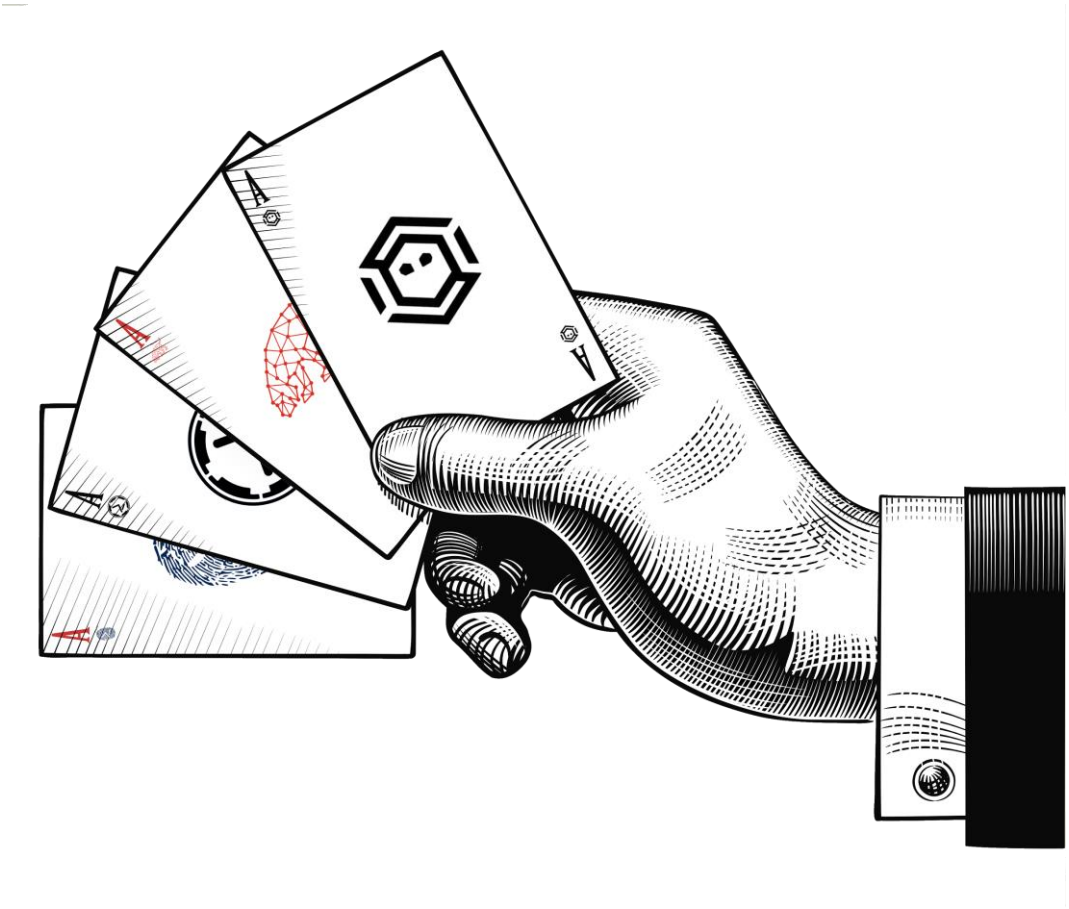


An **ACE** Up the Sleeve:

Designing Active Directory DACL Backdoors

Will Schroeder

Andy Robbins

Lee Christensen

Abstract

Active Directory (AD) object security descriptors are an untapped offensive landscape, often overlooked by attackers and defenders alike. While AD security descriptor misconfigurations can provide numerous paths that facilitate elevation of domain rights, they also present a unique chance to covertly deploy Active Directory persistence. It's often difficult to determine whether a specific AD security descriptor misconfiguration was set intentionally or implemented by accident. We present a taxonomy of control relationships that allow for specific node takeover, approaches for using BloodHound to help plan backdoor strategies, stealth primitives that include hiding discretionary access control list (DACL) enumeration rights and the existence of principals, and a series of backdoor case studies that chain multiple primitives for subtle domain persistence. "*If you can imagine it, it's likely already been done*" applies here- these backdoors have likely been deployed in environments for years without administrator knowledge. By bringing light to this persistence approach, we hope to raise awareness for both attackers and defenders alike of the persistence opportunities available through Active Directory security descriptor manipulation.

Introduction

With the increasing awareness of Golden[1] and Silver[2] Kerberos tickets, the industry has started to become aware of "malware-less" persistence techniques. That is, persistence strategies that don't involve code execution on systems in order to preserve future access to environments. While Golden and Silver Kerberos ticket attacks can provide persistence without any modifications or code execution in an environment, another avenue exists for facilitating Active Directory persistence. The security descriptor persistence approach **does** involve some type of modification to the environment; however, code execution is not required, and the changes will often survive operating system and domain functional level upgrades. This means that Active Directory security descriptor modifications provide an excellent opportunity for persistence in a domain with a minimal forensic footprint.

Active Directory objects are a class of securable object[3], meaning they contain a security descriptor[4]. In terms of persistence and privilege escalation in AD environments, we are particularly interested in analyzing the object owner and DACL fields of AD security descriptors.

Object owners can modify an object's DACL. A DACL is a list of Access Control Entries (ACEs) that mandate what principals (or "trustees") have what control rights over the object in question. Object owner and DACL control relationships can quickly explode in complexity for a modern domain, and combined with a lack of easy auditing opportunities mean that some type of security descriptor misconfiguration exists in most environments. The limited amount of previous work has mostly focused on the enumeration of these control relationships for domain privilege escalation; here we are covering the use of these misconfigurations for persistence purposes. This touches on another advantage of this approach- it's often difficult to tell if a security descriptor "misconfiguration" was implemented maliciously or implemented by accident. It goes without saying that in order to implement these changes, some type of domain elevated access is already needed, most commonly "Domain Admin" or equivalent access.

In order to build backdoors comprised of one or more chains of misconfigured security descriptors, this paper provides the proper technical background on object ownership, DACLs/ACEs, an easy way for domain-authenticated (but otherwise unprivileged) users to enumerate these ACEs, how to use BloodHound to map "normal" for an environment, a taxonomy of object takeover relationships, "stealth" primitives for use in backdoor chains, and several case studies that demonstrate an increasing complexity of security descriptor attack chains. We cap off with some defensive reflections and a look towards future research opportunities.

The control rights we're interested in are generally broken into three main categories: generic rights, standard/"control" rights that allow for taking control of an object itself, and object-specific rights that

[1] http://passing-the-hash.blogspot.com/2014/08/mimikatz-and-golden-tickets-whats-bfd.html
[2] https://adsecurity.org/?p=2011
[3] https://msdn.microsoft.com/en-us/library/windows/desktop/aa379557(v=vs.85).aspx
[4] https://msdn.microsoft.com/en-us/library/cc230366.aspx

apply in specific ways for nodes we care about. Generic rights include GenericAll and GenericWrite, which implicitly grant particular object-specific rights. The control rights we care about are WriteDacl and WriteOwner, which allow for the modification of the DACL and the owner of an object, respectively. Since the owner of an Active Directory object implicitly grants complete control of an object, ownership modification is a valuable object takeover primitive. Object-specific rights are also a valuable persistence primitive, but the specific rights vary based on the target AD object. The AD objects focused on in the scope of this whitepaper are users, groups, computers, containers (e.g. OUs and domain objects), and GPOs.

The main two stealth primitives developed during our research include hiding the security descriptor (including the DACL) and hiding the principal from existing privileged users. By modifying object ownerships and setting specific "read Deny" ACE entries on AD objects, we can complicate the retrieval of backdoored objects' security descriptors by privileged users such as members of "Domain Admins". While potentially still recoverable, this complicates defenders' abilities to find these purposely implemented backdoors. Likewise, we can modify permissions of the AD container holding the principal, preventing easy triage of the user (the trustee/principal) holding the malicious rights.

Throughout this paper, please keep this quote from Matt Graeber's 2015 BlackHat talk "*Abusing Windows Management Instrumentation (WMI) to Build a Persistent, Asynchronous, and Fileless Backdoor*[5]*"* in mind:

As an offensive researcher, if you can dream it, someone has likely already done it… and that someone isn't the kind of person who speaks at security cons.

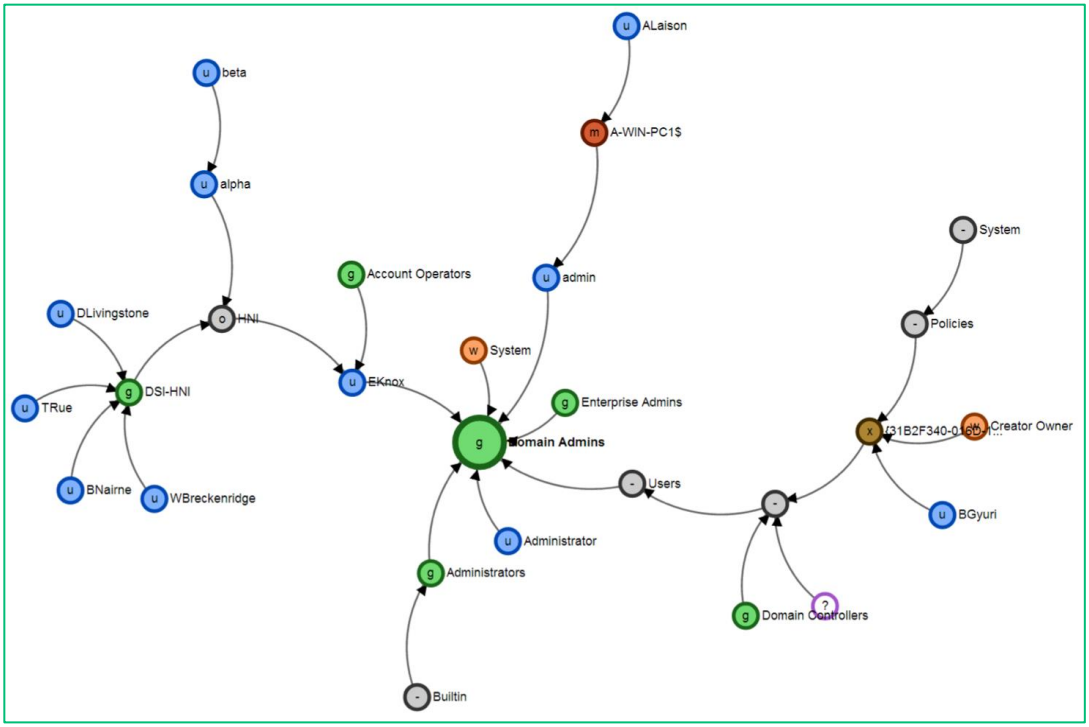
While we are not aware of any public examples of these types of backdoors being used "in the wild," we fully believe that we are not the first group to think of this idea. Our belief is that at least some advanced adversaries have likely been using security-descriptor-based persistence approaches for as long as access control has existed in Windows domains. These may have been in your environment for years.

[5] https://www.blackhat.com/docs/us-15/materials/us-15-Graeber-Abusing-Windows-Management-Instrumentation-WMI-To-Build-A-Persistent Asynchronous-And-Fileless-Backdoor-wp.pdf

Background

There is comparatively little existing security research concerning Active Directory security descriptors, and nearly none that examines the possibility of using them as a persistence approach. One of the first projects to ever cover these types of control relationships was the "*Chemins de contrôle en environnement Active Directory*[6]*"* (translated: "Control Paths in an Active Directory Environment") presentation at the 2014 Symposium on Information and Communications Technology Security (SSTIC) by Emmanuel Gras and Lucas Bouillot. Their whitepaper[7] (in French) breaks down many of the same control relationships we will cover here.

Their AD-control-paths[8] project grants one method for collecting, visualizing, and analyzing these control relationships. The project includes a collection of binaries to enumerate the ACEs of domain objects, as well a defined schema and Neo4j graph database approach for visualization:



Above: Figure 19 on page 71 of the "Chemins de contrôle en environnement Active Directory" whitepaper.

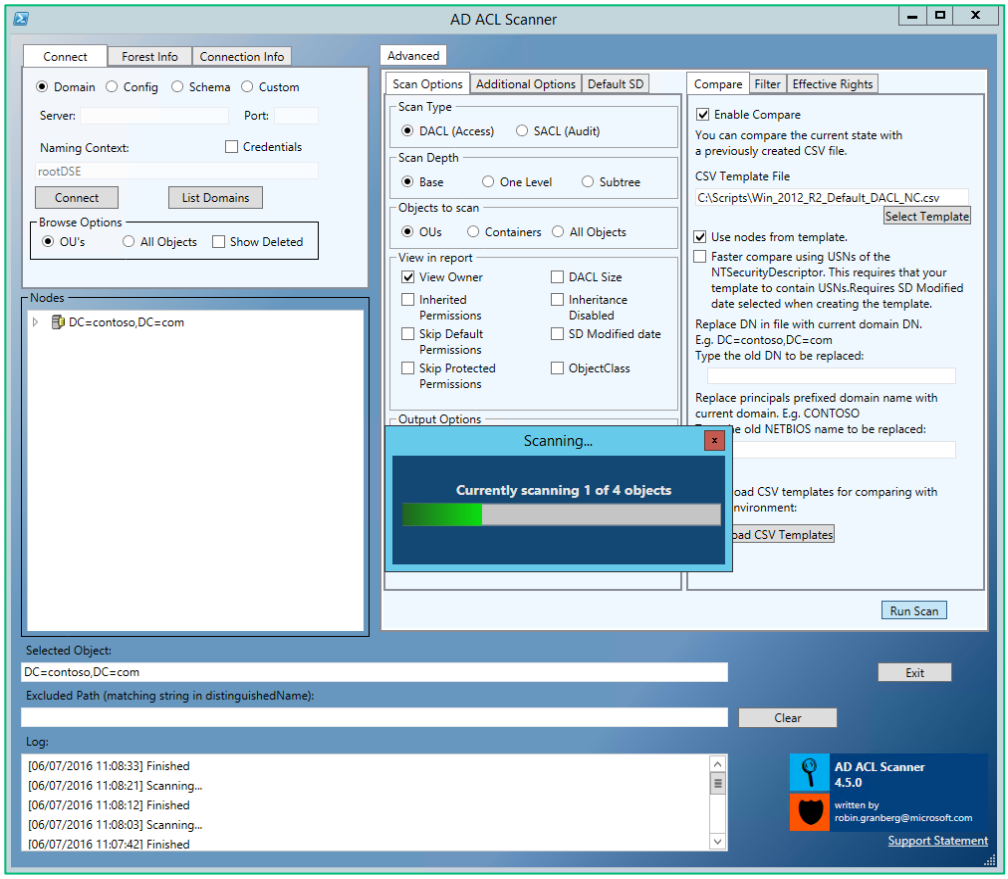
This 2014 work was groundbreaking, and remains one of the only public coverages of this subject area. However, it is not without some issues:

[6] https://www.sstic.org/2014/presentation/chemins_de_controle_active_directory/
[7] https://www.sstic.org/media/SSTIC2014/SSTIC-actes/chemins_de_controle_active_directory/SSTIC2014-Article-chemins_de_controle_active_directory-gras_bouillot.pdf
[8] https://github.com/ANSSI-FR/AD-control-paths

- It requires that a series of binaries be on disk during the collection process. From an offensive and (to an extent) a defensive perspective, our general philosophy is to remain off of disk as much as possible in order to minimize artifacts.

- As the project is based heavily on raw Neo4j and has several setup steps, there are issues with the usability of the project, which has potentially hindered adoption. While its technical capabilities are phenomenal, the use of the project is sometimes difficult for newcomers to grasp.

The second project that covers ACL auditing and analysis is "Forensics: Active Directory ACL investigation[9]" by Robin Granberg, as well as his associated "AD ACL Scanner[10]" toolset. The article covers many dangerous permissions in the "What kind of permissions are more of a risk than others?" section, which includes all of the specific control relationships we will cover in this paper. The AD ACL Scanner is a PowerShell-based GUI toolset that allows for easy enumeration of object rights:



Above: The AD ACL scanner interface enumerating DACLs in a domain.

AD ACL Scanner also includes a diff-ing capability where templates of standard DACL configurations for specific object types are used to help filter out standard settings and make it easier to find

[9] https://blogs.technet.microsoft.com/pfesweplat/2017/01/28/forensics-active-directory-acl-investigation/
[10] https://github.com/canix1/ADACLScanner

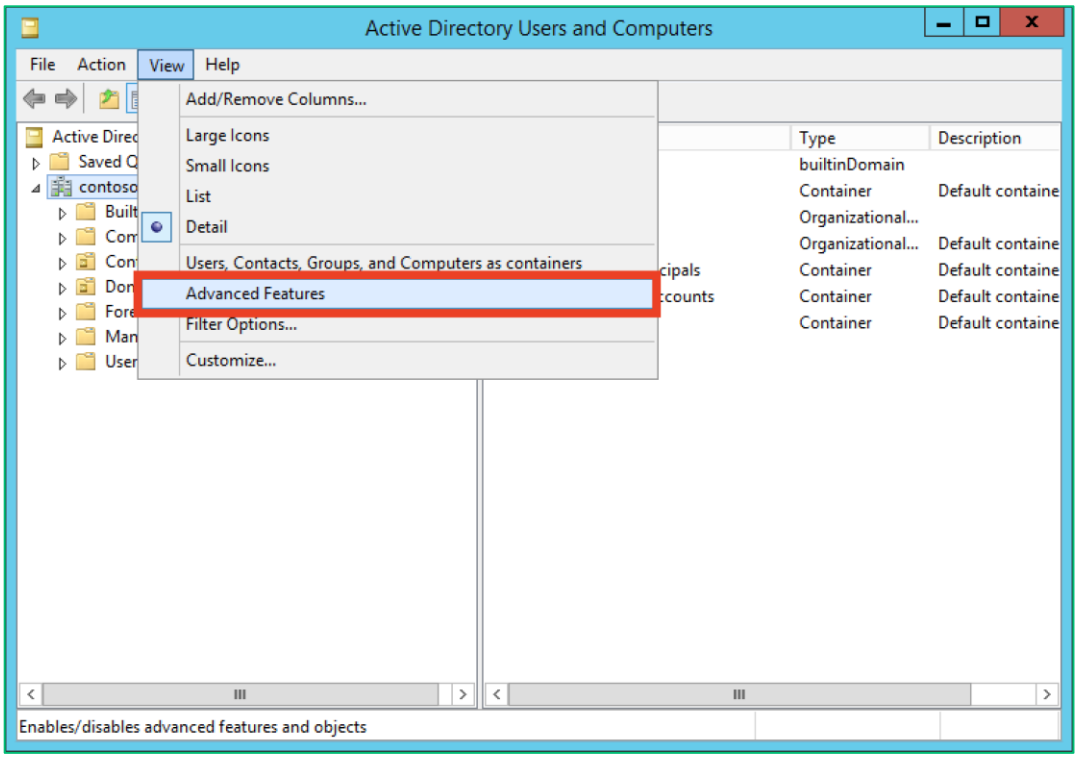
misconfigurations. This project appears to be fairly comprehensive and actively maintained; however, it misses out on the ability to visualize these control relationships in terms of risk or chains of misconfigurations. If we take the "Defenders think in lists. Attackers think in graphs"[11] philosophy as an approach to this problem, then a graph-based solution allows us to better visualize how these control relationships factor into attack chains.

Another project that deals with Active Directory auditing is the BTA project[12], which describes itself as an "an open-source Active Directory security audit framework," released by Philippe Biondi and Joffrey Czarny of the Airbus Group. The authors of the project presented at Black Hat Arsenal in 2015 with "ACTIVE DIRECTORY BACKDOORS: Myth or Reality."[13] The backdoors covered include domain admin membership, AdminSDHolder abuse, and show the start of a generalized auditing framework for Active Directory. It has since expanded to include things like "Who has extended rights (userForceChangePassword, SendAs, etc.)" and has some diffing capabilities as well to detect changes in points of time. However, as the project requires a ntds.dit Active Directory database to extract out the information needed, it is of less use offensively, while it remains a great defensive resource.

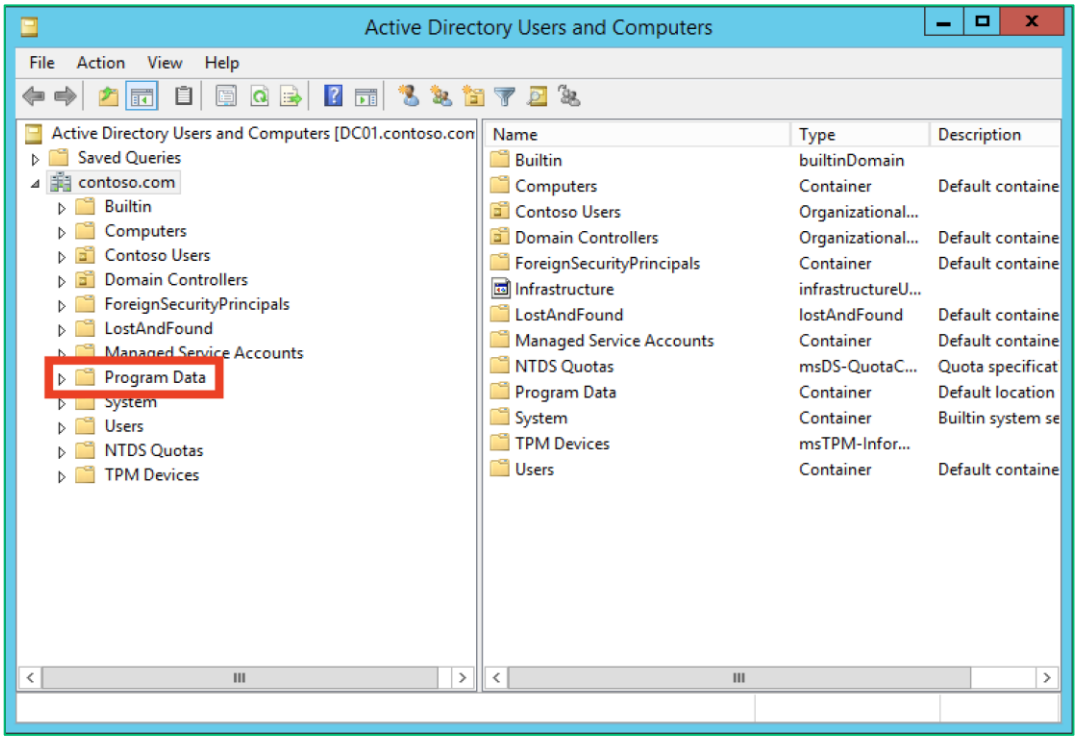
One of the only references we could find related to offensive use of AD ACLs was the 2010 Russian post "Бэкдор в active directory своими руками[14]" (roughly translated: "Backdoor in active directory with their own hands.") As none of the authors of this whitepaper speak Russian, we had to depend on Google Translate to examine the post. From what we can tell, the approach described a method for creating an invisible, privileged user within Active Directory. The methodology starts by creating a new domain user account with an account name that appears to be legitimate, or otherwise difficult to identify as malicious based on name only: "ExchangeLegacyReceiver." Then, the password on the account is set to never expire and the object owner is set to itself. Next, the user is put into a high privilege security group, with the given examples being either "Remote Desktop Users" or "Enterprise Admins". The stealth mechanism used by the Russian work is to place the user into a group that is not displayed by default within Active Directory Users and Computers (ADUC). Finally, the writer states that the privileged, backdoor account should be invisible both in ADUC and with ads.exe.

While the Russian blog post certainly sets a precedent for stealthy, agentless back-doors in Active Directory, several factors led us to conclude that the backdoor and stealth mechanisms could be further obscured. First, the post writer acknowledges that the user will appear as a member of the privileged account it is added to. The approach relies on the name of the account to dissuade investigation. We would rather the stealth mechanism rely on hiding the AD object, or a mechanism that makes the existence of the object less obvious. Second, the directory container which the writer proposes moving the object into is visible when selecting "Advanced Features" under view in ADUC:

[11] https://blogs.technet.microsoft.com/johnla/2015/04/26/defenders-think-in-lists-attackers-think-in-graphs-as-long-as-this-is-true-attackers-win/
[12] https://bitbucket.org/iwseclabs/bta/
[13] https://bitbucket.org/iwseclabs/bta/downloads/BH_Arsenal_US-15-bta.pdf
[14] https://habrahabr.ru/post/90990/



Before: The "Program Data" container is not visible. In ADUC, an administrator selects "Advanced Features" under "View".



After: The "Program Data" container is visible and its contents enumerable.

During the development of this material, Sean Metcalf released a post titled "*Scanning for Active Directory Privileges & Privileged Accounts*[15]" In the rather comprehensive post, Sean breaks down enumeration AD ACL information, as well as several of the control relationships covered in this whitepaper.

Securable Objects

A securable object is defined by Microsoft[16] as an object that can have a security descriptor. A security descriptor is a binary data structure that can vary in length and exact contents, but always contains, at a minimum, a header of control bits, the security identifier (SID) of the object owner, and the SID of the object's primary group. Most modern AD environments ignore the primary group section of the security descriptor. The security descriptor can also contain a discretionary access control list (DACL) and/or system access control list (SACL), though these are not technically required. [MS-ADTS][17] 6.1.3 outlines the requirements for AD object security descriptors, some of which we outline here.

```
typedef struct _SECURITY_DESCRIPTOR {
  UCHAR  Revision;
  UCHAR  Sbz1;
  SECURITY_DESCRIPTOR_CONTROL  Control;
  PSID  Owner;
  PSID  Group;
  PACL  Sacl;
  PACL  Dacl;
} SECURITY_DESCRIPTOR, *PISECURITY_DESCRIPTOR;
```

Above: The definition of the SECURITY_DESCRIPTOR structure[18].

Header Control Bits

The header control bits are defined by the 16-bit SECURITY_DESCRIPTOR_CONTROL[19] data type. These bits control various aspects of inheritance as well as other settings, and include two specific bits that are particularly interesting to us. The SE_DACL_PRESENT bit signals that a DACL is present in the security

[15] https://adsecurity.org/?p=3658
[16] https://msdn.microsoft.com/en-us/library/windows/desktop/aa379557(v=vs.85).aspx
[17] https://msdn.microsoft.com/en-us/library/cc223122.aspx
[18] https://msdn.microsoft.com/en-us/library/windows/hardware/ff556610(v=vs.85).aspx
[19] https://msdn.microsoft.com/en-us/library/windows/desktop/aa379566(v=vs.85).aspx

descriptor. As the documentation states, "*If this flag is not set, or if this flag is set and the DACL is NULL, the security descriptor allows full access to everyone.*"[20] The other bit of interest is the SE_DACL_PROTECTED bit (0x1000), which stops the included object DACL from being modified by inheritable ACEs. We theorized that this would have allowed for an interesting method of hiding effective access through inheritance manipulation, but tests proved unsuccessful. See the "Future Research" section for more information.

We also attempted to determine how to manually set these bits on AD objects, which would have had the same effect as a Null DACL; however, we were again unsuccessful. It appears that the Active Directory Service ignores some header control bits that clients specify when updating security descriptors via LDAP, which is an area for future research. It may be possible to manually set the bits through functions executed on the domain controller, or through raw editing of the NTDS.dit AD database. For a complete breakdown of all control bit functions, please refer to the Microsoft documentation[21].

Object Ownership

All AD security descriptors must have an owner specified as a security identifier (SID). Active Directory implicitly grants object owners WriteDacl and RIGHT_READ_CONTROL[22], granting the owner full control of the security descriptor of the object. As such, attackers wanting to gain access to an object can do so by compromising the target object's owner or by compromising anyone who can grant ownership to the target object (e.g. principals with the WriteDacl/WriteOwner rights, or SeTakeOwnership and SeRestorePrivilege privileges).

ACLs, DACLs, and SACLs

When most documentation refers to the term "Access Control List" (ACL), it is referring to the discretionary access control list (DACL) and the system access control list (SACL) of a particular object's security descriptor. An object's DACL and SACL are both collections of access control entries (ACEs).

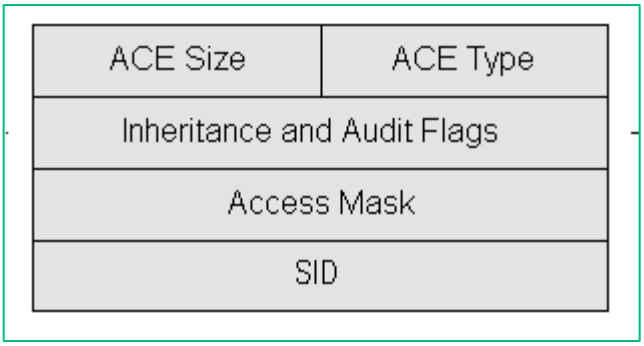
The SACL is used to "*specify the types of access attempts that generate audit records in the security event log of a domain controller*.[23]" While SACLs have great defensive potential, they are outside of the scope of this paper. The ACEs in an object's DACL define what security principals (also sometimes called a "trustee") have what rights on the target object, and are what our work focuses on. Of note, there is a difference between a Null DACL and an empty DACL. A Null DACL effectively grants all rights to all users, while a DACL that is present but doesn't contain any ACE entries denies rights to all users. NULL DACLs are not allowed in Active Directory per the spec (see MS-ADTS 6.1.3[24]).

[20] https://msdn.microsoft.com/en-us/library/windows/desktop/aa379566(v=vs.85).aspx
[21] https://msdn.microsoft.com/en-us/library/windows/desktop/aa379566(v=vs.85).aspx
[22] [MS-ADTS] 5.1.3.3.1 "Null vs Empty DACLs". https://msdn.microsoft.com/en-us/library/cc223515.aspx
[23] https://msdn.microsoft.com/en-us/library/ms677926(v=vs.85).aspx
[24] https://msdn.microsoft.com/en-us/library/cc223731.aspx

ACEs

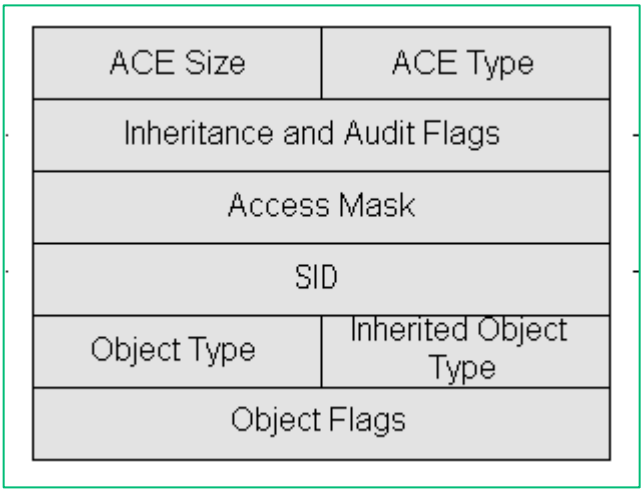
All ACEs include a 32-bit set of flags that control inheritance and auditing, an access mask that specifies the rights allowed on the object, and a security identifier (SID) that identifies the principal/trustee that has the specified rights. There are two types of ACEs: generic ACEs and object-specific ACEs. Object-specific ACEs allow for more granular control of inheritance specifics and the object-specific access rights that can't be captured with the generic right specification. Object-specific ACEs are applicable only to AD objects.

A generic ACE's structure is as follows:



Above: The layout of a generic ACE[25].

While an object-specific ACE's structure is:



Above: The layout of an object-specific ACE[26].

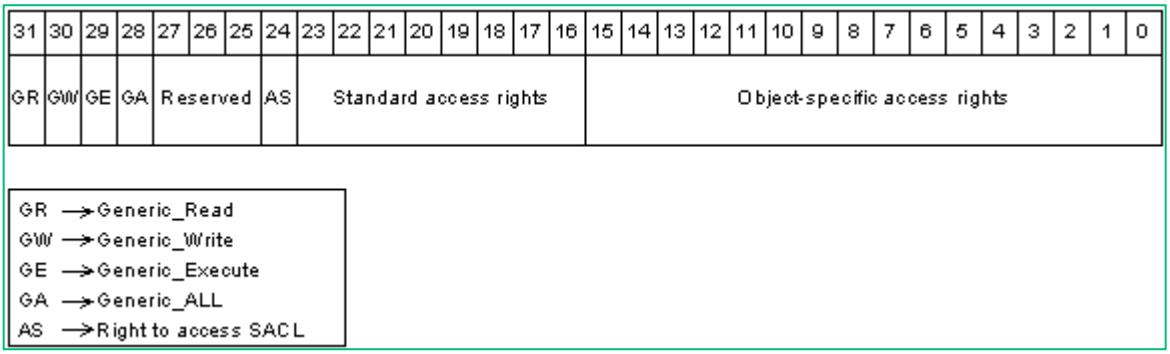
[25] http://searchwindowsserver.techtarget.com/feature/The-structure-of-an-ACE
[26] http://searchwindowsserver.techtarget.com/feature/The-structure-of-an-ACE

The “Object Type” contains a GUID that can be interpreted in one of three ways:

1. **A child object type**: if the access mask grants the RIGHT_DS_CREATE_CHILD right, the ObjectType GUID specifies the type of child object that can be created in the container. A GUID of all 0s or a lack of a GUID indicates that all child object types can be created.
2. **A property/property set**: If RIGHT_DS_READ_PROP or RIGHT_DS_WRITE_PROP are specified in the access mask, or if the property is a ‘confidential’ property and the RIGHT_DS_CONTROL_ACCESS is set, the ObjectType GUID refers to a specific property (or property set) that the principal is allowed to read/write, respectively.
3. **A specific extended right**: If DS_CONTROL_ACCESS is set, and the ObjectType GUID maps to an extended right registered in the current forest schema, then that specific extended right (such as force-resetting a user’s password) is granted.

The Access Mask

The access mask is the 32-bit component of the ACE that actually defines the rights on the object that the specified principal SID is granted. The generic structure for an access mask is:



Above: The bit breakout of the general case access mask[27].

Here’s what the access mask for an Active Directory object looks like (X == Reserved bits):

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
GR	GW	GX	GA	X	X	X	X	X	X	X	X	WO	WD	RC	DE	X	X	X	X	X	X	X	CR	LO	DT	WP	RP	VW	LC	DC	CC

Above: The bit breakout of the Active Directory object-specific access mask[28].

[27] https://msdn.microsoft.com/en-us/library/windows/desktop/aa374896(v=vs.85).aspx
[28] https://msdn.microsoft.com/en-us/library/cc223511.aspx

For Active Directory objects, here are how generic AD rights are interpreted (section 5.1.3.2 of the [MS-ADTS]: Active Directory Technical Specification[29]):

RIGHT_GENERIC_READ (GR, 0x80000000)	The right to read permissions and all properties of the object, and list the contents of the object in the case of containers. Equivalent to: RIGHT_READ_CONTROL | RIGHT_DS_LIST_CONTENTS | RIGHT_DS_READ_PROPERTY | RIGHT_DS_LIST_OBJECT Referred to as **GenericRead** elsewhere in this paper.
RIGHT_GENERIC_WRITE (GW, 0x40000000)	Includes the right to read permissions on the object, and the right to write all the properties on the object. Equivalent to: RIGHT_READ_CONTROL | RIGHT_DS_WRITE_PROPERTY | RIGHT_DS_WRITE_PROPERTY_EXTENDED Referred to as **GenericWrite** elsewhere in this paper.
RIGHT_GENERIC_EXECUTE (GX, 0x20000000)	The right to read permissions/list the contents of a container object. Equivalent to: RIGHT_READ_CONTROL | RIGHT_DS_LIST_CONTENTS Referred to as **GenericExecute** elsewhere in this paper.
RIGHT_GENERIC_ALL (GA, 0x10000000)	The right to create/delete child objects, read/write all properties, see any child objects, add and remove the object, and read/write with an extended right. Equivalent to: RIGHT_DELETE | RIGHT_READ_CONTROL | RIGHT_WRITE_DAC | |RIGHT_WRITE_OWNER | RIGHT_DS_CREATE_CHILD

[29] https://msdn.microsoft.com/en-us/library/cc223511.aspx

	| RIGHT_DS_DELETE_CHILD | RIGHT_DS_DELETE_TREE | RIGHT_DS_READ_PROPERTY | RIGHT_DS_WRITE_PROPERTY | RIGHT_DS_LIST_CONTENTS | RIGHT_DS_LIST_OBJECT | RIGHT_DS_CONTROL_ACCESS | RIGHT_DS_WRITE_PROPERTY_EXTENDED) Referred to as **GenericAll** elsewhere in this paper.

The bits of the 'standard access' section are interpreted as:

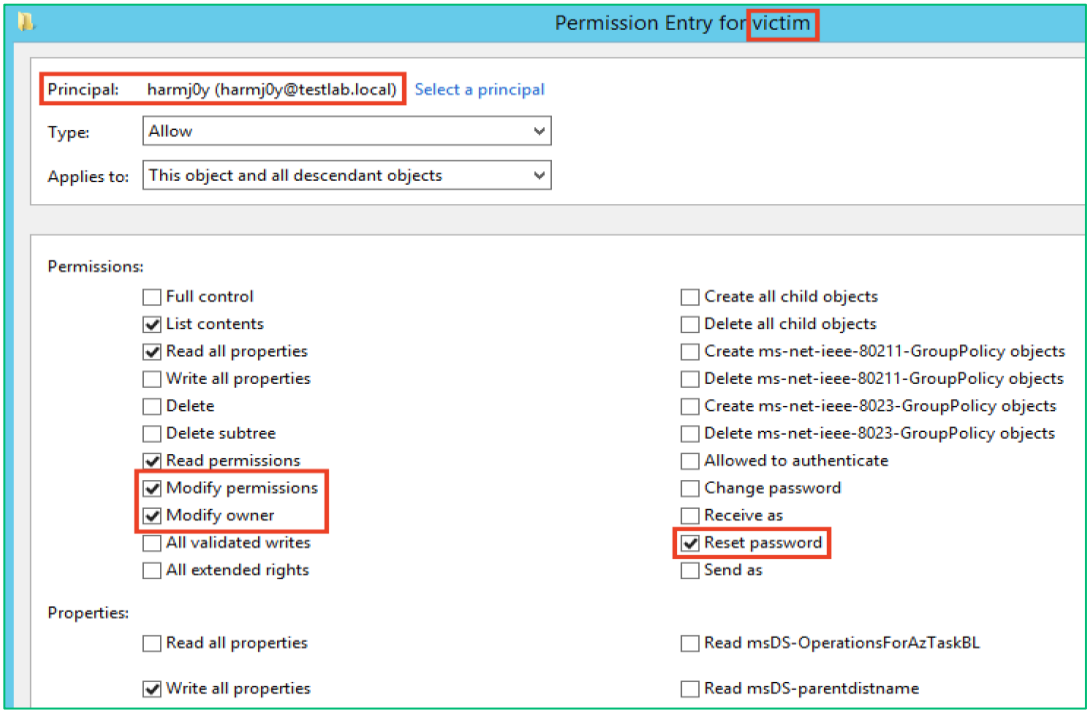
RIGHT_WRITE_OWNER (WO, 0x00080000)	The right to modify the owner section of the security descriptor. Of note, a user with this right can only change the owner to themselves - ownership cannot be transferred to other users with only this right. Referred to as **WriteOwner** elsewhere in this paper.
RIGHT_WRITE_DAC (WD, 0x00040000)	The right to modify the DACL for the object. Referred to as **WriteDacl** elsewhere in this paper.
RIGHT_READ_CONTROL (RC, 0x00020000)	The right to read all data from the security descriptor except the SACL. Referred to as **ReadControl** elsewhere in this paper.
RIGHT_DELETE (DE, 0x00010000)	The right to delete the object. Referred to as **Delete** elsewhere in this paper.

The final object-specific access masks bits are interpreted as:

RIGHT_DS_CONTROL_ACCESS (CR, 0x00000100)	A specific control access right (if the ObjectType GUID refers to an extended right registered in the forest schema) or the right to read a confidential property (if the ObjectType GUID refers to a confidential property). If the GUID is not present,

	then all extended rights are granted.
RIGHT_DS_LIST_OBJECT (LO, 0x00000080)	The right to list an object. If the user does not have this right and also does not have the RIGHT_DS_LIST_CONTENTS right on the object's parent container then the object is hidden from the user.
RIGHT_DS_DELETE_TREE (DT, 0x00000040)	The right to perform a delete-tree operation.
RIGHT_DS_WRITE_PROPERTY (WP, 0x00000020)	The right to write one or more properties of the object specified by the ObjectType GUID. If the ObjectType GUID is not present or is all 0s, then the right to write all properties is granted.
RIGHT_DS_READ_PROPERTY (RP, 0x00000010)	The right to read one or more properties of the object specified by the ObjectType GUID. If the ObjectType GUID is not present or is all 0s, then the right to read all properties is granted.
RIGHT_DS_WRITE_PROPERTY_EXTENDED (VW, 0x00000008)	The right to execute a validated write access right.
RIGHT_DS_LIST_CONTENTS (LC, 0x00000004)	The right to list all child objects of the object, if the object is a type of container.
RIGHT_DS_DELETE_CHILD (DC, 0x00000002)	The right to delete child objects of the object, if the object is a type of container. If the ObjectType contains a GUID, the GUID will reference the type of child object that can be deleted.
RIGHT_DS_CREATE_CHILD (CC, 0x00000001)	The right to create child objects under the object, if the object is a type of container. If the ObjectType contains a GUID, the GUID will reference the type of child object that can be created.

Here's what these rights actually look like in GUI form through Active Directory Users and Computers (ADUC):



Above: The graphical view of an ACE being interpreted in Active Directory Users and Computers (ADUC).

DS_CONTROL_ACCESS

The RIGHT_DS_CONTROL_ACCESS bit of the access mask is a special case that can be interpreted in a few different ways. If the GUID specified in the ObjectType maps to an extended right registered in the forest schema, then the specific extended right (sometimes called a 'control access right') is granted. This approach is taken so more granular rights can be expanded in future domain schemas, and actions that don't exactly map to the reading/writing of specific properties can be granted. For example, the User-Change-Password[30] right (GUID: ab721a53-1e2f-11d0-9819-00aa0040529b) enables a user to change his/her own password if the previous password value is known, versus the User-Force-Change-Password[31] right (GUID: 00299570-246d-11d0-a768-00aa006e0529) which permits the forceful reset of a user's password without knowing the previous value.

The ObjectType GUID can also refer to a property or property set. In this case, if the property or property set identified by the GUID is marked as confidential[32], then the DS_CONTROL_ACCESS grants the ability to read the attribute (RIGHT_DS_READ_PROPERTY does not grant access to confidential

[30] https://msdn.microsoft.com/en-us/library/ms684413(v=vs.85).aspx
[31] https://msdn.microsoft.com/en-us/library/ms684414(v=vs.85).aspx
[32] https://msdn.microsoft.com/en-us/library/cc223153.aspx

attributes). For example, this is the case with the Local Administrator Password Solution (LAPS)[33], which extends the forest schema to include the ms-Mcs-AdmPwd and ms-mcs-AdmPwdExpirationTime properties. The ms-Mcs-AdmPwd property is marked as confidential and stores the plaintext of the randomized local administrator password for the machine represented by the computer object. So, in order to enumerate users who have read access to this attribute, we search for ACE entries with DS_CONTROL_ACCESS flipped and the ObjectType GUID referring to the randomized GUID pointing to ms-Mcs-AdmPwd that is specific to the environment.

The Security Reference Monitor

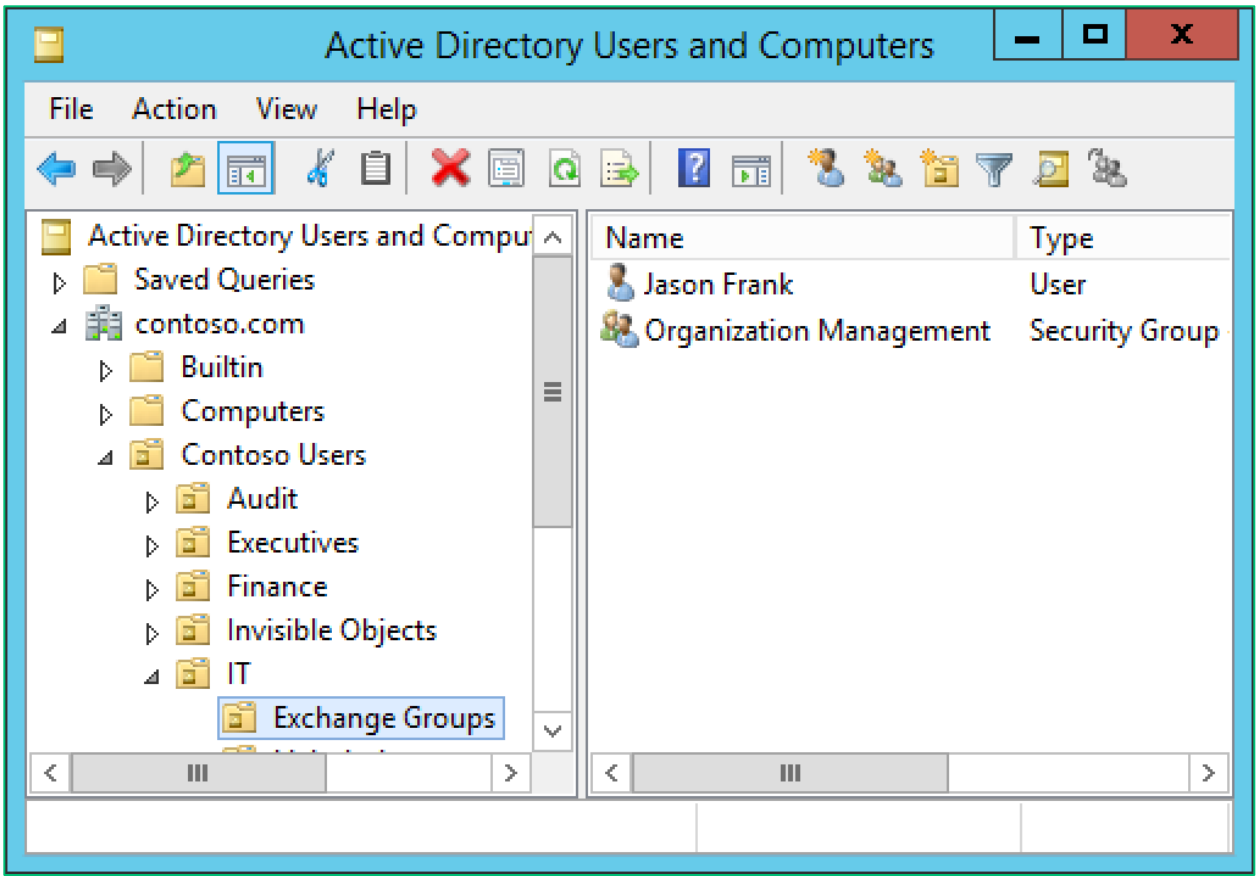
In Windows and Active Directory, access requests decisions are made by the Kernel-Mode Security Reference Monitor (SRM). Since Windows 2000, the SRM has supported object-based access control lists, property and property-set based read/write/modify privileges, and the logic for evaluating canonically-ordered ACEs when making access decisions (i.e., allow or deny a requested privilege). For example, consider the "Domain Admins" group, which will have the "Full Control" access privilege by default to every securable Active Directory object. When a member of the "Domain Admins" group requests the ability to change a user password, the SRM must decide whether that request should be granted or not. The SRM evaluates the DACL on the target user, determines that the "Domain Admins" group (and in turn, members of this group) has full control of the user, and then allows the change password process to continue.

When evaluating an object's DACL, the SRM will read the ACEs in canonical order, which orders ACEs as follows:

1. Explicitly defined DENY ACEs.
2. Explicitly defined ALLOW ACEs.
3. Inherited DENY ACEs.
4. Inherited ALLOW ACEs.

Inherited ACEs, which are by far the most common ACEs we've encountered, are further complicated by being evaluated based on generational degrees of separation from the affected object. Consider the following OU tree structure:

[33] https://www.microsoft.com/en-us/download/details.aspx?id=46899



Above: The "contoso.com" domain contains the OU "Contoso Users", which contains the OU "IT", which contains the OU "Exchange Groups", which contains the user "Jason Frank".

The **Jason Frank** user may inherit ACEs from the objects above it, including the domain object and the user's parent, grandparent, and great grandparent OUs. Because ACEs inherited from generationally closer objects are given precedence, ACEs inherited from the "Exchange Groups" OU will effectively override ACEs inherited from the "IT" OU, even if this means a conflicting **ALLOW** ACE takes precedence over a **DENY** ACE.

Through understanding the order of evaluation the SRM uses for these access decisions, an attacker may more effectively hide malicious ACEs or even entire security principals from defenders. We discuss the implications of the SRM's evaluation of ACEs and its impact on an attacker's ability to more effectively hide objects in Active Directory in the section of this paper titled "Stealth Primitive - Hiding the Principal".

Security Descriptor Enumeration

During the backdoor planning and execution process, it's essential to have a way to easily and programmatically enumerate the security descriptors of various objects, as well as the object owner.

Luckily, one of the .NET classes to execute LDAP searches, DirectorySearcher[34], contains a property named SecurityMasks[35], which allows us to enumerate parts of the ntSecurityDescriptor property of an Active Directory object, even as a non-privileged user. Here are the available values for SecurityMasks:

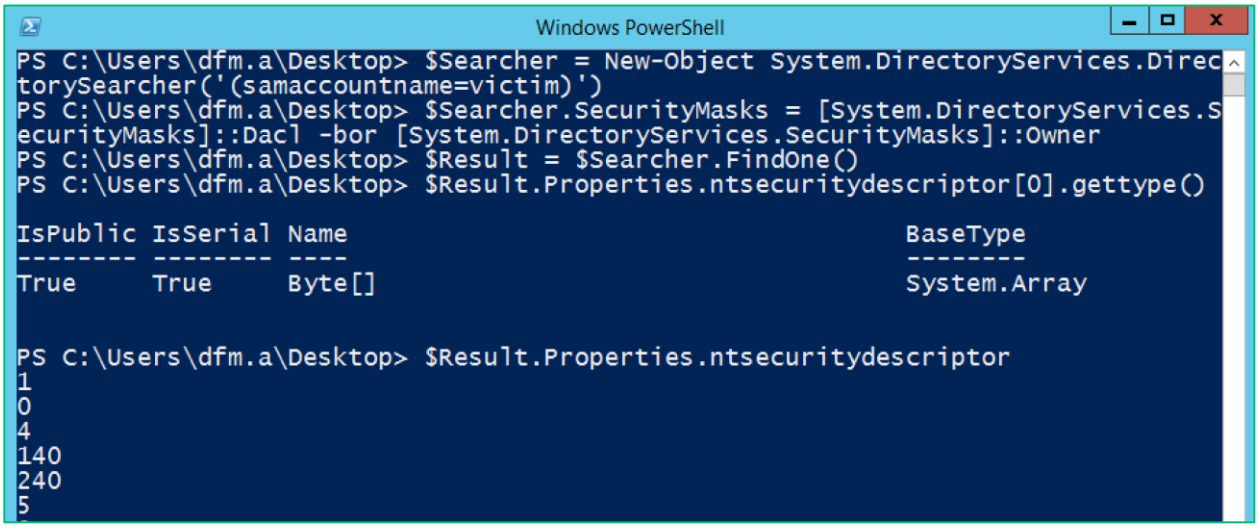
Dacl	Return the discretionary access control list (DACL).
Group	Return primary group data, not applicable for modern domains.
None	Don't return any security data (default behavior).
Owner	Return the security identifier (SID) of the object owner.
Sacl	Return the system access-control list (SACL).

These values can be combined to return multiple parts of the security descriptor. For example, here is how to retrieve the DACL and Owner fields from objects returned by an LDAP search in C#[36]:

```
using System.DirectoryServices;
...
DirectorySearcher src = new DirectorySearcher("…");
src.PropertiesToLoad = new string[] {ntSecurityDescriptor,…};
src.SecurityMasks = SecurityMasks.Dacl | SecurityMasks.Owner;
SearchResultCollection res = src.FindAll();
```

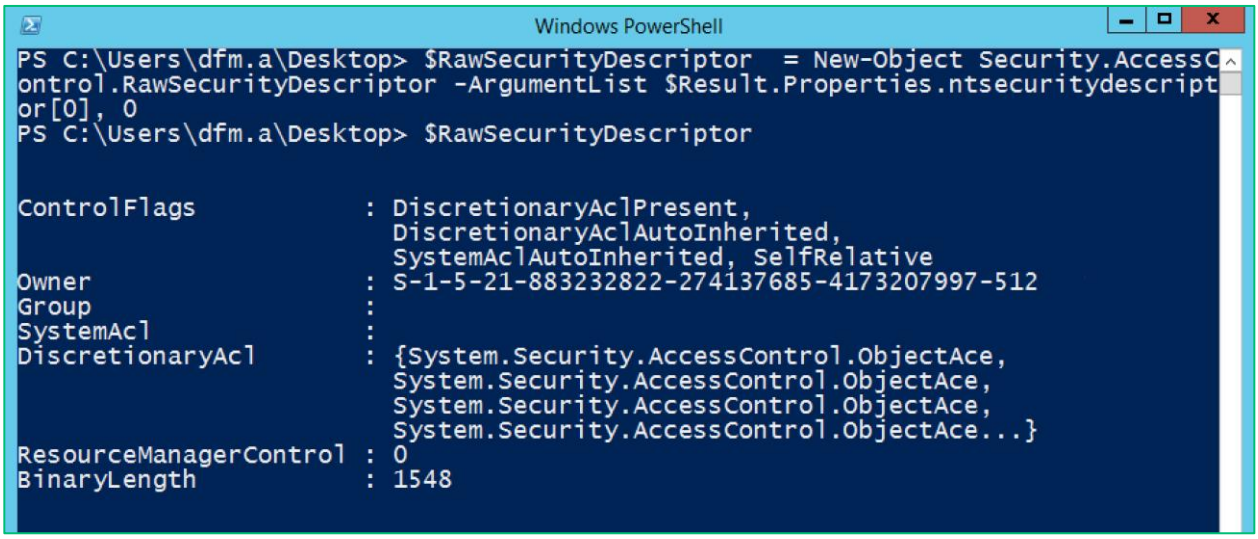
[34] https://msdn.microsoft.com/en-us/library/system.directoryservices.directorysearcher(v=vs.110).aspx
[35] https://msdn.microsoft.com/en-us/library/system.directoryservices.directorysearcher.securitymasks(v=vs.110).aspx
[36] https://msdn.microsoft.com/en-us/library/system.directoryservices.directorysearcher.securitymasks(v=vs.110).aspx

And here is how to enumerate the same information in PowerShell:



Above: PowerShell retrieval of the security descriptor binary blob for the "victim" user, containing the owner and DACL information for the object.

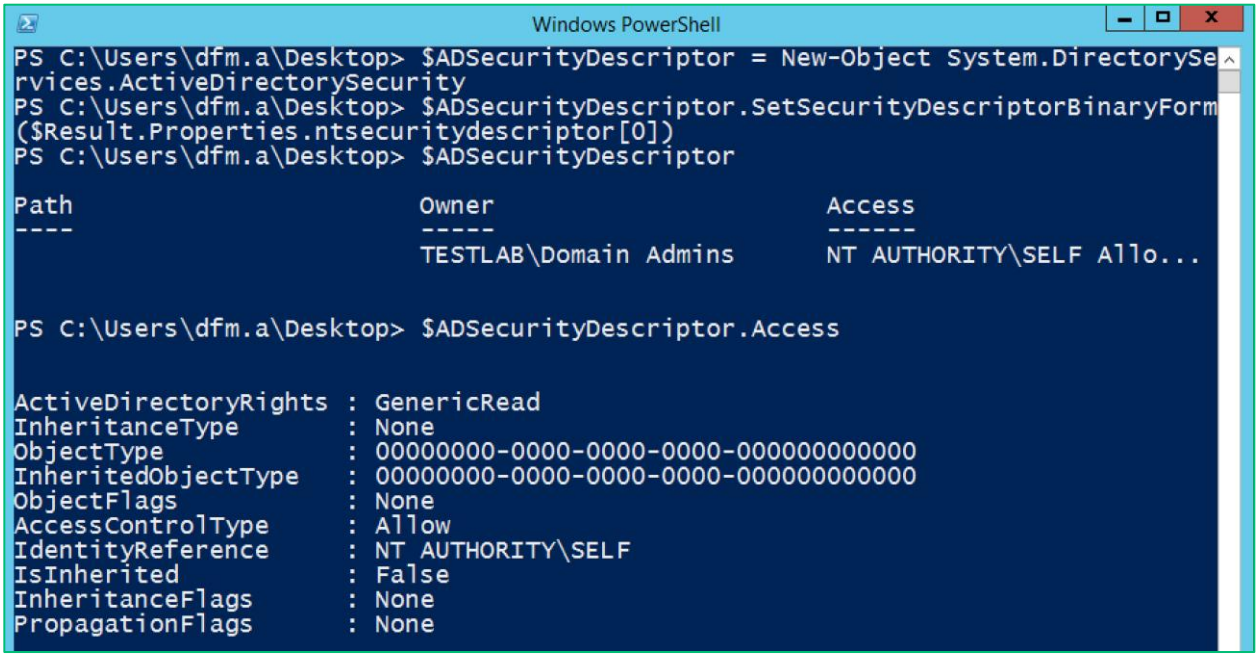
As you can see above, the security descriptor information will be returned as a binary blob stored in the **ntSecurityDescriptor** property of the resulting object. This information needs to be parsed into a human-readable form, which can be done one of two ways. One option is to create a new Security.AccessControl.RawSecurityDescriptor[37]:



Above: PowerShell parsing of the security descriptor binary blob for the "victim" user into a 'RawSecurityDescriptor'.

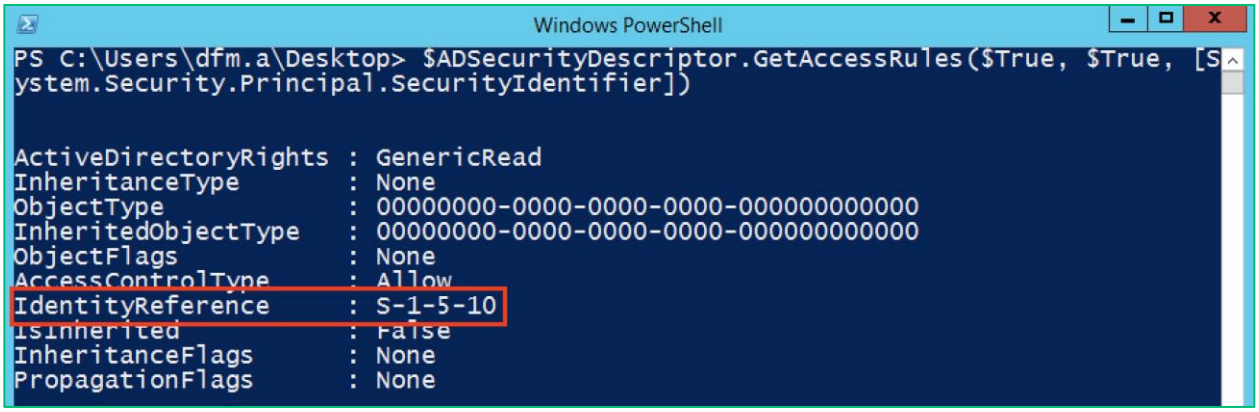
[37] https://msdn.microsoft.com/en-us/library/system.security.accesscontrol.rawsecuritydescriptor(v=vs.110).aspx

The alternative method is to use the more specific System.DirectoryServices.ActiveDirectorySecurity[38] class:



Above: PowerShell parsing of the security descriptor binary blob for the "victim" user into an 'ActiveDirectorySecurity' object'.

With this second approach, you can specify whether to return the security identifier or a resolved short name for the security principal/trustee listed in each ACE. This is accomplished with a specific call to the GetAccessRules() method[39]:



Above: Retrieving ACEs for the specified object through the GetAccessRules() method, specifying that security identifiers (SIDs) should be returned for the identity reference.

[38] https://msdn.microsoft.com/en-us/library/system.directoryservices.activedirectorysecurity(v=vs.110).aspx
[39] https://msdn.microsoft.com/en-us/library/system.security.accesscontrol.directoryobjectsecurity.getaccessrules(v=vs.110).aspx

PowerView[40] is a PowerShell 2.0-compatible toolkit written by one of the whitepaper authors that facilitates various domain/network reconnaissance tasks, and is a part of the PowerSploit project[41]. More information about PowerView can be found in a series of posts[42] written by its author. We will use PowerView throughout this paper to demonstrate security descriptor enumeration and backdoor weaponization. For DACL enumeration, PowerView's **Get-DomainObjectACL** function will return the ACEs for a particular object. The optional **ResolveGUIDs** flag will first perform a mapping of all right GUIDs currently registered in the domain and will translate all ObjectType GUIDs in resulting ACEs to their human-readable form.

```
PS C:\Users\dfm.a\Desktop> Get-DomainObjectAcl -Identity harmj0y -ResolveGUIDs |
 ? {$_.SecurityIdentifier -match $(ConvertTo-SID eviluser)}


AceQualifier           : AccessAllowed
ObjectDN               : CN=harmj0y,CN=Users,DC=testlab,DC=local
ActiveDirectoryRights  : WriteProperty
ObjectAceType          : Script-Path
ObjectSID              : S-1-5-21-883232822-274137685-4173207997-1111
InheritanceFlags       : None
BinaryLength           : 56
AceType                : AccessAllowedObject
ObjectAceFlags         : ObjectAceTypePresent
IsCallback             : False
PropagationFlags       : None
SecurityIdentifier     : S-1-5-21-883232822-274137685-4173207997-1115
AccessMask             : 32
AuditFlags             : None
IsInherited            : False
AceFlags               : None
InheritedObjectAceType : All
OpaqueLength           : 0
```

Above: Using PowerView's Get-DomainObjectAcl function to retrieve the ACEs for the harmj0y user with GUIDs for the ObjectAceType translated.

Of note, any domain authenticated user can enumerate the Owner and DACL of most objects in a default domain. While this is not relevant for our assumed case of elevated access, it is useful when enumerating control relationships for domain escalation.

[40] https://github.com/PowerShellMafia/PowerSploit/blob/dev/Recon/PowerView.ps1
[41] https://github.com/PowerShellMafia/PowerSploit
[42] http://www.harmj0y.net/blog/tag/powerview/

DACL (Mis)configurations

Our work was first motivated by a desire to find security descriptor misconfigurations to facilitate domain escalation. This was the original intent for ACL-based ingestion into the BloodHound[43] analysis platform. As we began to realize the potential for offensive backdooring, we realized that the same control relationships that we search for from a domain escalation perspective could form the building blocks of security descriptor backdoors. Of note, again, is that an attacker must to have the ability to modify parts of the security descriptor of the objects to which the backdoors are added. Gaining the required rights usually means needing some type of elevated access, often "Domain Admins" or something equivalent.

Here, we present our control relationship and object takeover taxonomy. Some of the relationships we care about vary based on the target object we're trying to compromise. We will start this breakdown first with rights that apply to all target object types (generic and control rights) and then will break out each object type we're targeting with the additional appropriate control relationships highlighted. Also, note that this breakdown *is not complete*: we are sure that there are additional takeover relationships, and are not claiming this collection to be a canonical reference at this point.

AD Generic Rights

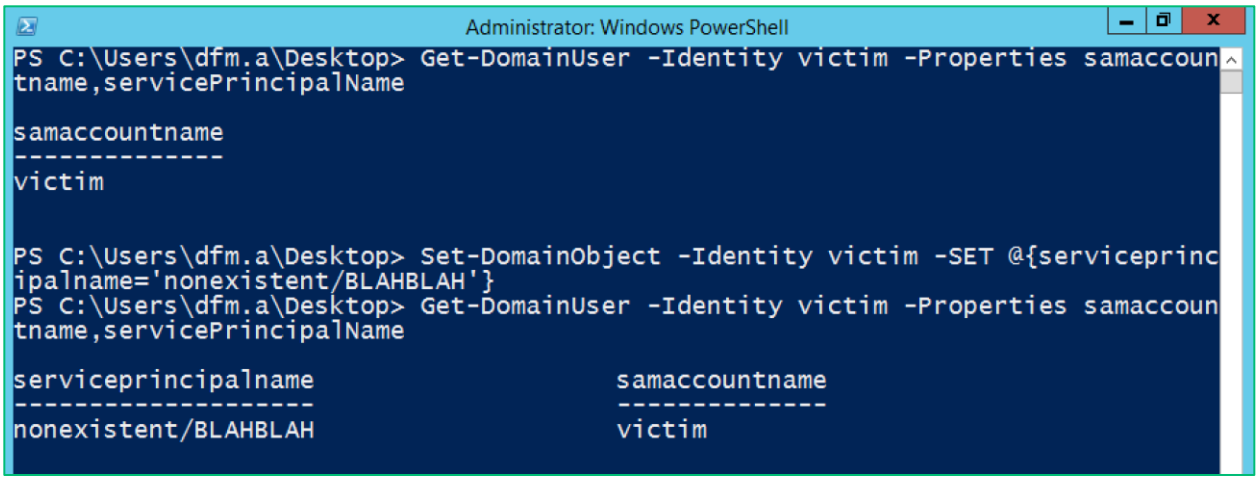
GenericAll (fully defined in the "Access Mask" section) grants complete control over a target object, including the "control rights" outlined below. This includes the **WriteDacl** and **WriteOwner** privileges, as well as any specific extended rights.

GenericWrite (also fully defined in the "Access Mask" section) is a combination of RIGHT_READ_CONTROL (the right to read the DACL), RIGHT_DS_WRITE_PROPERTY, and RIGHT_DS_WRITE_PROPERTY_EXTENDED applied to all properties.

WriteProperty with an ObjectType that doesn't contain a GUID also means that the principal has the right to modify all properties. While this case is technically not the equivalent of GenericWrite, in practice they are basically equivalent in most cases.

These generic rights can be abused with PowerView's **Set-DomainObject**. The **-Set** parameter allows for field modification (-Set @{"property1"="value1";"property2"="value2"}) while the **-Clear** parameter will clear a property's value. Here's an example of setting a target user's servicePrincipalName, Kerberoasting the account, and resetting the servicePrincipalName:

[43] https://github.com/BloodHoundAD/



Above: Using PowerView's Set-DomainObject function to manipulate the servicePrincipalName property of the 'victim' user.

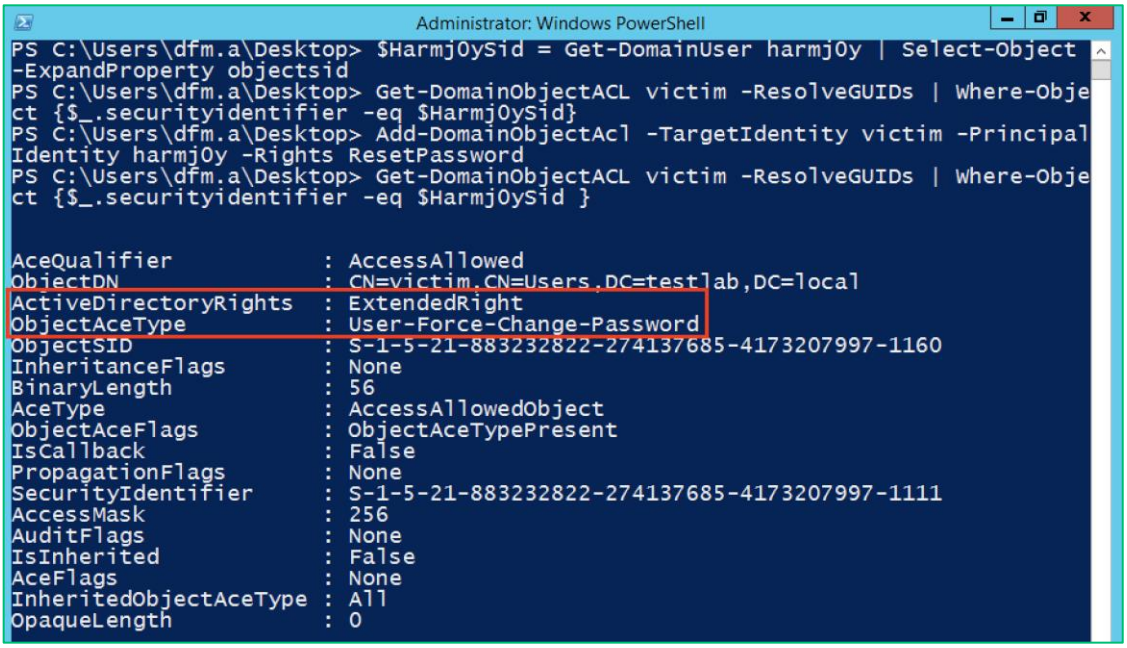
AD Control Rights

Two specific rights allow a principal to gain modify the security descriptor of a target object, even though the rights don't allow for any property modification or extended right access. These rights require a modification to the target object's security descriptor before performing the next step of the intended chain (e.g. the attacker would have to add ACEs to allow property modification or the execution of a specific extended right).

WriteDacl (formally RIGHT_WRITE_DAC) allows the principal to modify the DACL of the affected object. This means that an attacker can add or remove specific access control entries, allowing for them to grant themselves complete access to the object. Thus, WriteDacl is a right that enables additional rights in the chain.

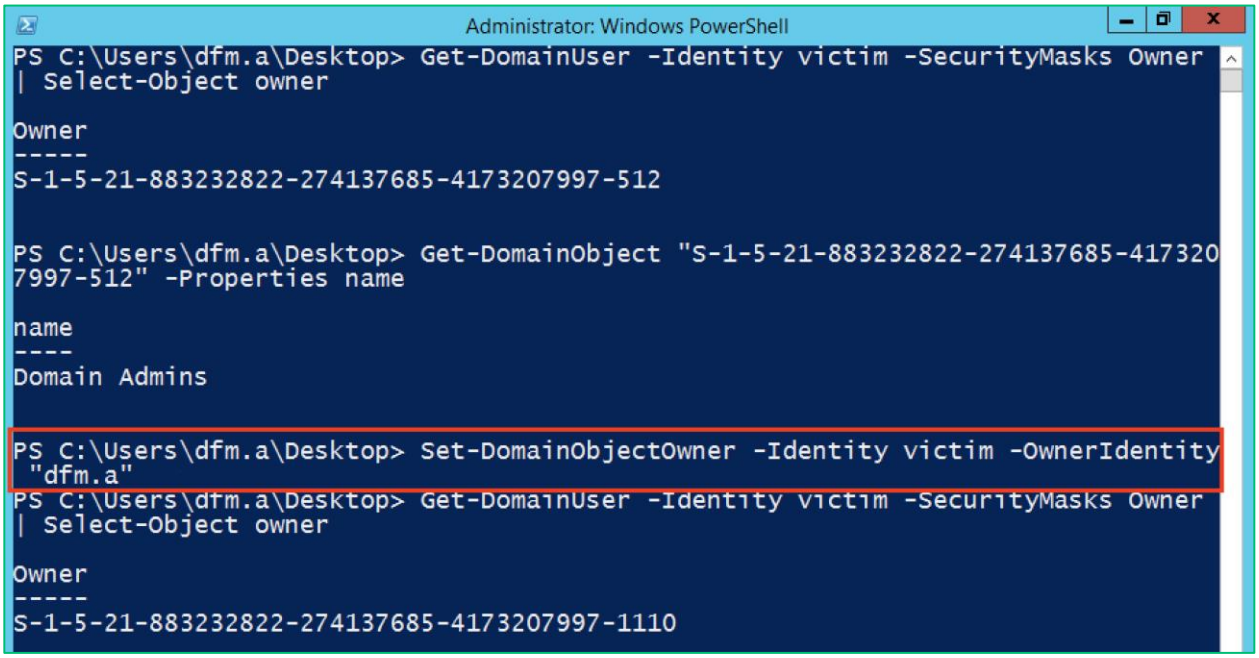
WriteOwner (formally RIGHT_WRITE_OWNER) allows the principal to modify the owner section of the object's security descriptor. Since the object owner has implicit GenericAll rights over the object, the principal can gain complete access to the object by modifying the object's owner. A user with this right can only change the owner to themselves - ownership cannot be transferred to other users. If the user has the SE_RESTORE_PRIVILEGE, however, the user with this right can change the owner to any object.

The WriteDacl control relationship can be abused with PowerView's **Add-DomainObjectAcl** function. This function has a **-Rights** parameter that currently accepts 'All', 'ResetPassword', 'WriteMembers', and 'DCSync' as aliases. The **-RightsGUID** parameter will accept a manual GUID representing the right to add to the target:



Above: Using PowerView's Add-DomainObjectACL function to grant the 'harmj0y' user force reset password rights over the 'victim' user.

The WriteOwner control relationship can be abused with PowerView's **Set-DomainObjectOwner**:



Above: Changing the owner of the 'victim' user to the attacking user 'dfm.a' using PowerView's Set-DomainObjectOwner function.

It is also worth noting that principals granted SE_TAKE_OWNERSHIP_PRIVILEGE can change any object's owner to themselves. In addition, principals granted SE_TAKE_OWNERSHIP_PRIVILEGE and SE_RESTORE_PRIVILEGE can change any object's owner to any other principal. As such, these privileges are all AD control right primitives.

Object-Specific Targeting

Some control relationships will only allow for the takeover of specific objects. For example, the User-Force-Change-Password doesn't make sense for a group policy object (GPO), but does make sense for a user object. This section will break down the objects in Active Directory that we care about for the purposes of our attack chains. Please note that what follows is a non-exhaustive list of object targeting strategies.

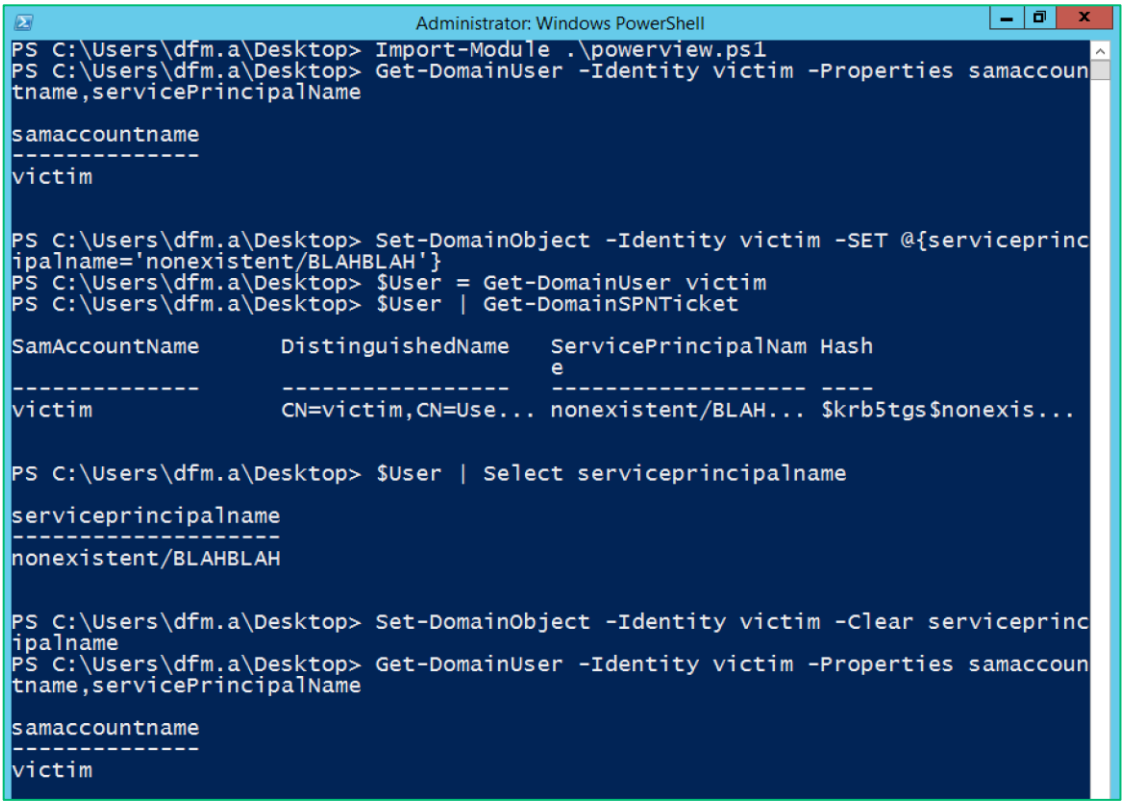
User Objects

To assume the rights of a user object, the logon credentials for the user account needs to be known or recovered by the attacker. There are two attack primitives that allow a principal to recover these credentials. The first is force-resetting the user's password without knowing the current password. This is granted by the User-Force-Change-Password right (GUID: 00299570-246d-11d0-a768-00aa006e0529), i.e. the RIGHT_DS_CONTROL_ACCESS bit is set and the ObjectType contains the previous GUID. GenericAll as well as RIGHT_DS_CONTROL_ACCESS with no GUID specified (i.e. 'all extended rights') will also grant this ability. This action is "destructive" in that you're making a reasonably loud change to the Active Directory environment- the user will be unable to log back into their account and will need to have their password reset by someone else in the organization. However, failed password attempts are not unheard of, and the user will likely believe that they just mistyped their password, not that it was reset maliciously.

The second attack primitive is "Targeted Kerberoasting[44]" where a target user's servicePrincipalName is set to a "nonsense" value, the account is Kerberoast'ed, the servicePrincipalName is reset to its original value, and the target user's password is cracked offline. This is obviously dependent on the user account having a password simple enough to crack in a reasonable amount of time, but this primitive has the big advantage of not being destructive and can be executed without the target user being made aware of the change. A principal with WriteDacl or WriteOwner rights on a target user account can execute this attack as well.

For execution with PowerView, **Set-DomainObject** can be used to set and unset a user's servicePrincipalName for Kerberoasting, and **Set-DomainUserPassword** can be used to force reset a user's password:

[44] http://www.harmj0y.net/blog/activedirectory/targeted-kerberoasting/



Above: Using PowerView's Set-DomainObject to set the 'victim' user's servicePrincipalName to a nonsense value, Kerberoasting the user so the password can be cracked offline, and clearing the servicePrincipalName property. We refer to this attack strategy as "Targeted Kerberoasting".

For more information on Kerberoasting, see Tim Medin's SANS HackFest 2014 talk "*Attacking Kerberos: Kicking the Guard Dog of Hades*[45]" or Sean Metcalf's blog post on the subject[46].

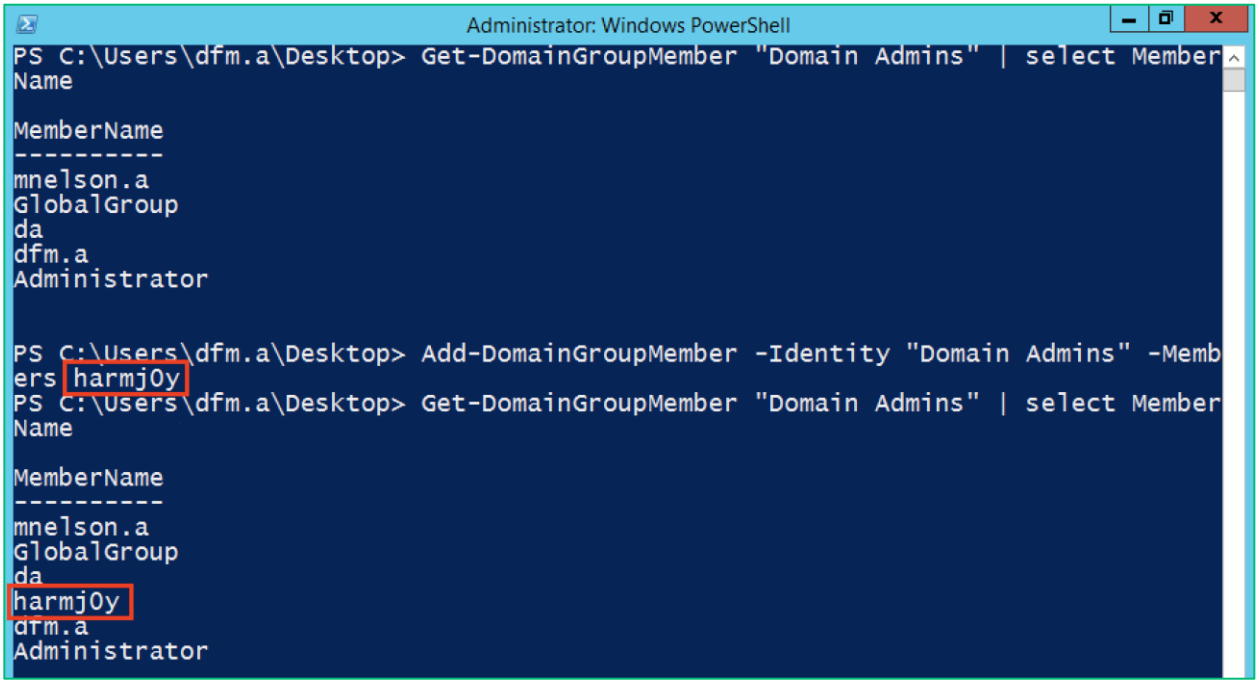
Group Objects

Groups are also security principals in Active Directory, but act like pseudo-containers. The way to take advantage of the rights/access a group has, as well as its possible nested relationships, is by adding a controllable user to the group membership. This is done through the modification of the **member** property of the group.

Specifically, we care about WriteProperty ACEs with the ObjectType being the GUID for the **member** property (GUID: bf9679c0-0de6-11d0-a285-00aa003049e2). Obviously, GenericAll and WriteProperty with no GUID (implies a properties) will implicitly grant this specific modification right as well, as do WriteDacl and WriteOwner.

[45] https://files.sans.org/summit/hackfest2014/PDFs/Kicking%20the%20Guard%20Dog%20of%20Hades%20-%20Attacking%20Microsoft%20Kerberos%20%20-%20Tim%20Medin(1).pdf
[46] https://adsecurity.org/?p=2293

To take advantage of this control relationship with PowerView, use the **Add-DomainGroupMember** function to modify group membership:



Above: Adding the 'harmj0y' user to the "Domain Admins" group using PowerView's Add-DomainGroupMember function.

Computer Objects

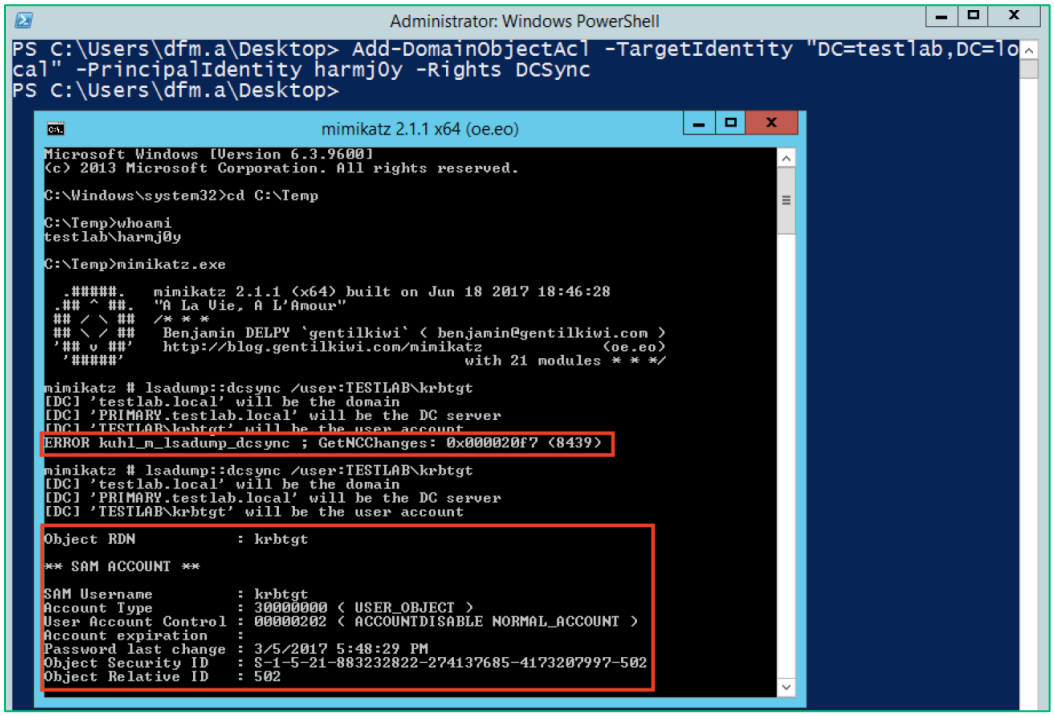
Unfortunately, we do not have a generic takeover primitive for computer objects. We believe this is likely possible and remains an open question for future research. Computer objects are a special type of user object, so force-resetting the machine's password would allow us to technically assume the computer account's rights; however, this causes the machine to effectively become disjoined from the domain until the machine password on the local system is resynced with the password for the account stored in Active Directory. Because this is not a relatively common occurrence (unlike user password resets), we do not view this approach as a viable primitive.

There is one exception to the above lack of attack primitives, but it's not present in the default schema of a forest. If Microsoft's Local Administrator Password Solution (LAPS) is installed, the plaintext password of the built-in local administrator account of the target computer object is stored in the confidential ms-Mcs-AdmPwd property on the computer object. By default, only elevated users (like Domain Administrators) have the right to read this property, but that right can be delegated to additional principals. The presence of this right allows the principal to read the plaintext password in ms-Mcs-AdmPwd, just as they would any other property. The password could then be used to compromise the target system. There is more background on LAPS, as well as ways to hide a LAPS-focused backdoors, in the "Backdoor Case Studies" section of this whitepaper.

Domain Objects

Domains are themselves represented in the Active Directory domain database, and reside at the root naming context “DC=domain,DC=local.” While we are aware that there are multiple ways to backdoor domain objects, the only one we cover in this whitepaper is backdooring domain objects to grant the rights associated with DCSync. Given two specific extended rights add to a domain object, DS-Replication-Get-Changes (GUID: 1131f6aa-9c07-11d1-f79f-00c04fc2dcd2) and DS-Replication-Get-Changes-All (GUID: 1131f6ad-9c07-11d1-f79f-00c04fc2dcd2), the specified principal can use the DC replication protocol to obtain the passwords hashes of any user/computer (amongst other things). The Mimikatz toolkit[47] contains an implementation of this credential theft technique (the **lsadump::dcsync** command) written by Benjamin Delpy and Vincent Le Toux.

With all that said, the relationships we care about are WriteDacl or WriteOwner on the domain object, ownership of the domain object, GenericAll on the domain object as it implicitly grants WriteDacl/WriteOwner, or existing DS-Replication-Get-Changes/ DS-Replication-Get-Changes-All rights. Given the right to modify the domain object’s DACL, two ACE entries can be added that grant the attacker the ability to DCSync a domain controller for any account hash.



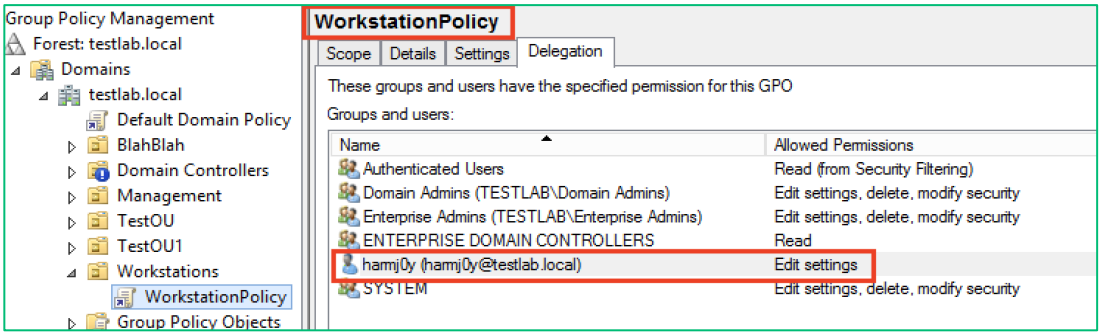
Above: Using PowerView’s Add-DomainObjectAcl function to add the two ACEs necessary for DCSync privileges to the root domain object with ‘harmj0y’ as the principal. The first highlighted attempt at executing DCSync fails, the rights are then added in the top window using PowerView, and the second highlighted DCSync attempt succeeds.

[47] https://github.com/gentilkiwi/mimikatz

Group Policy Objects

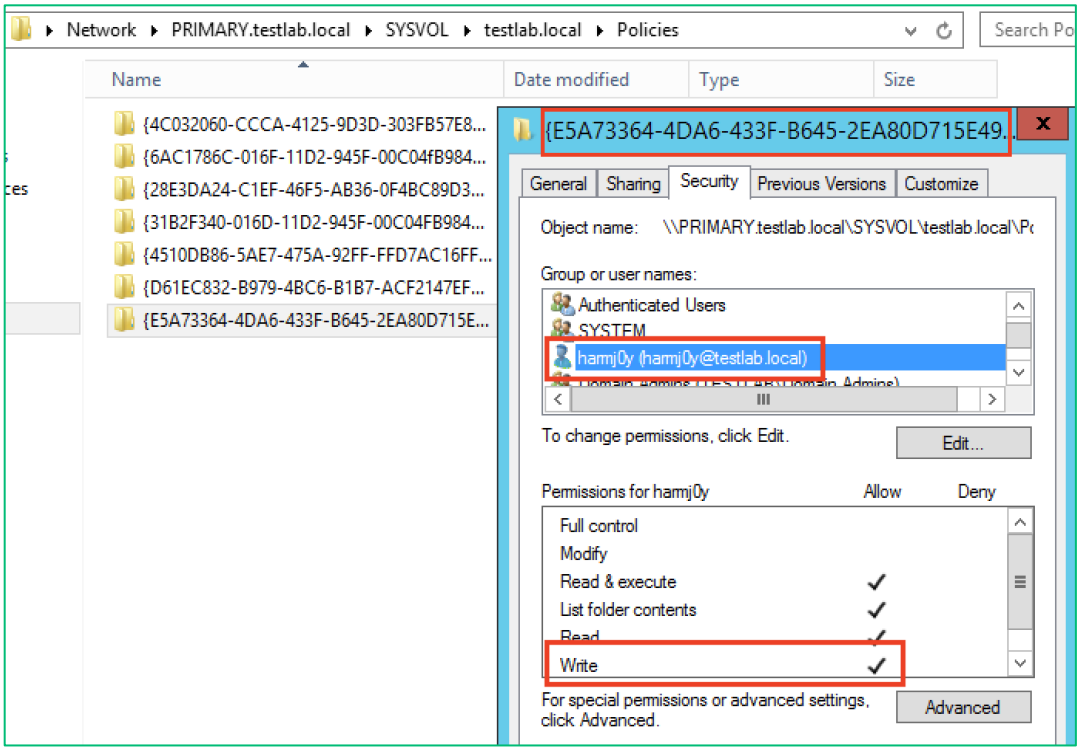
While we will not get into a full overview of Group Policy Objects (GPOs), the gist is that GPOs are collections of settings that are linked to an organizational unit (OU, this is the most common scenario), domain objects themselves (like the "Default Domain Policy"), and Active Directory sites[48] (this scenario is often forgotten). The settings in a GPO can apply to users or computers where the GPO is applied, and there are a myriad of ways to utilize a GPO to compromise a user account or gain code execution on an affected machine. For example, an "immediate" scheduled task can be pushed out to all machines through an applicable GPO, resulting in a scheduled task that runs once and then deletes itself. Suffice it to say that gaining edit rights over a GPO can easily result in the compromise of any users or computers the GPO is applied to.

The main relationship we care about with GPOs is WriteProperty to the GPC-File-Sys-Path property (GUID: f30e3bc1-9ff0-11d1-b603-0000f80367c1) of a GPO. According to Microsoft, "*This attribute specifies the Universal Naming Convention (UNC) path to the Group Policy Object template located in the system volume (SYSVOL)*"[49] meaning that the right to modify this property means the principal can modify the settings of the GPO. This is executed by manual editing of the associated GPO files at \\domain.com\SYSVOL\domain.com\Policies\{GPO_GUID}\. GenericAll, GenericWrite, WriteProperty without a GUID specified (i.e. write to ALL properties), and WriteDacl/WriteOwner rights can all facilitate this same effective access. On an interesting note, users granted WriteProperty rights to GPC-File-Sys-Path appear to have associated rights cloned to the GPO-specific folder on the filesystem where the associated SYSVOL is located:



Above: Granting the 'harmj0y' user rights to edit the 'WorkstationPolicy' GPO through the Group Policy Management console.

[48] https://technet.microsoft.com/en-us/library/cc754697(v=ws.11).aspx
[49] https://msdn.microsoft.com/en-us/library/cc219950.aspx



Above: The granted 'harmj0y' GPO edit rights being cloned onto the file system of the associated GPO folder in SYSVOL.

PowerView Abuse; redux

Here is the summary of the object takeover rights as weaponized through PowerView:

Right	Abuse Function
GenericWrite/GenericAll	Set-DomainObject
WriteProperty to specific properties	Set-DomainObject
WriteDacl	Add-DomainObjectAcl
WriteOwner	Set-DomainObjectOwner
CreateChild	Add-DomainGroupMember
User-Force-Change-Password	Set-DomainUserPassword

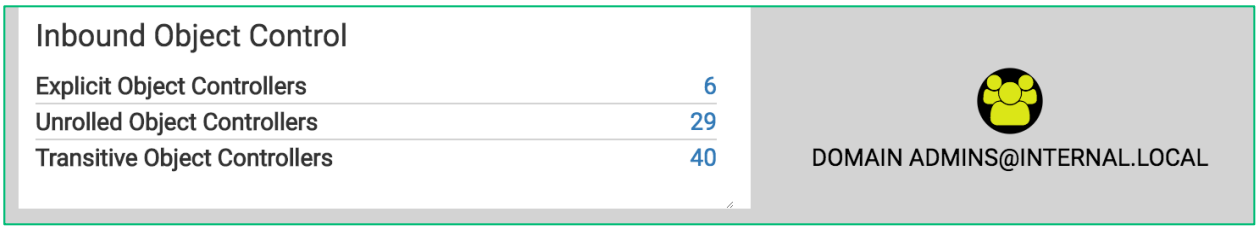
BloodHound Analysis

At DEF CON 24[50], the authors of this paper and Rohan Vazarkar released BloodHound[51], a free and open source tool which uses graph theory to reveal the hidden and often unintended privilege relationships in AD. Historically, BloodHound tracked AD security group memberships, local administrator group memberships, and user logon sessions. By collecting this information, an attacker may very quickly find the shortest derivative local administrator[52] style attack path to compromise a target node, if a path exists. Further, defenders may use the BloodHound interface and its attack graph to identify and remediate such attack paths, and easily illustrate the attack paths and privilege relationships to an audience not as familiar with Active Directory privilege delegation.

In May of 2017, we released version 1.3 of BloodHound[53] which added domain object DACL attack path analysis capabilities to the ingestors and interface. This update added several DACL privilege relationship edges to the attack graph, which are discussed earlier in this paper in the section titled "AD Control Rights." These privilege relationships are limited to the relevant ACEs which the authors of this paper have verified are abusable for taking control over other AD principals.
This update also added several DACL-related queries to each node type information tab. For instance, the interface makes it easy to analyze the effective inbound, abusable control relationships against any object, under the "Inbound Object Control" section of a user or group node tab:

- **Explicit Object Controllers:** The number of principals with abusable ACEs defined on the object's DACL. These principals would show up in the ADUC GUI when viewing the relevant object's DACL. Clicking the number displays the principals with "explicit" control of the object.
- **Unrolled Object Controllers**: The number of principals with abusable ACEs defined on the object's DACL, when accounting for security group delegation. For instance, if the "Domain Users" group has a control relationship over the "Domain Admins" group, the "unrolled object controllers" count against the "Domain Admins" group would equal the number of users in the "Domain Users" group.
- **Transitive Object Controllers:** The number of explicit object controllers, plus the number of unrolled object controllers, plus the number of other principals which can gain control of the affected user or group through an ACL-enabled attack path, without relying on a derivative local admin style attack.

[50] https://www.slideshare.net/AndyRobbins3/six-degrees-of-domain-admin-bloodhound-at-def-con-24
[51] https://github.com/bloodHoundAD/
[52] https://www.sixdub.net/?p=591
[53] https://wald0.com/?p=112

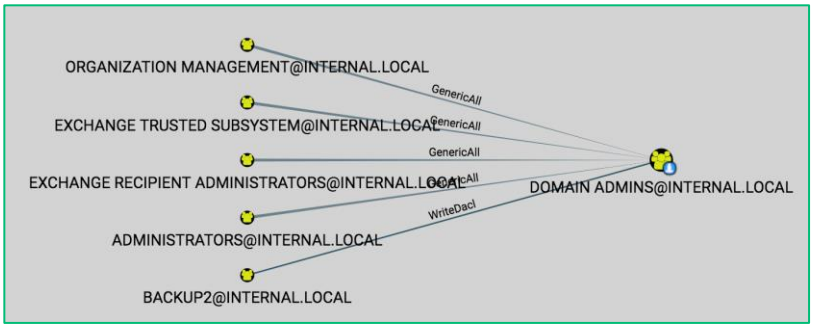


Above: The BloodHound interface tracks the principals which may take over the "Domain Admins" group by abusing ACL control relationships to this group.

By analyzing this data, an attacker may derive several key insights when determining how best to hide a DACL backdoor in Active Directory: the current state of DACL hygiene in the directory, the typical names used for principals with control over other objects, and good hiding places for objects hiding in "plain sight" within deeply nested security groups.

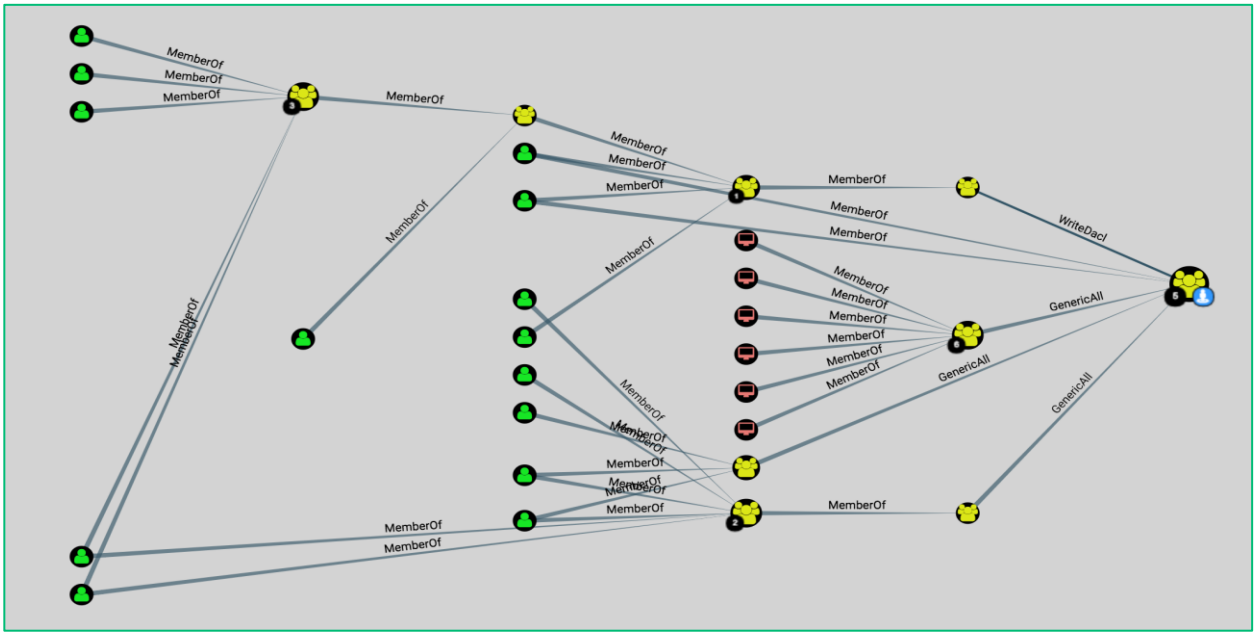
When assessing the current state of DACL hygiene, an easy starting point is the "Domain Admins" group in any domain. The "Domain Admins" group is a protected group, and thus its DACL will be replicated from the AdminSDHolder object. Ideally, from a defensive point of view, only the most trusted groups and users will have control relationships against the AdminSDHolder object, and in turn, every other protected object. However, in reality, the authors have observed several instances where a haphazard addition to the AdminSDHolder DACL has resulted in tens, hundreds, and even thousands of users gaining DACL control of the "Domain Admins" object, and every other protected object. In the example BloodHound database, we have added several DACL edges to illustrate what we have observed in real environments.

So far, the greatest offender we have seen of adding DACL control relationships to most, if not all, securable AD objects, is Microsoft Exchange Server (depending on the version). During the installation process, Exchange Server alters AD in several ways, including extending the AD schema to include the Exchange properties, adding several security groups to Active Directory, and, most critically, granting additional GenericAll (or Full Control) ACEs to almost every securable object in Active Directory. In fact, until Exchange Server 2007 SP1, the Exchange Server installation process included granting the "Exchange Enterprise Servers" group the WriteDacl permission against the root domain object, effectively giving any computer or other principal added to that group full control of the domain, and making Exchange servers equally as sensitive as domain controllers. The below figure shows a typical set of groups with control over the "Domain Admins" group, based on what we have seen in real environments. Several Exchange groups are granted control, as well as a seemingly random group called "Backup2":



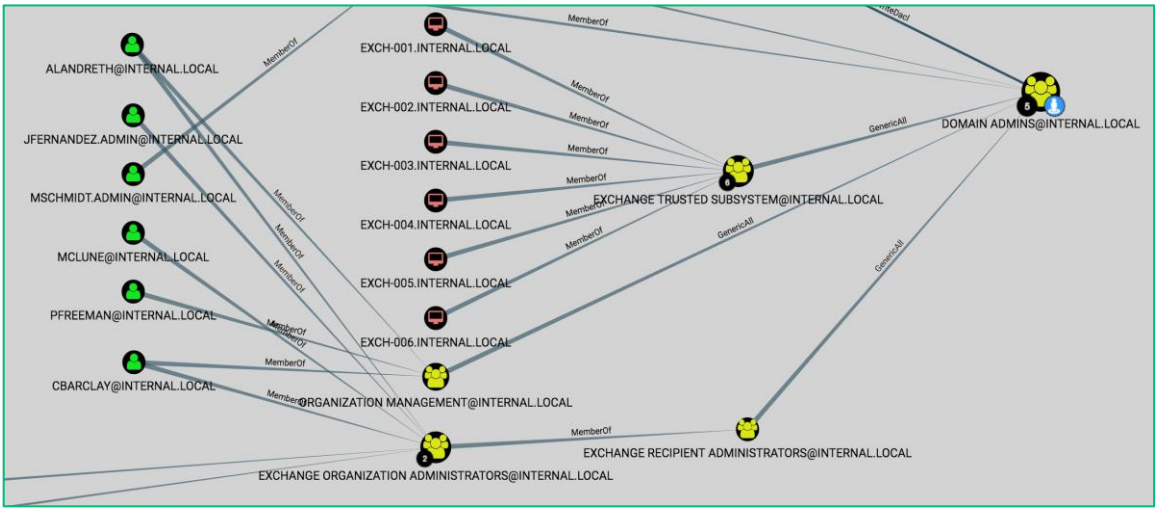
Above: The BloodHound interface showing the other groups which may take over the "Domain Admins" group by abusing ACL control relationships. Four groups have the "GenericAll" (or Full Control) privilege against the "Domain Admins" group, and one group has the "WriteDacl" privilege against the "Domain Admins" group.

The above view could fairly be equated with a trimmed-down view of the ADUC GUI of an object DACL, although we believe even in this aspect BloodHound provides an analyst with a more user-friendly experience. These groups have control of the "Domain Admins" group only tells a very small part of the story, as other users, groups and computers can, of course, be added to groups, and get the same rights as that group. BloodHound allows easy expansion to include the principals which effectively have control of the "Domain Admins" group through security group delegation:



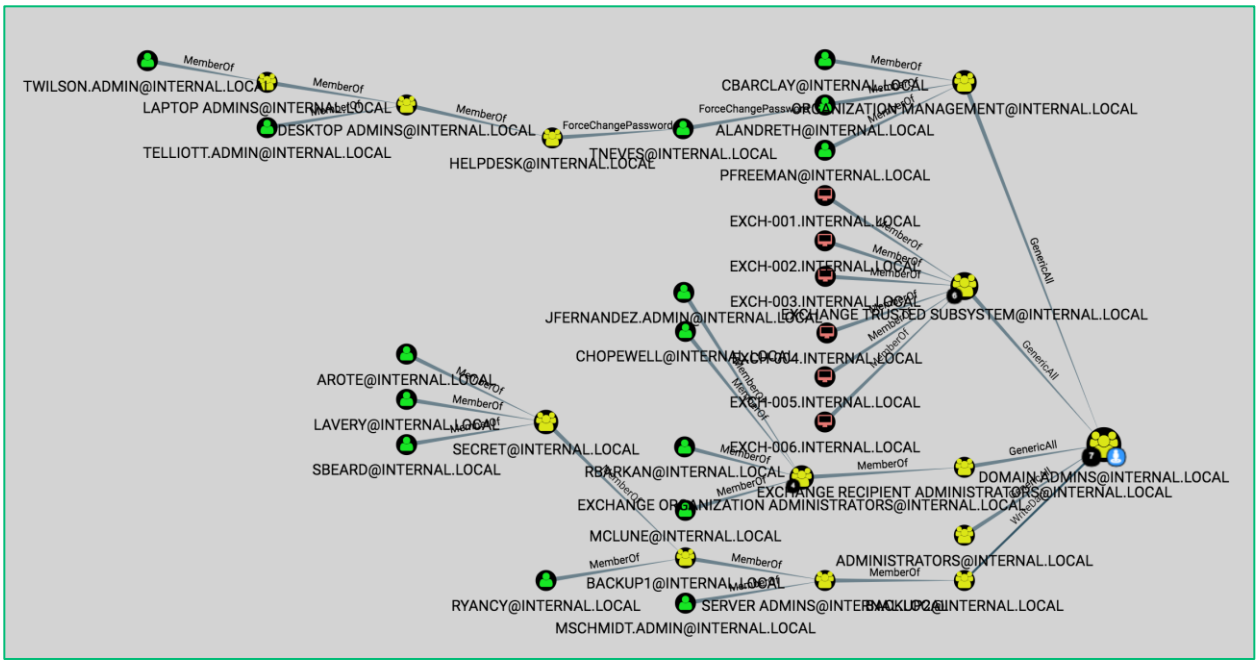
Above: The BloodHound interface shows the unrolled members of the groups with ACL control relationships against the "Domain Admins" group (far right). Of note are the several computer objects (colored in red) with effective control of the "Domain Admins" group.

An attacker may inspect this view in order to select backdoor candidates. For example, if we zoom into part of the above view, we can see several computer objects that may make good backdoor targets. Alternatively, an attacker may add a new object to the domain and "blend in" with existing accounts based on group memberships. Would a computer object called "Exch-007" immediately stick out when inspecting this view? Does the physical computer associated with the "Exch-003" computer object still exist? The answers to these questions may inform an adversary when deciding where to hide:



Above: Detail of the previous figure, showing several computers belonging to the security group "Exchange Trusted Subsystem", which in turn has the "GenericAll" (or Full Control) relationship to the "Domain Admins" group.

Finally, BloodHound enables easy analysis of *most* (but not all, yet) possible ACL-enabled attack paths that lead to a selected object. In the example below, our target object is the "Domain Admins" group. By clicking the number next to "Transitive Object Controllers", BloodHound queries and displays all the objects that can chain together an ACL-enabled attack path that is not based on the classic derivative local admin attack. This is especially interesting to an adversary, as the further away an object is from the "Domain Admins" group, the less likely it is that the defenders are paying close attention to that object. For instance, in the below screenshot, we can see that on the far left and upper right, a user named "TWILSON_ADMIN" has an ACL-enabled attack path resulting in ownership of the "Domain Admins" group. This attack path starts off due to this user belonging to a group called "LAPTOP ADMINS'. The controls, monitoring, and audit activities against this user are likely not as robust as those applied against Domain Administrator users. Regardless, this user is capable of taking over the "Domain Admins" group and, in turn, the domain, without using malware or compromising any computer in the domain.



Above: The BloodHound interface displaying the objects with a transitive object control attack path to the "Domain Admins" group. These are principals which may execute an ACL-enabled attack path to take over the "Domain Admins" group, without a derivative local admin style attack.

DACL-Based Backdoors

Before diving into strategies and stealth primitives for these types of backdoors, we wanted to step back for a second and reflect on the phrases “pure ACL attack-path” and “ACL-only attack paths.” As many of these concepts are new to most of the audience, we wanted to explain our interpretation of some of this new terminology.

We define an ACL “backdoor” as maliciously crafted ACE entries that allow for later domain or object compromise, while “attack path” is the execution of one or more control relationships to increase or otherwise expand an attacker’s current privilege. An attack path can be executed on a number of unintentional misconfigurations for the purposes of domain escalation, or the attack path can be the execution of a previously-implemented backdoor.

One issue is that to take over group, computer, or GPO nodes in an attack chain, an action must be executed that falls outside of ACL manipulation. For example, to subsume a group’s rights, a controllable principal must be added to its membership. Likewise, gaining control over a computer object (currently) involves code execution on the system, and attack paths involving GPOs involve editing of the GPO and often code execution in a specific machine or user context. Whether or not these actions qualify as a part of a “pure” ACL attack path is up to interpretation.

Therefore, we like the phrase “ACL-enabled” attack paths to avoid the purist argument, and we propose both strong and weak definitions. In our minds, the “strong” definition can be defined as “a domain attack path involving the abuse of one or more ACL control relationships that **don’t** require code execution on any additional machines.” That is, the “strong” definition of an ACL-enabled attack path means it can be executed from a single system, though may involve additional logon sessions. Our version of a “weak” definition would be “a domain attack path involving the abuse of one or more ACL control relationships.” Since it’s debatable what specific actions qualify as “ACL” based actions, we feel that this looser definition makes the most sense.

Objective

Our objective is to create various AD security-descriptor-enabled backdoors that:

- Facilitate the regaining of elevated control in the AD environment
- Blend in with normal security descriptor configurations (“hiding in plain sight”), or are otherwise hidden from easy enumeration by defenders

So assuming that defenders: a) are knowledgeable about these types of backdoor (unlikely) and b) have the tooling to enumerate, analyze, and remove these misconfigurations at scale (even more unlikely), we can either blend with “normal” DACL/ACE entries in a specific environment, prevent the backdoor from being enumerated, obfuscate/otherwise complicate the enumeration of the malicious principal used in the entry (or entries), or otherwise complicate the triage and removal of these types of actions.

This rest of section will cover various “stealth” (or anti-audit) primitives that we will use in the next section to build chains of malicious ACEs.

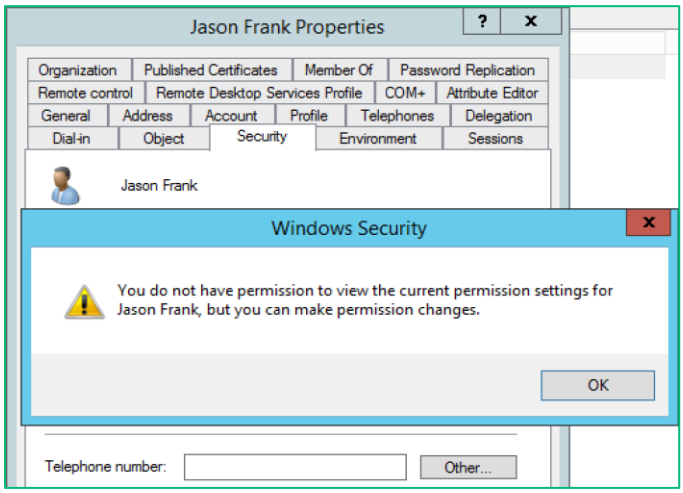
Stealth Primitive - Hiding the Security Descriptor

Remember the previous security reference monitor (SRM) breakdown and the “canonical” order of ACE evaluation:

1. Explicit DENY
2. Explicit ALLOW
3. Inherited DENY
4. Inherited ALLOW

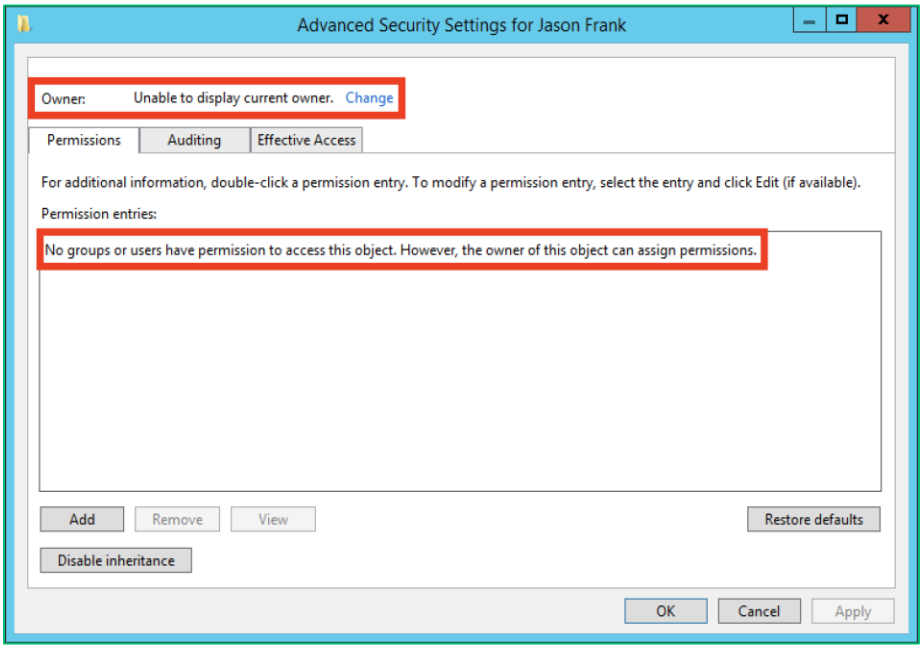
As explicit DENY rules take precedence over explicit allows, this grants us an opportunity to prevent the enumeration of our malicious security descriptor even by an elevated group such as “Domain Admins.” Also, remember that object owners implicitly have **GenericAll** rights to any object they own. This supersedes any explicit ACEs in the DACL. Since objects may have been created by the very privileged groups we’re attempting to hide the DACL from, we can’t forget to also modify the object owner. Also, remember that the **WriteOwner** right (implicit in **GenericAll**) means that principals can TAKE ownership of the objects, meaning they can only change the ownership to themselves. The principals are not able to “give” ownership to other principals (unless the principals also have SE_RESTORE_PRIVILEGE, that is).

The approach to hide an object’s DACL from elevated account enumeration thus involves: a) changing the owner of the object to an attacker controlled principal and then b) adding an explicit ACE to the target object that denies the “Everyone” principal (SID: S-1-1-0) RIGHT_READ_CONTROL (aka the “Read Permissions” right). Here’s what the object will look like in Active Directory Users and Computers (ADUC) MMC snap-in:

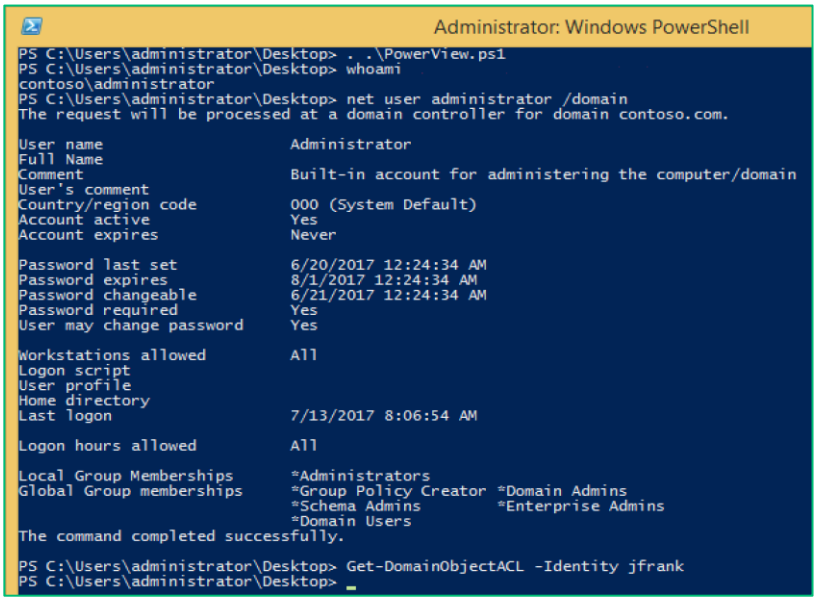


Above: When clicking the “Security” tab on the user object in ADUC, the above alert is displayed, informing the user that they do not have permission to view permissions on the object.

Here's how the enumeration of the DACL will look like from a "Domain Admins" context:



Above: The owner of the object is obscured from a domain admin user, along with the ACEs comprising the user DACL.

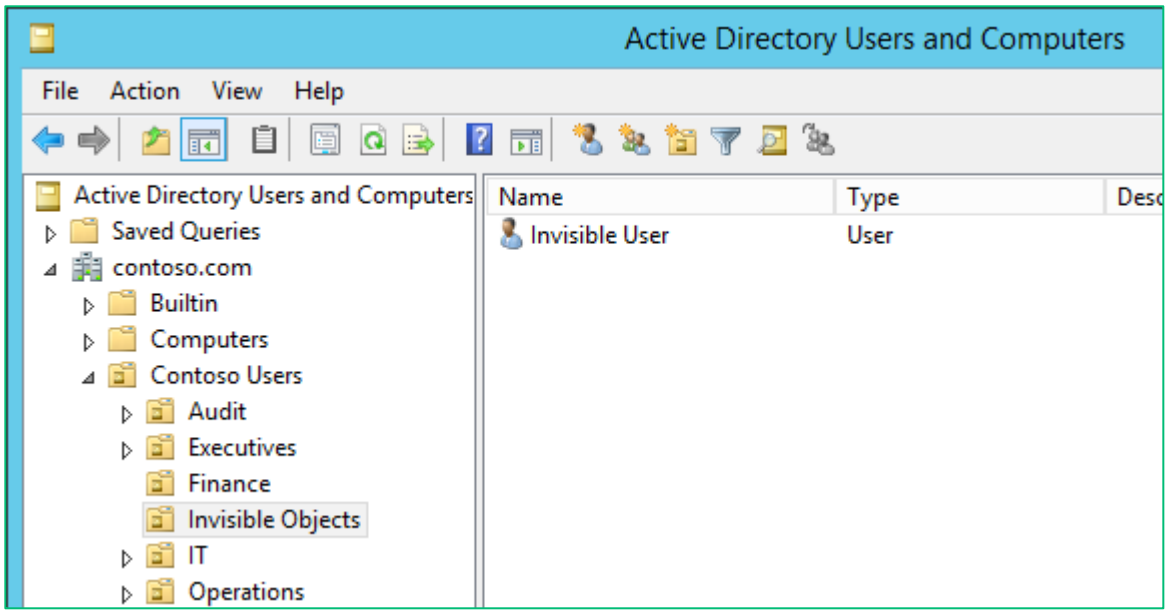


Above: Running PowerView from a "Domain Admin" context is still unable to enumerate the "jfrank" user.

Stealth Primitive - Hiding the Principal

Another primitive is hiding the principal/trustee listed in the security descriptor for the object. Even if an elevated user can retake ownership of a manipulated object, as in the previous example, we can still obscure the principal itself, preventing a defender from easily discovering who actually holds the right specified in the ACE.

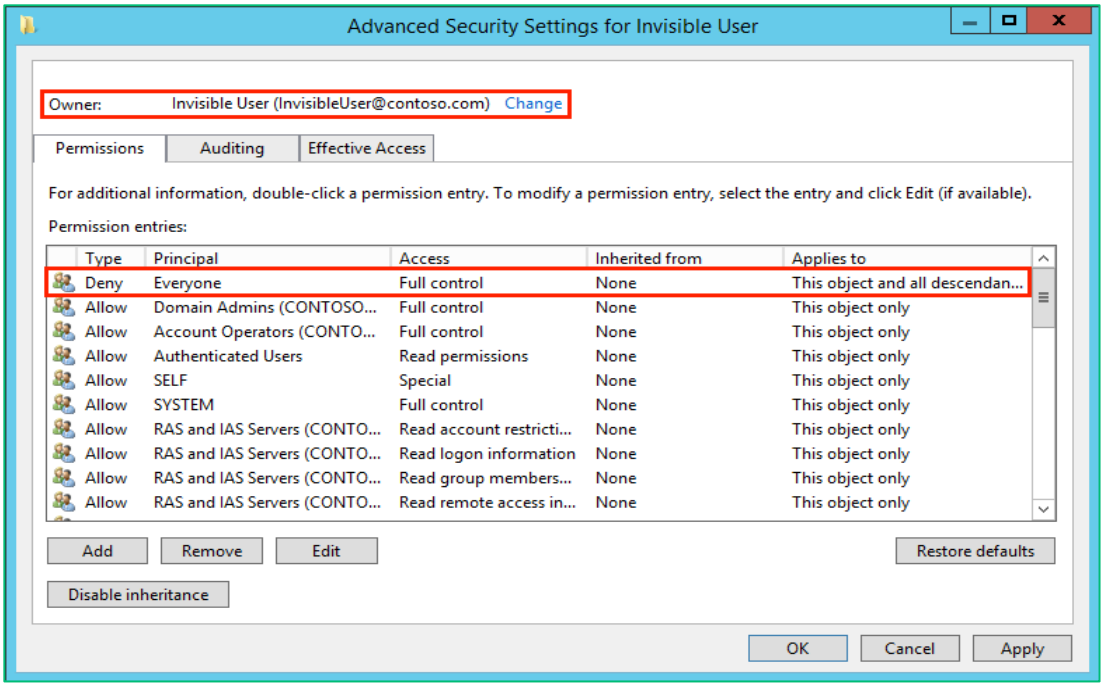
Consider an OU called “Invisible Objects”, containing a user called “Invisible User.” Upon initial creation of this OU and user, an admin may easily see the existence of the user in ADUC:



Above: ADUC displaying the contents of the “Invisible Objects” OU: one user called “Invisible User”.

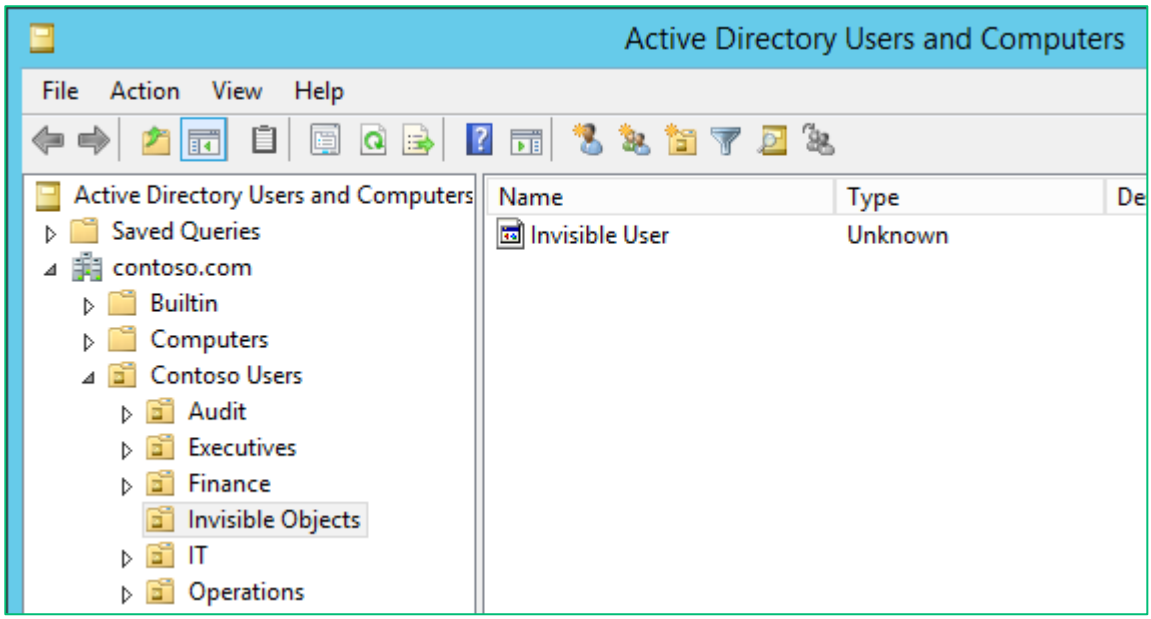
“Hiding” a principal is slightly more complicated than preventing the enumeration of a specific object DACL:’

1. Change the principal’s explicit owner to the attacker, or an attacker controlled account, due to the ownership issues described previously.
2. Grant explicit control of the principal to either itself, or another controlled principal.
3. On the organizational unit (OU) that contains this principal, deny RIGHT_DS_LIST_CONTENTS (“The right to list all child objects of the object, if the object is a type of container”).



Above: The owner of "Invisible User" has been set to itself and a new explicit "Deny" ACE has been added, denying "Everyone" full control of the user (I.e.: no privileges at all).

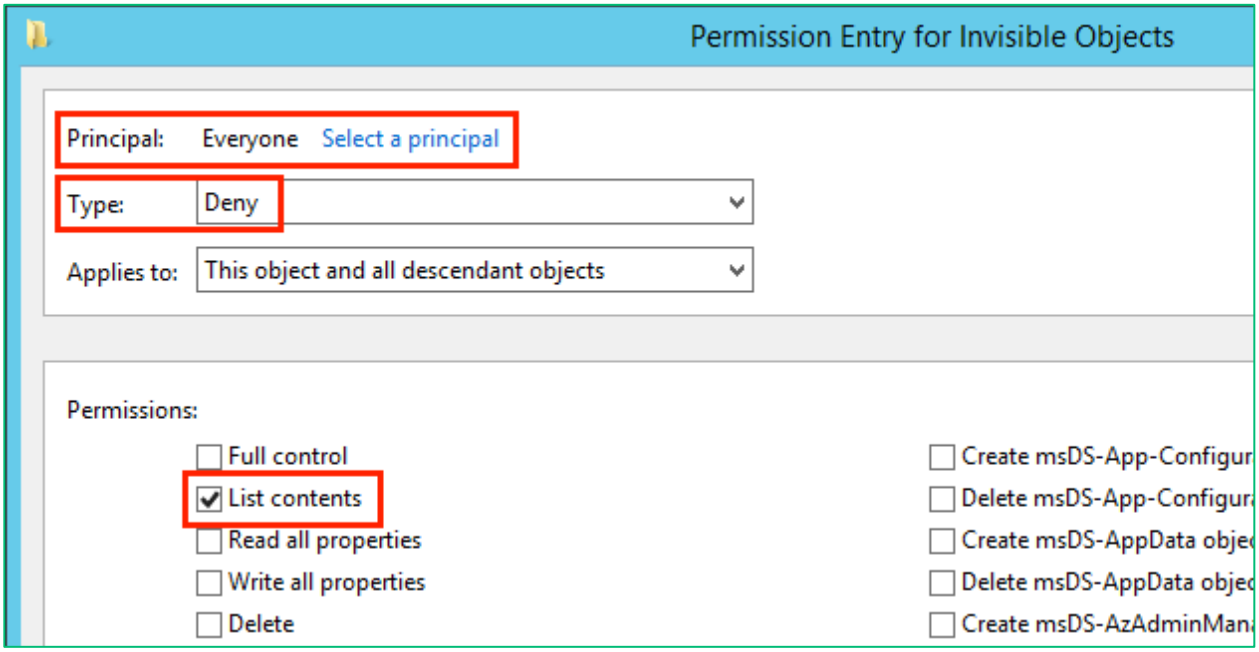
The reason for step 3 above is that if deny rights are implemented on the principal, any user that effectively has the LIST_CONTENTS rights on the container (most often an OU) that contains the principal, the user can still list the principal, though it will be rendered like this in ADUC:



Above: Viewing ADUC using a Domain Admin account, the "Invisible User" user object appears as an "Unknown" object type.

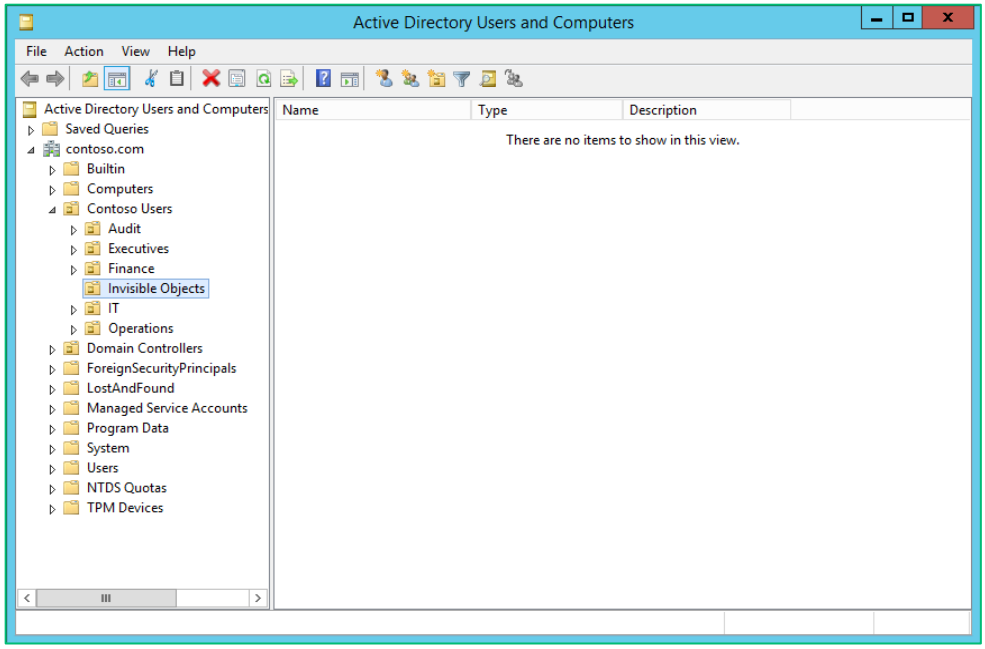
An admin may now easily see the existence of this strange, "unknown" type object in the OU structure. When a privileged user attempts to view the object in ADUC, the following text displays in the object property window: "*The Active Directory Domain Services object could not be displayed. Unable to view attribute or value. You may not have permissions to view this object.*" Then, if an admin tries to view the "Security" tab of the user, a pop up will inform the user that "*You do not have permission to view the current permission settings for <username>. It cannot be determined if you have the permission to make changes. Permission changes will be allowed but it cannot be guaranteed that the changes will successfully apply.*" Finally, when viewing the Security Descriptor GUI and DACL, the admin will be presented with a blank, non-informational view of the rights on this user; however, the admin retains the ability to change or claim ownership of the object through the SeTakeOwnership and SeBackupRestore privileges.

In order to more effectively obscure the existence of the object, we need to add an ACE to the container this object belongs to: in this example, it is the "Invisible Objects" OU. By denying the "Everyone" group the ability to list contents of this OU, the OU will effectively appear empty both in the GUI and via LDAP.



Above: A new Deny-type ACE is added to the OU, "Invisible Objects," blocking the ability for anyone in the domain to view the contents of the OU.

Now, a defender will not easily be able to identify the existence of the invisible user object, either through ADUC or LDAP:



Above: An administrator running ADUC attempts to list the contents of the "Invisible Users" OU, but the OU appears to be empty.

In this way, an attacker may be able to convince defenders that the security principal is some type of remnant and something to be ignored.

Stealth Primitive - A "Patsy" User

One additional approach involves the user of what we are calling a "patsy" (or "proxy") user. Instead of the attacker's account being used as the principal, and then attempting to hide that account from defenders, an attacker can use an account he/she controls as the principal in the malicious ACE. The attacker can implement any of the malicious ACEs on the patsy user itself to guarantee future account takeover, and then that account can be set as the principal in the "actual" malicious ACE. This forces defenders, even if they are even able to find the end ACE backdoor, to walk back chains of control relationships to triage the effective access.

This approach can extend beyond a single user as well. An attacker could implement numerous chains of control-focused ACEs that terminate in a user set as the principal on the actual object we're interested in. While BloodHound can walk these types of relationships back, this task is not trivial using other current toolsets.

Backdoor Case Studies

Now, let's bring everything together and show how all of these pieces can be used to create interesting AD DACL backdoors. Here is a review of the primitives we've covered:

- We know which specific ACEs on which object types can result in object takeover.
- We can hide/complicate the enumeration an object's security descriptor, even from users in elevated contexts (like "Domain Admins").
- We can hide/obscure a security principal/trustee in a way that makes it difficult to triage.
- We can use one or more "patsy" users to complicate walking the chains of control ACE relationships back.

Sean Metcalf calls this type of approach "Sneaky Active Directory Persistence Tricks"[54] and lays some of the groundwork that we expand upon.

Let's see what we can build!

A DCSync Backdoor

As described in the "Domain Objects" part of the previous "Object-Specific Targeting" section, DCSync is Benjamin Delpy and Vincent Le Toux's implementation of the replication protocol used by domain controllers to synchronize data. This has been fully implemented into the Mimikatz toolset[55], and allows for the retrieval of the password hash for any user in the domain, given the requesting user context has the rights to retrieve this information. Many in the offensive community believe that "Domain/Enterprise Admin" rights, or the equivalent, are needed to execute this synchronization operation. However, this is not the case.

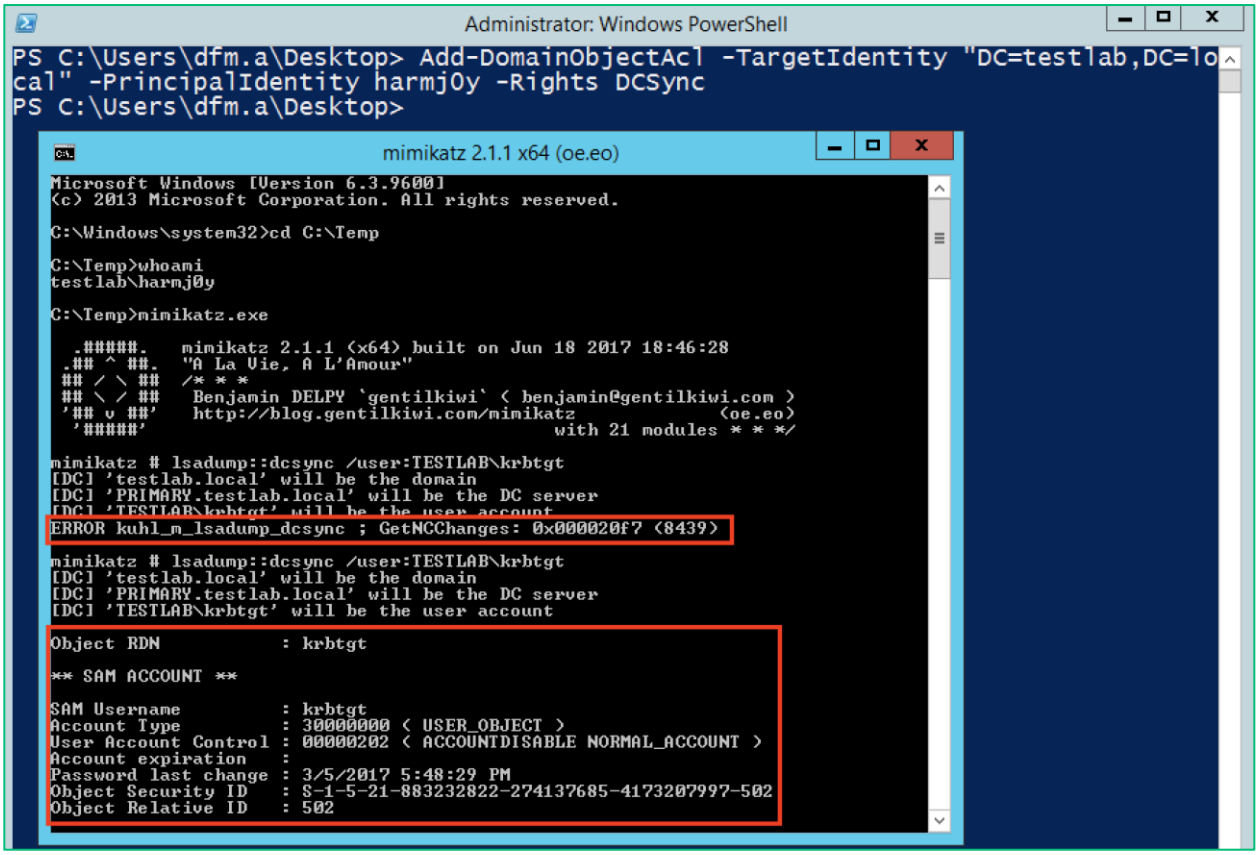
In effect, DCSync rights boil down to adding two extended rights on the domain object itself, located at the root naming context "DC=domain,DC=local". The two extended rights are DS-Replication-Get-Changes[56] (GUID: 1131f6aa-9c07-11d1-f79f-00c04fc2dcd2) and DS-Replication-Get-Changes-All[57] (1131f6ad-9c07-11d1-f79f-00c04fc2dcd2). These rights are, by default, granted to "Enterprise Domain Admins" and "Domain Controllers[58]". Sean Metcalf describes these rights on page 60 of his "*Red vs. Blue: Modern Active Directory Attacks & Defense*"[59] presentation at DerbyCon 2015.

Since replication rights depend on these two ACEs on the domain object itself, and not on group membership or other criteria, our first backdoor is a relatively simple one. An attacker simply adds these

[54] https://adsecurity.org/?p=1929
[55] https://github.com/gentilkiwi/mimikatz
[56] https://msdn.microsoft.com/en-us/library/ms684354(v=vs.85).aspx
[57] https://msdn.microsoft.com/en-us/library/ms684355(v=vs.85).aspx
[58] https://redmondmag.com/articles/2001/01/01/active-directory-data-guarding-the-family-jewels.aspx
[59] http://dansolutions.com/wp-content/uploads/2015/10/DerbyCon-2015-Metcalf-RedvsBlue-ADAttackAndDefense-Presented-Final.pdf

two extended-right ACE entries to the domain object for a principal/trustee that the attacker already controls, and said principal can retrieve the password of ANY user in the domain at will.

PowerView already weaponizes this through its **Add-DomainObjectAcl** function with the **-Rights DCSync** parameter:



Above: Using PowerView's Add-DomainObjectAcl function to add the two ACEs necessary for DCSync privileges to the root domain object with 'harmj0y' as the principal. The first highlighted attempt at executing DCSync fails, the rights are then added in the top window using PowerView, and the second highlighted DCSync attempt succeeds.

AdminSDHolder

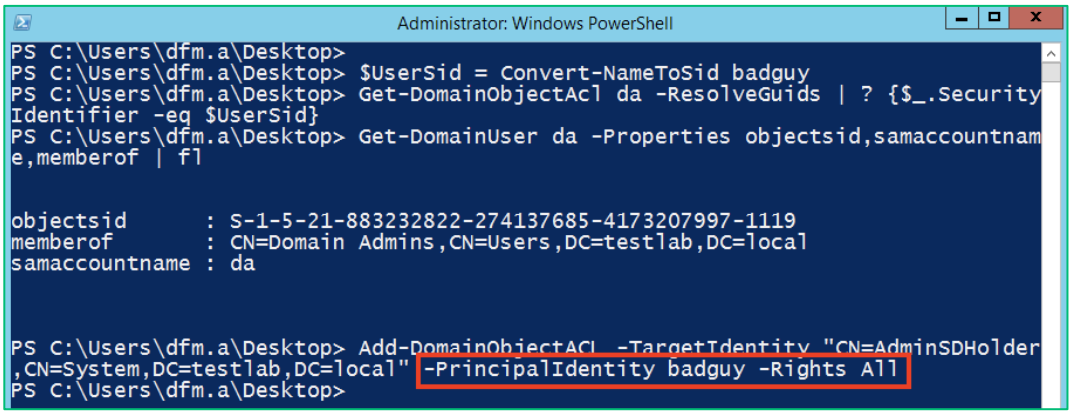
AdminSDHolder[60] is a special Active Directory object located at **CN=AdminSDHolder,CN=System,DC=domain,DC=com**. The stated purpose of this object is to protect certain privileged accounts from unintended security descriptor modification. Every 60 minutes, a special background AD task called SDProp (Security Descriptor propagator) recursively enumerates membership for a specific set of protected groups, checks the access control lists for all accounts discovered, and clones the security descriptor of the AdminSDHolder object to any protected objects

[60] https://technet.microsoft.com/en-us/library/2009.09.sdadminholder.aspx

with a differing security descriptor. Any account/group which is (or once was) a part of a protected group has their "adminCount" property set to 1, even if the object is moved out of that protected group. However, according to Microsoft[61], simply having the adminCount property set to 1 is not sufficient to trigger the AdminSDHolder protection. For additional information on AdminSDHolder and SDProp, please see Sean Metcalf's post on the subject[62] or the AD technical specification[63].

One effect of this AdminSDHolder/SDProp protection is that if any security descriptors are modified for protected groups, such as "Domain Admins" or "Enterprise Admins," the security descriptor template for those accounts will be reset within 60 minutes. If a malicious ACE is discovered by defenders on a single protected group or user, say a particular user in "Domain Admins," and defenders remove the malicious ACE, it will be recloned. However, this grants attackers another opportunity: if any ACEs are modified for the AdminSDHolder object itself, those permissions will be cloned to all protected users/groups within 60 minutes. Sean Metcalf detailed this object his "*Red vs. Blue: Modern Active Directory Attacks & Defense*"[64] presentation at DerbyCon 2015 on slides 51-53, and Philippe Biondi and Joffrey Czarny also detailed this backdoor approach in their 2015 Black Hat Arsenal talk "ACTIVE DIRECTORY BACKDOORS: Myth or Reality"[65] on slides 15-18. This is an attack primitive that the authors have explored previously[66], but we will add in an additional stealth primitive to expand the approach.

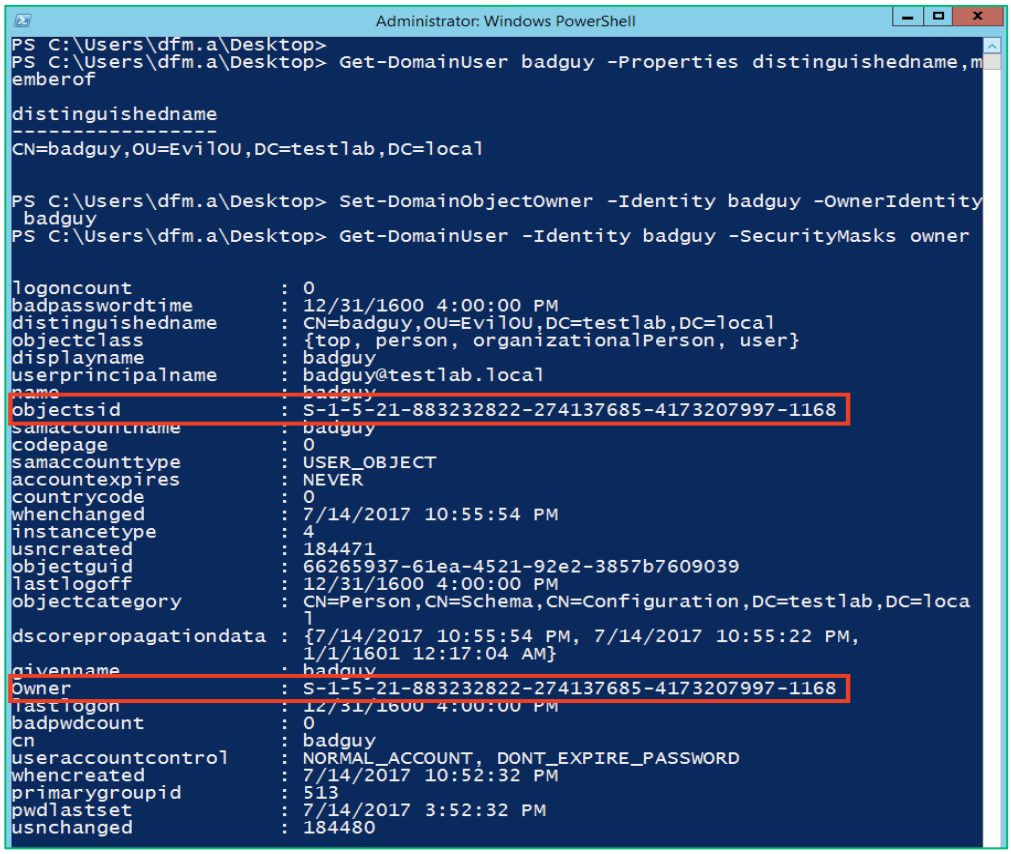
To implement the backdoor, the attacker first chooses some type of account/group takeover right described (GenericWrite, GenericAll, User-Force-Change-Password, WriteMembers, etc.) and adds that right to the **CN=AdminSDHolder,CN=System,DC=domain,DC=com** object. For our example, we will use GenericAll:



Above: Showing that a Domain Admin user 'da' does not have any ACE where 'badguy' is a principal. Then, Add-DomainObjectAcl is used to add a GenericAll ACE for 'badguy' to the AdminSDHolder object.

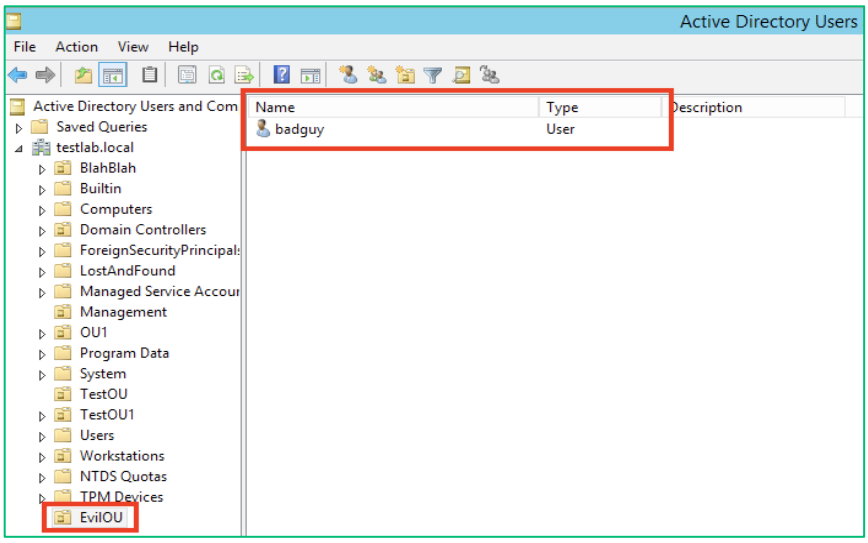
[61] https://blogs.technet.microsoft.com/askds/2009/05/07/five-common-questions-about-adminsdholder-and-sdprop/
[62] https://adsecurity.org/?p=1906
[63] [MS-ADTS] 3.1.1.6.1 "AdminSDHolder". https://msdn.microsoft.com/en-us/library/dd240052.aspx
[64] http://dansolutions.com/wp-content/uploads/2015/10/DerbyCon-2015-Metcalf-RedvsBlue-ADAttackAndDefense-Presented-Final.pdf
[65] https://bitbucket.org/iwseclabs/bta/downloads/BH_Arsenal_US-15-bta.pdf
[66] http://www.harmj0y.net/blog/redteaming/abusing-active-directory-permissions-with-powerview/

Then the attacker “hides” their account using the “Hiding the Principal” stealth primitive covered in the previous section. First, the owner of the ‘badguy’ principal is changed to ‘badguy’ itself. And note that ‘badguy’ is no particular groups:

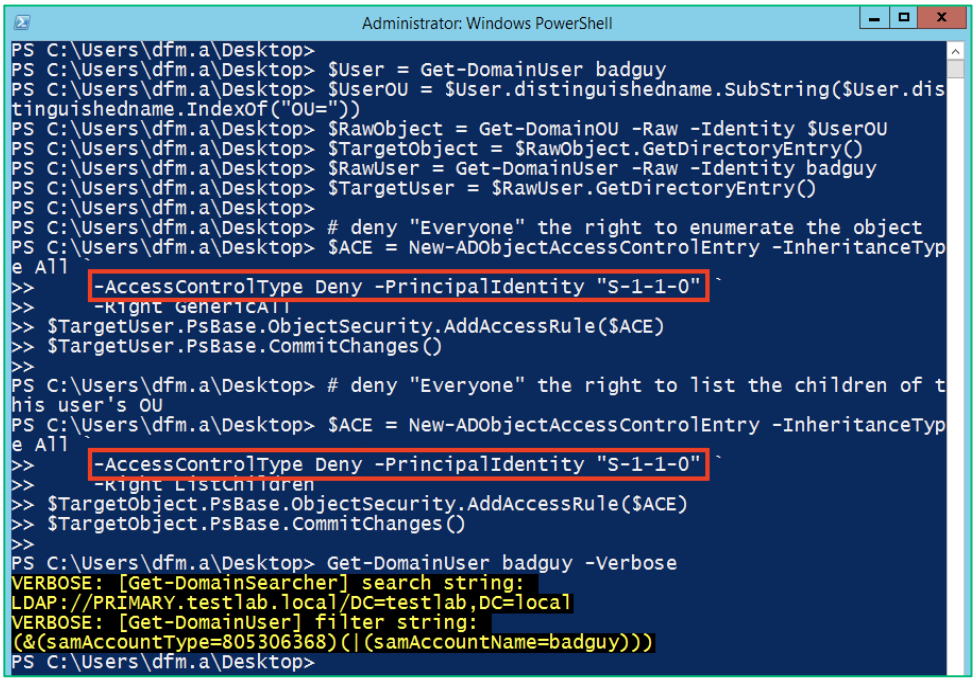


Above: Using PowerView’s Set-DomainObjectOwner to modify the ownership of the ‘badguy’ object.

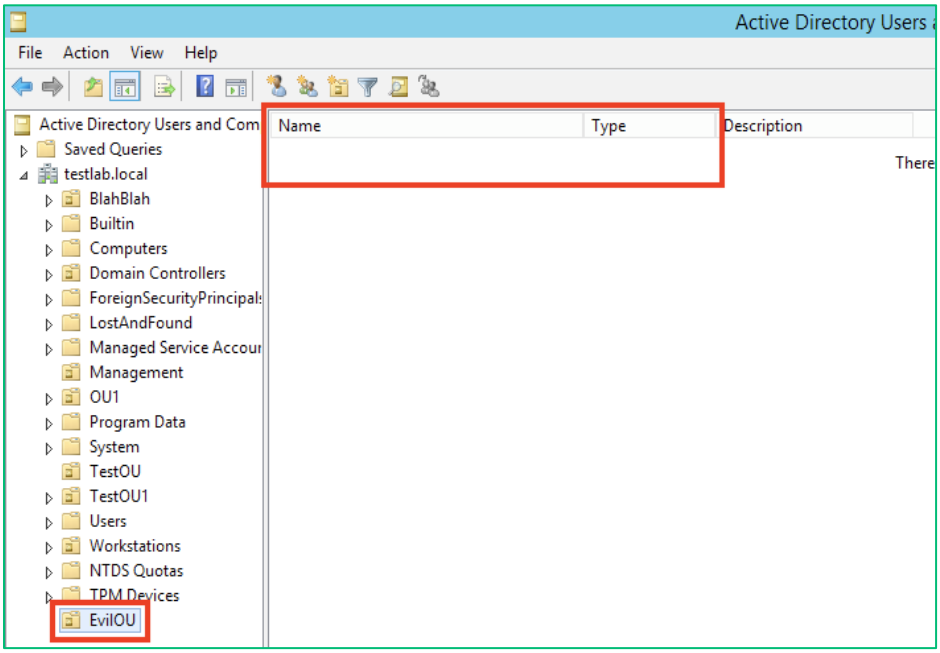
Next, we’ll hide the object from its OU listing as well as adding an explicit deny to the object itself for ‘Everyone’:



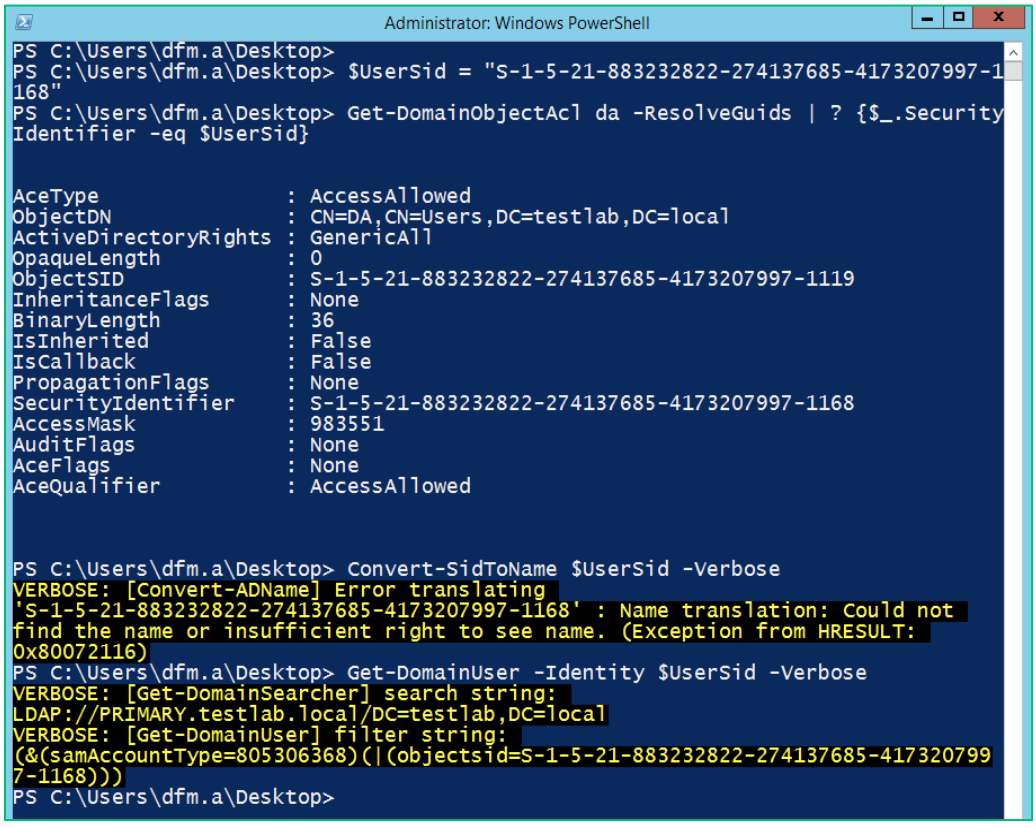
Above: The 'badguy' being listed in the 'TestOU' OU before manipulation.



Above: Adding a deny entry for ListChildren on the 'badguy's parent OU where the 'Everyone' SID (S-1-1-0) is the principal, as well as a deny for GenericAll to 'Everyone' on the 'badguy' object itself. After modifying the ACE, the object is no longer discoverable through LDAP.

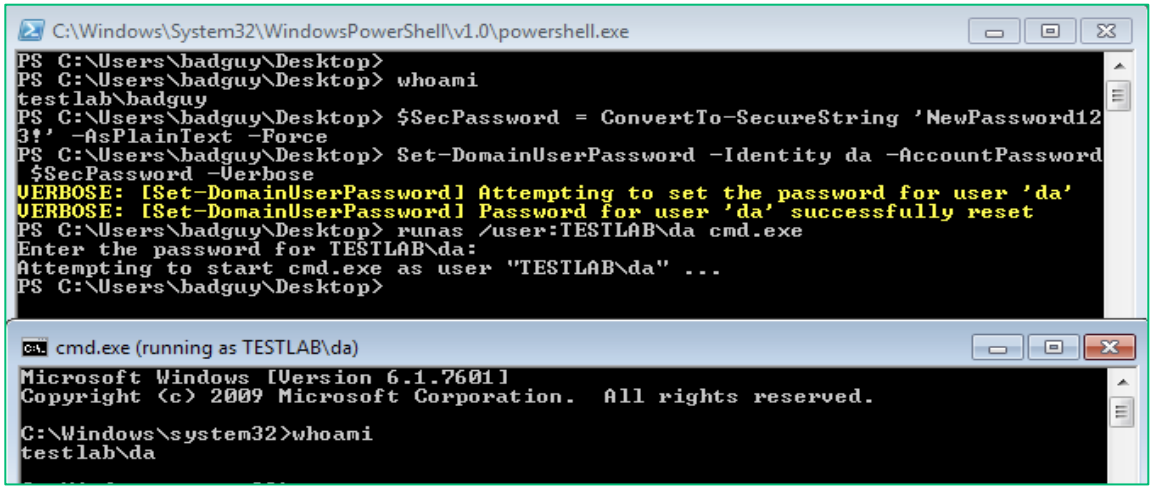


Above: The 'badguy' object is now hidden from its parent OU listing.



Above: The malicious ACE with 'badguy' as the principal present on a user in 'Domain Admins' but the current elevated context is unable to resolve the SID.

To execute the backdoor, the attacker executes a password reset from the 'badguy' context, resetting the password for a user in 'Domain Admins':



Above: The malicious ACE with 'badguy' as the principal present on a user in 'Domain Admins' but the current elevated context is unable to resolve the SID.

The end result is that the attacker-controlled account gains the ability to modify the membership of every SDProp-protected group (again, including "Domain/Enterprise Admins"), the ability to force reset the password for any protected user that is a member of any of these groups, and the ability to force Kerberoasting of the password for any user in these protected groups. However, even if the malicious ACE is discovered, it's difficult to triage the principal in the ACE.

A Case of Subverting LAPS

In May of 2015, Microsoft released the Local Administrator Password Solution (LAPS), an IT administration tool used to securely randomize and manage local administrator passwords. LAPS accomplishes this through two major components: an Active Directory schema change and a client side group policy extension. After installing these, IT administrators can then configure LAPS via the AdmPwd.PS PowerShell module.

The LAPS Active Directory schema change adds the properties ms-Mcs-AdmPwd and ms-Mcs-AdmPwdExpirationTime to computer objects. Most interesting to us is the ms-Mcs-AdmPwd property, as it contains the plaintext password of the local administrator account. In order to read ms-Mcs-AdmPwd property, a principal must have the DS_CONTROL_ACCESS right to the entire target computer object or to the ms-Mcs-AdmPwd property on the target computer object. Usually IT administrators do this by using the AdmPwd.PS cmdlet Set-AdmPwdReadPasswordPermission, which creates an ACE on a target OU granting a principal DS_CONTROL_ACCESS and ReadProperty rights to the ms-Mcs-AdmPwd property on all computers in the OU.

To help audit who has rights to read the ms-Mcs-AdmPwd property (and therefore the local admin password of LAPS-secured machines), the AdmPwd.PS PowerShell module contains a cmdlet named Find-AdmPwdExtendedRights. The intended purpose for this cmdlet is to enumerate, given a certain OU passed in as parameter, which AD principals can read the ms-Mcs-AdmPwd properties. The suggested use of this cmdlet is to audit who can read LAPS local administrator passwords. However, upon doing this research, we found several flaws where this cmdlet will not detect users with DS_CONTROL_ACCESS and we identified scenarios this cmdlet does not account for that could lead to attackers granting themselves access to the ms-Mcs-AdmPwd property.

When detecting who has a specific AD right, two things should be taken into consideration:

1) What principals can grant the right to themselves or other principals
2) What existing ACEs grant the specified right and what objects do the ACEs apply to

In regard to consideration #1, the first "flaw" we found is that that the Find-AdmPwdExtendedRights does not account for who has security descriptors control rights. For example, any principal can modify the DACL of a LAPS-protected OU or computer if the principal is the owner of the object, has the WriteDacl or WriteOwner rights to the object, or has SE_TAKE_OWNERSHIP_PRIVILEGE. With the ability to modify the DACL, the principal could simply grant itself DS_CONTROL_ACCESS to the ms-Mcs-AdmPwd property. As the cmdlet is named Find-AdmPwd*ExtendedRights* (emphasis on the extended rights), it's hard to really call this a flaw in the cmdlet. However, several articles tout using this cmdlet during security audits to determine who can read LAPS passwords, but in reality, this cmdlet does not take into account principals who can grant themselves this privilege.

The second set of flaws we found in Find-AdmPwdExtendedRights deal with some bugs in how it determines where the DS_CONTROL_ACCESS right has been granted (consideration #2 from above). Adding an allow ACE explicitly or via inheritance to a LAPS-protected computer with an access mask of DS_CONTROL_ACCESS or GenericAll (which implicitly grants DS_CONTROL_ACCESS) will allow a principal to read the ms-Mcs-AdmPwd property. To determine which principals have these rights, Find-AdmPwdExtendedRights analyzes explicit allow ACEs based on four things:

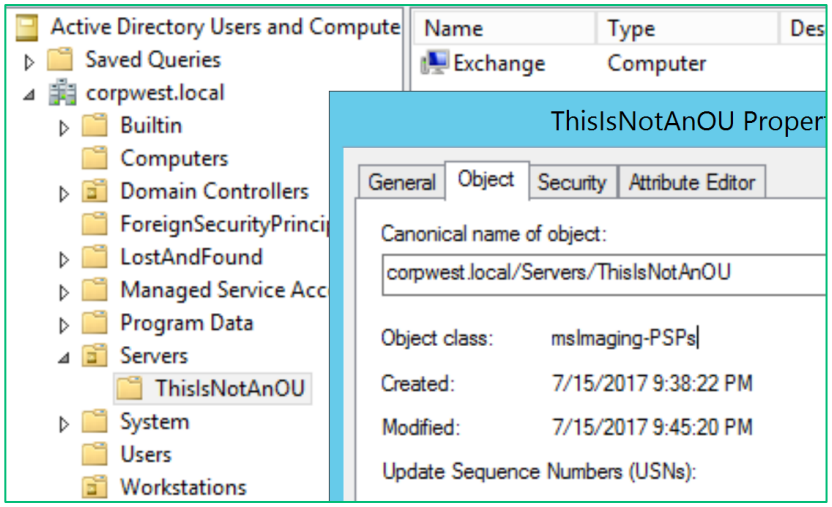
1) AD object type
2) ACE access mask (i.e. the granted right)
3) ACE ObjectType
4) ACE InheritedObjectType

In regard to AD object types, Find-AdmPwdExtendedRights only analyzes ACEs applied to OUs or computers. All other container objects are ignored. Consequently, an attacker could add a malicious ACE to a non-OU container that grants the attacker's account DS_CONTROL_ACCESS to the ms-Mcs-AdmPwd property.

One example of a non-OU container is the Computers container - the default container that AD adds all new computers to. Since Find-AdmPwdExtendedRights only analyzes explicit rights on OU containers,

it's impossible for the cmdlet to even analyze the "Computers" container. Consequently, if an attacker adds a malicious ACE to the Computers container, it will never be detected by this cmdlet.

Another example of malicious container is the msImaging-PSPs object. The following screenshots how an attacker could leverage this container type to evade the Find-AdmPwdExtendedRights cmdlet as well.



Above: 1) Compromised IT admin adds an msImaging-PSPs object "ThisIsNotAnOU," a type of container, to an existing OU and moves the computer "Exchange" from the Server OU into it.

```
PS C:\> whoami
corpwest\itadmin

PS C:\>
$RawObject = Get-DomainObject -Raw ThisIsNotAnOU
$TargetObject = $RawObject.GetDirectoryEntry()
$ACE = New-ADObjectAccessControlEntry -AccessControlType Allow `
    -Right ExtendedRight -PrincipalIdentity johnsmith -InheritanceType All
$TargetObject.PsBase.ObjectSecurity.AddAccessRule($ACE)
$TargetObject.PsBase.CommitChanges()

PS C:\> Find-AdmPwdExtendedRights -Identity Servers -IncludeComputers | fl *


ObjectDN              : OU=Servers,DC=corpwest,DC=local
ExtendedRightHolders : {NT AUTHORITY\SYSTEM, CORPWEST\Domain Admins, CORPWEST\ServerAdmins}

ObjectDN              : CN=Exchange,CN=ThisIsNotAnOU,OU=Servers,DC=corpwest,DC=local
ExtendedRightHolders : {NT AUTHORITY\SYSTEM, CORPWEST\Domain Admins}
```

Above: 2) Attacker uses IT admin to add an ACE to the "ThisIsNotAnOU" container, granting the low-privileged user "johnsmith" DS_CONTROL_ACCESS to all objects in the container. Note that Find-AdmPwdExtendedRights does not detect "johnsmith" as the having DS_CONTROL_ACCESS right ("johnsmith" is not in ExtendedRightHolders field).

```
PS C:\> whoami
corpwest\johnsmith
PS C:\> (Get-DomainUser johnsmith).memberof -eq $null
True
PS C:\> Get-DomainComputer Exchange -Properties ms-Mcs-AdmPwd

ms-mcs-admpwd
-------------
n.H54m-]Bq;46#3dtV2&
```

Above: 3) The low-privileged user "johnsmith" accessing the ms-Mcs-AdmPwd property

In regard to OU and computer object DACL ACEs, Find-AdmPwdExtendedRights analyzes ACEs that grant DS_CONTROL_ACCESS and GenericAll to the principal- the rights necessary to read the ms-Mcs-AdmPwd property. However, the cmdlet makes several flawed logic decisions when analyzing the ACE ObjectType and ACE InheritedObjectType fields that an attacker could abuse to subvert detection.

For ACEs with an access mask that grants GenericAll rights, the cmdlet will report a principal only if the ACE InheritedObjectType applies to all objects or only to computer objects. Notably, the cmdlet fails to account for ACEs added to computer object DACLs where the InheritedObjectType refers to a nonexistent object. The following table outlines this logic and its flaws:

AdmPwd.PS GenericAll Logic Flaws	OA	OC	OP	OAC	OCC	OPC
InheritedObjectType == Any object	✔	✔	✔	✔	✔	✔
InheritedObjectType == Computers	✔	✔	✔	✔	✔	✔
InheritedObjectType == Nonexistent Object InheritanceType == All,Descendants, or SelfAndChildren	**X**	**X**	**X**	N/A	N/A	N/A

Above: ACE ObjectType(Top) vs ACE InheritedObjectType(Left) when analyzing GenericAll ACEs. The ✔ and X symbols indicate whether Find-AdmPwdExtendedRights cmdlet correctly reports the principal that is granted GenericAll or not, respectively.

Table Definitions:

- ***OA** - ACE where ObjectType == Any Object, added to a computer object DACL*
- ***OC** - ACE where ObjectType == Computer Object, added to a computer object DACL*
- ***OP** - ACE where ObjectType == ms-Mcs-AdmPwd property, added to a computer object DACL*
- ***OAC** - ACE where ObjectType == Any Object, added to a container object DACL*

- ***OCC*** *- ACE where ObjectType == Computer Object, added to a container object DACL*
- ***OPC*** *- ACE where ObjectType == ms-Mcs-AdmPwd property, added to a container object DACL*
- ***N/A*** *= Not applicable since the ACE will never grant a principal the rights needed to view the ms-Mcs-AdmPwd property*

For ACEs with an access mask that grants DS_CONTROL_ACCESS rights, the cmdlet will report a principal under a couple circumstances:

- If the ACE applies to the ms-Mcs-AdmPwd property and is inherited to computers object descendants
- If the ACE applies to all objects and is inherited to all descendants or only to computers object.

Just as with GenericAll checks, the cmdlet fails to detect ACEs added to computer objects DACLs where the InheritedObjectType refers to a nonexistent object. Notably the cmdlet's logic fails to report any object where the ACE ObjectType only applies to computers. In addition, the cmdlet does not report when the ACE ObjectType applies to the ms-Mcs-AdmPwd property and to computer objects. The following table outlines this logic and its flaws:

AdmPwd.PS DS_CONTROL_ACCESS Logic Flaws	OA	OC	OP	OAC	OCC	OPC
InheritedObjectType == Any object	✔	X	X	✔	X	✔
InheritedObjectType == Computers	✔	X	✔	✔	X	✔
InheritedObjectType == Nonexistent Object InheritanceType == All,Descendants, or SelfAndChildren	X	X	X	N/A	N/A	N/A

Above: ACE ObjectType(Top) vs ACE InheritedObjectType(Left) when analyzing DS_CONTROL_ACCESS ACEs. The ✔ and X symbols indicate whether Find-AdmPwdExtendedRights cmdlet correctly reports the principal that is granted DS_CONTROL_ACCESS or not, respectively.

To exploit any of the above noted flaws, an attacker would need to strategically craft an ACE and apply it to an AD computer object or container. As an example, consider the scenario described in the table above where an attacker creates an ACE granting the DS_CONTROL_ACCESS right, with an ObjectType that applies to the ms-Mcs-AdmPwd property, and an InheritedObjectType applying to any descendant object. A privileged attacker could exploit this by adding this ACE to a target LAPS-secured computer, granting all users the ability to read the local admin password of the target computer while at the same time subverting the Find-AdmPwdExtendedRights cmdlet. We demonstrate this in the following screenshots:

```
PS C:\> whoami
corpwest\itadmin

PS C:\>
$RawObject = Get-DomainComputer -Raw Exchange
$TargetObject = $RawObject.GetDirectoryEntry()

$Guids = Get-DomainGUIDMap
$AdmPropertyGuid = $Guids.GetEnumerator() | ?{$_.value -eq 'ms-Mcs-AdmPwd'} | select -ExpandProperty name

$ACE = New-ADObjectAccessControlEntry -AccessControlType Allow `
    -PrincipalIdentity "Domain Users" -Right ExtendedRight `
    -ObjectType $AdmPropertyGuid -InheritedObjectType ([Guid]::Empty) -InheritanceType All

$TargetObject.PsBase.ObjectSecurity.AddAccessRule($ACE)
$TargetObject.PsBase.CommitChanges()

PS C:\> Find-AdmPwdExtendedRights -IncludeComputers -Identity Servers | fl *


ObjectDN              : OU=Servers,DC=corpwest,DC=local
ExtendedRightHolders : {NT AUTHORITY\SYSTEM, CORPWEST\Domain Admins, CORPWEST\ServerAdmins}

ObjectDN              : CN=Exchange,OU=Servers,DC=corpwest,DC=local
ExtendedRightHolders : {NT AUTHORITY\SYSTEM, CORPWEST\Domain Admins}
```

Above: A privileged attacker granting "Domain Users" DS_CONTROL_ACCESS to the "Exchange" computer. Note that the Find-AdmPwdExtendedRights cmdlet does not detect "Domain Users" as having the DS_CONTROL_ACCESS right (the group isn't listed in the ExtendedRightHolders field of Find-AdmPwdExtendedRights's output)

```
PS C:\> whoami
corpwest\johnsmith
PS C:\> (Get-DomainUser johnsmith).memberof -eq $null
True
PS C:\> Get-DomainComputer Exchange -Properties ms-Mcs-AdmPwd

ms-mcs-admpwd
-------------
n.H54m-]Bq;46#3dtV2&
```

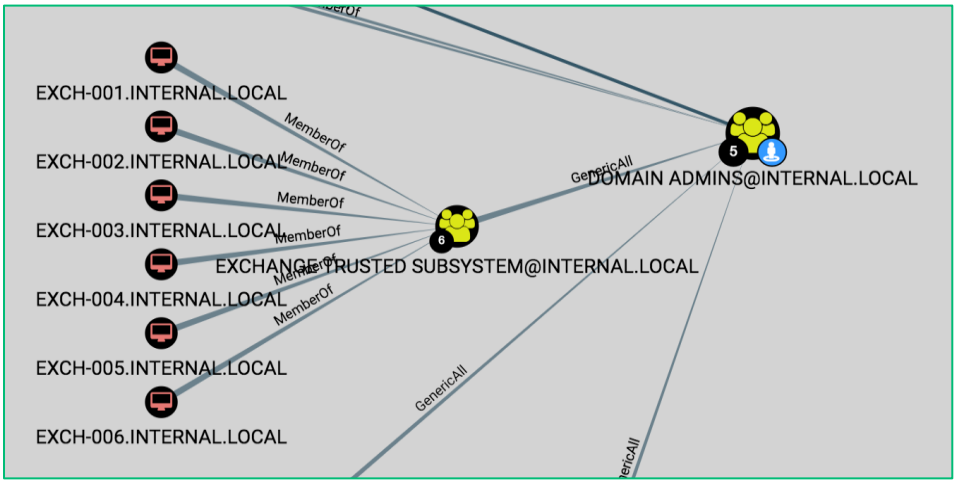
Above: The attacker leveraging an unprivileged user account to obtain the local admin password of the "Exchange" computer via an ACE backdoor.

As shown, an attacker can subvert detection by the Find-AdmPwdExtendedRights cmdlet by exploit logic flaws in detecting who has the DS_CONTROL_ACCESS right. This cmdlet is a great demonstration that detecting who has access to a specific Active Directory right is not a straightforward task. While these flaws are specific to LAPS, they can more generally apply to any tool that attempts to detect AD security descriptor misconfiguration flaws.

Exchange Strikes Back

The installation process for Microsoft Exchange Server includes several modifications to the Active Directory schema, default object class parameters, and the addition of several Exchange-related security groups and control relationships. In several enterprise environments, we have observed these privileges to include control over the AdminSDHolder object, and, in turn, every protected object (i.e. Domain

Admins, Enterprise Admins, and the members of these groups). For example, in many environments, all of the Exchange server computer objects are added to a security group called “Exchange Trusted Subsystem”, and this group is then given “GenericAll” (or Full Control) over the Domain Admins group and all the users of the Domain Admins group. If an adversary elevates to the SYSTEM user on any Exchange server in these environments, they will be able to abuse this privilege to execute the DCSync attack (due to full control over the domain object) or directly take over any other user or group in the domain.

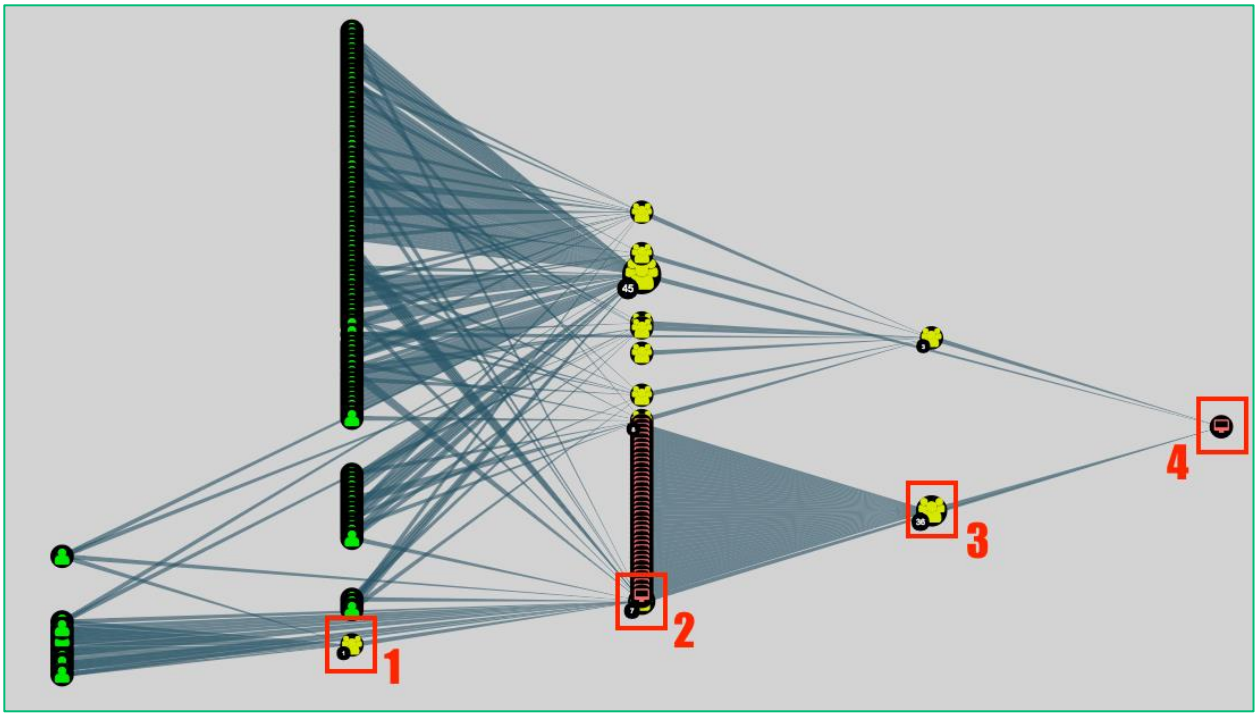


Above: A typical scenario in many enterprise networks: the Exchange Trusted Subsystem, with all Exchange computer objects as members, has full control over the Domain Admins group.

The level of control Exchange grants itself in Active Directory is dependent on the version of Exchange Server being installed. Exchange 2016 and 2010 assume full control over every user, computer, and group in Active Directory except for any protected object, including AdminSDHolder; however, Exchange 2016 still adds at least 53 ACEs to the AdminSDHolder object, which may still provide some way to assume control of AdminSDHolder and, in turn, any protected Active Directory object. Until Exchange Server 2017 SP1, Exchange additionally assumed control of the domain object with a “WriteDACL” ACE. As mentioned, we have observed several instances of the Exchange Trusted Subsystem simply having full control of all objects in the domain, including the domain object.

This backdoor relies on riding that liberal control over other objects granted to Exchange servers, and requires two steps:

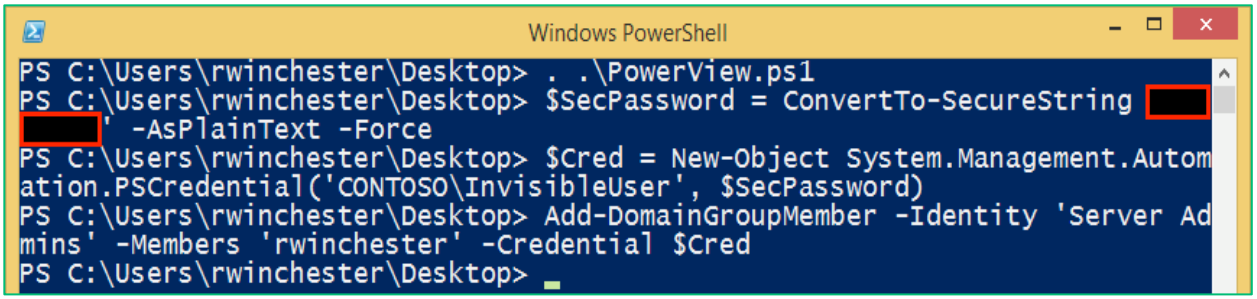
First, add our backdoor user to a nested group with local admin rights to an Exchange server. Security Group nesting easily grows out of control in most environments, making auditing user and group privileges an incredibly difficult and tedious task. In this demonstration, we will select a group three degrees separated from the system it has admin rights to: “Server Admins” is added to “US-West Server Admins”, which is added to “Exchange Server Admins”. Finally, “Exchange Server Admins” is granted local admin rights to an exchange server called EXCH001. Our backdoor, invisible user will be added to the “Server Admins” group, with its password set to never expire.



Above: An anonymized example from a real enterprise network. The group on the far left (1) is added to the group in the center (2), which is added to the group to the right (3). This group (3) is granted local admin rights to the computer (4). Because this computer belonged to the Exchange Trusted Subsystem, it had full control of the entire domain.

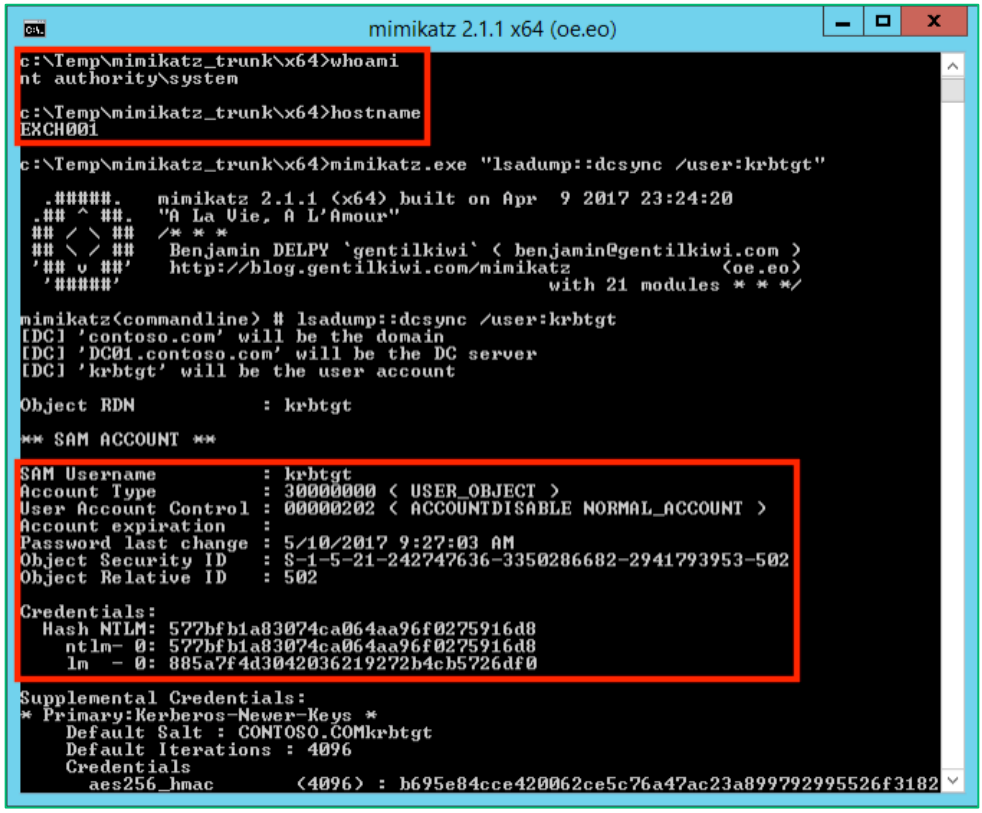
Second, hide the backdoor user using the methodology described in the "Stealth Primitive – Hiding the Backdoor User" section of this paper. Please see that section for detailed information on hiding the backdoor user.

To execute this backdoor, we first need domain authenticated access as any user. Because of LDAP's ability to supply alternate credentials, we can impersonate our backdoor, invisible user from any domain-joined context. We'll impersonate the user, then add our current user to the "Server Admins" group, effectively granting our new, initial access user local admin rights on the Exchange server.



Above: Using the alternate credentials of the "InvisibleUser" user, rwinchester adds himself as a member of the "Server Admins" group.

Next, we assume the identity of the "SYSTEM" user on the affected Exchange server. An attacker has numerous options for this step, which are outside the scope of this paper. Then, the attacker may ride the existing, extreme amount of control Exchange servers have over other Active Directory objects. The options for executing this backdoor are vast and varied. An attacker may choose to use this privilege to reset a high privilege user and then assume that user's identity, or add another user to a high privilege group, or even push evil GPOs to users in a certain OU. In several environments, the Exchange server account may even have DCSync privileges, granting us the ability to pull the krbtgt NT hash and create golden tickets.



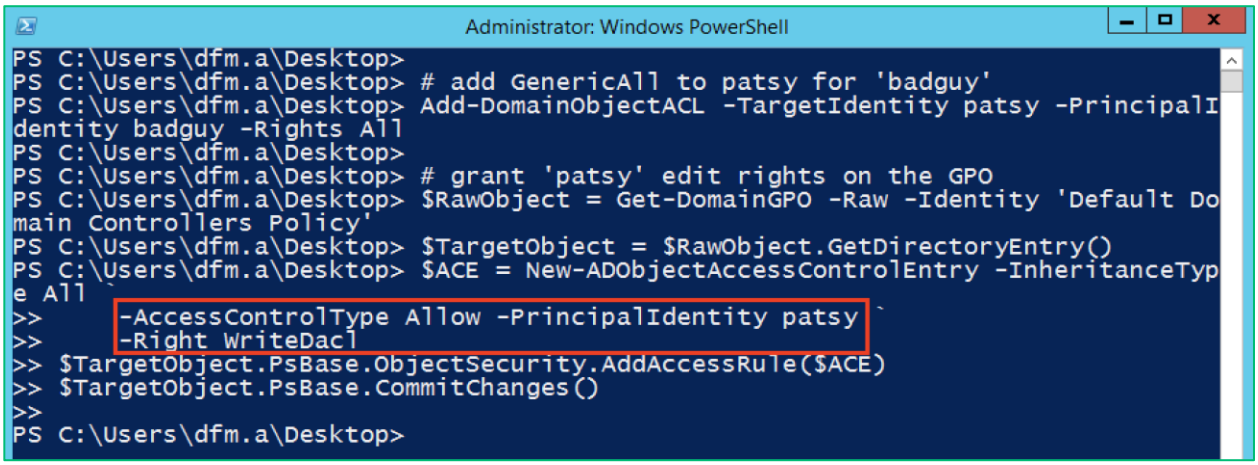
Above: Using Mimikatz's DCSync command, the SYSTEM user on the Exchange server steals the NT hash for the KRBTGT.

Abusing GPOs and Constrained Delegation

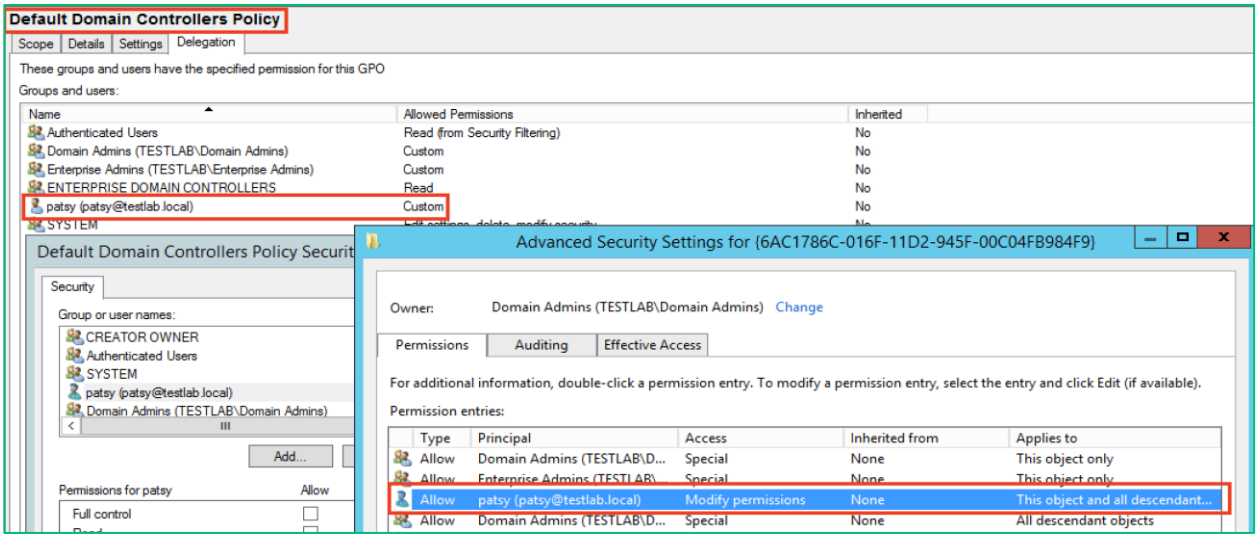
Our final case study is the most complex, and takes advantage of the "patsy" user approach. A core part of the attack chain is the abuse of the SeEnableDelegationPrivilege, which is described in depth by one of the authors here[67]. The more general use of GPOs for domain persistence is described in Sean Metcalf's "Sneaky Active Directory Persistence #17: Group Policy"[68] post.

[67] http://www.harmj0y.net/blog/activedirectory/the-most-dangerous-user-right-you-probably-have-never-heard-of/
[68] https://adsecurity.org/?p=2716

The implementation of the backdoor comprises of two parts. First, the attacker grants themselves **GenericAll** rights to any user in the domain. This user does not have to be a member of any privileged group and will function as our "patsy" user. Second, the attacker grants this "patsy" user the **WriteDacl** right to the "Default Domain Controllers" Group Policy Object (GUID: 6AC1786C-016F-11D2-945F-00C04FB984F9). This is the entirety of the backdoor implementation.



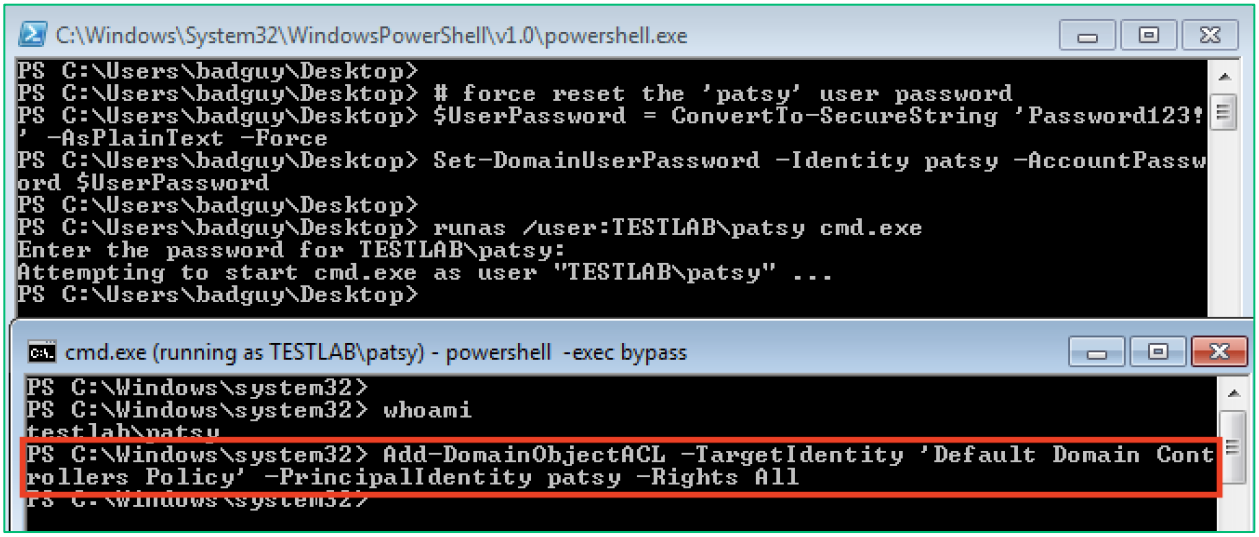
Above: Granting 'harmj0y' all rights to the 'patsy' user, and granting the 'patsy' user the right to edit the DACL of the "Default Domain Controllers Policy".



Above: The DACL edit rights granted to the 'patsy' user reflected in the group policy management console.

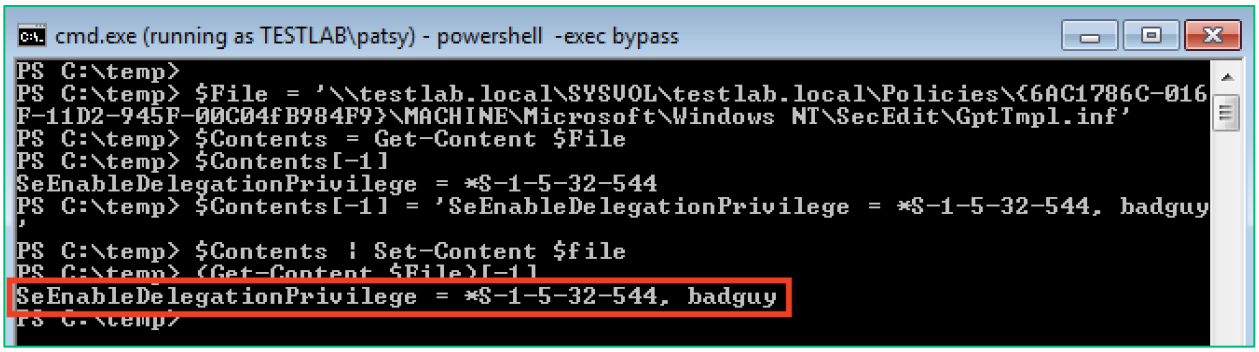
To exercise the backdoor, the attacker first force-resets the "patsy" user's password, or executes the "targeted Kerberoasting" attack to recover the user's password. The key is regaining the ability to authenticate as this user. From there, the attacker authenticates as the "patsy" user and uses takes

advantage of the previously implemented **WriteDacl** right to add a malicious ACE to the "Default Domain Controllers" GPO which grants the "patsy" user the right to edit the GPO itself:



Above: The attacker 'badguy' force-resets the password for the 'patsy' user, authenticates as 'patsy,' and then uses that authentication context to grant all rights for 'patsy' to the 'Default Domain Controllers Policy' GPO.

Then, from the same "patsy" user context, the attacker grants the "badguy" user the SeEnableDelegationPrivilege privilege right in the \\DOMAIN\sysvol\testlab.local\Policies\{6AC1786C-016F-11D2-945F-00C04fB984F9}\MACHINE\Microsoft\Windows NT\SecEdit\GptTmpl.inf group policy file:

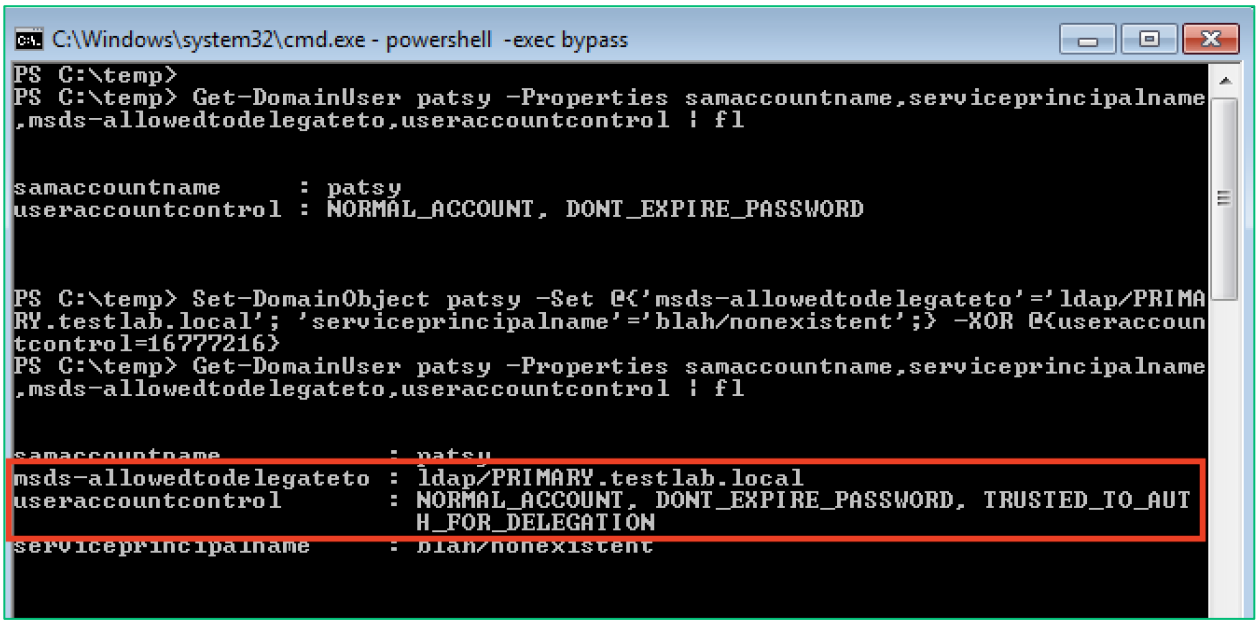


Above: Modifying the default domain controllers policy to grant the 'badguy' user SeEnableDelegationPrivilege.

This now grants the 'badguy' user the ability to execute a constrained delegation[69] attack using Benjamin Delpy's Mimikatz and Kekeo[70] projects that would result in the ability to DCSync any account

[69] http://www.harmj0y.net/blog/activedirectory/s4u2pwnage/
[70] https://github.com/gentilkiwi/kekeo/

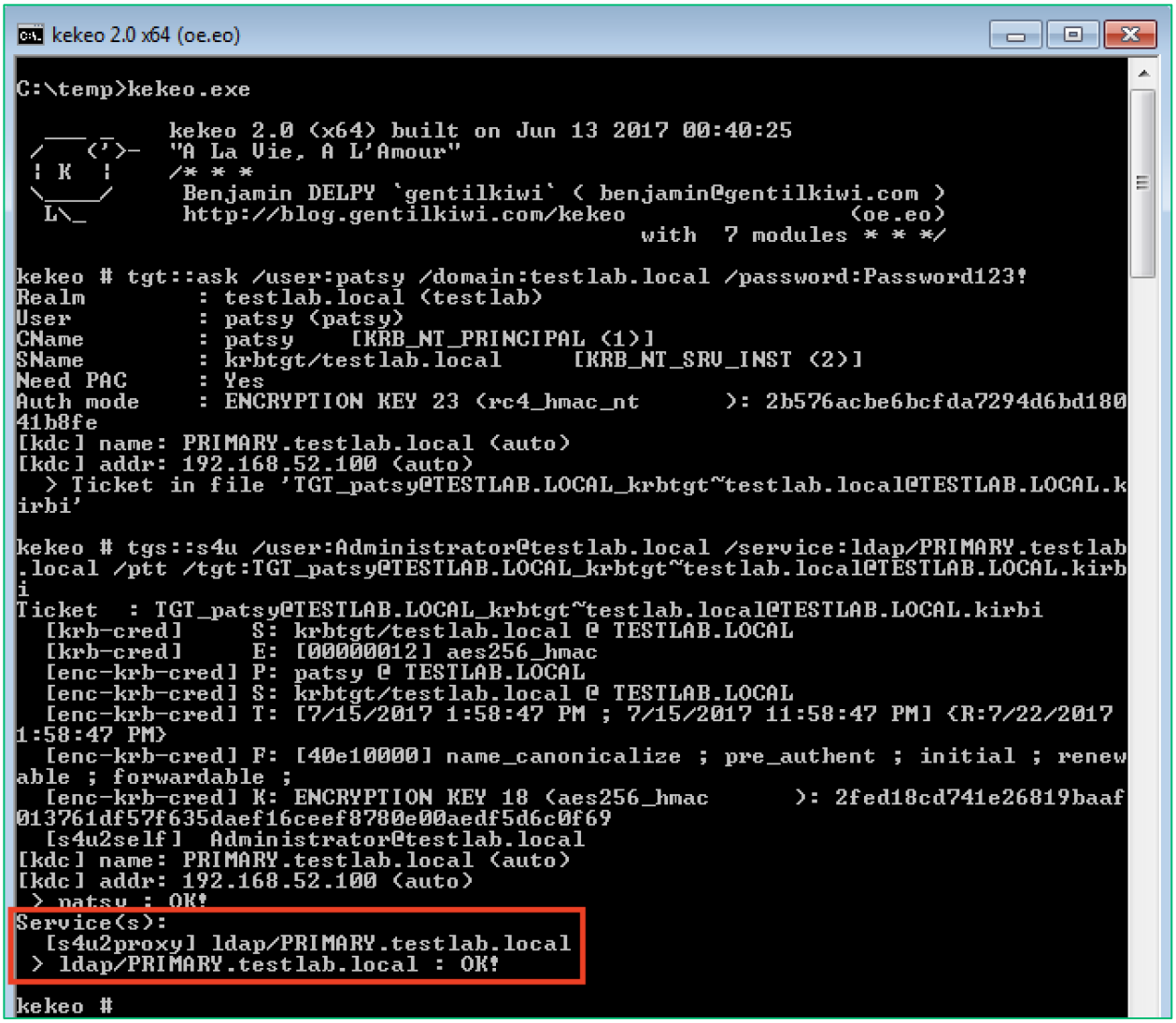
credential in the domain. First, 'msds-allowedtodelegateto' for 'patsy' is changed to ldap/DOMAIN_CONTROLLER. This property can only be modified by a user who has SeEnableDelegationPrivilege, which we just granted the 'badguy' user. We also set a nonsense SPN as it's a requirement for the later Kerberos flow[71], and flip the user account control bit 16777216 (TRUSTED_TO_AUTH_FOR_DELEGATION):



Above: Using the 'badguy' user's SeEnableDelegationPrivilege to set the msds-allowedtodelegate property of 'patsy'.

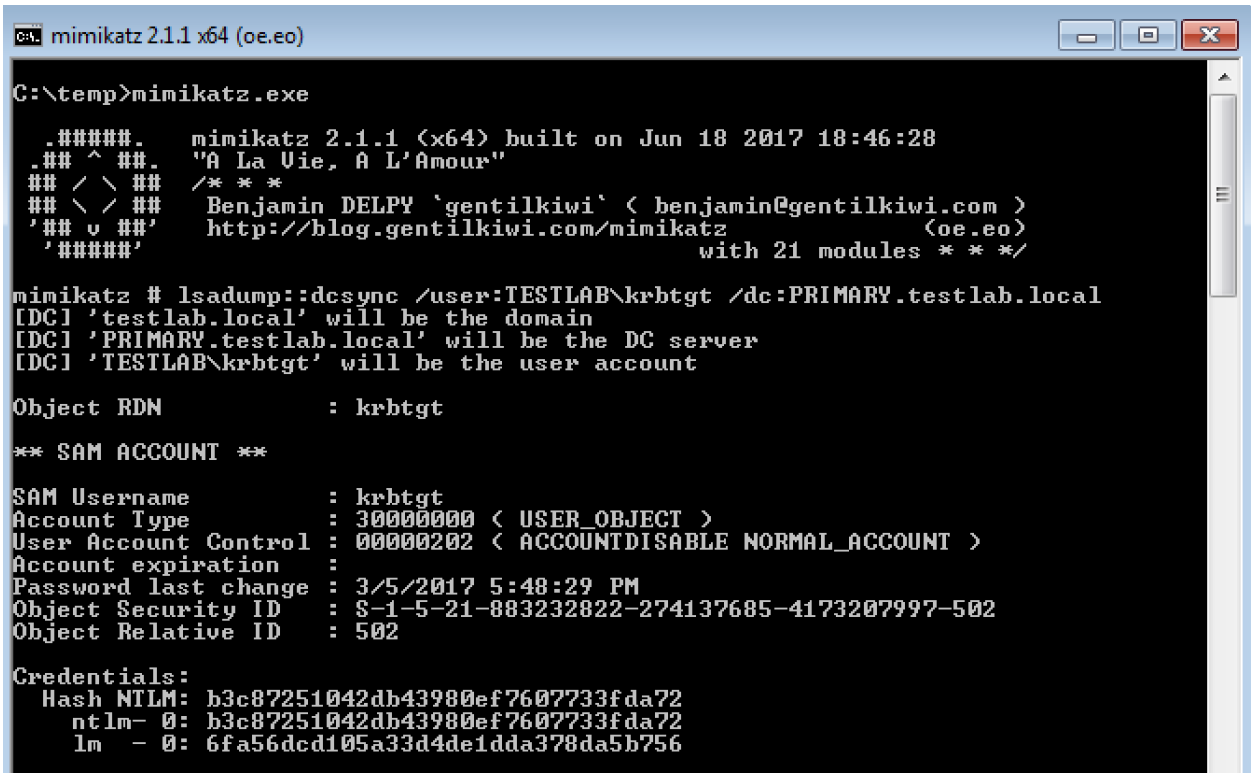
From here, the 'badguy' attacker uses Kekeo to request a ticket-granting-ticket (TGT) for the 'patsy' user, and then uses Kekeo's **tgs::s4u** module to request a ticket for LDAP service access to the domain controller through constrained delegation:

[71] http://www.microsoft.com/technet/prodtechnol/windowsserver2003/technologies/security/tkerberr.mspx



Above: Using Kekeo to inject a ticket for ldap/PRIMARY.testlab.local through constrained delegation.

And finally, with this ticket injected, DCSync for any user can be executed against the PRIMARY domain controller:



Above: DCSync is successfully executed against PRIMARY.testlab.local to retrieve the account hash for the krbtgt account.

More information on the specifics of this attack approach are covered here[72] and here[73].

[72] http://www.harmj0y.net/blog/activedirectory/the-most-dangerous-user-right-you-probably-have-never-heard-of/
[73] http://www.harmj0y.net/blog/activedirectory/s4u2pwnage/

Defenses

Some defenders may believe the detection of these types of backdoors is a lost cause, but several defensive approaches can help find these types of persistence. The primary method for detection and investigation remains properly tuned event logs for domain controllers. It is outside the scope of this paper to comprehensively cover event log tuning for these types of attacks, but the authors intend to perform follow up research on this for future publication. However, we will highlight some resources and approaches to facilitate defense. Note that we will not focus on how to "prevent your domain being owned," but rather the detection of the implementation steps of these backdoors.

One interesting defensive tool is the use of AD replication metadata. When a change is made to a domain object on a domain controller in AD, those changes are replicated to other domain controllers in the same domain (see the "Directory Replication" section here[74]). As part of the replication process, metadata about the replication is preserved in two constructed attributes (attributes where the end value is calculated from other attributes). These two properties are msDS-ReplAttributeMetaData (for normal attribute) and msDS-ReplValueMetaData (for linked attributes). The data in these two properties are stored as XML:

```
PS C:\Users\dfm.a\Desktop> Get-DomainUser harmj0y -Properties msDS-ReplAttribute
MetaData | Select-Object -ExpandProperty msds-replattributemetadata
<DS_REPL_ATTR_META_DATA>
        <pszAttributeName>lastLogonTimestamp</pszAttributeName>
        <dwVersion>7</dwVersion>
        <ftimeLastOriginatingChange>2017-06-05T05:20:06Z</ftimeLastOriginatingCh
ange>
        <uuidLastOriginatingDsaInvocationID>3f310a2d-7b38-4fdb-bf19-76cb7fe0b48b
</uuid
LastOriginatingDsaInvocationID>
        <usnOriginatingChange>102931</usnOriginatingChange>
        <usnLocalChange>102931</usnLocalChange>
        <pszLastOriginatingDsaDN>CN=NTDS Settings,CN=PRIMARY,CN=Servers,CN=Defau
lt-Fir
st-Site-Name,CN=Sites,CN=Configuration,DC=testlab,DC=local</pszLastOriginatingD
saDN>
</DS_REPL_ATTR_META_DATA>

<DS_REPL_ATTR_META_DATA>
        <pszAttributeName>objectCategory</pszAttributeName>
        <dwVersion>1</dwVersion>
        <ftimeLastOriginatingChange>2017-03-07T19:56:27Z</ftimeLastOriginatingCh
ange>
        <uuidLastOriginatingDsaInvocationID>3f310a2d-7b38-4fdb-bf19-76cb7fe0b48b
</uuid
LastOriginatingDsaInvocationID>
        <usnOriginatingChange>25630</usnOriginatingChange>
        <usnLocalChange>25630</usnLocalChange>
        <pszLastOriginatingDsaDN>CN=NTDS Settings,CN=PRIMARY,CN=Servers,CN=Defau
lt-Fir
st-Site-Name,CN=Sites,CN=Configuration,DC=testlab,DC=local</pszLastOriginatingD
saDN>
</DS_REPL_ATTR_META_DATA>
```

Above: Retrieving XML-formatted replication metadata for the 'harmj0y' user.

[74] https://msdn.microsoft.com/en-us/library/hh872004.aspx

You can see that we get an array of XML text blobs that describes the modification events. PowerView's brand new **Get-DomainObjectAttributeHistory** function will automatically query **msDS-ReplAttributeMetaData** for one or more objects and parse out the XML blobs into custom PSObjects:

```
PS C:\Users\dfm.a\Desktop> Get-DomainObjectAttributeHistory harmj0y


ObjectDN              : CN=harmj0y,CN=Users,DC=testlab,DC=local
AttributeName         : lastLogonTimestamp
LastOriginatingChange : 2017-06-05T05:20:06Z
Version               : 7
LastOriginatingDsaDN  : CN=NTDS Settings,CN=PRIMARY,CN=Servers,CN=Default-First
                        -Site-Name,CN=Sites,CN=Configuration,DC=testlab,DC=loca
                        l

ObjectDN              : CN=harmj0y,CN=Users,DC=testlab,DC=local
AttributeName         : objectCategory
LastOriginatingChange : 2017-03-07T19:56:27Z
Version               : 1
LastOriginatingDsaDN  : CN=NTDS Settings,CN=PRIMARY,CN=Servers,CN=Default-First
                        -Site-Name,CN=Sites,CN=Configuration,DC=testlab,DC=loca
                        l
```

Above: Using PowerView's Get-DomainObjectAttributeHistory function to retrieve parsed replication metadata for the 'harmj0y' user.

Metadata won't magically tell you an entire story, but it can start to point you in the right direction, with the bonus of being pre-existing functionality already present in your domain. For most triage situations, the process will be:

1. Use AD replication metadata to detect changes to objective properties that *might* indicate malicious behavior.
2. Collect detailed event logs from the domain controller linked to the change (as indicated by the metadata) in order to track down who performed the modification and what the value was changed to.

Example event log IDs of interest are 4735/4737/4755 for modifications to domain local, global, and universally scoped security groups and 4738 ("A user account was changed") for modification to specific user properties.

Also, system access control lists (SACLs), the other ACL component, are ripe for defensive use. SACLs can implement custom auditing of specific AD changes. We believe these have not been properly investigated because of the perceived difficulty in implementation for use at scale in enterprises, as well as the additional noise introduced. However, we believe that it may be possible to implement very specific SACLs for just the malicious primitives outlined in this paper, reducing the overhead of implementation, maintenance, and event noise. This is a future research area for the authors.

Future Research

Two DACL neutering approaches were investigated that proved unfruitful: setting a Null DACL on an object (which would effectively allow everyone all access) and flipping the SE_DACL_PRESENT header control bit in the security descriptor to produce a similar effect. Section 6.1.3[75] ("Security Descriptor Requirements") of the Active Directory Technical Specification ([MS-ADTS]) explicitly states that NULL DACLs are disallowed. In our testing, SE_DACL_PRESENT failed as well. Contradictory, though, the specification also outlines checking for NULL DACLs and the SE_DACL_PRESENT when evaluating access control (see [MS-ADTS] 5.1.3.3 "Checking Access"). These discrepancies warrant further investigation.

Another area of future research will be investigating methods of combatting the stealth primitives described. We believe, but have not properly tested, that there should be additional options to regain object access or enumerating DACLs with explicit "deny" rules if a defender is operating from an elevated context on a domain controller itself. We hope to discover how to detect and mitigate every backdoor action and stealth primitive described in this paper.

Related to this work on securable Active Directory objects, the authors have also started some research into host-based securable objects and their associated implications. We believe that security-descriptor-based backdoors added to host-based securable objects can be implemented in conjunction with AD-security-descriptor-based backdoors to create even more subtle attack scenarios.

Finally, as mentioned, defensive use of SACLs to detect the attacks described is another area of future work. In the future, the authors will produce complete guidance on performing this type of tuning and monitoring for even large enterprises.

[75] https://msdn.microsoft.com/en-us/library/cc223731.aspx

Conclusion

The Active Directory access control model is an untapped resource for covert persistence in a domain. As mentioned, a huge advantage of using this approach is that it's often difficult to tell if a security descriptor "misconfiguration" was implemented maliciously or implemented by accident. In addition, malicious security descriptor configurations are likely to survive operating system and domain functional level upgrades.

With a minimal amount of time and elevated domain credentials, the possibilities for creative security-descriptor-based domain backdoors are limited only by an attacker's imagination. By using the control relationship taxonomy outlined in this paper, combined with the stealth primitives of hiding object DACLs and/or principals, hard to triage backdoors of this nature can be obscured from defenders. The BloodHound analysis project can find some of these types of malicious attack chains, but this lacks the context of "who made this change." While defense certainly is possible in the form of properly tuned domain controller event log collection, this collection needs to be in place at the time the backdoor is implemented.

It is the authors' opinion that these types of backdoors have likely been implemented in the wild, but we have not found any public confirmation of this suspicion. Remember: "*As an offensive researcher, if you can dream it, someone has likely already done it…… and that someone isn't the kind of person who speaks at security cons.*"

Acknowledgements

We'd like to thank the following people for their previous work, guidance, and moral support during our research into this subject:

- Lucas Bouillot and Emmanuel Gras for Active Directory Control Paths
- Sean Metcalf for his work on Active Directory Security as well as the groundwork for several of the backdoor case studies
- Rohan Vazarkar for his great work on building and maintaining the BloodHound interface
- Matt Graeber for content review
- Jeff Dimmock for content review
- Jason Frank for content review
- Robin Granberg for his writings on sensitive Active Directory ACL entries