment, reas: #2749]
2749	192.619254	192.168.88.129	192.168.88.128	DRSUAPI	466 DsGetNCChanges response, Error: WERR_INVALID_HANDLE[Long frame (6016 bytes)]
2754	192.623704	192.168.88.128	192.168.88.135	DCERPC	582 Bind: call_id: 12, Fragment: Single, 2 context items: DRSUAPI V4.0 (32bit NDR), DRSUAPI V4.0 (64bit NDR)
2756	192.624349	192.168.88.135	192.168.88.128	DCERPC	284 Bind_ack: call_id: 12, Fragment: Single, max_xmit: 5840 max_recv: 5840, 2 results: Provider rejection, Acceptance
2757	192.624664	192.168.88.128	192.168.88.135	DCERPC	227 Alter_context: call_id: 12, Fragment: Single, 1 context items: DRSUAPI V4.0 (64bit NDR)
2758	192.624955	192.168.88.135	192.168.88.128	DCERPC	118 Alter_context_resp: call_id: 12, Fragment: Single, max_xmit: 5840 max_recv: 5840, 1 results: Acceptance
2759	192.625082	192.168.88.128	192.168.88.135	DRSUAPI	338 DsReplicaSync request
2760	192.625450	192.168.88.135	192.168.88.128	DRSUAPI	178 DsReplicaSync response
2765	192.626306	192.168.88.135	192.168.88.128	DCERPC	580 Bind: call_id: 12, Fragment: Single, 2 context items: DRSUAPI V4.0 (32bit NDR), DRSUAPI V4.0 (64bit NDR)
2767	192.626804	192.168.88.128	192.168.88.135	DCERPC	284 Bind_ack: call_id: 12, Fragment: Single, max_xmit: 5840 max_recv: 5840, 2 results: Provider rejection, Acceptance
2768	192.626994	192.168.88.135	192.168.88.128	DCERPC	227 Alter_context: call_id: 12, Fragment: Single, 1 context items: DRSUAPI V4.0 (32bit NDR)
2769	192.627138	192.168.88.128	192.168.88.135	DCERPC	118 Alter_context_resp: call_id: 12, Fragment: Single, max_xmit: 5840 max_recv: 5840, 1 results: Acceptance
2770	192.627218	192.168.88.135	192.168.88.128	DCERPC	126 Alter_context: call_id: 12, Fragment: Single, 1 context items: DRSUAPI V4.0 (32bit NDR)
2771	192.627283	192.168.88.128	192.168.88.135	DCERPC	110 Alter_context_resp: call_id: 12, Fragment: Single, max_xmit: 5840 max_recv: 5840, 1 results: Acceptance
2772	192.627487	192.168.88.135	192.168.88.128	DRSUAPI	640 DsGetNCChanges request[Long frame (424 bytes)]
2776	192.628704	192.168.88.128	192.168.88.135	DRSUAPI	822 DsGetNCChanges response
2778	192.630830	192.168.88.135	192.168.88.128	DRSUAPI	338 DsReplicaSync request
2779	192.631107	192.168.88.128	192.168.88.135	DRSUAPI	178 DsReplicaSync response
2780	192.631339	192.168.88.135	192.168.88.128	DRSUAPI	658 DsGetNCChanges request[Long frame (456 bytes)]
2782	192.631921	192.168.88.128	192.168.88.135	DRSUAPI	1898 DsGetNCChanges response, Error: WERR_INVALID_HANDLE[Long frame (2312 bytes)]
2793	192.733127	192.168.88.129	192.168.88.128	DCERPC	214 Bind: call_id: 2, Fragment: Single, 3 context items: EPMV4 V3.0 (32bit NDR), EPMV4 V3.0 (64bit NDR), EPMV4 V3.0 (6c671c2c-9812-4540-8300-000000000000)
2794	192.733570	192.168.88.128	192.168.88.129	DCERPC	162 Bind_ack: call_id: 2, Fragment: Single, max_xmit: 5840 max_recv: 5840, 3 results: Provider rejection, Acceptance, Negotiate ACK
2795	192.733770	192.168.88.129	192.168.88.128	EPH	322 Map request, RPC_NETLOGON, 32bit NDR
2796	192.734087	192.168.88.128	192.168.88.129	EPH	322 Map response, RPC_NETLOGON, 32bit NDR, RPC_NETLOGON, 32bit NDR
2800	192.735083	192.168.88.129	192.168.88.128	DCERPC	214 Bind: call_id: 2, Fragment: Single, 3 context items: RPC_NETLOGON V1.0 (32bit NDR), RPC_NETLOGON V1.0 (64bit NDR), RPC_NETLOGON V1.0 (6c671c2c-9812-4540-8300-000000000000)
2801	192.735335	192.168.88.128	192.168.88.129	DCERPC	162 Bind_ack: call_id: 2, Fragment: Single, max_xmit: 5840 max_recv: 5840, 3 results: Provider rejection, Acceptance, Negotiate ACK
2802	192.735486	192.168.88.129	192.168.88.128	RPC_NE_	204 NetServerReqChallenge request
2803	192.735776	192.168.88.128	192.168.88.129	RPC_NE_	90 NetServerReqChallenge response
2804	192.735952	192.168.88.129	192.168.88.128	RPC_NE_	258 NetServerAuthenticate3 request
2805	192.737039	192.168.88.128	192.168.88.129	RPC_NE_	98 NetServerAuthenticate3 response
2806	192.737828	192.168.88.129	192.168.88.128	DCERPC	176 Alter_context: call_id: 4, Fragment: Single, 1 context items: RPC_NETLOGON V1.0 (64bit NDR)
2807	192.738113	192.168.88.128	192.168.88.129	DCERPC	130 Alter_context_resp: call_id: 4, Fragment: Single, max_xmit: 5840 max_recv: 5840, 1 results: Acceptance
2808	192.738304	192.168.88.129	192.168.88.128	RPC_NE_	302 NetLogonDummyRoutine1 request[Malformed Packet]
2809	192.738662	192.168.88.128	192.168.88.129	RPC_NE_	174 NetLogonDummyRoutine1 response[Malformed Packet]
2810	192.738898	192.168.88.129	192.168.88.128	RPC_NE_	414 NetLogonSendToSam request[Malformed Packet]
2811	192.741510	192.168.88.128	192.168.88.129	DRSUAPI	722 DsGetNCChanges request[Long frame (528 bytes)]
2812	192.741607	192.168.88.129	192.168.88.128	RPC_NE_	158 NetLogonSendToSam response[Malformed Packet]
2813	192.741811	192.168.88.129	192.168.88.1	SANR	198 SetUserPwInfo response

Rindert Kramer @

May 30, 2024 9:43:29 AM

In the [previous blog post](#), we looked at how a password reset event can be captured and decrypted. Our quest to achieve eternal persistence in an Active Directory (AD) domain while being passive and undetectable is not yet complete. If the password of the krbtgt account is reset, we could end up in a scenario where the credentials that we have obtained thus far no longer work to decrypt network data.

To achieve our ultimate goal of eternal persistence, we need to take a look at how password changes are replicated. In this blog post we cover one aspect of AD's password replication process, and in the next blog we dive into the generic replication process between domain controllers.

Before we dive into details, we need to understand how passwords are replicated. If you want to skip the fluffy details, scroll down to the 'Understanding the RPC call' section of the blog.

Replication

A domain has domain controllers (DC) that handle all logic on behalf of the domain. If you want to authenticate, you prove your identity to the DC after which you will be granted access, or not. Realistically, a domain should have multiple domain controllers, either for

reliability purposes or to minimize latency in the authentication process.

Domain controllers in the same domain should have an exact copy of all the data that is in the domain, except for some data only relevant for the DC itself. If we query DC A and DC B for an attribute on a user account, the returned value must be the same. If a user changes their telephone number or any other attribute, this attribute must be shared with other domain controllers in the domain. This is done using replication, where domain controllers share updated data with their replication partners. In smaller sized domains, this often means that every domain controller replicates to all other domain controllers, but this could vary depending on how the domain is constructed and the amount of domain controllers.

There are a few types of replication. For this scenario we go into detail for the following replication types:

1.
 1. Normal replication
 2. Urgent replication
 3. Immediate replication

i. Normal replication: This is the default behavior for most changed values in attributes. Every given interval (which defaults to 15 seconds[1]), the domain controller that holds the new information will send out a replication notification to replication partners, saying **Hey man, want some data?** The partnered domain controller will respond, saying **Gimme the good stuff!** after which the replication process will begin.

ii. Urgent replication: The process for urgent replication is basically the same as normal replication. However, it differs depending on the attribute, as it will send out a notification regardless of the interval. For example, when the password policy of the domain is changed on DC A, it will send out a replication notification immediately to the configured replication partners, which will respond with **Gimme the good stuff!**

iii. Immediate replication: Urgent replication is designed to be faster than regular replication, but sometimes that is not fast enough. It takes multiple packets for the new information to be replicated to all other domain controllers in the domain and for specific types of information, this needs to be faster. Security information, such as secrets, should be available immediately. Every domain has a primary domain controller configured that has the final answer regarding secrets. It does a few more things, but that is outside the scope of this blog post. If you're interested, search Google for FSMO roles 😊
If a user changes their password on DC A, the new password will be immediately sent to the primary domain controller (PDC) of the domain, without sending a replication notification. The new password will be sent using a Remote Procedure Call (RPC) called **NetrLogonSendToSam**. This behavior can be observed in Wireshark:

2735	192.598296	192.168.88.1	192.168.88.129	SAHR	222 GetUserPwInfo request
2736	192.598673	192.168.88.129	192.168.88.1	SAHR	286 GetUserPwInfo response
2737	192.599882	192.168.88.1	192.168.88.129	SAHR	1082 SetUserInfo2 request[Malformed Packet]
2739	192.615607	192.168.88.129	192.168.88.128	DRSUAPI	338 DsReplicaSync request
2740	192.616321	192.168.88.128	192.168.88.129	DRSUAPI	178 DsReplicaSync response
2741	192.616552	192.168.88.128	192.168.88.129	DRSUAPI	658 DsGetNCChanges request[Long frame (456 bytes)]
2742	192.618983	192.168.88.129	192.168.88.128	DCERPC	1510 Response: call_id: 18, Fragment: 1st, Ctx: 1 [DCE/RPC 1st fragment, reas: #2749]
2749	192.619254	192.168.88.129	192.168.88.128	DRSUAPI	468 DsGetNCChanges response, Error: WERR_INVALID_HANDLE[Long frame (6816 bytes)]
2754	192.623704	192.168.88.128	192.168.88.135	DCERPC	582 Bind: call_id: 12, Fragment: Single, 2 context items: DRSUAPI V4.0 (32bit NDR), DRSUAPI V4.0 (64bit NDR)
2756	192.624349	192.168.88.135	192.168.88.128	DCERPC	284 Bind_ack: call_id: 12, Fragment: Single, max_xmit: 5840 max_recv: 5840, 2 results: Provider rejection, Acceptance
2757	192.624664	192.168.88.128	192.168.88.135	DCERPC	227 Alter_context: call_id: 12, Fragment: Single, 1 context items: DRSUAPI V4.0 (64bit NDR)
2758	192.624955	192.168.88.135	192.168.88.128	DCERPC	118 Alter_context_resp: call_id: 12, Fragment: Single, max_xmit: 5840 max_recv: 5840, 1 results: Acceptance
2759	192.625082	192.168.88.128	192.168.88.135	DRSUAPI	338 DsReplicaSync request
2760	192.625450	192.168.88.135	192.168.88.128	DRSUAPI	178 DsReplicaSync response
2765	192.626306	192.168.88.135	192.168.88.128	DCERPC	580 Bind: call_id: 12, Fragment: Single, 2 context items: DRSUAPI V4.0 (32bit NDR), DRSUAPI V4.0 (64bit NDR)
2767	192.626804	192.168.88.128	192.168.88.135	DCERPC	284 Bind_ack: call_id: 12, Fragment: Single, max_xmit: 5840 max_recv: 5840, 2 results: Provider rejection, Acceptance
2768	192.626994	192.168.88.135	192.168.88.128	DCERPC	227 Alter_context: call_id: 12, Fragment: Single, 1 context items: DRSUAPI V4.0 (32bit NDR)
2769	192.627138	192.168.88.128	192.168.88.135	DCERPC	118 Alter_context_resp: call_id: 12, Fragment: Single, max_xmit: 5840 max_recv: 5840, 1 results: Acceptance
2770	192.627218	192.168.88.135	192.168.88.128	DCERPC	126 Alter_context: call_id: 12, Fragment: Single, 1 context items: DRSUAPI V4.0 (32bit NDR)
2771	192.627283	192.168.88.128	192.168.88.135	DCERPC	110 Alter_context_resp: call_id: 12, Fragment: Single, max_xmit: 5840 max_recv: 5840, 1 results: Acceptance
2772	192.627487	192.168.88.135	192.168.88.128	DRSUAPI	642 DsGetNCChanges request[Long frame (424 bytes)]
2776	192.628704	192.168.88.128	192.168.88.135	DRSUAPI	822 DsGetNCChanges response
2778	192.630639	192.168.88.135	192.168.88.128	DRSUAPI	338 DsReplicaSync request
2779	192.631107	192.168.88.128	192.168.88.135	DRSUAPI	178 DsReplicaSync response
2780	192.631339	192.168.88.128	192.168.88.135	DRSUAPI	658 DsGetNCChanges request[Long frame (456 bytes)]
2782	192.632021	192.168.88.135	192.168.88.128	DRSUAPI	1038 DsGetNCChanges response, Error: WERR_INVALID_HANDLE[Long frame (2312 bytes)]
2793	192.733127	192.168.88.129	192.168.88.128	DCERPC	214 Bind: call_id: 2, Fragment: Single, 3 context items: EPMV4 V3.0 (32bit NDR), EPMV4 V3.0 (64bit NDR), EPMV4 V3.0 (6cb71c2c-9812-4540-8300-000000000000)
2794	192.733570	192.168.88.128	192.168.88.129	DCERPC	162 Bind_ack: call_id: 2, Fragment: Single, max_xmit: 5840 max_recv: 5840, 3 results: Provider rejection, Acceptance, Negotiate ACK
2795	192.733770	192.168.88.129	192.168.88.128	EPH	222 Map request, RPC_NETLOGON, 32bit NDR
2796	192.734087	192.168.88.128	192.168.88.129	EPH	322 Map response, RPC_NETLOGON, 32bit NDR, RPC_NETLOGON, 32bit NDR
2800	192.735083	192.168.88.129	192.168.88.128	DCERPC	214 Bind: call_id: 2, Fragment: Single, 3 context items: RPC_NETLOGON V1.0 (32bit NDR), RPC_NETLOGON V1.0 (64bit NDR), RPC_NETLOGON V1.0 (6cb71c2c-9812-4540-8300-000000000000)
2801	192.735335	192.168.88.128	192.168.88.129	DCERPC	162 Bind_ack: call_id: 2, Fragment: Single, max_xmit: 5840 max_recv: 5840, 3 results: Provider rejection, Acceptance, Negotiate ACK
2802	192.735486	192.168.88.129	192.168.88.128	RPC_NE...	284 NetrServerReqChallenge request
2803	192.735776	192.168.88.128	192.168.88.129	RPC_NE...	98 NetrServerReqChallenge response
2804	192.735952	192.168.88.129	192.168.88.128	RPC_NE...	258 NetrServerAuthenticate3 request
2805	192.737039	192.168.88.128	192.168.88.129	RPC_NE...	98 NetrServerAuthenticate3 response
2806	192.737828	192.168.88.129	192.168.88.128	DCERPC	176 Alter_context: call_id: 4, Fragment: Single, 1 context items: RPC_NETLOGON V1.0 (64bit NDR)
2807	192.738113	192.168.88.128	192.168.88.129	DCERPC	130 Alter_context_resp: call_id: 4, Fragment: Single, max_xmit: 5840 max_recv: 5840, 1 results: Acceptance
2808	192.738304	192.168.88.129	192.168.88.128	RPC_NE...	302 NetrLogonDummyRoutine1 request[Malformed Packet]
2809	192.738662	192.168.88.128	192.168.88.129	RPC_NE...	174 NetrLogonDummyRoutine1 response[Malformed Packet]
2810	192.738898	192.168.88.129	192.168.88.128	RPC_NE...	414 NetrLogonSendToSam request[Malformed Packet]
2811	192.741510	192.168.88.128	192.168.88.129	DRSUAPI	722 DsGetNCChanges request[Long frame (528 bytes)]
2812	192.741607	192.168.88.128	192.168.88.129	RPC_NE...	158 NetrLogonSendToSam response[Malformed Packet]
2813	192.741811	192.168.88.129	192.168.88.1	SAHR	198 SetUserInfo2 response

The first red block shows the password reset being invoked. Before the response is sent back to the client, the domain controller (192.168.88.129) connects to the PDC (192.168.88.128) and forwards the new password using the **NetrLogonSendToSam** RPC call. If the password reset event happens on the PDC, then this RPC is not invoked since the new password is already known on the PDC.

This process allows the DC to do a second opinion when an authentication attempt fails. If **Piet** has changed his password on DC A and immediately after tries to authenticate on DC B, DC B will check with the PDC if the credentials are correct. This behavior can be observed in the following image:

DESKTOP-SIGT7GL.loc...	ADDC03.rebel.local	LDAP	599 bindRequest(143) "<ROOT>" , NTLMSSP_AUTH, User: \piet.pietersonsasl
ADDC03.rebel.local	ADPDC01.rebel.local	RPC_NETLOGON	910 NetrLogonSamLogonEx request[Malformed Packet]
ADPDC01.rebel.local	ADDC03.rebel.local	RPC_NETLOGON	174 NetrLogonSamLogonEx response, STATUS_WRONG_PASSWORD
ADDC03.rebel.local	DESKTOP-SIGT7GL.loc...	LDAP	165 bindResponse(143) invalidCredentials (8009030C: LdapErr: DSID-0C090569,

In the example, we can see that **ADDC03** forwards the (invalid) credential to **ADPDC01** - which is the PDC - only to have it verified that the password is indeed incorrect.

So what exactly is this **NetrLogonSendToSam** RPC call?

Understanding the RPC call

Googling this RPC call leads us to the technical documentation site of Microsoft[2], detailing the RPC call:

```

1 NTSTATUS NetrLogonSendToSam(
2     [in, unique, string] LOGONSRV_HANDLE PrimaryName,
3     [in, string] wchar_t* ComputerName,
4     [in] PNETLOGON_AUTHENTICATOR Authenticator,
5     [out] PNETLOGON_AUTHENTICATOR ReturnAuthenticator,
6     [in, size_is(OpaqueBufferSize)]
7     UCHAR * OpaqueBuffer,
8     [in] ULONG OpaqueBufferSize
9 );

```

Looking at the call specifications, we're interested in what's inside the opaque buffer, but the article above does not mention how the buffer is constructed. However, take note of the following comment: *A buffer to be passed to the Security Account Manager (SAM) service on the PDC. The buffer is encrypted on the wire.*

After doing some desk research, we find the documentation for SAM Server-to-Server messages[3]. The base message is constructed with the message type, size and the message itself, as can be seen in the following image:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
MessageType																															
MessageSize																															
Message (variable)																															
...																															

There various message types, but message type *PasswordUpdate Request Message*[4] looks quite promising. This message is constructed as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31																
Flags																																															
Size																																															
AccountRid																																															
PasswordExp																Reserved																															
OffsetLengthArray (variable)																																															
...																																															
Data (variable)																																															
...																																															

While reading the documentation, we noticed something interesting: *All bits that can be set, as specified below, can be set in any combination by the requestor with the exception of LM and NT; these bits MUST both be set or both be cleared.*[4] This sounds interesting and could be an indication that the data contains both NTLM and LM hashes.

We now know how the data should look more or less, let's see if we can dissect the data that's inside the `NetrLogonSendToSam` RPC call.

Deconstructing the RPC call

If you want to create test data, you can find more details about how to do this at the end of this blog post.

After resetting a password on a non PDC, we can see that the password is sent to the PDC:

ADDC10.rebel.local	adpdc01.rebel.local	RPC_NETL...	414	NetrLogonSendToSam request[Malformed Packet]
adpdc01.rebel.local	ADDC10.rebel.local	RPC_NETL...	158	NetrLogonSendToSam response[Malformed Packet]

Wireshark is able to partially decrypt the packet, but not fully dissect the message. Some data is successfully decrypted, but a majority of the message is gibberish:

00 00 02 00 00 00 00 00	16 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00	16 00 00 00 00 00 00 00
5c 00 5c 00 41 00 44 00	50 00 44 00 43 00 30 00	\\-\\-A-D- P-D-C-0-
31 00 2e 00 72 00 65 00	62 00 65 00 6c 00 2e 00	1-.r-e-b-e-l-.
6c 00 6f 00 63 00 61 00	6c 00 00 00 00 00 00 00	l-o-c-a-l-.....
07 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
07 00 00 00 00 00 00 00	41 00 44 00 44 00 43 00A-D-D-C-
31 00 30 00 00 00 00 00	49 d7 79 2d 85 ef 29 43	1-0-.....I-y-..)C
bc 54 31 65 00 00 00 00	74 00 00 00 00 00 00 00	-T1e-...t-.....
9e 88 8e 03 ef 51 cb 10	cd 23 84 25 e6 64 1b 26Q-..-#-%-d-&
9b 26 34 33 e3 02 fd 3e	2b 90 15 b0 e7 e5 07 3c	-&43-...>+.....<
84 07 21 8e 18 4d 40 a3	4c 5d f4 3c 68 7d 16 d2	..!..M@. L]-<h}..
ae 3f a9 3e 0c 34 37 7d	19 e7 92 47 c3 43 b9 23	..?..>-47}...G-C-#
e5 60 80 23 7d 03 27 5a	ad 00 48 62 b6 81 c1 d0	..`-#}-'Z..Hb....
10 81 b4 df 1a c6 dd 6b	6d 06 8f 4b ac 6e 75 3dk m--K-nu=
aa c0 44 e4 c8 f4 4e 74	11 2d 0e aa e7 67 18 17	..D-..Nt --...g..
e9 ea 18 06 74 00 00 00	00 00 00 00 00 00 00 00t-...

Looking back at the technical documentation, a hint was already given: *The buffer is encrypted on the wire*. Let's decrypt the complete packet ourselves, shall we?

Decrypting the data

The RPC call is invoked over a secure channel. Data is encrypted using a session key, the message digest and a sequence number. Using Wireshark it is possible to find the session key, by expanding **Auth Info** → **Secure Channel Verifier** and expand the expert section.

<div> <div>Auth Info: NETLOGON Secure Channel, Packet privacy, AuthContextId(1)</div> <div> Auth type: NETLOGON Secure Channel (68) Auth level: Packet privacy (6) Auth pad len: 8 Auth Rsvd: 0 Auth Context ID: 1 </div> </div> <div> <div>Secure Channel Verifier</div> <div> Sign algorithm: Unknown (0x0013) Seal algorithm: Unknown (0x001a) Flags: 0000 Sequence No: d722678586cba87f Packet Digest: cf90945a3220b0d2 Nonce: bc82d776d623fae3 </div> </div> <div> <div>[Expert Info (Chat/Security): Using session key learned in frame 4506 (8bd40864) from keytab principal [Using session key learned in frame 4506 (8bd40864) from keytab principal krbtgt@REBEL.LOCAL]</div> <div> [Severity level: Chat] [Group: Security] </div> </div>
--

When taking a look at the frame number, it becomes apparent that the frame has the actual session key in it.

```
▼ [Expert Info (Chat/Security): session key (8bd4086484b2f594e1eb41f937f0a5a1)]
  [session key (8bd4086484b2f594e1eb41f937f0a5a1)]
  [Severity level: Chat]
  [Group: Security]
```

To decrypt the contents of the RPC call, the package digest must be used as Initialization Vector (IV) to decrypt the sequence number. This value must be concatenated with itself and can then be used as IV. The session key needs to be XOR'd with `0xf0` after which we can decrypt RPC call. More info can be found here:

<https://github.com/billchaison/securechannel>

Using this info, the message can be decrypted manually and results in (more or less) the same output as Wireshark:

The image shows a Wireshark packet capture of a Network Data Representation (NDR) structure. The packet is 100 bytes long. The NDR structure is as follows:

Offset	Hex	ASCII
00000000	13 A1 6D F3 B4 5B 10 0F 1F D4 B2 5D 6D FB 7A 1B	..m..[.....]m.z.
00000010	00 00 00 00 00 00 00 00 16 00 00 00 00 00 00 00
00000020	5C 00 5C 00 41 00 44 00 50 00 44 00 43 00 30 00	.\.A.D.P.D.C.0.
00000030	31 00 2E 00 72 00 65 00 62 00 65 00 6C 00 2E 00	1..r.e.b.e.l..
00000040	6C 00 6F 00 63 00 61 00 6C 00 00 00 00 00 00 00	l.o.c.a.l.....
00000050	07 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000060	07 00 00 00 00 00 00 00 41 00 44 00 44 00 43 00A.D.D.C.
00000070	31 00 30 00 00 00 00 00 49 D7 79 2D 85 EF 29 43	1.0.....I.y-..)C
00000080	BC 54 31 65 00 00 00 00 74 00 00 00 00 00 00 00	.T1e....t.....
00000090	9E 88 8E 03 EF 51 CB 10 CD 23 84 25 E6 64 1B 26Q...#.%d&
000000A0	9B 26 34 33 E3 02 FD 3E 2B 90 15 B0 E7 E5 07 3C	.&43...>+.....<
000000B0	84 07 21 8E 18 4D 40 A3 4C 5D F4 3C 68 7D 16 D2	..!..M@.L].<h}..
000000C0	AE 3F A9 3E 0C 34 37 7D 19 E7 92 47 C3 43 B9 23	.?.>.47}...G.C.#
000000D0	E5 60 80 23 7D 03 27 5A AD 00 48 62 B6 81 C1 D0	.`.#}.'Z..Hb....
000000E0	10 81 B4 DF 1A C6 DD 6B 6D 06 8F 4B AC 6E 75 3Dkm..K.nu=
000000F0	AA C0 44 E4 C8 F4 4E 74 11 2D 0E AA E7 67 18 17	..D...Nt-...g..
00000100	E9 EA 18 06 74 00 00 00 00 00 00 00 00 00 00 00	...t.....

Labels in the image:

- LOGONSRV_HANDLE**: Points to the first 8 bytes of the packet.
- ComputerName**: Points to the 16 bytes starting at offset 0x20.
- Authenticator**: Points to the 16 bytes starting at offset 0x40.
- MessageType**: Points to the 4 bytes starting at offset 0x80.
- MessageSize**: Points to the 4 bytes starting at offset 0x84.
- Message**: Points to the 88 bytes starting at offset 0x88.

The data is encoded using *Network Data Representation* (NDR). During testing, we were able to workaround the need for a decoder, but given the limited sample size it could be possible that an NDR decoder is needed for more reliable decoding.

The `MessageType` field contains - well - the type of message. The following options are available[3]:

Value
PASSWORD_UPDATE_MSG
0x00000000
RESET_PWD_COUNT_MSG
0x00000001
FWD_PASSWORD_UPDATE_MSG
0x00000002
FWD_LASTLOGON_TS_UPDATE_MSG
0x00000003
RESET_SMART_CARD_ONLY_PWD
0x00000004

The **message** in the example above is the opaque buffer. The buffer itself is re-encrypted using the same session key, but without being XOR'd with **0xf0**. If we decrypt the buffer with this key and an empty IV, we can see some readable data. This resembles with the structure that we identified earlier on.

	Flags	Size	RID	Pwd_Exp	Reserved	
00000000	91 00 00 00	30 00 00 00	E6 3E 00 00	00 00 00 00	00 00 00 000....>.....
00000010	00 00 00 00	1C 00 00 00	1C 00 00 00	00 00 00 00	00 00 00 00
00000020	1C 00 00 00	10 00 00 00	2C 00 00 00	10 00 00 00	00 00 00 00,.....
00000030	70 00 69 00	65 00 74 00	2E 00 70 00	69 00 65 00	00 00 00 00	p.i.e.t..p.i.e.
00000040	74 00 65 00	72 00 73 00	6F 00 6E 00	CC 90 45 31	00 00 00 00	t.e.r.s.o.n...E1
00000050	7B 45 BE 4C	76 1F 61 5C	52 CA B2 54	6A 71 65 C2	00 00 00 00	{E.Lv.a\R..Tjqe.
00000060	D2 E0 B6 6D	D4 6A 33 C3	F6 5A 1A B1	52 83 D6 2E	00 00 00 00	...m.j3..Z..R..
00000070	83 C1 F8 80	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00

OffsetLengthArray

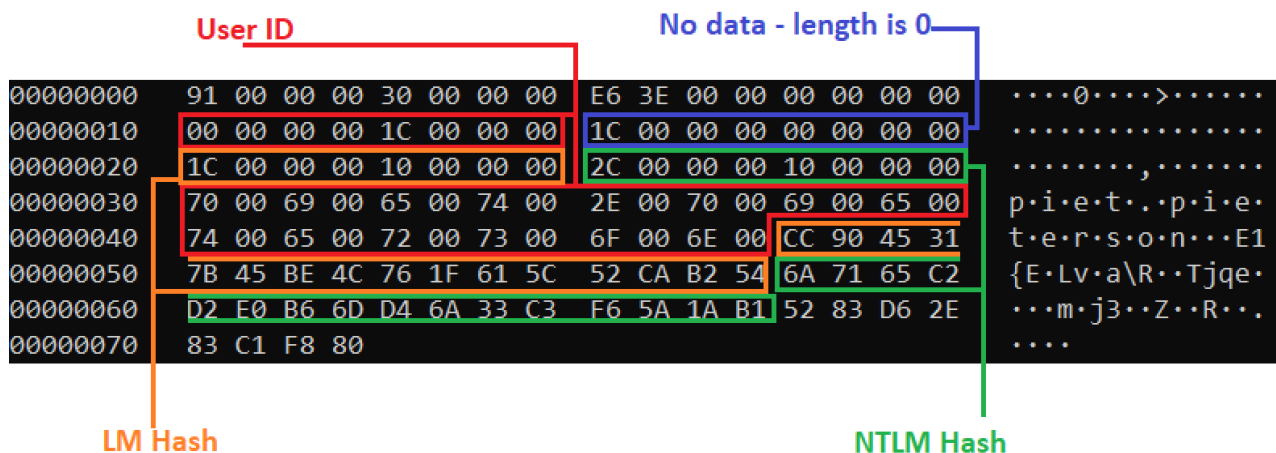
Data

Looking at the technical documentation, there are a few things that stand out:

- When **MessageType** is set to **0x00000000**, the message should contain an NTLM hash and an LM hash
- No reference to a **sAMAccountName** or other identity reference can be found in the documentation, but the identity is referenced in the buffer. In the example above, the identity is set to **pieter.pieterson**.

The `OffsetLengthArray` contains information what info can be found where. It is an array with members of 8 bytes. The first 4 bytes indicate where the data in the buffer starts and the other 4 bytes refer to the length of the data. The order of which data are stored is fixed. Seen from the end of the array:

- NTLM hash
- LM hash
- UserId



By passively sniffing network traffic, we successfully decrypted and recovered the NTLM and LM hash of the new password set on the account of `piet.pieterson`. There are more ways to gather hashes in an AD environment, however, it is quite rare to obtain LM hashes. Even if the AD environment has been configured to not store and use LM hashes, domain controllers will still send the LM hash over the wire to the PDC, only to be omitted when saving it to `ntds.dit`. Of course, configuring a password longer than 14 characters will mitigate this issue and will result in an LM hash for value null.

By using TShark, this whole process has been automated. In the following snippet we see a password reset event for the user `piet.pieterson` followed by the `NetrLogonSendToSam` data sent to the PDC.

```
C:\Code\PassiveAggression\PassiveAggression\bin\Debug\net7.0\PassiveAggression.exe
```

```
[+] Got password reset event:
    Username:    piet.pieterson
    New password: Hellothere1!

[+] NetrLogonSendToSam data:
    User:    piet.pieterson
    rID:     16102
    LM:      cc9045317b45be4c761f615c52cab254
    NTLM:    6a7165c2d2e0b66dd46a33c3f65a1ab1
```


Source of the tool, pcaps and key data can be found on our GitHub page:
<https://github.com/huntandhackett/PassiveAggression>


But wait, there's more!

Being able to recover password hashes is nice, but intercepting and decoding the password reset event itself ultimately yields the same level of access. However, there is a surprising element to this call that allows us to step up our level of passive persistence.

If a domain is compromised, one of the key remediation steps is to reset the password of the `krbtgt` account twice. Intercepting these password reset events is not useful, since according to the Microsoft documentation [5] *"The password that you specify isn't significant because the system will generate a strong password automatically independent of the password that you specify."*

However, once reset on the domain controller, the password replication process is the same. That means that the new password of the `krbtgt` account is sent to the PDC using - you guessed it - the `NetrLogonSendToSam` RPC call!

We'll go over why this is not a fluke or bad practice in the next blog post but for now, let's enjoy the glorious sight of a new `krbtgt` hash, delivered at our digital doorstep:

 C:\Code\PassiveAggression\PassiveAggression\bin\Debug\net7.0\PassiveAggression.exe

```
[+] NetrLogonSendToSam data:
    User:   krbtgt
    rID:    502
    LM:     aad3b435b51404eeaad3b435b51404ee
    NTLM:   ad840b4a42cb50b8d854afb62e1220e8
```

By comparing this hash with the hash in the `ntds.dit` database, we know that this is the new hash of the `krbtgt` account and not the hash representation of the clear text password that we entered during the password reset.

Resetting this password directly on the PDC prevents the data being shared using the `NetrLogonSendToSam` RPC call. However, other domain controllers in the domain would still need to be notified of the newly configured password, otherwise replication will stop working. This is where other forms of replication comes to play and will be looked at in the next blog post.

Checking our goals

In the previous blog post, a few goals were established to determine if a new technique would be useful. This new method can be used to survive a remediation process, but only if the password of the `krbtgt` account is reset on a non-primary domain controller. If a primary domain controller is used to invoke the password reset, then the remediation process would be effective. Therefore, this method cannot be used to survive the

remediation process. Further, this technique yields outdated password hashes, which are very useful - especially non-null LM hashes - but serve a different purpose. In some scenarios – such as setting up a NETLOGON session where the NTLM hash is used to generate an encryption key to be used with AES128-CFB encryption – this is fine. But authenticating using the Kerberos protocol would result in tickets being created using obsolete encryption or hashing algorithms, which is not considered opsec safe.

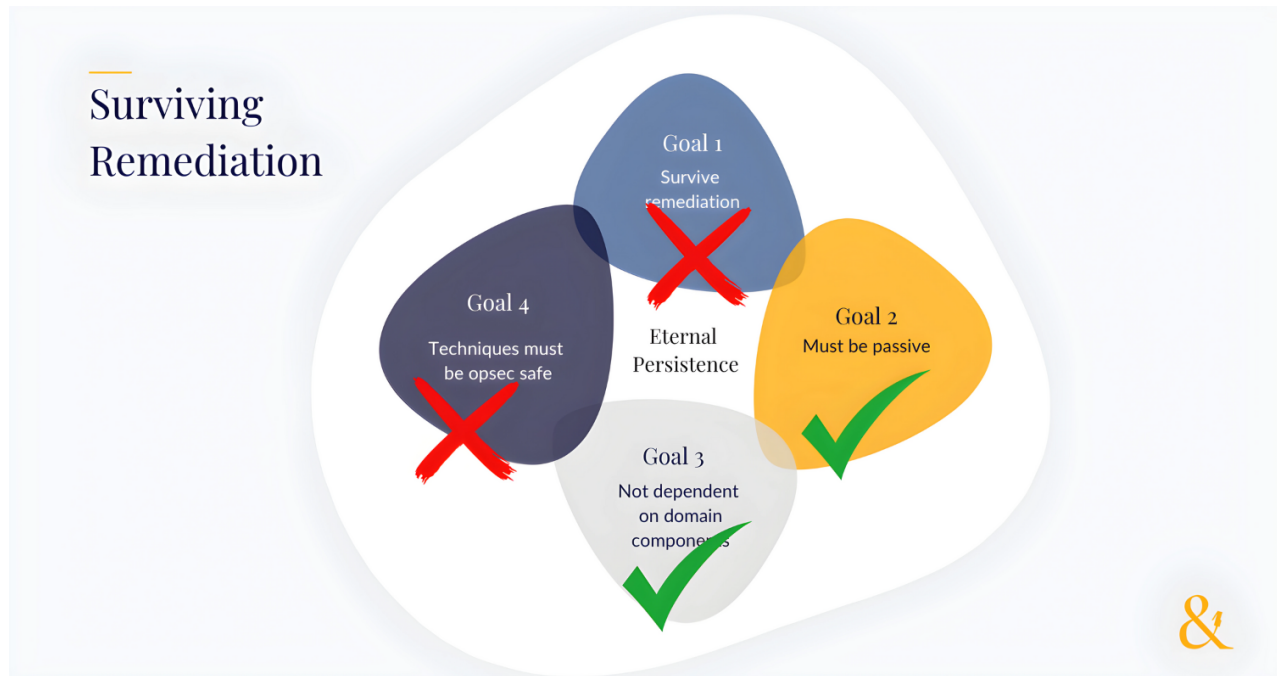


Figure 1. Results vs pre-agreed goals for eternal persistence

Create test data

Set up a Wireshark and an Active Directory environment. Refer to the previous blog post for pointers on how to do this.

Determine which domain controller holds the PDC FSMO role by running `netdom query fsmo` on a domain joined computer or a domain controller. Take note of another domain controller not having this role, connect to it and initiate a password reset of a user account. Examples can be found in the previous blog post.

After a few seconds, you should see `RPC_NETLOGON` API calls popping up in Wireshark, including the `NetrLogonSendToSam` RPC call:

ADDC10.rebel.local	adpdc01.rebel.local	RPC_NETL...	414 NetrLogonSendToSam request[Malformed Packet]
adpdc01.rebel.local	ADDC10.rebel.local	RPC_NETL...	158 NetrLogonSendToSam response[Malformed Packet]

References

Keep me informed

[Sign up for the newsletter](#)

