5

SECURITY DESCRIPTORS

In the last chapter we discussed the security access token, which describes the user's identity to the SRM. In this chapter, you'll learn how *security descriptors* define a resource's security. A security descriptor does several things. It specifies the owner of a resource, allowing the SRM to grant specific rights to users who are accessing their own data. It also contains the *discretionary access control (DAC)* and *mandatory access control (MAC)*, which grant or deny access to users and groups. Finally, it can contain entries that generate auditing events. Almost every kernel resource has a security descriptor, and user-mode applications can implement their own access control through security descriptors without needing to create a kernel resource.

Understanding the structure of security descriptors is crucial to understanding the security of Windows, as they're used to secure every kernel object and many user-mode components, such as services. You'll even find security descriptors used across network boundaries to secure remote resources. While developing a Windows application or researching Windows security, you'll inevitably have to inspect or create a security descriptor, so having a clear understanding of what a security descriptor contains will save you a lot of time. To help with this, I'll start by describing the structure of a security descriptor in more detail.

The Structure of a Security Descriptor

Windows stores security descriptors as binary structures on disk or in memory. While you'll rarely have to manually parse these structures, it's worth understanding what they contain. A security descriptor consists of the following seven components:

- The revision
- Optional resource manager flags
- Control flags
- An optional owner SID
- An optional group SID
- An optional discretionary access control list
- An optional system access control list

Let's look at each of these in turn. The first component of any security descriptor is the *revision*, which indicates the version of the security descriptor's binary format. There is only one version, so the revision is always set to the value 1. Next is an optional set of flags for use by a resource manager. You'll almost never encounter these flags being set; however, they are used by Active Directory, so we'll talk more about them in Chapter 11.

The resource manager flags are followed by a set of *control flags*. These have three uses: they define which optional components of the security descriptor are valid, how the security descriptors and components were created, and how to process the security descriptor when applying it to an object. Table 5-1 shows the list of valid flags and their descriptions. We'll cover many of the terms in this table, such as inheritance, in more detail in the following chapter.

Table 5-1: Valid Control Flags and Their Values and Descriptions

Name	Value	Description
`OwnerDefaulted`	`0x0001`	The owner SID was assigned through a default method.
`GroupDefaulted`	`0x0002`	The group SID was assigned through a default method.
`DaclPresent`	`0x0004`	The DACL is present in the security descriptor.
`DaclDefaulted`	`0x0008`	The DACL was assigned through a default method.
`SaclPresent`	`0x0010`	The SACL is present in the security descriptor.
`SaclDefaulted`	`0x0020`	The SACL was assigned through a default method.
`DaclUntrusted`	`0x0040`	When combined with `ServerSecurity`, the DACL is untrusted.
`ServerSecurity`	`0x0080`	The DACL is replaced with a server ACL (more on the use of this in Chapter 6).
`DaclAutoInheritReq`	`0x0100`	DACL auto-inheritance for child objects is requested.
`SaclAutoInheritReq`	`0x0200`	SACL auto-inheritance for child objects is requested.
`DaclAutoInherited`	`0x0400`	The DACL supports auto-inheritance.
`SaclAutoInherited`	`0x0800`	The SACL supports auto-inheritance.
`DaclProtected`	`0x1000`	The DACL is protected from inheritance.
`SaclProtected`	`0x2000`	The SACL is protected from inheritance.
`RmControlValid`	`0x4000`	The resource manager flags are valid.
`SelfRelative`	`0x8000`	The security descriptor is in a relative format.

After the control flags comes the *owner SID*, which represents the owner of the resource. This is typically the user's SID; however, ownership can also be assigned to a group, such as the *Administrators* group. Being the owner of a resource grants you certain privileges, including the ability to modify the resource's security descriptor. By ensuring the owner has this capability, the system prevents a user from locking themselves out of their own resources.

The *group SID* is like the owner SID, but it's rarely used. It exists primarily to ensure POSIX compatibility (a concern in the days when Windows still had a POSIX subsystem) and plays no part in access control for Windows applications.

The most important part of the security descriptor is the *discretionary access control list (DACL)*. The DACL contains a list of *access control entries (ACEs)*, which define what access a SID is given. It's considered *discretionary* because the user or system administrator can choose the level of access granted. There are many different types of ACEs. We'll discuss these further in "Access Control List Headers and Entries" on page XX; for now, you just need to know that the basic information in each ACE includes the following:

- The SID of the user or group to which the ACE applies
- The type of ACE
- The access mask to which the SID will be allowed or denied access

The final component of the security descriptor is the *security access control list (SACL)*, which stores auditing rules. Like the DACL, it contains a list of ACEs, but rather than determining access based on whether a defined SID matches the current user's, it determines the rules for generating audit events when the resource is accessed. Since Windows Vista, the SACL has also been the preferred location in which to store additional non-auditing ACEs, such as the resource's mandatory label.

Two final elements to point out in the DACL and SACL are the `DaclPresent` and `SaclPresent` control flags. These flags indicate that the DACL and SACL, respectively, are present in the security descriptor. Using flags allows for the setting of a *NULL ACL*, where the present flag is set but no value has been specified for the ACL field in the security descriptor. A NULL ACL indicates that no security for that ACL has been defined and causes the SRM to effectively ignore it. This is distinct from an empty ACL, where the present flag is set and a value for the ACL is specified but the ACL contains no ACEs.

The Structure of a SID

Until now, we've talked about SIDs as opaque binary values or strings of numbers. In this section, we'll look more closely at what a SID contains. The diagram in Figure 5-1 shows a SID as it's stored in memory.

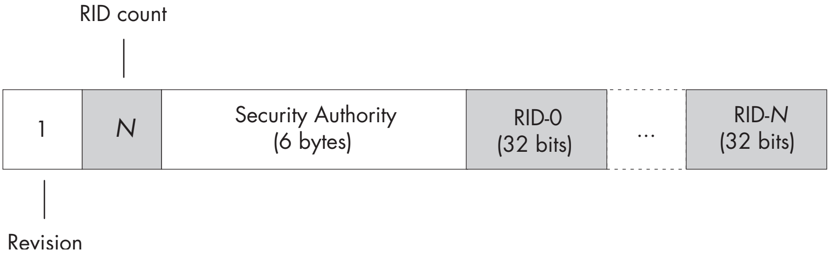


Figure 5-1: The security identifier (SID) structure in memory

There are four components to a binary SID:

Revision A value that is always set to `1`, as there is no other defined version number

Relative identifier count The number of RIDs in the SID

Security authority A value representing the party that issued the SID

Relative identifiers Zero or more 32-bit numbers that represent the user or group

The security authority can be any value, but Windows has predefined some commonly used ones. All well-known authorities start with five `0` bytes followed by a value from Table 5-2.

Table 5-2: Well-Known Authorities and Their Values

Name	Final value	Example name
`Null`	`0`	*NULL SID*
`World`	`1`	*Everyone*
`Local`	`2`	*CONSOLE LOGON*
`Creator`	`3`	*CREATOR OWNER*
`Nt`	`5`	*BUILTIN\Users*
`Package`	`15`	*APPLICATION PACKAGE AUTHORITY\Your internet connection*
`MandatoryLabel`	`16`	*Mandatory Label\Medium Mandatory Level*
`ScopedPolicyId`	`17`	*N/A*
`ProcessTrust`	`19`	*TRUST LEVEL\ProtectedLight-Windows*

After the security authority come the relative identifiers. A SID can contain one or more RIDs, with the domain RIDs followed by the user RIDs.

Let's walk through how the SID is constructed for a well-known group, *BUILTIN\Users*. Note that the domain component is separated from the group name with a backslash. In this case, the domain is *BUILTIN*. This is a predefined domain represented by a single RID, `32`. Listing 5-1 builds the domain SID for the *BUILTIN* domain from its components by using the `Get-NtSid` PowerShell command, then uses the `Get-NtSidName` command to retrieve the system-defined name for the SID.

```
PS> $domain_sid = Get-NtSid -SecurityAuthority Nt -RelativeIdentifier 32
PS> Get-NtSidName $domain_sid
Domain  Name    Source  NameUse Sddl
------  ----    ------  ------- ----
BUILTIN BUILTIN Account Domain  S-1-5-32
```

Listing 5-1: Querying for the BUILTIN *domain SID*

The *BUILTIN* domain's SID is a member of the `Nt` security authority. We specify this security authority using the `SecurityAuthority` parameter and specify the single RID using the `RelativeIdentifier` parameter.

We then pass the SID to the `Get-NtSidName` command. The first two columns of the output show the domain name and the name of the SID. In this case, those values are the same; this is just a quirk of the *BUILTIN* domain's registration.

The next column indicates the location from which the name was retrieved. In this example, the source, `Account`, indicates that the name was retrieved from LSASS. If the source were `WellKnown`, this would indicate that PowerShell knew the name ahead of time and didn't need to query LSASS.

The fourth column, `NameUse`, indicates the SID's type. In this case, it's `Domain`, which we might have expected. The final column is the SID in its SDDL format.

Any RIDs specified for SIDs following the domain SID identify a particular user or group. For the *Users* group, we use a single RID with the value `545` (predefined by Windows). Listing 5-2 creates a new SID by adding the `545` RID to the base domain's SID.

```
PS> $user_sid = Get-NtSid -BaseSid $domain_sid -RelativeIdentifier 545
PS> Get-NtSidName $user_sid
Domain  Name  Source  NameUse Sddl
------  ----  ------  ------- ----
BUILTIN Users Account Alias   S-1-5-32-545

PS> $user_sid.Name
BUILTIN\Users
```

Listing 5-2: Constructing a SID from a security authority and RIDs

The output now shows *Users* as the SID name. Also notice that `NameUse` in this case is set to `Alias`. This indicates that the SID represents a local, built-in group, as distinct from `Group`, which represents a user-defined group. When we print the `Name` property on the SID, it outputs the fully qualified name, with the domain and the name separated by a backslash.

You can find lists of known SIDs in Microsoft's technical documentation and on other websites. However, Microsoft sometimes adds SIDs without documenting them. Therefore, I encourage you to test multiple security authority and RID values to see what other users and groups you can find. Merely checking for different SIDs won't cause any damage. For example, try replacing the user RID in Listing 5-2 with `544`. This new SID represents the *BUILTIN\Administrators* group, as shown in Listing 5-3.

```
PS> Get-NtSid -BaseSid $domain_sid -RelativeIdentifier 32, 544
Name                   Sid
----                   ---
BUILTIN\Administrators S-1-5-32-544
```

Listing 5-3: Querying the Administrators *group SID using* `Get-NtSid`

Remembering the security authority and RIDs for a specific SID can be tricky, and you might not recall the exact name to query by using the `Name` parameter, as described in Chapter 2. Therefore, `Get-NtSid` implements a mode that can query a SID from a known set. For example, to query the SID of the *Administrators* group, you can use the command shown in Listing 5-4.

```
PS> Get-NtSid -KnownSid BuiltinAdministrators
Name                   Sid
----                   ---
BUILTIN\Administrators S-1-5-32-544
```

Listing 5-4: Querying the known Administrators *group SID using* `Get-NtSid`

You'll find SIDs used throughout the Windows operating system. It's crucial that you understand how they're structured, as this will allow you to quickly assess what a SID might represent. For example, if you identify a SID with the `Nt` security authority and its first RID is `32`, you can be sure it's representing a built-in user or group. Knowing the structure also allows you to identify and extract SIDs from crash dumps or memory in cases where better tooling isn't available.

Absolute and Relative Security Descriptors

The kernel supports two binary representation formats for security descriptors: absolute and relative. We'll examine both in this section, and consider the advantages and disadvantages of each.

Both formats start with the same three values: the revision, the resource manager flags, and the control flags. The `SelfRelative` flag in the control flags determines which format to use, as shown in Figure 5-2.

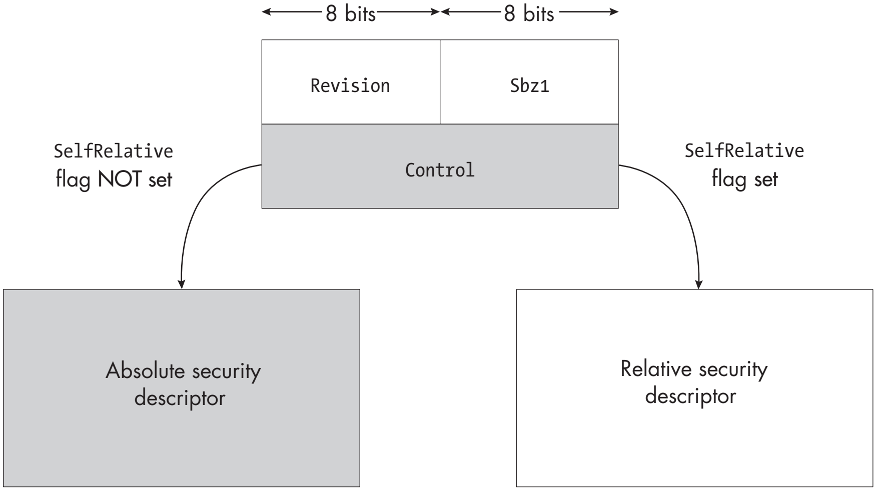


Figure 5-2: Selecting the security descriptor format based on the `SelfRelative` control flag

The total size of the security descriptor's header is 32 bits, split between two 8-bit values, the revision and `Sbz1`, and the 16-bit control flags. The security descriptor's resource manager flags are stored in `Sbz1`; these are only valid if the `RmControlValid` control flag is set, although the value will be present in either case. The rest of the security descriptor is stored immediately after the header.

The simplest format, the absolute security descriptor, is used when the `SelfRelative` flag is not set. After the common header, the absolute format defines four pointers to reference in memory: the owner SID, the group SID, the DACL, and the SACL, in that order, as shown in Figure 5-3.

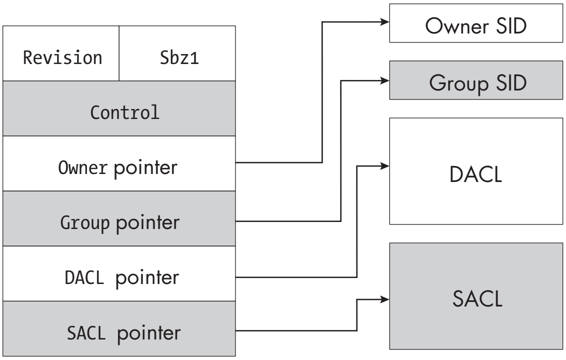


Figure 5-3: The structure of an absolute security descriptor

Each pointer references an absolute memory address at which the data is stored. The size of the pointer therefore depends on whether the application is 32- or 64-bit. It's also possible to specify a NULL value for the pointer to indicate that the value is not present. The owner and group SID values are stored using the binary format defined in the previous section.

When the `SelfRelative` flag is set, the security descriptor instead follows the relative format. Instead of referencing its values using absolute memory addresses, a relative security descriptor stores these locations as positive offsets relative to the start of its header. Figure 5-4 shows how a relative security descriptor is constructed.

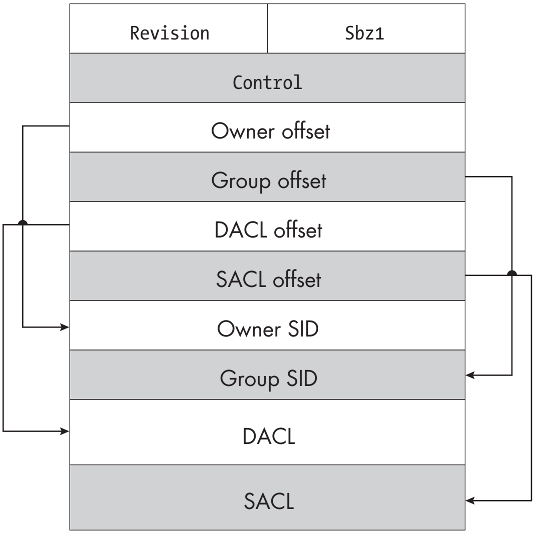


Figure 5-4: The structure of a relative security descriptor

These values are stored in contiguous memory. The ACL format, which we'll explore in the following section, is already a relative format and therefore doesn't require any special handling when used in a relative security descriptor. Each offset is always 32 bits long, regardless of the system's bit size. If an offset is set to 0, the value doesn't exist, as in the case of NULL for an absolute security descriptor.

The main advantage of an absolute security descriptor is that you can easily update its individual components. For example, to replace the owner SID, you'd allocate a new SID in memory and assign its memory address to the owner pointer. In comparison, modifying a relative security descriptor in the same way might require adjusting its allocated memory if the new owner SID structure is larger than the old one.

On the other hand, the big advantage of a relative security descriptor is that it can be built in a single contiguous block of memory. This allows you to serialize the security descriptor to a persistent format, such as a file or a registry key. When you're trying to determine the security of a resource, you might need to extract its security descriptor from memory or a persistent store. By understanding the two formats, you can determine how to read the security descriptor into something you can view or manipulate.

Most APIs and system calls accept either security descriptor format, determining how to handle a security descriptor automatically by checking the value of the SelfRelative flag. However, you'll find some exceptions in which an API takes only one format or another; in that case, if you pass the API a security descriptor in the wrong format, you'll typically receive an error such as STATUS_INVALID_SECURITY_DESCR. Security descriptors returned from an API will almost always be in relative format due to the simplicity of their memory management. The system provides the APIs RtlAbsoluteToSelfRelativeSD and RtlSelfRelativeToAbsoluteSD to convert between the two formats if needed.

The PowerShell module handles all security descriptors using a SecurityDescriptor object, regardless of format. This object is written in .NET and converts to a relative or absolute security descriptor only when it's required to interact with native code. You can determine whether a SecurityDescriptor object was generated from a relative security descriptor by inspecting the SelfRelative property.

Access Control List Headers and Entries

The DACL and SACL make up most of the data in a security descriptor. While these elements have different purposes, they share the same basic structure. In this section we'll cover how they're arranged in memory, leaving the details of how they contribute to the access check process to Chapter 6.

The Header

All ACLs consist of an ACL header followed by a list of zero or more ACEs in one contiguous block of memory. Figure 5-5 shows this top-level format.

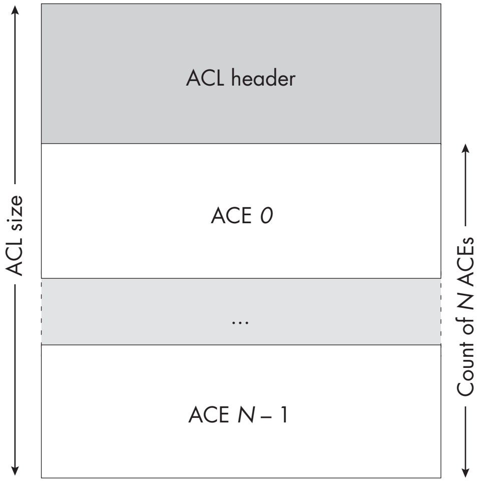


Figure 5-5: A top-level overview of the ACL structure

The ACL header contains a revision, the total size of the ACL in bytes, and the number of ACE entries that follow the header. Figure 5-6 shows the header structure.

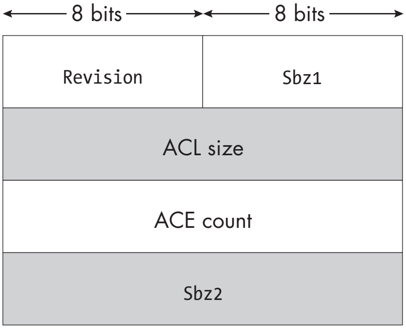


Figure 5-6: The structure of the ACL header

The ACL header also contains two reserved fields, `Sbz1` and `Sbz2`, both of which should always be `0`. They serve no purpose in modern versions of Windows and are there in case the ACL structure needs to be extended. Currently, the `Revision` field can have one of three values, which determine the ACL's valid ACEs. If an ACL uses an ACE that the revision doesn't support, the ACL won't be considered valid. Windows supports the following revisions:

Default The default ACL revision. Supports all the basic ACE types, such as `Allowed` and `Denied`. Specified with the `Revision` value `2`.

Compound Adds support for compound ACEs to the default ACL revision. Specified with the `Revision` value `3`.

Object Adds support for object ACEs to the compound. Specified with the `Revision` value `4`.

The ACE List

Following the ACL header is the list of ACEs, which determines what access the SID has. ACEs are of variable length but always start with a header that contains the ACE type, additional flags, and the ACE's total size. The header is followed by data specific to the ACE type. Figure 5-7 shows this structure.

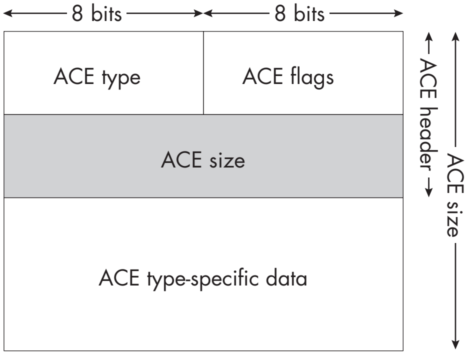


Figure 5-7: The ACE structure

The ACE header is common to all ACE types. This allows an application to safely access the header when processing an ACL. The ACE type value can then be used to determine the exact format of the ACE's type-specific data. If the application doesn't understand the ACE type, it can use the size field to skip the ACE entirely (we'll discuss how types affect access checking in Chapter 7).

Table 5-3 lists the supported ACE types, the minimum ACE revision they are valid in, and whether they are valid in the DACL or the SACL.

Table 5-3: Supported ACE Types, Minimum ACL Revisions, and Locations

ACE type	Value	Minimum revision	ACL	Description
`Allowed`	0x0	Default	DACL	Grants access to a resource
`Denied`	0x1	Default	DACL	Denies access to a resource
`Audit`	0x2	Default	SACL	Audits access to a resource
`Alarm`	0x3	Default	SACL	Alarms upon access to a resource; unused
`AllowedCompound`	0x4	Compound	DACL	Grants access to a resource during impersonation
`AllowedObject`	0x5	Object	DACL	Grants access to a resource with an object type
`DeniedObject`	0x6	Object	DACL	Denies access to a resource with an object type
`AuditObject`	0x7	Object	SACL	Audits access to a resource with an object type
`AlarmObject`	0x8	Object	SACL	Alarms upon access with an object type; unused
`AllowedCallback`	0x9	Default	DACL	Grants access to a resource with a callback
`DeniedCallback`	0xA	Default	DACL	Denies access to a resource with a callback
`AllowedCallbackObject`	0xB	Object	DACL	Grants access with a callback and an object type
`DeniedCallbackObject`	0xC	Object	DACL	Denies access with a callback and an object type
`AuditCallback`	0xD	Default	SACL	Audits access with a callback
`AlarmCallback`	0xE	Default	SACL	Alarms upon access with a callback; unused
`AuditCallbackObject`	0xF	Object	SACL	Audits access with a callback and an object type
`AlarmCallbackObject`	0x10	Object	SACL	Alarms upon access with a callback and an object type; unused
`MandatoryLabel`	0x11	Default	SACL	Specifies a mandatory label
`ResourceAttribute`	0x12	Default	SACL	Specifies attributes for the resource
`ScopedPolicyId`	0x13	Default	SACL	Specifies a central access policy ID for the resource
`ProcessTrustLabel`	0x14	Default	SACL	Specifies a process trust label to limit resource access
`AccessFilter`	0x15	Default	SACL	Specifies an access filter for the resource

While Windows officially supports all these ACE types, the kernel does not use the `Alarm` types. User applications can specify their own ACE types, but various APIs in user and kernel mode check for valid types and will generate an error if the ACE type isn't known.

An ACE's type-specific data falls primary into one of three formats: normal ACEs, such as `Allowed` and `Denied`; compound ACEs; and object ACEs. A *normal ACE* contains the following fields after the header, with the field's size indicated in parentheses:

Access mask (32-bit) The access mask to be granted or denied based on the ACE type

SID (variable size) The SID, in the binary format described earlier in this chapter

Compound ACEs are for use during impersonation. These ACEs can grant access to both the impersonated caller and the process user at the same time. The only valid type for them is `AllowedCompound`. Even though the latest versions of Windows still support compound ACEs, they're effectively undocumented and presumably deprecated. I've included them in this book for completeness. Their format is as follows:

Access mask (32-bit) The access mask to be granted

Compound ACE type (16-bit) Set to `1`, which means the ACE is used for impersonation

Reserved (16-bit) Always `0`

Server SID (variable size) The server SID in binary format; matches the service user

SID (variable size) The SID in a binary format; matches the impersonated user

Microsoft introduced the *object ACE* format to support access control for Active Directory Services. Active Directory uses a 128-bit GUID to represent a directory service object type; the object ACE determines access for specific types of objects, such as computers or users. For example, using a single security descriptor, a directory could grant a SID the access needed to create one type of object but not another. The object ACE format is as follows:

Access mask (32-bit) The access mask to be granted or denied based on the ACE type

Flags (32-bit) Used to indicate which of the following GUIDs are present

Object type (16-byte) The object type GUID; present only if the flag in bit 0 is set

Inherited object type (16-byte) The inherited object GUID; present only if the flag in bit 1 is set

SID (variable size) The SID in a binary format

ACEs can be larger than their types' defined structures, and they may use additional space to stored unstructured data. Most commonly, they use this unstructured data for the callback ACE types, such as `AllowedCallback`, which defines a conditional expression that determines whether the ACE should be active during an access check. We can inspect the data that would be generated from a conditional expression using the `ConvertFrom-NtAceCondition` PowerShell command, as shown in Listing 5-5.

```
PS> ConvertFrom-NtAceCondition 'WIN://TokenId == "XYZ"' | Out-HexDump -ShowAll
          00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  - 0123456789ABCDEF
-----------------------------------------------------------------------------
00000000: 61 72 74 78 F8 1A 00 00 00 57 00 49 00 4E 00 3A  - artx.....W.I.N.:
00000010: 00 2F 00 2F 00 54 00 6F 00 6B 00 65 00 6E 00 49  - ./././.T.o.k.e.n.I
00000020: 00 64 00 10 06 00 00 00 58 00 59 00 5A 00 80 00  - .d......X.Y.Z...
```

Listing 5-5: Parsing a conditional expression and displaying binary data

We refer to these ACEs as *callback ACEs* because prior to Windows 8 an application needed to call the `AuthzAccessCheck` API to handle them. The API accepted a callback function that would be invoked to determine whether to include a callback ACE in the access check. Since Windows 8, the kernel access check has built-in support for conditional ACEs in the format shown in Listing 5-5, although user applications are free to specify their own formats and handle these ACEs manually.

The primary use of the ACE flags is to specify inheritance rules for the ACE. Table 5-4 shows the defined ACE flags.

Table 5-4: ACE Flags with Values and Descriptions

ACE flag	Value	Description
`ObjectInherit`	`0x1`	The ACE can be inherited by an object.
`ContainerInherit`	`0x2`	The ACE can be inherited by a container.
`NoPropagateInherit`	`0x4`	The ACE's inheritance flags are not propagated to children.
`InheritOnly`	`0x8`	The ACE is used only for inheritance, and not for access checks.
`Inherited`	`0x10`	The ACE was inherited from a parent container.
`Critical`	`0x20`	The ACE is critical and can't be removed. Applies only to allow ACEs.
`SuccessfulAccess`	`0x40`	An audit event should be generated for a successful access.
`FailedAccess`	`0x80`	An audit event should be generated for a failed access.
`TrustProtected`	`0x40`	When used with an `AccessFilter` ACE, this flag prevents modification.

The inheritance flags take up only the lower 5 bits, leaving the top 3 bits for ACE-specific flags.

Constructing and Manipulating Security Descriptors

Now that you're familiar with the structure of a security descriptor, let's look at how to construct and manipulate them using PowerShell. By far the most common reason to do this is to view a security descriptor's contents so you can understand the access applied to a resource. Another important use case is if you need to construct a security descriptor to lock down a resource. The PowerShell module used in this book aims to make constructing and viewing security descriptors as simple as possible.

Creating a New Security Descriptor

To create a new security descriptor, you can use the `New-NtSecurityDescriptor` command. By default, it creates a new `SecurityDescriptor` object with no owner, group, DACL, or SACL set. You can use the command's parameters to add these parts of the security descriptor, as shown in Listing 5-6.

```
PS> $world = Get-NtSid -KnownSid World
PS> $sd = New-NtSecurityDescriptor -Owner $world -Group $world -Type File
PS> $sd | Format-Table
Owner    DACL ACE Count SACL ACE Count Integrity Level
-----    -------------- -------------- ---------------
Everyone NONE           NONE           NONE
```

Listing 5-6: Creating a new security descriptor with a specified owner

We first get the SID for the *World* group. When calling `New-NtSecurityDescriptor` to create a new security descriptor, we use this SID to specify its `Owner` and `Group`. We also specify the name of the kernel object type this security descriptor will be associated with; this step makes some of the later commands easier to use. In this case, we'll assume it's a `File` object's security descriptor.

We then display the security descriptor, formatting the output as a table. As you can see, the `Owner` field is set to `Everyone`. The `Group` value isn't printed by default, as it's not as important. Neither a DACL nor a SACL is currently present in the security descriptor, and there is no integrity level specified.

To add some ACEs, we can use the `Add-NtSecurityDescriptorAce` command. For normal ACEs, we need to specify the ACE type, the SID, and the access mask. Optionally, we can also specify the ACE flags. The script in Listing 5-7 adds some ACEs to our new security descriptor.

```
❶ PS> $user = Get-NtSid
❷ PS> Add-NtSecurityDescriptorAce $sd -Sid $user -Access WriteData, ReadData
  PS> Add-NtSecurityDescriptorAce $sd -KnownSid Anonymous -Access GenericAll
  -Type Denied
  PS> Add-NtSecurityDescriptorAce $sd -Name "Everyone" -Access ReadData
❸ PS> Add-NtSecurityDescriptorAce $sd -KnownSid World -Access Delete
  -Type Audit -Flags FailedAccess
❹ PS> Set-NtSecurityDescriptorIntegrityLevel $sd Low
❺ PS> Set-NtSecurityDescriptorControl $sd DaclAutoInherited, SaclProtected
```

```
❻ PS> $sd | Format-Table
  Owner    DACL ACE Count SACL ACE Count Integrity Level
  -----    -------------- -------------- ---------------
  Everyone 3              2              Low

❼ PS> Get-NtSecurityDescriptorControl $sd
  DaclPresent, SaclPresent, DaclAutoInherited, SaclProtected

❽ PS> Get-NtSecurityDescriptorDacl $sd | Format-Table
  Type    User                         Flags Mask
  ----    ----                         ----- ----
  Allowed GRAPHITE\user                None  00000003
  Denied  NT AUTHORITY\ANONYMOUS LOGON None  10000000
  Allowed Everyone                     None  00000001

❾ PS> Get-NtSecurityDescriptorSacl $sd | Format-Table
  Type           User                               Flags        Mask
  ----           ----                               -----        ----
  Audit          Everyone                           FailedAccess 00010000
  MandatoryLabel Mandatory Label\Low Mandatory Level None         00000001
```

Listing 5-7: Adding ACEs to the new security descriptor

We start by getting the SID of the current user with `Get-NtSid` ❶. We use this SID to add a new `Allowed` ACE to the DACL ❷. We also add a `Denied` ACE for the anonymous user by specifying the `Type` parameter, followed by another `Allowed` ACE for the *Everyone* group. We then modify the SACL to add an audit ACE ❸ and set the mandatory label to the `Low` integrity level ❹. To finish creating the security descriptor, we set the `DaclAutoInherited` and `SaclProtected` control flags ❺.

We can now print details about the security descriptor we've just created. Displaying the security descriptor ❻ shows that the DACL now contains three ACEs and the SACL two, and the integrity level is `Low`. We also display the control flags ❼ and the lists of ACEs in the DACL ❽ and SACL ❾.

Ordering the ACEs

Because of how access checking works, there is a canonical ordering to the ACEs in an ACL. For example, all `Denied` ACEs should come before any `Allowed` ACEs, as otherwise the system might grant access to a resource improperly, based on which ACEs come first. The SRM doesn't enforce this canonical ordering; it trusts that any application has correctly ordered the ACEs before passing them for an access check. ACLs should order their ACEs according to the following rules:

1. All `Denied`-type ACEs must come before `Allowed` types.
2. The `Allowed` ACEs must come before `Allowed` object ACEs.
3. The `Denied` ACEs must come before `Denied` object ACEs.
4. All non-inherited ACEs must come before ACEs with the `Inherited` flag set.

In Listing 5-7, we added a `Denied` ACE to the DACL after we added an `Allowed` ACE, failing the first order rule. We can ensure the DACL is canonicalized by using the `Edit-NtSecurity` command with the `CanonicalizeDacl` parameter. We can also test whether a DACL is already canonical by using the `Test-NtSecurityDescriptor` PowerShell command with the `DaclCanonical` parameter. Listing 5-8 illustrates the use of both commands.

```
PS> Test-NtSecurityDescriptor $sd -DaclCanonical
False

PS> Edit-NtSecurityDescriptor $sd -CanonicalizeDacl
PS> Test-NtSecurityDescriptor $sd -DaclCanonical
True

PS> Get-NtSecurityDescriptorDacl $sd | Format-Table
Type    User                         Flags Mask
----    ----                         ----- ----
Denied  NT AUTHORITY\ANONYMOUS LOGON None  10000000
Allowed GRAPHITE\user                None  00000003
Allowed Everyone                     None  00000001
```

Listing 5-8: Canonicalizing the DACL

If you compare the list of ACEs in Listing 5-8 with the list in Listing 5-7, you'll notice that the `Denied` ACE has been moved from the middle to the start of the ACL. This ensures that it will be processed before any `Allowed` ACEs.

Formatting Security Descriptors

You can print the values in the security descriptor manually, though the `Format-Table` command, but this is time-consuming. Another problem with manual formatting is that the access masks won't be decoded, so instead of `ReadData`, for example, you'll see `00000001`. It would be nice to have a simple way of printing out the details of a security descriptor and formatting them based on the object type. That's what `Format-NtSecurityDescriptor` is for. You can pass it a security descriptor, and the command will print it to the console. Listing 5-9 provides an example.

```
PS> Format-NtSecurityDescriptor $sd -ShowAll
Type: File
Control: DaclPresent, SaclPresent

<Owner>
 - Name  : Everyone
 - Sid   : S-1-1-0

<Group>
 - Name  : Everyone
 - Sid   : S-1-1-0
```

```
<DACL> (Auto Inherited)
- Type   : Denied
 - Name  : NT AUTHORITY\ANONYMOUS LOGON
 - SID   : S-1-5-7
 - Mask  : 0x10000000
 - Access: GenericAll
 - Flags : None

 - Type  : Allowed
 - Name  : GRAPHITE\user
 - SID   : S-1-5-21-2318445812-3516008893-216915059-1002
 - Mask  : 0x00000003
 - Access: ReadData|WriteData
 - Flags : None

 - Type  : Allowed
 - Name  : Everyone
 - SID   : S-1-1-0
 - Mask  : 0x00000001
 - Access: ReadData
 - Flags : None

<SACL> (Protected)
 - Type  : Audit
 - Name  : Everyone
 - SID   : S-1-1-0
 - Mask  : 0x00010000
 - Access: Delete
 - Flags : FailedAccess

<Mandatory Label>
 - Type  : MandatoryLabel
 - Name  : Mandatory Label\Low Mandatory Level
 - SID   : S-1-16-4096
 - Mask  : 0x00000001
 - Policy: NoWriteUp
 - Flags : None
```

Listing 5-9: Displaying the security descriptor using `Format-NtSecurityDescriptor`

We pass the `ShowAll` parameter to `Format-NtSecurityDescriptor` to ensure that it displays the entire contents of the security descriptor; by default it won't output the SACL or less common ACEs, such as `ResourceAttribute`. Note that the output kernel object type matches the `File` type we specified when creating the security descriptor in Listing 5-6. Specifying the kernel object type allows the formatter to print the decoded access mask for the type rather than a generic hex value.

The next line in the output shows the current control flags. These are calculated on the fly based on the current state of the security descriptor; later, we'll discuss how to change these control flags to change the security descriptor's behavior. The control flags are followed by the owner and group SIDs and the DACL, which account for most of the output. Any

DACL-specific flags appear next to the header; in this case, these indicate that we set the DaclAutoInherited flag. Next, the output lists each of the ACEs in the ACL in order, starting with the type of ACE. Because the command knows the object type, it prints the decoded access mask for the type as well as the original access mask in hexadecimal.

Next is the SACL, which shows our single audit ACE as well as the SaclProtected flag. The final component shown is the mandatory label. The access mask for a mandatory label is the mandatory policy, and it's decoded differently from the rest of the ACEs that use the type-specific access rights. The mandatory policy can be set to one or more of the bit flags shown in Table 5-5.

Table 5-5: Mandatory Policy Values

Name	Value	Description
NoWriteUp	0x00000001	A lower integrity level caller can't write to this resource.
NoReadUp	0x00000002	A lower integrity level caller can't read this resource.
NoExecuteUp	0x00000004	A lower integrity level caller can't execute this resource.

By default, Format-NtSecurityDescriptor can be a bit verbose. To shorten its output, specify the Summary parameter, which will remove as much data as possible while keeping the important information. Listing 5-10 demonstrates.

```
PS> Format-NtSecurityDescriptor $sd -ShowAll -Summary
<Owner> : Everyone
<Group> : Everyone
<DACL>
<DACL> (Auto Inherited)
NT AUTHORITY\ANONYMOUS LOGON: (Denied)(None)(GenericAll)
GRAPHITE\user: (Allowed)(None)(ReadData|WriteData)
Everyone: (Allowed)(None)(ReadData)
<SACL> (Protected)
Everyone: (Audit)(FailedAccess)(Delete)
<Mandatory Label>
Mandatory Label\Low Mandatory Level: (MandatoryLabel)(None)(NoWriteUp)
```

Listing 5-10: Displaying the security descriptor in summary format

I mentioned in Chapter 2 that for ease of use the PowerShell module used in this book uses simple names for most common flags, but that you can display the full SDK names if you prefer (for example, to compare the output with native code). To display SDK names when viewing the contents of a security descriptor with Format-NtSecurityDescriptor, use the SDKName property, as shown in Listing 5-11.

```
PS> Format-NtSecurityDescriptor $sd -SDKName -SecurityInformation Dacl
Type: File
Control: SE_DACL_PRESENT|SE_SACL_PRESENT|SE_DACL_AUTO_INHERITED|SE_SACL_PROTECTED
<DACL> (Auto Inherited)
 - Type  : ACCESS_DENIED_ACE_TYPE
 - Name  : NT AUTHORITY\ANONYMOUS LOGON
 - SID   : S-1-5-7
 - Mask  : 0x10000000
 - Access: GENERIC_ALL
 - Flags : NONE

 - Type  : ACCESS_ALLOWED_ACE_TYPE
 - Name  : GRAPHITE\user
 - SID   : S-1-5-21-2318445812-3516008893-216915059-1002
 - Mask  : 0x00000003
 - Access: FILE_READ_DATA|FILE_WRITE_DATA
 - Flags : NONE

 - Type  : ACCESS_ALLOWED_ACE_TYPE
 - Name  : Everyone
 - SID   : S-1-1-0
 - Mask  : 0x00000001
 - Access: FILE_READ_DATA
 - Flags : NONE
```

Listing 5-11: Formatting a security descriptor with SDK names

One quirk of `File` objects is that their access masks have two naming conventions, one for files and one for directories. You can request that `Format-NtSecurityDescriptor` print the directory version of the access mask by using the `Container` parameter, or more generally, by setting the `Container` property of the security descriptor object to `True`. Listing 5-12 shows the impact of setting the `Container` parameter on the output.

```
PS> Format-NtSecurityDescriptor $sd -ShowAll -Summary -Container
<Owner> : Everyone
<Group> : Everyone
<DACL>
NT AUTHORITY\ANONYMOUS LOGON: (Denied)(None)(GenericAll)
❶ GRAPHITE\user: (Allowed)(None)(ListDirectory|AddFile)
Everyone: (Allowed)(None)(ListDirectory)
--snip--
```

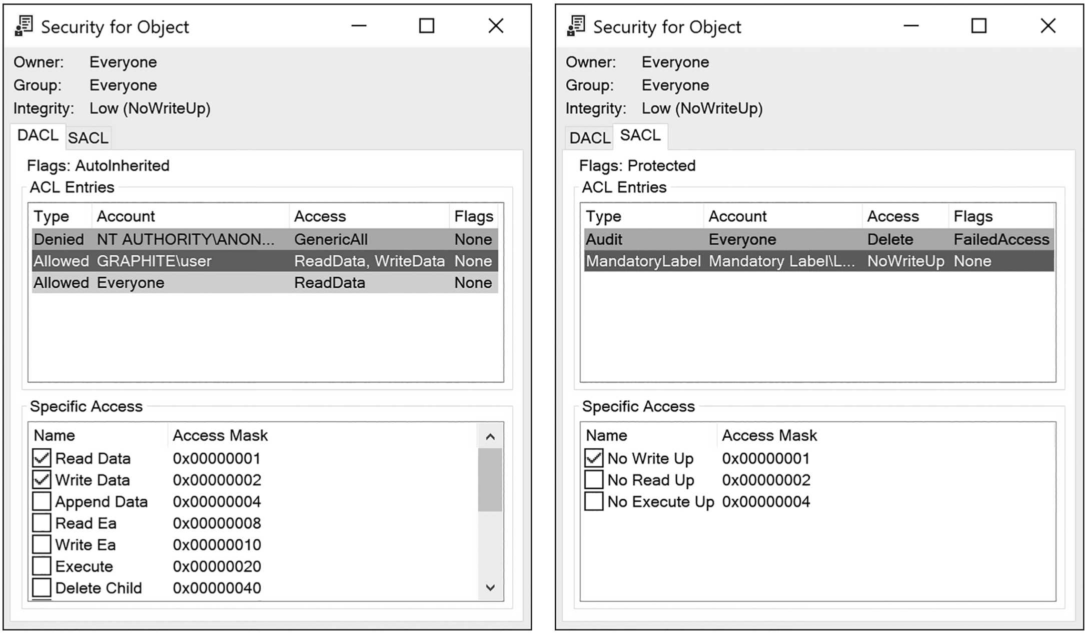
Listing 5-12: Formatting the security descriptor as a container

Note how the output line changes from `ReadData|WriteData` to `ListDirectory|AddFile` ❶ when we format it as a container. The `File` type is the only object type with this behavior in Windows. This is important to security, as you could easily misinterpret `File` access rights if you formatted the security descriptor for a directory as a file, or vice versa.

If a GUI is more your thing, you can start a viewer using the following `Show-NtSecurityDescriptor` command:

```
PS> Show-NtSecurityDescriptor $sd
```

Running the command should open the dialog shown in Figure 5-8.



Owner: Everyone
Group: Everyone
Integrity: Low (NoWriteUp)

Flags: AutoInherited

Type	Account	Access	Flags
Denied	NT AUTHORITY\ANON...	GenericAll	None
Allowed	GRAPHITE\user	ReadData, WriteData	None
Allowed	Everyone	ReadData	None

Name	Access Mask
☑ Read Data	0x00000001
☑ Write Data	0x00000002
□ Append Data	0x00000004
□ Read Ea	0x00000008
□ Write Ea	0x00000010
□ Execute	0x00000020
□ Delete Child	0x00000040

Owner: Everyone
Group: Everyone
Integrity: Low (NoWriteUp)

Flags: Protected

Type	Account	Access	Flags
Audit	Everyone	Delete	FailedAccess
MandatoryLabel	Mandatory Label\L...	NoWriteUp	None

Name	Access Mask
☑ No Write Up	0x00000001
□ No Read Up	0x00000002
□ No Execute Up	0x00000004

Figure 5-8: A GUI displaying the security descriptor

The dialog summarizes the security descriptor's important data. At the top are the owner and group SIDs resolved into names, as well as the security descriptor's integrity level and mandatory policy. These match the values we specified when creating the security descriptor. In the middle is the list of ACEs in the DACL (left) or SACL (right), depending on which tab you select, with the ACL flags at the top. Each entry in the list includes the type of ACE, the SID, the access mask in generic form, and the ACE flags. At the bottom is the decoded access. The list populates when you select an ACE in the ACL list.

Converting to and from a Relative Security Descriptor

We can convert a security descriptor object to a byte array in the relative format using the `ConvertFrom-NtSecurityDescriptor` command. We can then print its contents to see what the underlying structure really is, as shown in Listing 5-13.

```
PS> $ba = ConvertFrom-NtSecurityDescriptor $sd
PS> $ba | Out-HexDump -ShowAll
          00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  - 0123456789ABCDEF
-----------------------------------------------------------------------------
```

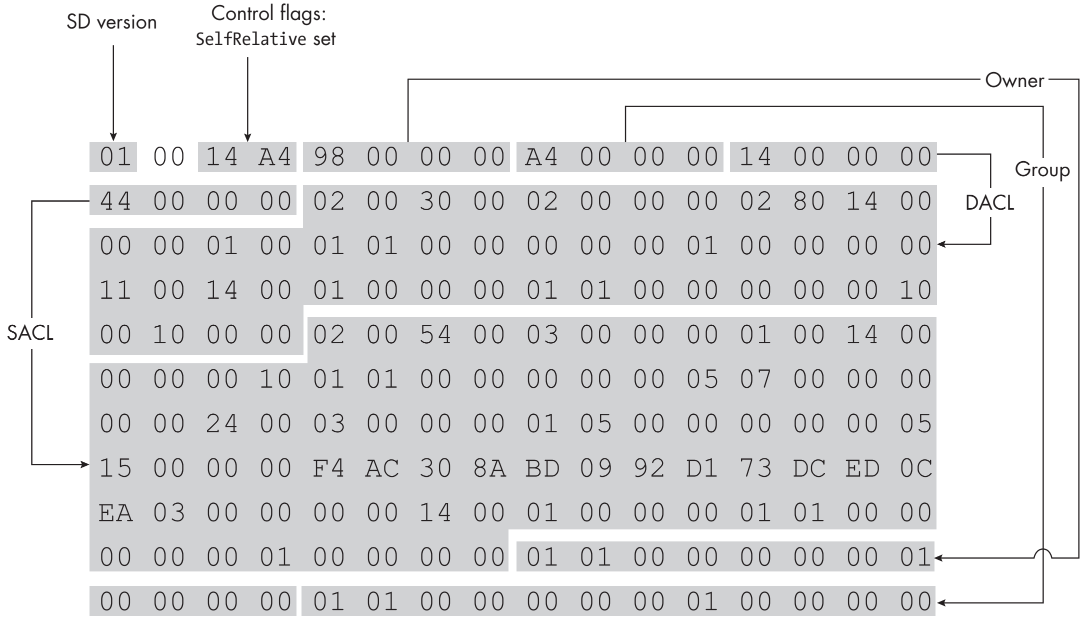
```
00000000: 01 00 14 A4 98 00 00 00 A4 00 00 00 14 00 00 00  - ................
00000010: 44 00 00 00 02 00 30 00 02 00 00 00 02 80 14 00  - D.....0.........
00000020: 00 00 01 00 01 01 00 00 00 00 00 01 00 00 00 00  - ................
00000030: 11 00 14 00 01 00 00 00 01 01 00 00 00 00 00 10  - ................
00000040: 00 10 00 00 02 00 54 00 03 00 00 00 01 00 14 00  - ......T.........
00000050: 00 00 00 10 01 01 00 00 00 00 00 05 07 00 00 00  - ................
00000060: 00 00 24 00 03 00 00 00 01 05 00 00 00 00 00 05  - ..$.............
00000070: 15 00 00 00 F4 AC 30 8A BD 09 92 D1 73 DC ED 0C  - ......0.....s...
00000080: EA 03 00 00 00 00 14 00 01 00 00 00 01 01 00 00  - ................
00000090: 00 00 00 01 00 00 00 00 01 01 00 00 00 00 00 01  - ................
000000A0: 00 00 00 00 01 01 00 00 00 00 00 01 00 00 00 00  - ................
```

Listing 5-13: Converting an absolute security descriptor to relative format and displaying its bytes

We can convert the byte array back to a security descriptor object using New-NtSecurityDescriptor and the Byte parameter:

```
PS> New-NtSecurityDescriptor -Byte $ba
```

As an exercise, I'll leave it to you to pick apart the hex output to find the various structures of the security descriptor based on the descriptions provided in this chapter. To get you started, Figure 5-9 highlights the major structures.



```
01 00 14 A4 98 00 00 00 A4 00 00 00 14 00 00 00
44 00 00 00 02 00 30 00 02 00 00 00 02 80 14 00
00 00 01 00 01 01 00 00 00 00 00 01 00 00 00 00
11 00 14 00 01 00 00 00 01 01 00 00 00 00 00 10
00 10 00 00 02 00 54 00 03 00 00 00 01 00 14 00
00 00 00 10 01 01 00 00 00 00 00 05 07 00 00 00
00 00 24 00 03 00 00 00 01 05 00 00 00 00 00 05
15 00 00 00 F4 AC 30 8A BD 09 92 D1 73 DC ED 0C
EA 03 00 00 00 00 14 00 01 00 00 00 01 01 00 00
00 00 00 01 00 00 00 00 01 01 00 00 00 00 00 01
00 00 00 00 01 01 00 00 00 00 00 01 00 00 00 00
```

Figure 5-9: An outline of the major structures in the relative security descriptor hex output

You'll need to refer to the layout of the ACL and SID structures to manually decode the rest.

The Security Descriptor Definition Language

In Chapter 2, we discussed the basics of the Security Descriptor Definition Language (SDDL) format for representing SIDs. The SDDL format can represent the entire security descriptor, too. As the SDDL version of a security descriptor uses ASCII text, it's somewhat human readable, and unlike the binary data shown in Listing 5-13, it can be easily copied. Because it's common to see SDDL strings used throughout Windows, let's look at how to represent a security descriptor in SDDL and how you can read it.

You can convert a security descriptor to SDDL format by specifying the `ToSddl` parameter to `Format-NtSecurityDescriptor`. This is demonstrated in Listing 5-14, where we pass the security descriptor we built in the previous section. You can also create a security descriptor from an SDDL string using `New-NtSecurityDescriptor` with the `-Sddl` parameter.

```
PS> $sddl = Format-NtSecurityDescriptor $sd -ToSddl -ShowAll
PS> $sddl
O:WDG:WDD:AI(D;;GA;;;AN)(A;;CCDC;;;S-1-5-21-2318445812-3516008893-216915059-
1002)(A;;CC;;;WD)S:P(AU;FA;SD;;;WD)(ML;;NW;;;LW)
```

Listing 5-14: Converting a security descriptor to SDDL

The SDDL version of the security descriptor contains four optional components. You can identify the start of each component by looking for the following prefixes:

`O:`	Owner SID
`G:`	Group SID
`D:`	DACL
`S:`	SACL

In Listing 5-15, we split the output from Listing 5-14 into its components to make it easier to read.

```
PS> $sddl -split "(?=O:)|(?=G:)|(?=D:)|(?=S:)|(?=\()"
O:WD
G:WD
D:AI
  (D;;GA;;;AN)
  (A;;CCDC;;;S-1-5-21-2318445812-3516008893-216915059-1002)
  (A;;CC;;;WD)
S:P
  (AU;FA;SD;;;WD)
  (ML;;NW;;;LW)
```

Listing 5-15: Splitting up the SDDL components

The first two lines represent the owner and group SIDs in SDDL format. You might notice that these don't look like the SDDL SIDs we're used to seeing, as they don't start with `S-1-`. That's because these strings are two-character aliases that Windows uses for well-known SIDs to reduce the size of an SDDL string. For example, the owner string is WD, which we could convert back to the full SID using Get-NtSid (Listing 5-16).

```
PS> Get-NtSid -Sddl "WD"
Name     Sid
----     ---
Everyone S-1-1-0
```

Listing 5-16: Converting an alias to a name and SID

As you can see, the WD alias represents the *Everyone* group. Table 5-6 shows the aliases for a few well-known SIDs. You can find a more comprehensive list of all supported SDDL aliases in Appendix B.

Table 5-6: Examples of Well-Known SIDs and Their Aliases

SID alias	Name	SDDL SID
AU	*NT AUTHORITY\Authenticated Users*	S-1-5-11
BA	*BUILTIN\Administrators*	S-1-5-32-544
IU	*NT AUTHORITY\INTERACTIVE*	S-1-5-4
SY	*NT AUTHORITY\SYSTEM*	S-1-5-18
WD	*Everyone*	S-1-1-0

If a SID has no alias, Format-NtSecurityDescriptor will emit the SID in SDDL format, as shown in Listing 5-15. Even SIDs without aliases can have names defined by LSASS. For example, the SID in Listing 5-15 belongs to the current user, as shown in Listing 5-17.

```
PS> Get-NtSid -Sddl "S-1-5-21-2318445812-3516008893-216915059-1002" -ToName
GRAPHITE\user
```

Listing 5-17: Looking up the name of the SID

Next in Listing 5-15 is the representation of the DACL. After the D: prefix, the ACL in SDDL format looks as follows:

```
ACLFlags(ACE0)(ACE1)...(ACEn)
```

The ACL flags are optional; the DACL's are set to AI and the SACL's are set to P. These values map to security descriptor control flags and can be one or more of the strings in Table 5-7.

Table 5-7: ACL Flag Strings Mapped to Security Descriptor Control Flags

ACL flag string	DACL control flag	SACL control flag
P	DaclProtected	SaclProtected
AI	DaclAutoInherited	SaclAutoInherited
AR	DaclAutoInheritReq	SaclAutoInheritReq

I'll describe the uses of these three control flags in Chapter 6. Each ACE is enclosed in parentheses and is made up of multiple strings separated by semicolons, following this general format:

```
(Type;Flags;Access;ObjectType;InheritedObjectType;SID[;ExtraData])
```

The Type is a short string that maps to an ACE type. Table 5-8 shows these mappings. Note that SDDL format does not support certain ACE types, so they're omitted from the table.

Table 5-8: Mappings of Type Strings to ACE Types

ACE type string	ACE type
A	Allowed
D	Denied
AU	Audit
AL	Alarm
OA	AllowedObject
OD	DeniedObject
OU	AuditObject
OL	AlarmObject
XA	AllowedCallback
XD	DeniedCallback
ZA	AllowedCallbackObject
XU	AuditCallback
ML	MandatoryLabel
RA	ResourceAttribute
SP	ScopedPolicyId
TL	ProcessTrustLabel
FL	AccessFilter

The next component is Flags, which represents the ACE flags. The audit entry in the SACL from Listing 5-15 shows the flag string FA, which represents FailedAccess. Table 5-9 shows other mappings.

Table 5-9: Mappings of Flag Strings to ACE Flags

ACE flag string	ACE flag
OI	ObjectInherit
CI	ContainerInherit
NP	NoPropagateInherit
IO	InheritOnly
ID	Inherited
CR	Critical

(continued)

Table 5-9: Mappings of Flag Strings to ACE Flags *(continued)*

ACE flag string	ACE flag
`SA`	`SuccessfulAccess`
`FA`	`FailedAccess`
`TP`	`TrustProtected`

Next is `Access`, which represents the access mask in the ACE. This can be a number in hexadecimal (`0x1234`), octal (`011064`), or decimal (`4660`) format, or a list of short access strings. If no string is specified, then an empty access mask is used. Table 5-10 shows the access strings.

Table 5-10: Mappings of Access Strings to Access Masks

Access string	Access name	Access mask
`GR`	`Generic Read`	`0x80000000`
`GW`	`Generic Write`	`0x40000000`
`GX`	`Generic Execute`	`0x20000000`
`GA`	`Generic All`	`0x10000000`
`WO`	`Write Owner`	`0x00080000`
`WD`	`Write DAC`	`0x00040000`
`RC`	`Read Control`	`0x00020000`
`SD`	`Delete`	`0x00010000`
`CR`	`Control Access`	`0x00000100`
`LO`	`List Object`	`0x00000080`
`DT`	`Delete Tree`	`0x00000040`
`WP`	`Write Property`	`0x00000020`
`RP`	`Read Property`	`0x00000010`
`SW`	`Self Write`	`0x00000008`
`LC`	`List Children`	`0x00000004`
`DC`	`Delete Child`	`0x00000002`
`CC`	`Create Child`	`0x00000001`

Note that the available access strings do not cover the entire access mask range. This is because SDDL was designed to represent the masks for directory service objects, which don't define access mask values outside of a limited range. This is also why the names of the rights are slightly confusing; for example, `Delete Child` does not necessarily map to an arbitrary object type's idea of deleting a child, and you can see in Listing 5-15 that the `File` type's specific access maps to directory service object access, even though it has nothing to do with Active Directory.

To better support other types, the SDDL format provides access strings for common file and registry key access masks, as shown in Table 5-11. If the available access strings can't represent the entire mask, the only option is to represent it as a number string, typically in hexadecimal format.

Table 5-11: Access Strings for File and Registry Key Types

Access string	Access name	Access mask
FA	File All Access	0x001F01FF
FX	File Execute	0x001200A0
FW	File Write	0x00120116
FR	File Read	0x00120089
KA	Key All Access	0x000F003F
KR	Key Read	0x00020019
KX	Key Execute	0x00020019
KW	Key Write	0x00020006

For the `ObjectType` and `InheritedObjectType` components, used with object ACEs, SDDL uses a string format for the GUIDs. The GUIDs can be any value. For example, Table 5-12 contains a few well-known ones used by Active Directory.

Table 5-12: Well-Known Object Type GUIDs Used in Active Directory

GUID	Directory object
19195a5a-6da0-11d0-afd3-00c04fd930c9	Domain
bf967a86-0de6-11d0-a285-00aa003049e2	Computer
bf967aba-0de6-11d0-a285-00aa003049e2	User
bf967a9c-0de6-11d0-a285-00aa003049e2	Group

Here is an example ACE string for an `AllowedObject` ACE with the `ObjectType` set:

```
(OA;;CC;2f097591-a34f-4975-990f-00f0906b07e0;;WD)
```

After the `InheritedObjectType` component in the ACE is the SID. As detailed earlier in this chapter, this can be a short alias if it's a well-known SID, or the full SDDL format if not.

In the final component, which is optional for most ACE types, you can specify a conditional expression if using a callback ACE or a security attribute if using a `ResourceAttribute` ACE. The conditional expression defines a Boolean expression that compares the values of a token's security attribute. When evaluated, the result of the expression should be true or false. We saw a simple example in Listing 5-5: `WIN://TokenId == "XYZ"`, which compares the value of the security attribute `WIN://TokenId` with the string value `XYZ` and evaluates to true if they're equal. The SDDL expression syntax has four different attribute name formats for the security attribute you want to refer to:

Simple Used for local security attributes; for example, `WIN://TokenId`

Device Used for device claims; for example, `@Device.ABC`

User Used for user claims; for example, `@User.XYZ`

Resource User for resource attributes; for example, `@Resource.QRS`

The comparison values in the conditional expressions can accept several different types, as well. When converting from SDDL to a security descriptor, the condition expression will be parsed, but because the type of the security attribute won't be known at this time, no validation of the value's type can occur. Table 5-13 shows examples for each conditional expression type.

Table 5-13: Example Values for Different Conditional Expression Types

Type	Examples
Number	Decimal: `100`, `-100`; octal: `0100`; hexadecimal: `0x100`
String	`"ThisIsAString"`
Fully qualified binary name	`{"O=MICROSOFT CORPORATION, L=REDMOND, S=WASHINGTON",1004}`
SID	`SID(BA)`, `SID(S-1-0-0)`
Octet string	`#0011223344`

The syntax then defines operators to evaluate an expression, starting with the unary operators in Table 5-14.

Table 5-14: Unary Operators for Conditional Expressions

Operator	Description
`Exists` *`ATTR`*	Checks whether the security attribute *ATTR* exists
`Not_Exists` *`ATTR`*	Inverse of `Exists`
`Member_of {`*`SIDLIST`*`}`	Checks whether the token groups contain all SIDs in *SIDLIST*
`Not_Member_of {`*`SIDLIST`*`}`	Inverse of `Member_of`
`Device_Member_of {`*`SIDLIST`*`}`	Checks whether the token device groups contain all SIDs in *SIDLIST*
`Not_Device_Member_of {`*`SIDLIST`*`}`	Inverse of `Device_Member_of`
`Member_of_Any {`*`SIDLIST`*`}`	Checks whether the token groups contain any SIDs in *SIDLIST*
`Not_Member_of_Any {`*`SIDLIST`*`}`	Inverse of `Not_Member_of_Any`
`Device_Member_of_Any {`*`SIDLIST`*`}`	Checks whether the token device groups contain any SIDs in *SIDLIST*
`Not_Device_Member_of_Any {`*`SIDLIST`*`}`	Inverse of `Device_Member_of_Any`
`!(`*`EXPR`*`)`	The logical not of an expression

In Table 5-14, *ATTR* is the name of an attribute to test, *SIDLIST* is a list of SID values enclosed in braces {}, and *EXPR* is another conditional subexpression. Table 5-15 shows the infix operators the syntax defines.

Table 5-15: Infix Operators for Conditional Expressions

Operator	Description
ATTR Contains *VALUE*	Checks whether the security attribute contains the value
ATTR Not_Contains *VALUE*	Inverse of Contains
ATTR Any_of {*VALUELIST*}	Checks whether the security attribute contains any of the values
ATTR Not_Any_of {*VALUELIST*}	Inverse of Any_of
ATTR == *VALUE*	Checks whether the security attribute equals the value
ATTR != *VALUE*	Checks whether the security attribute does not equal the value
ATTR < *VALUE*	Checks whether the security attribute is less than the value
ATTR <= *VALUE*	Checks whether the security attribute is less than or equal to the value
ATTR > *VALUE*	Checks whether the security attribute is greater than the value
ATTR >= *VALUE*	Checks whether the security attribute is greater than or equal to the value
EXPR && *EXPR*	The logical AND between two expressions
EXPR || *EXPR*	The logical OR between two expressions

In Table 5-15, *VALUE* can be either a single value from Table 5-13 or a list of values enclosed in braces. The Any_of and Not_Any_of operators work only on lists, and the conditional expression must always be placed in parentheses in the SDDL ACE. For example, if you wanted to use the conditional expression shown back in Listing 5-5 with an AccessCallback ACE, the ACE string would be as follows:

```
(ZA;;GA;;;WD;(WIN://TokenId == "XYZ"))
```

The final component represents a security attribute for the ResourceAttribute ACE. Its general format is as follows:

```
"AttrName",AttrType,AttrFlags,AttrValue(,AttrValue...)
```

The AttrName value is the name of the security attribute, AttrFlags is a hexadecimal number that represents the security attribute flags, and AttrValue is one or more values specific to the AttrType, separated by commas. The AttrType is a short string that indicates the type of data contained in the security attribute. Table 5-16 shows the defined strings, with examples.

Table 5-16: Security Attribute SDDL Type Strings

Attribute type	Type name	Example value
TI	Int64	Decimal: 100, -100; octal: 0100; hexadecimal: 0x100
TU	UInt64	Decimal: 100; octal: 0100; hexadecimal: 0x100
TS	String	"XYZ"
TD	SID	BA, S-1-0-0
TB	Boolean	0, 1
RX	OctetString	#0011223344

To give an example, the following SDDL string represents a ResourceAttribute ACE with the name Classification. It contains two string values, TopSecret and MostSecret, and has the CaseSensitive and NonInheritable flags set:

```
S:(RA;;;;;WD;("Classification",TS,0x3,"TopSecret","MostSecret"))
```

The last field in Listing 5-15 to define is the SACL. The structure is the same as that described for the DACL, although the types of ACEs supported differ. If you try to use a type that is not allowed in the specific ACL, parsing the string will fail. In the SACL example in Listing 5-15, the only ACE is the mandatory label. The mandatory label ACE has its own access strings used to represent the mandatory policy, as shown in Table 5-17.

Table 5-17: Mandatory Label Access Strings

Access string	Access name	Access mask
NX	No Execute Up	0x00000004
NR	No Read Up	0x00000002
NW	No Write Up	0x00000001

The SID represents the integrity level of the mandatory label; again, special SID aliases are defined. Anything outside the list shown in Table 5-18 needs to be represented as a full SID.

Table 5-18: Mandatory Label Integrity Level SIDs

SID alias	Name	SDDL SID
LW	Low integrity level	S-1-16-4096
ME	Medium integrity level	S-1-16-8192
MP	MediumPlus integrity level	S-1-16-8448
HI	High integrity level	S-1-16-12288
SI	System integrity level	S-1-16-16384

The SDDL format doesn't preserve all information you can store in a security descriptor. For example, the SDDL format can't represent the `OwnerDefaulted` or `GroupDefaulted` control flags, so these are discarded. SDDL also doesn't support some ACE types, so I omitted those from Table 5-8.

As mentioned previously, if an unsupported ACE type is encountered while converting a security descriptor to SDDL, the conversion process will fail. To get around this problem, the `ConvertFrom-NtSecurityDescriptor` PowerShell command can convert a security descriptor in relative format to base64, as shown in Listing 5-18. Using base64 preserves the entire security descriptor and allows it to be copied easily.

```
PS> ConvertFrom-NtSecurityDescriptor $sd -AsBase64 -InsertLineBreaks
AQAUpJgAAACkAAAAFAAAAEQAAAACADAAAgAAAAKAFAAAAAEAAQEAAAAAAAEAAAAAEQAUAAEAAAAB
AQAAAAAAEAAQAAACAFQAAwAAAAEAFAAAAAAQAQEAAAAAAAUHAAAAAAAkAAMAAAABBQAAAAAABRUA
AADOrDCKvQmSOXPc7QzqAwAAAAAUAAEAAAABAQAAAAAAAQAAAAABAQAAAAAAAQAAAAABAQAAAAAA
AQAAAAA=
```

Listing 5-18: Converting a security descriptor to a base64 representation

To retrieve the security descriptor, you can pass `New-NtSecurityDescriptor` the `Base64` parameter.

Worked Examples

Let's finish this chapter with some worked examples that use the commands you've learned about here.

Manually Parsing a Binary SID

The PowerShell module comes with commands you can use to parse SIDs that are structured in various forms. One of those forms is a raw byte array. You can convert an existing SID to a byte array using the `ConvertFrom-NtSid` command:

```
PS> $ba = ConvertFrom-NtSid -Sid "S-1-1-0"
```

You can also convert the byte array back to a SID using the `Byte` parameter to the `Get-NtSid` command, as shown here. The module will parse the byte array and return the SID:

```
PS> Get-NtSid -Byte $ba
```

Although PowerShell can perform these conversions for you, you'll find it valuable to understand how the data is structured at a low level. For example, you might identify code that parses SIDs incorrectly, which could lead to memory corruption; through this discovery, you might find a security vulnerability.

The best way to learn how to parse a binary structure is to write a parser, as we do in Listing 5-19.

```
❶ PS> $sid = Get-NtSid -SecurityAuthority Nt -RelativeIdentifier 100, 200, 300
  PS> $ba = ConvertFrom-NtSid -Sid $sid
  PS> $ba | Out-HexDump -ShowAll
            00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  - 0123456789ABCDEF
  -----------------------------------------------------------------------------
  00000000: 01 03 00 00 00 00 00 05 64 00 00 00 C8 00 00 00  - ........d.......
  00000010: 2C 01 00 00                                      - ,...

  PS> $stm = [System.IO.MemoryStream]::new($ba)
❷ PS> $reader = [System.IO.BinaryReader]::new($stm)

  PS> $revision = $reader.ReadByte()
❸ PS> if ($revision -ne 1) {
      throw "Invalid SID revision"
  }

❹ PS> $rid_count = $reader.ReadByte()
❺ PS> $auth = $reader.ReadBytes(6)
  PS> if ($auth.Length -ne 6) {
      throw "Invalid security authority length"
  }

  PS> $rids = @()
❻ PS> while($rid_count -gt 0) {
      $rids += $reader.ReadUInt32()
      $rid_count–
  }

❼ PS> $new_sid = Get-NtSid -SecurityAuthorityByte $auth -RelativeIdentifier $rids
  PS> $new_sid -eq $sid
  True
```

Listing 5-19: Manually parsing a binary SID

For demonstration purposes, we start by creating an arbitrary SID and converting it to a byte array ❶. Typically, though, you'll receive a SID to parse in some other way, such as from the memory of a process. We also print the SID as hex. (If you refer to the SID structure shown in Figure 5-1, you might already be able to pick out its various components.)

Next, we create a `BinaryReader` to parse the byte array in a structured form ❷. Using the reader, we first check whether the revision value is set to `1` ❸; if it isn't, we throw an error. Next in the structure is the RID count as a byte ❹, followed by the 6-byte security authority ❺. The `ReadBytes` method can return a short reader, so you'll want to check that you read all six bytes.

We now enter a loop to read the RIDs from the binary structure and append them to an array ❻. Next, using the security authority and the RIDs, we can run `Get-NtSid` to construct a new SID object ❼ and verify that the new SID matches the one we started with.

This listing gives you an example of how to manually parse a SID (or, in fact, any binary structure) using PowerShell. If you're adventurous, you could implement your own parser for the binary security descriptor formats, but that's outside the scope of this book. It's simpler to use the New-NtSecurityDescriptor command to do the parsing for you.

Enumerating SIDs

The LSASS service does not provide a publicly exposed method for querying every SID-to-name mapping it knows about. While the official Microsoft documentation provides a list of known SIDs, these aren't always up to date and won't include the SIDs specific to a computer or enterprise network. However, we can try to enumerate the mappings using brute force. Listing 5-20 defines a function, Get-AccessSids, to brute-force a list of the SIDs for which LSASS has a name.

```
  PS> function Get-AccountSids {
      param(
          [parameter(Mandatory)]
        ❶ $BaseSid,
          [int]$MinRid = 0,
          [int]$MaxRid = 256
      )

      $i = $MinRid

      while($i -lt $MaxRid) {
          $sid = Get-NtSid -BaseSid $BaseSid -RelativeIdentifier $i
          $name = Get-NtSidName $sid
        ❷ if ($name.Source -eq "Account") {
              [PSCustomObject]@{
                  Sid = $sid;
                  Name = $name.QualifiedName;
                  Use = $name.NameUse
              }
          }
          $i++
      }
  }

❸ PS> $sid = Get-NtSid -SecurityAuthority Nt
  PS> Get-AccountSids -BaseSid $sid
  Sid          Name                    Use
  ----         ----                    ---
  S-1-5-1      NT AUTHORITY\DIALUP     WellKnownGroup
  S-1-5-2      NT AUTHORITY\NETWORK    WellKnownGroup
  S-1-5-3      NT AUTHORITY\BATCH      WellKnownGroup
  --snip--

❹ PS> $sid = Get-NtSid -BaseSid $sid -RelativeIdentifier 32
  PS> Get-AccountSids -BaseSid $sid -MinRid 512 -MaxRid 1024
  Sid          Name                    Use
  ----         ----                    ---
  S-1-5-32-544 BUILTIN\Administrators Alias
  S-1-5-32-545 BUILTIN\Users          Alias
```

```
S-1-5-32-546 BUILTIN\Guests         Alias
--snip--
```

Listing 5-20: Brute-forcing known SIDs

The function accepts a base SID and the range of RID values to test ❶. It then creates each SID in the list and queries for its name. If the name's source is `Account`, which indicates the name was retrieved from LSASS, we output the SID's details ❷.

To test the function, we call it with the base SID, which contains the `Nt` authority but no RIDs ❸. We get the list of retrieved names and SIDs from LSASS. Notice that the SIDs in the output are not domain SIDs, as you might expect, but `WellKnownGroup` SIDs. For our purposes, the distinction between `WellKnownGroup`, `Group`, and `Alias` is not important; they're all groups.

Next, we try brute-forcing the *BUILTIN* domain SID ❹. In this case, we've changed the RID range based on our preexisting knowledge of the valid range, but you're welcome to try any other range you like. Note that you could automate the search by inspecting the `NameUse` property in the returned objects and calling `Get-AccountsSids` when its value is `Domain`. I leave this as an exercise for the reader.

Wrapping Up

We started this chapter by delving into the structure of the security descriptor. We detailed its binary structures, such as SIDs, and looked at access control lists and the access control entries that make up the discretionary and system ACLs. We then discussed the differences between absolute and relative security descriptors and why the two formats exist.

Next, we explored the use of the `New-NtSecurityDescriptor` and `Add-NtSecurityDescriptorAce` commands to create and modify a security descriptor so that it contains whatever entries we require. We also saw how to display security descriptors in a convenient form using the `Format-NtSecurityDescriptor` command.

Finally, we covered the SDDL format used for representing security descriptors. We discussed how to represent the various types of security descriptor values, such as ACEs, and how you can write your own. Some tasks we haven't yet covered are how to query a security descriptor from a kernel object and how to assign a new one. We'll get to these topics in the next chapter.