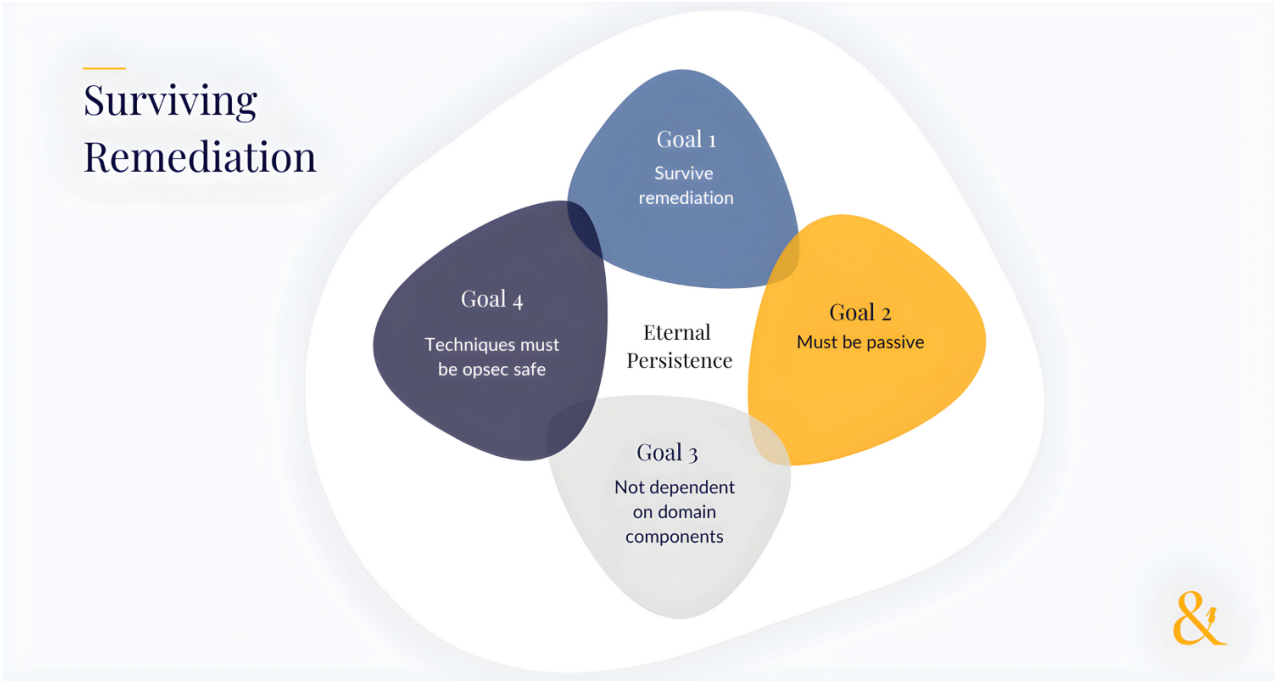
How to achieve eternal persistence in an Active Directory environment - Part 1

huntandhackett.com/blog/how-to-achieve-eternal-persistence

Rindert Kramer



Rindert Kramer @
May 23, 2024 12:07:15 PM

In this three-part blog series, we will explore different approaches to achieving passive persistence in an Active Directory (AD) environment. Specifically, we'll examine whether it is possible to survive a remediation process that contains steps such as rotating passwords, resetting AD service accounts, revoking logon sessions and removing backdoors on AD components such as domain controllers.

In this blog - the first in the series - we tackle the scenario of password rotation for compromised accounts, exploring how attackers can intercept and adapt to these changes. Subsequent blogs will delve into AD's password replication processes, as well as generic replication processes between domain controllers. Our goal is to examine whether it's possible to achieve "eternal persistence" – a state where an attacker remains embedded in the network without detection, regardless of the defensive measures taken. Ultimately, we aim to equip blue teamers and security professionals with the knowledge required to understand, detect, and defend against these sophisticated threats.

Introduction

During security tests (e.g. pentests, red team assignments), getting domain administrator privileges in an Active Directory (AD) domain results in a solid foundation to further achieve objectives that were agreed upon during the scoping process of the assignment. Having the highest privileges in a domain allows an attacker to access any resource within that domain. While Active Directory provides an organization the means and tools to organize Identity and Access management, it is therefore also the highest link in a company's chain of trust. If the highest link is compromised, there is no other authority to rely on to regain trust and integrity, which can be a serious issue in post compromise scenarios.

During a redteam, attempts to further advance through the network or maintain persistence can be thwarted by aware blue teams, who can patch points of entry, reset passwords, revoke sessions or rotate the `krbtgt` service account. If this happens, there are probably other ways to compromise the domain again, but that often involves interacting with services and components, leading to events being logged or triggering alerts. This got us wondering; is it possible to survive a remediation process that contains steps such as rotating passwords, resetting AD service accounts, revoking logon sessions and removing backdoors on AD components such as domain controllers?

This blog series will dive into the inner workings of AD, aiming to find a way to survive a remediation process. For this, we set the following goals:

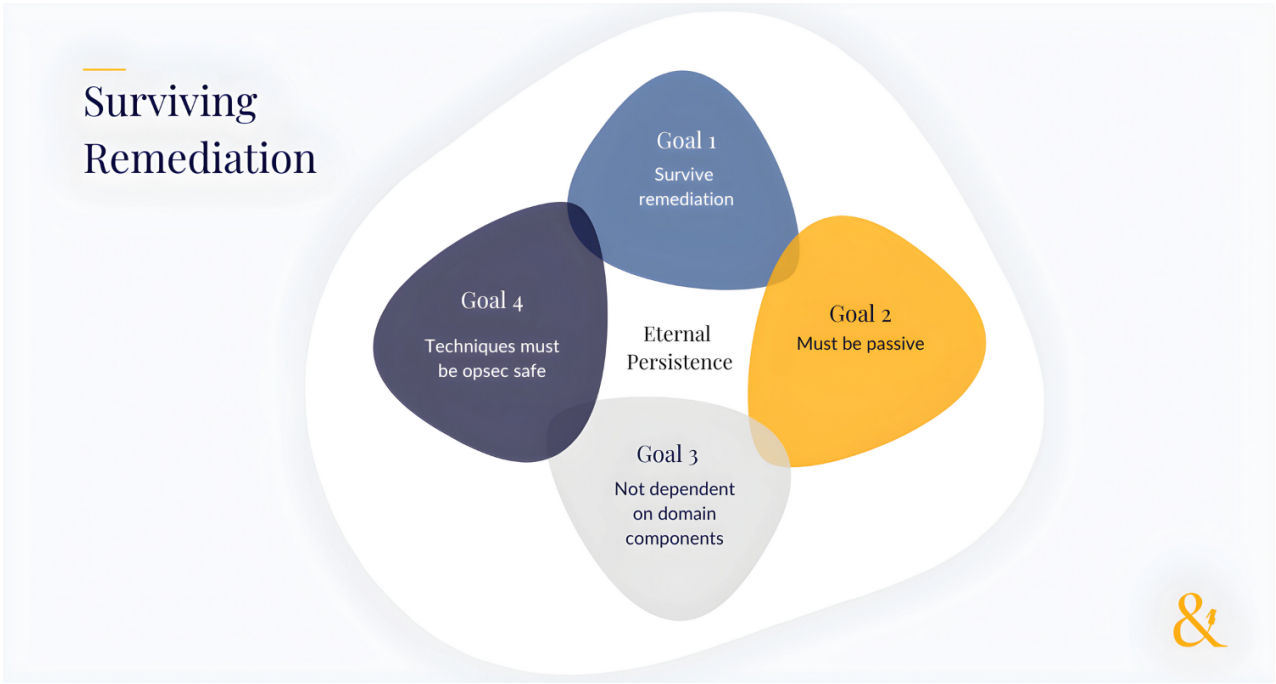


Figure 1. Goals for surviving the remediation process and achieving eternal persistence

1. **Survive remediation:** Using the technique, we should be able to survive a remediation process.
2. **It must be passive:** Surviving the remediation process should not rely on interacting with domain components, such as Domain Controllers. Less interaction leads to less noise, which means increased stealth.

3. **It should not depend on domain components:** The working of a technique should not rely on domain components, such as backdoors installed on domain controllers or servers. If a domain controller is cleaned or replaced with a new server, the technique should not be impacted by this.
4. **Techniques must be opsec safe:** If the technique results in the possession of an up –to date NTLM hash of the `krbtgt` account where the majority of logon events use AES keys to encrypt data, this won't be considered opsec safe. The use of an outdated encryption algorithm in lots of authentication requests where the majority use up-to-standard techniques is considered a big give-away.

This blog series will explore techniques for achieving passive persistence in an AD environment. Some techniques could result in an *eternal persistence* scenario, where the attacker does not need to have access to domain controllers or domain joined machines. In this scenario, current mitigation techniques are not sufficient to fully eradicate the attacker from the network. The only prerequisites are a full compromise of the domain, extracting the hashes from the AD database and access to network traffic being sent to/ received from domain controllers. This can be achieved in many ways such as listening on network equipment which the domain controllers are connected to.

We will begin with a scenario where the passwords of compromised accounts are being rotated. Having access to clear-text credentials is sometimes needed to achieve a certain goal, if Kerberos authentication or NTLM authentication are not supported. If passwords are rotated, one can extract the new password hash from the `ntds.dit` database, but that means interacting with the domain controller, which we want to avoid.

Because we have access to password data of all users and systems within the domain, and since we also have access to network traffic sent to and received from the domain controllers, another way to retrieve the new password is by decoding the password reset event. Instructions on how to create test data can be found at the end of this blog post.

Decoding a password reset

During a password reset, the system on which the reset was issued queries the remote Security Account Manager (SAM) database on the domain controller it is connected to using the SAM RPC endpoint. This is a database that stores identity-related information on a Windows system.

This looks something like the following network flow in Wireshark:

	Action	Details
1.	Protocol negotiation	Handshake between client and server to determine SMB version and dialect
2.	Session setup	Client authenticates to server. This is where session keys are established and pre-authentication hashes are calculated. This is used to prevent tampering of future SMB traffic.

3.	Tree connect	Connect to IPC$ file share
4.	Request file	Request handle to the SAM file on the server
5.	Get info	Get standard info about the file
6.	Bind	Bind to server to do RPC calls. This includes syntax negotiation and returning an authentication binding handle
7.	Connect5	Obtain a file handle to the SAM file. This includes the permissions needed to perform the password reset
8.	EnumDomains	Enumerate available domains
9.	LookupDomain	Lookup Security Identifier (sID) of the domain
10.	OpenDomain	Obtain handle to domain
11.	LookupNames	Lookup sID of the user
12.	OpenUser	Obtain handle to user
13.	GetUserPwdInfo	Obtain password policy of the domain
14.	SetUserInfo2	Set password to user account

On standalone machines, the SAM database also contains credentials. Domain controllers store their credentials in a different database (`ntds.dit`) but the SAM RPC interface can be used to query domain information and invoke other actions, such as password resets. Upon receiving and validating the call, the domain controller will update the `ntds.dit` database accordingly.

Now, let's dive into the `SetUserInfo2` RPC call and see if we can unravel its secrets.

Setting user info

When an admin makes changes to a user account, the aforementioned flow will result in the changes being applied and subsequently replicated to all domain controllers throughout the domain. But what exactly is shared with the domain controller on which the info is set?

Doing desk research about the `SetUserInfo2` RPC call yields surprisingly few results. This seems to be a known RPC call to reset passwords for user account in an Active Directory environment, but other than a small reference in the `MS-RSMC` section of the Microsoft documentation site, there isn't a lot of information on what exactly this call is and how it is constructed. Our educated guess is that this name stems from Samba's tool rpcclient which invokes the actual RPC call, and named the command `SetUserInfo2`, which was subsequently used by other tools as well. However, there are no other references to support our claim.

What we do know for sure is the opnum - or the operation number of the call - is 58. Using this information, we can find more documentation about this call[1] and we can see how this function should be invoked:

```
long SamrSetInformationUser2(
   [in] SAMPR_HANDLE UserHandle,
   [in] USER_INFORMATION_CLASS UserInformationClass,
   [in, switch_is(UserInformationClass)]
     PSAMPR_USER_INFO_BUFFER Buffer
 );
```

The first parameter would be the handle to the user account (result of the `OpenUser` RPC call). The second parameter refers the `USER_INFORMATION_CLASS` enum[2], which looks something like this:

```
 typedef  enum _USER_INFORMATION_CLASS
 {
   <...>shortened for readability</...>
   UserAllInformation = 21,
   UserInternal4Information = 23,
   UserInternal5Information = 24,
   UserInternal4InformationNew = 25,
   UserInternal5InformationNew = 26
   UserInternal7Information = 31,
   UserInternal8Information = 32
 } USER_INFORMATION_CLASS,
  *PUSER_INFORMATION_CLASS;
```

During testing, we only encountered the `UserInternal4InformationNew` user class being referenced.

Searching for the `SAMPR_USER_INTERNAL4_INFORMATION_NEW` structure, we see it is defined[3] as follows:

```
typedef struct _SAMPR_USER_INTERNAL4_INFORMATION_NEW {
   SAMPR_USER_ALL_INFORMATION I1;
   SAMPR_ENCRYPTED_USER_PASSWORD_NEW UserPassword;
 } SAMPR_USER_INTERNAL4_INFORMATION_NEW,
  *PSAMPR_USER_INTERNAL4_INFORMATION_NEW;
```

The `SAMPR_USER_ALL_INFORMATION` field is a struct containing updated values for attributes configured on the user account, such as department, `UserComment` and more. The `SAMPR_ENCRYPTED_USER_PASSWORD_NEW`[4] field is a buffer that carries an encrypted buffer.

```
typedef struct _SAMPR_ENCRYPTED_USER_PASSWORD_NEW {
  unsigned char Buffer[(256 * 2) + 4 + 16];
} SAMPR_ENCRYPTED_USER_PASSWORD_NEW,
 *PSAMPR_ENCRYPTED_USER_PASSWORD_NEW;
```

We now have a basic understanding of how these calls are established and what kind of data is transferred between client and server during a password reset event. The decrypted buffer should contain a clear text password - let's see if we can extract it.

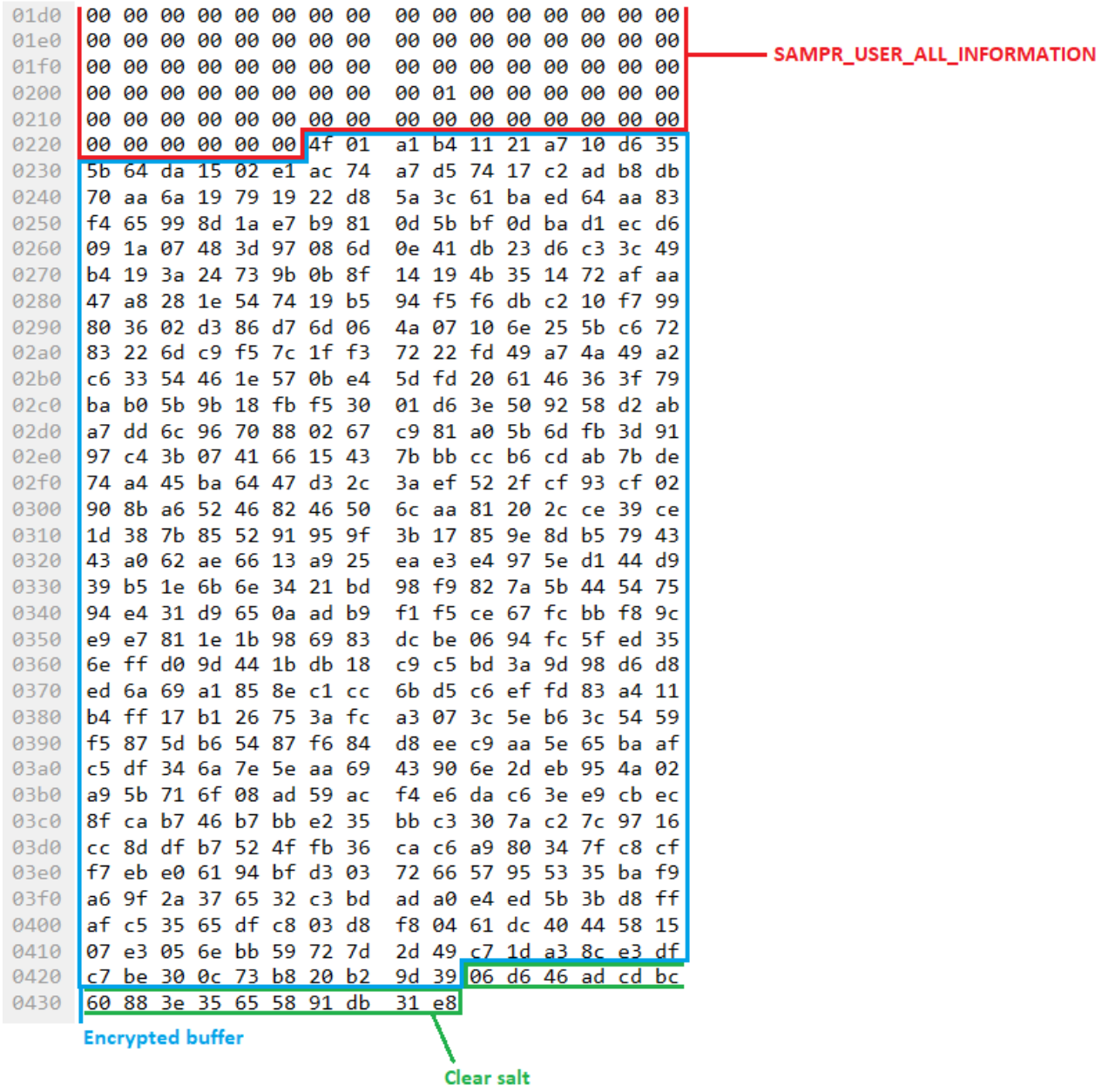
Decrypting the buffer

Reading the documentation, we uncover the following: The `SAMPR_USER_INTERNAL4_INFORMATION_NEW` structure holds all attributes of a user, along with an encrypted password. The encrypted password uses a salt to improve the encryption algorithm[3].

The encrypted buffer has a fixed size of `(256 *2) +4 +16 =` 532 bytes:

1. Buffer: 512 bytes
2. Length: 4 bytes
3. Clear salt: 16 bytes

This correlates with data in Wireshark, where the whole structure is filled with 00s, and the last remaining 532 bytes filled with random data.



Selecting the right key

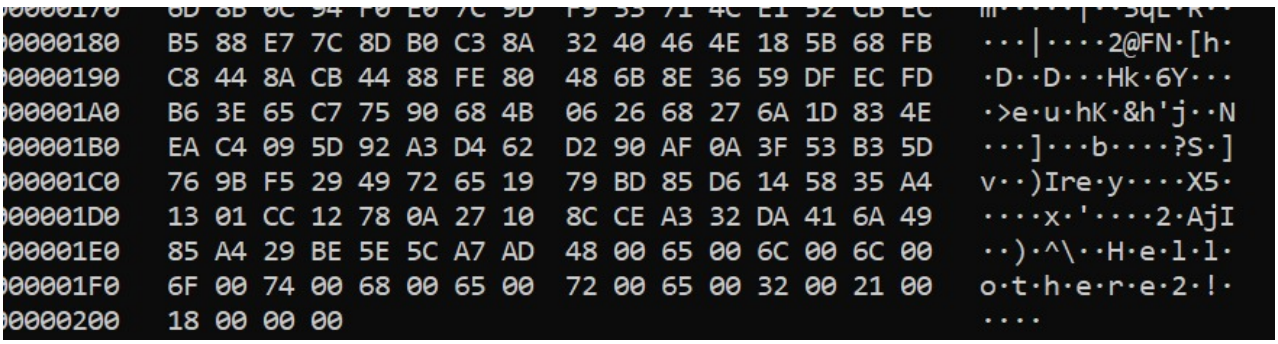
To create a decryption key, we need the key that was established during session negotiation and the clear salt. Next, we need to compute an `Application key` that is derived from the aforementioned key, a label and a context. The label and the context are concatenated to a byte array and the session key is used as secret to compute a `HMACSHA256` hash, the first 16 bytes of which will be used as an application key. This is illustrated within the following function, taken from the impacket framework[5]:

```python
def KDF_CounterMode(KI, Label, Context, L):
    h = 256
    r = 32

    n = L // h

    if n == 0:
        n = 1

    if n > (pow(2,r)-1):
        raise Exception("Error computing KDF_CounterMode")

    result = b''
    K      = b''

    for i in range(1,n+1):
       input = pack('>L', i) + Label + b'\x00' + Context + pack('>L',L)
       K = hmac.new(KI, input, hashlib.sha256).digest()
       result = result + K

    return result[:(L//8)]
```

Depending on the SMB dialect, the `Label` and `Context` values differ.

	Label	Context
SMB Dialect 3.1.1	`SMBAppKey\x00`	Pre-authentication hash
Other	`SMB2APP\x00`	`SmbRpc\x00`

The result of the `Key Derivation Function` is used with the clear salt to compute an MD5 hash. This hash can then be used to decrypt the buffer and length of the clear-text password. In the screenshot below, we can see that the clear text password is `Hellothere2!` and the length of the string is 24 bytes.



Tshark

While Wireshark is great for researching packets and flows manually, we want to do this automatically. Writing a proof-of-concept that can parse network traffic is a different beast and not what we're focusing on right now. Luckily, Wireshark is shipped with a tool that can do that for us: Tshark.

Tshark has the same capabilities as Wireshark but can be invoked from the command line. We wrote a function that invokes Tshark using the following parameters:

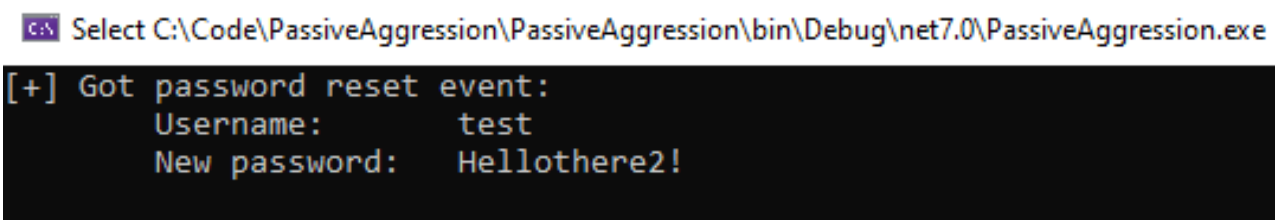
Parameter	Description
-2	Two stage process. This allows Tshark to defragment network packets into a single packet
-r	Path to pcap file. Recommended approach, since doing this live might result in issues
-K	Path to keytab file. Tshark will decrypt NTLM/ Kerberos data when possible
-Y	Wireshark filter to only include traffic we need
-T	Output in Json format. This allows for deserialization
-J	Select protocols we need
-x	Return raw data. Some decrypted fields are not returned properly, which this parameters resolves

This returns a deserialized object that contains all fields and values that were captured by Tshark.

To make it usable, we need to process a few packets:

- We need the SMB dialect and the pre-authentication hash (if applicable) to derive an application key
- We need the `NTLMSSP` session key that is established during the session setup
- The `SamrSetInformationUser2` RPC call only contains a handle to the user account. We need to store the contents of the `LookupNames` RPC call, since this will contain the username for which handle is requested later on.

Stitch all events together and we are now able to process password reset events:



Sample code, pcaps and keytab files can be found on our GitHub page: https://github.com/huntandhackett/PassiveAggression

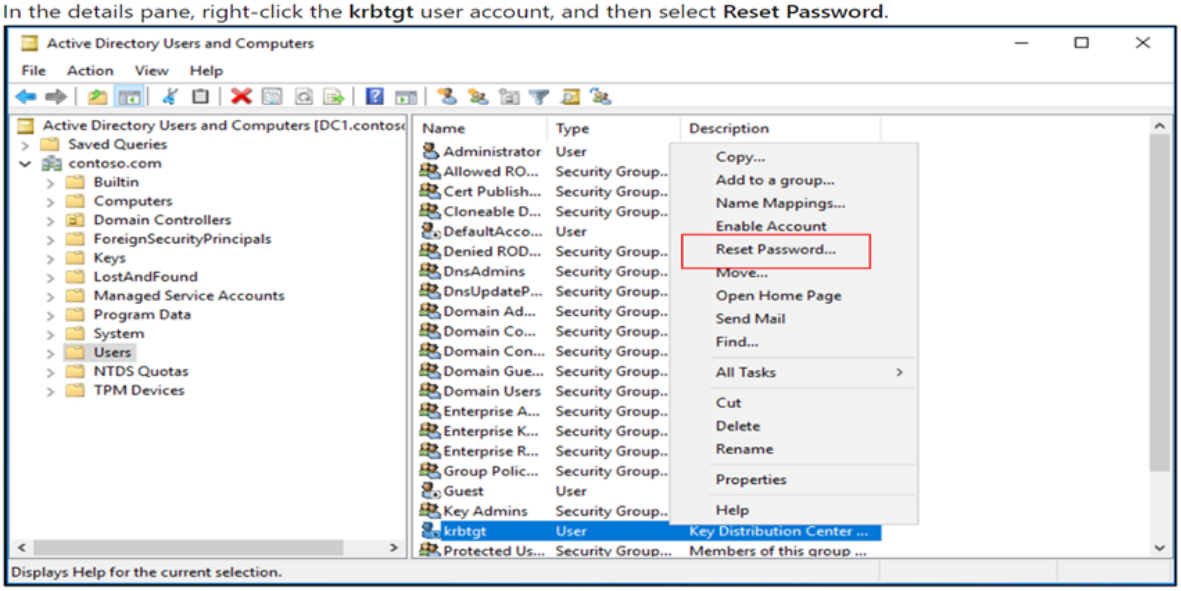
This method focuses only on password reset events. There are more ways to reset passwords resulting in different network types, password change events, and so on. While not covered in this post, methodologically the approach would be the same.

What about krbtgt?

After a domain compromise, the password of the `krbtgt`account must be reset twice. However, as can be seen in the following screenshot taken from the Microsoft documentation site[6], the domain controller will create a random and strong password which will be used instead.

Reset the krbtgt password

1. Select **Start**, point to **Control Panel**, point to **Administrative Tools**, and then select **Active Directory Users and Computers**.
2. Select **View**, and then select **Advanced Features**.
3. In the console tree, double-click the domain container, and then select **Users**.
4. In the details pane, right-click the **krbtgt** user account, and then select **Reset Password**.



5. In **New password**, type a new password, retype the password in **Confirm password**, and then select **OK**. The password that you specify isn't significant because the system will generate a strong password automatically independent of the password that you specify.

Closing thoughts

This method does not provide the means to survive a remediation process. All steps in this post can be done completely passively and there is no reliance on domain components, such as domain controllers. If passwords of admin accounts change, this mechanism provides the means to recover the new password. However, resetting the password of the `krbtgt` account will prevent the attacker from persisting in the network and thus not surviving the remediation process.

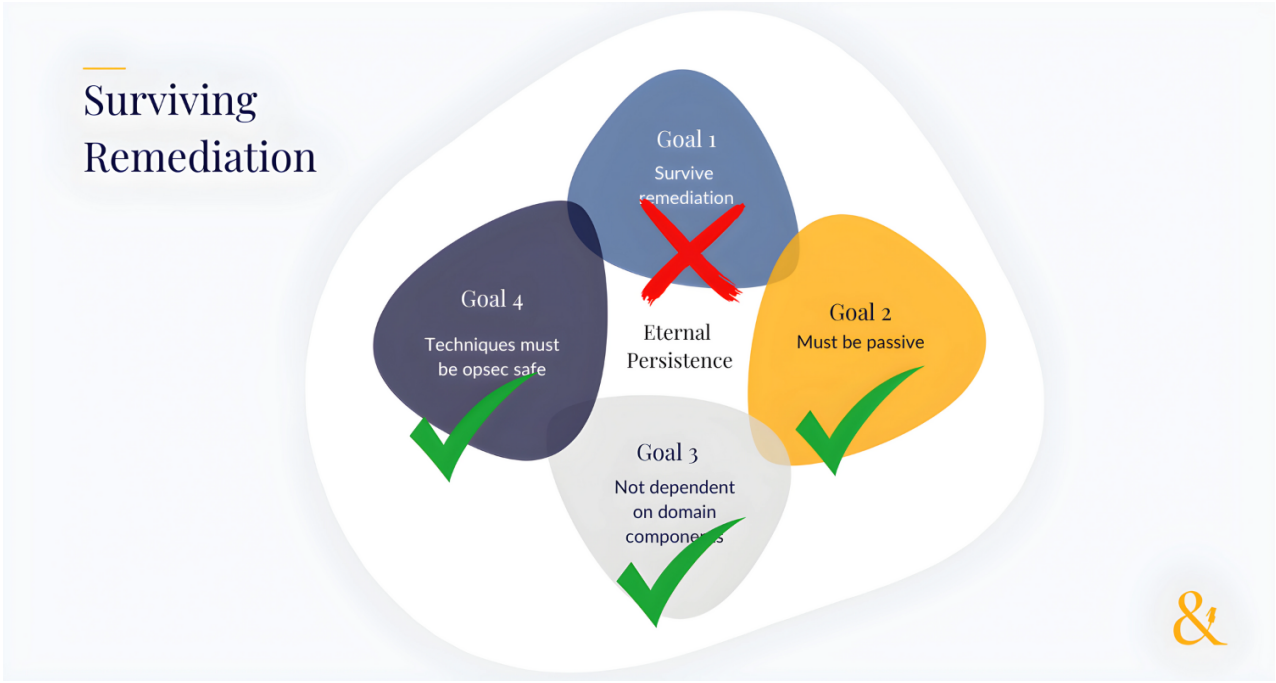


Figure 2. Ability to survive remediation according to specified criteria

In the next blog posts, we will go into detail about how different replication processes can be used to harvest credentials, increasing our level of persistence.

Creating test data

Configure Wireshark to decrypt the network traffic we need. For this, we need a keytab file with all relevant hashes. Copy all `RC4`, `AES128` and `AES256` hashes of the `krbtgt` user, domain admin accounts and all domain controller computer accounts. Use this tool to generate a keytab file.

In Wireshark, go to *Edit → Preferences → Protocols,* then:

- *TCP →* Enable *Reassemble out-of-order segments*
- *KRB5 → Try to decrypt Kerberos blobs* and specify the keytab you just created

Create a lab environment with:

- 2 domain controllers
- Workstation

Configure both domain controllers and workstation to be on the same network. Make sure you are able to tap into this network using Wireshark.
Start a new Wireshark capture and power on the domain controllers. During startup, the domain controllers will initiate a key exchange, which will be intercepted and decrypted by Wireshark for later use.

Next, `useldp.exe`, `dsa.msc` or the following PowerShell snippet to reset a user account.

```powershell
$ldapPath    = "LDAP://IP_OF_DC_THAT_IS_NOT_A_PDC"  # Replace with your LDAP server. Can be both FQDN or IP. Use IP when having DNS issues.
$username    = "test"                               # Replace with the username of the user whose password you want to reset
$newPassword = "Welcome01!"                         # Replace with the new password you want to set

# Account to initiate reset with
$sUsername = "Administrator"
$sPassword = "Welcome02!"

# Connect to the DC, find the user and set the password
$entry  = New-Object DirectoryServices.DirectoryEntry($ldapPath, $sUsername, $sPassword)
$search = New-Object DirectoryServices.DirectorySearcher($entry)
$search.Filter = "(&(objectCategory=user)(sAMAccountName=$username))"

$result = $search.FindOne()

if ($result -ne $null) {
    $userEntry = $result.GetDirectoryEntry()
    $userEntry.Invoke("SetPassword", @($newPassword))
    $userEntry.CommitChanges()

    Write-Host "Password reset successful."
}
```

After a few seconds, you should be able to see the password reset of the user account you selected.

References