

How to Achieve Eternal Persistence Part 3: How to access and recover replicated secrets

& huntandhackett.com/blog/how-to-achieve-eternal-persistence-part-3

Rindert Kramer

Rindert Kramer @

Jun 6, 2024 10:12:38 AM

This is the third blog post in the series How to Achieve Eternal Persistence. In the [first blog](#), we looked at decoding captured password reset events, while the [previous blog](#) post detailed how to decode the `netrLogonSendToSam` RPC call. This call transfers updated password hashes from user accounts to the primary domain controller in the domain, including that of the `krbtgt` service account. Having access to the password hash of this service account yields a lot of possibilities, varying from decrypting traffic to forging our own Kerberos tickets.

The only downside that we currently face is that the hash is an NTLM hash, which is old and will be deprecated in newer Windows versions. In specific scenarios it can also be used as key to encrypt data using the AES128 encryption algorithm - since both key lengths are equal in size, however, requesting Kerberos tickets using the NTLM hash is a major giveaway because it defaults to RC4 encryption. This type of key is not suitable for our cause and the use of it should be avoided if stealth is the goal.

To get other types of keys, we need to look at replication traffic. After a password has been changed, the new password must be shared with other domain controllers (DCs). The DC on which the password has been changed will send out an urgent replication request to its replication partner. This request tells the receiving DC "Hey, got some data for you". The receiving DC will then issue a replication request to the issuing DC to receive the updated data. This is different than the message types detailed in the previous blog post, where the DC will send the new credentials to the PDC first.

This request is wrapped in a struct that looks something like the following[1]:

```

1  typedef struct {
2      UUID uuidDsaObjDest;
3      UUID uuidInvocIdSrc;
4      DSNAME* pNC;
5      USN_VECTOR usnvecFrom;
6      UPTODATE_VECTOR_V1_EXT* pUpToDateVecDest;
7      ULONG ulFlags;
8      ULONG cMaxObjects;
9      ULONG cMaxBytes;
10     ULONG ulExtendedOp;
11     ULARGE_INTEGER liFsmoInfo;
12     PARTIAL_ATTR_VECTOR_V1_EXT* pPartialAttrSet;
13     PARTIAL_ATTR_VECTOR_V1_EXT* pPartialAttrSetEx;
14     SCHEMA_PREFIX_TABLE PrefixTableDest;
15 } DRS_MSG_GETCHGREQ_V8;

```

This request contains the object references, the attributes and something like the version counter of the data the issuing DC currently has. This is taken into account and updated data will be returned to the issuing DC. That reply is built according to the following structure[2]:

```

1  typedef struct {
2      UUID uuidDsaObjSrc;
3      UUID uuidInvocIdSrc;
4      DSNAME* pNC;
5      USN_VECTOR usnvecFrom;
6      USN_VECTOR usnvecTo;
7      UPTODATE_VECTOR_V2_EXT* pUpToDateVecSrc;
8      SCHEMA_PREFIX_TABLE PrefixTableSrc;
9      ULONG ulExtendedRet;
10     ULONG cNumObjects;
11     ULONG cNumBytes;
12     REPLENTINFLIST* pObjects;
13     BOOL fMoreData;
14     ULONG cNumNcSizeObjects;
15     ULONG cNumNcSizeValues;
16     DWORD cNumValues;
17     REPLVALINF_V1* rgValues;
18     DWORD dwDRSError;
19 } DRS_MSG_GETCHGREPLY_V6;

```

There are multiple iterations of both requests and replies, adding more features and fields with newer versions of Windows. These versions were encountered the most during testing, but your mileage may vary.

Replication data

Replication data is sent over the network using the Network Data Representation (NDR) syntax. This is a common practice for API calls in the Windows RPC protocol to send structured data over the wire.

If you have Wireshark set up correctly to decrypt Kerberos messages (see previous blog post for pointers to set this up), enter the following filter `drsuapi.opnum == 3 && dcerpc.pkt_type == 2`. This filter will display only responses (`pkt_type == 2`) to replication requests (`opnum == 3`), and you should be able to see the decrypted replication data:

0000	06 00 00 00 06 00 00 00	b8 ce f8 1e b0 1a d6 4bK
0010	93 7e 5b 73 72 e8 12 26	88 bb c9 64 8f aa 57 48	..[sr..&...d..WH
0020	8c 87 a4 3c 34 55 41 b2	00 00 02 00 00 00 00 00	...<4UA.....
0030	47 a0 00 00 00 00 00 00	00 00 00 00 00 00 00 00	G.....
0040	47 a0 00 00 00 00 00 00	68 a0 00 00 00 00 00 00	G.....h.....
0050	00 00 00 00 00 00 00 00	68 a0 00 00 00 00 00 00h.....
0060	04 00 02 00 28 00 00 00	08 00 02 00 00 00 00 00(.....
0070	00 00 00 00 b0 05 00 00	00 00 00 00 00 00 00 00
0080	00 00 00 00 00 00 00 00	00 00 00 00 ac 00 02 00
0090	00 00 00 00 12 00 00 00	5c 00 00 00 18 00 00 00\.....
00a0	0a 11 cc a9 8e 80 03 4f	91 be c5 f6 04 eb 0d b00.....
00b0	01 04 00 00 00 00 00 05	15 00 00 00 46 8c 68 b7F..h..
00c0	e9 68 0a 92 24 96 c2 a2	00 00 00 00 11 00 00 00	..h..\$......
assembled TCP (1612 bytes)		Decrypted GSS-Krb5 CFX DCE (1504 bytes)	Decrypted stub data (1504 bytes)

This data is the raw NDR formatted data of the `DRS_MSG_GETCHGREPLY_V*` structure mentioned earlier. In order to iterate through all the objects and attributes, we need to decode it.

There are various tools that can decode NDR data, including:

- Sharpkatz[3]
- Mimikatz[4]
- Impacket[5]
- WindowsProtocolTestSuites[6]
- NtApiDotNet[7]

While **Sharpkatz** and **Mimikatz** are able to decode NDR data, they handle the complete process, utilizing Windows Kernel API calls to craft the request message, invoke the RPC call and process the response. Replicating this (*Got it?*) so it would work with only raw bytes, filling in pointers and such seemed a bit too tedious. **Impacket** has an NDR decoder which a lot of Impacket tools rely on. This worked in some scenarios, but resulted in decoding issues most of the time.

WindowsProtocolTestSuites is a repository maintained by Microsoft that contains samples and examples for a variety of internal Windows protocols. This includes examples for encoding and decoding NDR data streams. This also worked in some scenarios, but resulted in decoding issues most of the time.

This led me to **NtApiDotNet**. This project, created by James Forshaw of Google Project Zero, can extrapolate an NDR encoder and decoder from a binary with RPC server logic

in it. This means that if we create a binary that invokes the same RPC calls that are invoked during the replication process, **NtApiDotNet** should be able to generate code to encode data into NDR formatted data or decode NDR data back into the original structure.

We took that route and created the binaries to be used with the **NtApiDotNet** project. Details on how to replicate this can be found in this GitHub repository:

<https://github.com/huntandhackett/PassiveAggression>. This repository also contains pointers on how to set things up correctly.

Once the binary has been compiled successfully, make sure you have a copy of **NtApiDotNet**. Go to the output folder, import the **NtApiDotNet** module and check if the RPC server is recognized:

```
[C:\tst] $ Get-RpcServer .\repl.exe
```

Name	UUID	Ver	Procs	EPs	Service	Running
repl.exe	4870536e-23fa-4cd5-9637-3f1a1699d3dc	1.0	32	0		False

Next, use the steps documented in [this blog post](#) to compile an encoder and decoder:

```
1 PS> $rpc = Get-RpcServer repl.exe -ResolveStructureNames
2 PS> Format-RpcComplexType $rpc.ComplexTypes -Pointer
```

This should result in a C# class file with methods to deserialize the NDR data back into the right struct. After some refactoring for easier readability and removing unused functions, I was able to deserialize NDR data:

```
byte[] inputBytes = Misc.HexStringToBytes(hexStream);

// Running Format-RpcComplexType results in a class
// where _Unmarshal_Helper is helper class to deserialize NDR data
_Unmarshal_Helper helper = new _Unmarshal_Helper(inputBytes);

result.reply = helper.ReadReferentValue<DRS_MSG_GETCHGREPLY_V6>(
    new Func<DRS_MSG_GETCHGREPLY_V6>(helper.Read_DRS_MSG_GETCHGREPLY_V6),
    false);
```

This worked well, until it didn't. This pattern became a bit too repetitive for my liking and was a bit confusing. After staring the issue right in the eye for maybe a day, it finally became clear.

(Pointer) size matters

To understand why deserializing some data worked while other data didn't, we need some basic understanding of how RPC calls are established.

Step 1: Endpoint mapping

The client sends a mapping request to the server. The client includes the RPC interface it wants to connect to. The server responds with endpoint information, including the port the client can connect to.

Step 2: Binding

This is where the client establishes a connection with the server and the RPC interface it wants to connect to. This phase includes authentication details and syntax negotiation.

Step 3: Altering context

Clients may send an *alter context request* if additional settings or features must be negotiated, such as authentication or a different encryption algorithm.

Step 4: RPC call

This is the step where the actual RPC call is invoked under the negotiated context.

Step 5: Cleanup

Client or server sends unbind request.

Now let's zoom in on the second step. During the bind call, the client will send two context items. These items contain the languages the client can speak. In the example below, the client can handle both 32bit and 64bit NDR syntax.

- Version: 5
 - Version (minor): 0
 - Packet type: Bind (11)
 - Packet Flags: 0x17
 - Data Representation: 10000000 (Order: Little-endian, Char: ASCII, Float: IEEE)
 - Frag Length: 1821
 - Auth Length: 1697
 - Call ID: 5
 - Max Xmit Frag: 5840
 - Max Recv Frag: 5840
 - Assoc Group: 0x00004806
 - Num Ctx Items: 2
 - Ctx Item[1]: Context ID:0, FRSTRANS, 32bit NDR
 - Context ID: 0
 - Num Trans Items: 1
 - Abstract Syntax: FRSTRANS V1.0
 - Transfer Syntax[1]: 32bit NDR V2
 - Transfer Syntax: 32bit NDR UUID:8a885d04-1ceb-11c9-9fe8-08002b104860
 - ver: 2
 - Ctx Item[2]: Context ID:1, FRSTRANS, 64bit NDR
 - Context ID: 1
 - Num Trans Items: 1
 - Abstract Syntax: FRSTRANS V1.0
 - Transfer Syntax[1]: 64bit NDR V1
 - Transfer Syntax: 64bit NDR UUID:71710533-beba-4937-8319-b5dbef9ccc36
 - ver: 1

When looking at the response, the server rejects the 32bit NDR syntax option and **acks** the 64bit option:

- ▼ Distributed Computing Environment / Remote Procedure Call (DCE/RPC) Bin
 - Version: 5
 - Version (minor): 0
 - Packet type: Bind_ack (12)
 - > Packet Flags: 0x17
 - > Data Representation: 10000000 (Order: Little-endian, Char: ASCII, Fl
 - Frag Length: 260
 - Auth Length: 168
 - Call ID: 5
 - Max Xmit Frag: 5840
 - Max Recv Frag: 5840
 - Assoc Group: 0x00004806
 - Scndry Addr len: 6
 - Scndry Addr: 57677
 - Num results: 2
 - ▼ Ctx Item[1]: Provider rejection, PNIO (Implicit Ar)
 - Ack result: Provider rejection (2)
 - Ack reason: Proposed transfer syntaxes not supported (2)
 - Transfer Syntax: PNIO (Implicit Ar)
 - Syntax ver: 0
 - ▼ Ctx Item[2]: Acceptance, 64bit NDR
 - Ack result: Acceptance (0)
 - Transfer Syntax: 64bit NDR
 - Syntax ver: 1

This means that for all other future calls in the established RPC context, data is transferred over the wire using NDR x64 bit syntax. This correlates with data being decoded successfully, had pointers 32bits in size:

Cancel count: 0	0000	06 00 00 00	06 00 00 00	b8 ce f8 1e b0 1a d6 4bK
[Opnum: 3]	0010	93 7e 5b 73 72 e8 12 26	88 bb c9 64 8f aa 57 48	~[sr-& ...d..WH	
[Request in frame: 1846]	0020	8c 87 a4 3c 34 55 41 b2	00 00 02 00 00 00 00 00	...<4UA-	
[Time from request: 0.000931000 seconds]	0030	38 b0 00 00 00 00 00 00	00 00 00 00 00 00 00 00	8.....	
Encrypted stub data: 6215dfb00130a08c93770f2b	0040	38 b0 00 00 00 00 00 00	41 b0 00 00 00 00 00 00	8..... A.....	
▼ Complete stub data (1536 bytes)	0050	00 00 00 00 00 00 00 00	41 b0 00 00 00 00 00 00 A.....	
Payload stub data (1536 bytes)	0060	04 00 02 00 28 00 00 00	08 00 02 00 00 00 00 00(.....	
▼ Auth Info: Kerberos SSP, Packet privacy, Auth	0070	00 00 00 00 d8 05 00 00	00 00 00 00 00 00 00 00	
Auth type: Kerberos SSP (16)	0080	00 00 00 00 00 00 00 00	00 00 00 00 ac 00 02 00	
Auth level: Packet privacy (6)	0090	00 00 00 00 24 00 00 00	80 00 00 00 00 00 00 00\$.....	
Auth pad len: 0	00a0	43 1c 31 fb ae 15 3f 4b	89 ca ba 65 11 fc bd 33	C.1...?K ...e...3	
Auth Rsvd: 0	00b0	01 04 00 00 00 00 00 05	15 00 00 00 46 8c 68 b7F..h..	
Auth Context ID: 0	00c0	e9 68 0a 92 24 96 c2 a2	00 00 00 00 23 00 00 00	.h..\$....#...	
> GSS-API Generic Security Service Applicatio	00d0	44 00 43 00 3d 00 44 00	6f 00 6d 00 61 00 69 00	D.C.=D- o-m-a-i-	
▼ DRSUAPI, DsGetNCChanges	00e0	6e 00 44 00 6e 00 73 00	5a 00 6f 00 6e 00 65 00	n-D-n-s- Z-o-n-e-	
Operation: DsGetNCChanges (3)	00f0	73 00 2c 00 44 00 43 00	3d 00 72 00 65 00 62 00	s.,-D-C=-r-e-b-	
	0100	65 00 6c 00 2c 00 44 00	43 00 3d 00 6c 00 6f 00	e-l.,-D- C=-l-o-	
	0110	63 00 61 00 6c 00 00 00	06 00 00 00 00 00 00 00	c-a-l...	

Packets that were not able to decode use 64bits pointers, as can be seen below:


```

Cancel count: 0
[Opnum: 3]
[Request in frame: 1802]
[Time from request: 0.001691000 seconds]
Encrypted stub data: 77e9d03a0d9774b197f364e
  Complete stub data (2972 bytes)
    Payload stub data (2972 bytes)
  Auth Info: Kerberos SSP, Packet privacy, Aut
    Auth type: Kerberos SSP (16)
    Auth level: Packet privacy (6)
    Auth pad len: 4
    Auth Rsvd: 0
    Auth Context ID: 0
    > GSS-API Generic Security Service Applicat
  DRSUAPI, DsGetNCChanges
    Operation: DsGetNCChanges (3)
    [Request in frame: 1802]
    level: 6
    DsGetNCChanges
      0000 06 00 00 00 00 00 00 00 06 00 00 00 00 00 00 00 .....
      0010 63 18 60 0b 01 89 e2 41 8c 42 8b 07 69 fd e1 ff c`...A`B`i...
      0020 27 e5 19 40 97 4f ee 40 8b 50 52 2b e4 57 82 17 '...@`O`@`-PR+-W...
      0030 00 00 02 00 00 00 00 00 64 60 03 00 00 00 00 00 ..... d`.....
      0040 00 00 00 00 00 00 00 00 64 60 03 00 00 00 00 00 ..... d`.....
      0050 69 60 03 00 00 00 00 00 00 00 00 00 00 00 00 00 i`.....
      0060 69 60 03 00 00 00 00 00 00 00 02 00 00 00 00 00 i`.....
      0070 28 00 00 00 00 00 00 00 00 00 02 00 00 00 00 00 (.....
      0080 00 00 00 00 01 00 00 00 f0 07 00 00 00 00 00 00 .....
      0090 00 00 02 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
      00a0 00 00 00 00 00 00 00 00 00 02 00 00 00 00 00 .....
      00b0 00 00 00 00 00 00 00 24 00 00 00 00 00 00 00 ..... $.....
      00c0 80 00 00 00 00 00 00 43 1c 31 fb ae 15 3f 4b ..... C`1...?K
      00d0 89 ca ba 65 11 fc bd 33 01 04 00 00 00 00 00 05 ...-e...3.....
      00e0 15 00 00 00 46 8c 68 b7 e9 68 0a 92 24 96 c2 a2 ...-F`h`-h`$...
      00f0 00 00 00 23 00 00 00 44 00 43 00 3d 00 44 00 ...#... D`C`=-D`
      0100 6f 00 6d 00 61 00 69 00 6e 00 44 00 6e 00 73 00 o-m-a-i-n-D-n-s-
      0110 5a 00 6f 00 6e 00 65 00 73 00 2c 00 44 00 43 00 Z-o-n-e`-s`-D-C-
      0120 3d 00 72 00 65 00 62 00 65 00 6c 00 2c 00 44 00 =-r-e-b`-e-l`-D-
      0130 43 00 3d 00 6c 00 6f 00 63 00 61 00 6c 00 00 00 C=-l-o`-c-a-l...

```

This meant that the reason why Impacket, ProtoSDK and the C# class `NtApiDotNet` products didn't work is because they only support 32bit NDR message syntax. Since we're dealing with intercepted data and cannot alter the binding process, we have to deal with what the client and server negotiated.

In some scenarios the client may send an `alter_context` request only providing a 32bit NDR syntax context, which the server accepts. RPC with all its quirks is a mind boggling journey on its own, so we did not spend a lot of time finding out how and why that is. If you do know why that is and in what context that occurs, please contact us at jobs@huntandhackett.com.

Making it work

Creating a new 64bit NDR decoder from scratch was not something to be desired and since the chosen path was already deep down the rabbit hole with the `NtApiDotNet` approach, going with this solution seemed the best approach.

Aligning pointers

Tooling such as Mimikatz and Sharpkatz can invoke the `GetNCChanges` RPC call and request lots of data from a domain controller. When doing so, pointer sizes never exceed 32bits. With this in mind, we assume for now that the last 4 bytes of a pointer in the NDR stream are always null bytes.

Using class `NdrUnMarshalBuffer` from the `NtApiDotNet` library, we can load all NDR into a memory stream. This class exposes methods to consume data from the stream. Calculating alignment and consuming the bytes to get data aligned again was straightforward, but in some cases actual data was consumed causing misalignment. Since the data was already consumed and the library did not have any method to restore the data onto the stream, I extended the library with a peek method that restored the position of the stream.


```

1 public byte[] PeekBuffer(int length)
2 {
3     if ((long)length > _stm.RemainingLength())
4         throw new IndexOutOfRangeException("Given length bigger than remaining
length");
5
6     long currentPosition = _stm.Position;
7
8     byte[] res = new byte[length];
9     res = ReadFixedByteArray(length);
10
11     _stm.Position = currentPosition;
12
13     return res;
14 }

```

This allowed for peeking at the stream without actually consuming the data.

Conformant arrays

The other obstacle had something to do with conformant arrays. Let's assume the following structs:

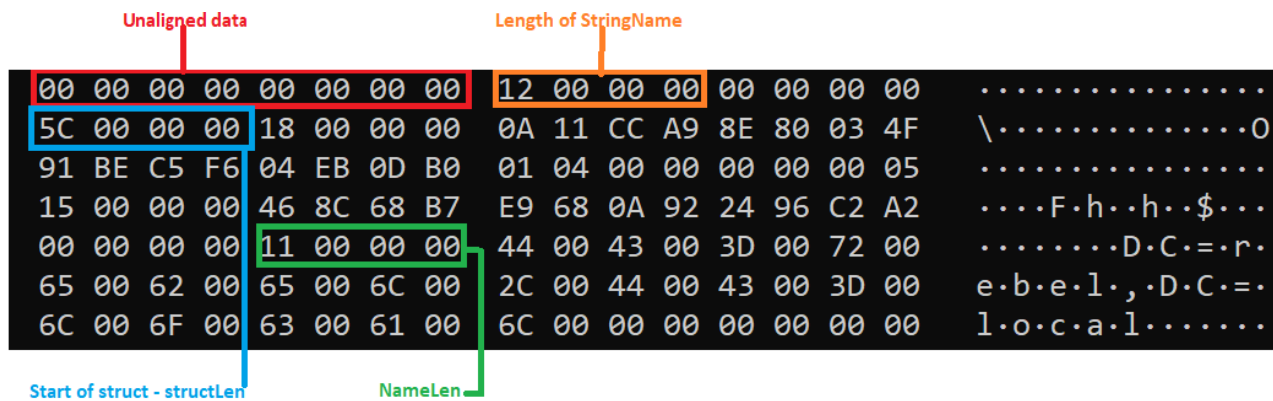
```

1 struct _NT4SID {
2     public sbyte[] Data;
3 }
4
5 struct _DSNAME {
6     public int structLen;
7     public int SidLen;
8     public System.Guid Guid;
9     public _NT4SID Sid;
10    public int NameLen;
11    public char[] StringName;
12 }

```

With a conformant array, the length of - in this case - `StringName` is placed before the actual start of the struct. When this array is read from the stream, the length of `StringName` will be read and placed onto a stack. Next, the complete struct is read from the stream and processed. When the `StringName` field is processed, the length referent is used that was put before the start of the struct. This works if the size of the pointers are 32bit, but when they're 64 bit this value will be different, meaning that the `StringName` field will be null or complete gibberish.

This is clear when looking at the following hex dump. When reading a conformant array, the length of `StringName` will be read, but the first 8 bytes are all 00s because the stack is misaligned.



Fixing the alignment by consuming 8 bytes from the stream before the struct is deserialized would fix the issue in this scenario. However, this struct is referenced frequently and the amount of alignment it needs varies. Structurally consuming 8 bytes would result in misalignment somewhere else later on in the stream.

Fixing this issue was trivial, since the data on the stream is linear and we happen to know the length of the `StringName` field, which is stored in `NameLen` and only needs to account for the trailing 00-byte.

However, this did cause another issue. When reading conformant arrays, the deserializer checks if there are items left on the stack. If so, the conformant array length will not be put on the stack, resulting in other structs not being parsed. To fix this, two functions were added to the `NtApiDotNet` library:

```

1 public void ClearConformanceValues()
2 {
3     _conformance_values = null;
4 }
5
6 public int[] GetConformanceValuesArray()
7 {
8     return _conformance_values;
9 }

```

The first method manually resets the stack, making sure that future conformant arrays will be read successfully. The second function returns the stack to the caller. The caller can then verify if the item on the stack (if any) is what is to be expected. If not, the stack can be cleared using the first method. A pull request has been created[8] to get this merged into the project, but at the time of writing it still needs to be processed by one of the maintainers of the repository.

After grinding through all structs and aligning all pointers correctly, 64bit NDR messages can now be decoded successfully.

Decoding replication traffic

In the following screenshot, we can see the password change of user `test`. We see that the domain controller first sends the password reset event to the primary domain controller by invoking the `NetrLogonSendToSam` RPC call. This call includes both NTLM and LM hashes and the relative User ID (rID) of the user account. Then, the secret is replicated. This does not include the LM hash, but does include the NTLM hash and AES keys.

```
[+] Got password reset event:
    Username:      test
    New password:   Hellothere2!

[+] NetrLogonSendToSam data:
    User:  test
    rID:   16601
    LM:    cc9045317b45be4c4466813f96e77910
    NTLM:  5d8b397985e472b9876753e18c71f54a

[+] Replicated secret:
    DN:      CN=test,CN=Users,DC=rebel,DC=local
    SID:     S-1-5-21-3077082182-2450155753-2730661412-16601
    GUID:    34d8d1c3-3bec-4f39-ae2e-dabe153163b9

    NTLM:    5d8b397985e472b9876753e18c71f54a
    salt:     REBEL.LOCALtest
    aes256:   19d61183620231c79bd8b5caa16d30b37d7ade865e5b7621e5403908608206df
    aes128:   96238354b0ddbd6ee18a85133b56e4d2
    des:      6d2cb6b358c17cfd
```

In the beginning of the first blog post, we stated that having eternal passive persistence was the ultimate goal. For that, we definitely want access to all key material of the `krbtgt` account, but as seen in this blog post, replication traffic is encrypted and can only be decrypted if you have access to the keys of the domain controller's computer accounts and the `krbtgt` account. Domain controllers will have access to these keys by default, since they own their own copy of the NTDS database.

Let's say that in a domain with two domain controllers, A and B, we initiate a password reset on the `krbtgt` account on DC A. How is DC B informed of this new password that DC A will create?


The answer is simple: via replication. Just like passwords of any regular user account, the new password of the `krbtgt` account will be shared using the replication process. Since replication traffic must be encrypted and DC B does not yet know the new password of the `krbtgt` account, DC A will use the previous password of the `krbtgt` account.

This creates a rather interesting catch-22 issue. If we, as an attacker, have access to all key material in a domain and have access to the domain controllers or *devices through which inter DC traffic is routed* we can intercept, decrypt and decode all password changes, including that of the `krbtgt` and domain controller computer accounts, resulting in practically eternal persistence. It is not necessary to have access to a domain controller, for example the following scenarios should suffice:

- Access to (virtual) switches that switches network traffic between domain controllers
- Access to hypervisors that routes network traffic from and to domain controllers

- Access to span ports on a network device that mirrors all network traffic, including traffic to/ from domain controllers
- Access to a 3rd party security provider that has access to all network flows

This can be demonstrated using the image below:

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

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

[+] Replicated secret:
    DN:     CN=krbtgt,CN=Users,DC=rebel,DC=local
    SID:    S-1-5-21-3077082182-2450155753-2730661412-502
    GUID:   3e96722b-4efd-4293-a965-5baa1fc57c1c

    NTLM:   ad840b4a42cb50b8d854afb62e1220e8
    salt:    REBEL.LOCALkrbtgt
    aes256: 52cc7216f6167c3575351eef15586821aa0bd7cd1c70eec38ff07f285dc06182
    aes128: 27a90c8d39c5b8ac8eeeb819b69ac31
    des:     a77570d30829d06e
```

At this point, you might be thinking “Having access to the *krbtgt* credential does not mean you can decrypt traffic sent to and received from domain controllers. You need the machine password for that as well”. That would be correct.

Let’s reset the password of a domain controller using this command:

```
netdom resetpwd /s:other.dc.local /ud:rebel\Administrator /pd:*
```

This will reset the password of the domain controller it is run on, using the domain controller specified in the */s* switch. (Fun fact, it seems that it uses the *SamrSetInformationUser2* RPC call to reset the password). Other domain controllers are notified of this new password via replication, which we can intercept and decode:

```
[+] Replicated secret:
    DN:     CN=ADDC10,OU=Domain Controllers,DC=rebel,DC=local
    SID:    S-1-5-21-3077082182-2450155753-2730661412-16103
    GUID:   63cbb5cf-43b9-420b-b571-e3fceeefdfc01

    NTLM:   c8316113be3b98166f8373998add23a
    salt:    REBEL.LOCALhostaddc10.rebel.local
    aes256: 8d3a59e8223ea8091050001fc3ec555f876fb636abd0c83fd5eae355f97161ba
    aes128: 1b9fedc30fa52179693b5ce8d06f2985
    des:     13a13ea1fd6220ce
```

Being able to decode replication data passively has a significant impact. It not only allows for intercepting password changes of user accounts, machine accounts and the *krbtgt* AD service account, you can also become your own passive domain controller where you

can keep track of all replicated data and use this as a method to exfiltrate data. If the replication decoder is running on a switch or router, it can exfiltrate all the data with little chance of being detected.

Checking our goals

In the first blog post of this series, we set a few goals to grade the solutions. This method can be run on any device that receives traffic sent to and received from domain controllers, is completely passive, contains keys used with modern encryption types and most importantly: it survives a remediation process. Regardless of whether the `krbtgt` account is replicated or the password of a domain controller is changed and communicated to other domain controllers, this should work because of the way AD works.



Figure 1. Results vs predetermined goals for eternal persistence

This has a serious impact on how remediation should be carried out, especially if the actor is sophisticated enough to conduct these kind of attacks.

Remediation

After reading this blog series, you might be wondering and thinking "But doing proper remediation would prevent all of this, right?" That would technically be correct, but in reality it depends on the remediation process. This kind of attack could survive a remediation process where:

- current domain controllers are demoted and replaced with new domain controllers
- *all* passwords are reset twice
- *all* servers, workstations and other AD assets and objects are cleaned and sanitized

The reason why this method could survive such a remediation process, is - as stated before - it can be executed on any device in the network, as long as network traffic between domain controllers can be intercepted. This can be a router, switch, hypervisor or a security provider with access to network data.

To fully remediate such a scenario, one could think of disconnecting all domain controllers from existing network equipment and generate new credential data twice (krbtgt keys, domain controller account passwords, etc) and make sure to fully replicate data across other domain controllers. This could increase remediation costs, effort and complexity, especially if domain controllers are located in multiple data centers.

Given their passive nature, these kinds of attacks are hard to detect. However, it can only be executed after full compromise of the AD domain which underlines the importance of security aspects such as prevention, detection and timely response. If you have questions about our research, require expert advice, or you are simply curious about improving your own organisation's resilience, [contact us here](#).

References



Keep me informed

[Sign up for the newsletter](#)

