Auditing Windows Environments
PowerShell XML output,
windows security, ossams

Cody Dumont
Auditing Windows Environments with PowerShell XML Output

GIAC (GCWN) Gold Certification

Author: Cody Dumont, cdumont@nwnit.com
Advisor: Aman Hardikar .M,

Accepted: 3 Jan, 2012

Abstract
Auditing with PowerShell is a major component to the future on Windows Security. As part of the Open Source Security Assessment Management System (OSSAMS) project, this paper analyzes the initial development of the PowerShell framework used to collect DACL’s from AD objects. The objective for OSSAMS is normalizing data for a streamlined analysis. The data will be collected from routers, switches, firewall, security tools, directory services, and other information systems. This paper outlines the initial framework used within PowerShell to audit MS AD and other MS systems. The restrictions on the framework are the customer, or organization being assessed, would only need to create a user account for the assessor. The computer doing the assessment cannot join the domain. The paper discusses the SID, .Net Classes, and the coding process in-depth.

Keywords: PowerShell, Active Directory, XML, DACL, SID
1 Introduction

A security professional often performs security assessments for customers and will use many tools to collect data. Each tool stores data in a separate format; this requires the assessor to develop a proprietary automated process or use a manual process to correlate all the data. This process includes custom parsing XML files into spreadsheets, manual reviews of some data and other time consuming tasks. This manual process often results in missed vulnerabilities or extensive time spent putting all pieces of the puzzle together. To solve this problem, a team of security professionals decided to create a Relational Database Management System (RDMS) to normalize all the data inputs and then store the data in a common database for a more accurate assessment. This project is called Open Source Security Assessment Management System (OSSAMS).

The heart of the project is the database structure, but the brain of the project is the framework developed by the OSSAMS team1 to normalize the data. The data parsing is a combination of Perl, Python, and PowerShell (PS1) scripts. These scripts will read the output from other tools, most commonly in XML and normalize the data into common and supportable data structure used by the OSSAMS database. This paper specifically focuses on the use of the PS1 to create a framework, collecting data from AD, normalizing, and storing the data into OSSAMS database. The initial framework will be used to collect Dynamic Access Control Lists (DACL) for the Active Directory objects.

1.1 What is OSSAMS

The OSSAMS project is a framework used to normalize data from many security tools and then correlate the information in a structured manner to allow the security professional and customer to see a more accurate security posture. The data sources are from device configurations (i.e. firewall, router, etc), questionnaires, security tools such as Nessus® & NMAP®, and manual reviews (i.e. policy reviews). Once all the data is collected, the security professional is able to review all the data based on three (3) primary pivot points, which are:

1 Original Team Members – Cody Dumont, Darryl Williams, & Adrien de Beaupre

Cody Dumont, cody@melcara.com
• Project / Assessment
• IP Address / Hostname / FQDN
• User / Group / Role

The function of the OSSAMS will not be to provide answers to the security professional, for example to tell the security professional what is wrong and how to fix the problem. The function of OSSAMS is to correlate all the data from many sources and provide the security professional with an in-depth view of the information system being reviewed. One of the initial concerns for OSSAMS is to avoid creating another system to do the thinking for the security professional, but to provide all the information to the security professional to make more accurate assessment of the customer’s security posture.

1.2 Project Scope Boundaries

When conducting security assessments or penetration tests, the security professional is often not permitted to join a PC to the customer’s AD domain. While using PS1 to preform the assessment, this can be somewhat unfortunate. The PS1 environment supports the use of a customized module called a Snap-In. There are many snap-ins for many functions, for example Quest Software created a snap-in to easily manage an AD environment. However, all of these plugins require the workstation running the PS1 script to be a member of the domain, or to have established trust relations with the domain to work properly. As mentioned earlier the ability to join the domain is seldom an option during the assessment. Therefore, for the purposes of this paper, the computer will not be a member of the AD domain.

The plugins, which provide support to AD, require Active Directory Web Service (ADWS) or the Active Directory Management Gateway Service (ADMGS) to be configured. As the security professional can’t rely on this service to be running and can’t ask the customer to enable it, the scripts must run without these services enabled.

1.3 Windows Assessment Methodology

When conducting assessments or audits, the security professional should have a methodology or strategy. The methodology or strategy should dictate the type of information to be collected and analyzed. Additionally the methodology or strategy

Cody Dumont, cody@melcara.com
should be combined with generally accepted best practices and standards for comparison against the customer’s implementation. This paper will not focus on the results of the data collection, meaning what is the best configuration for the customer, but will focus on the data collection process and methodology for collecting the data.

2 Development Process

The development process began with some basic PS1 scripts to loop through collected data and to gain a better understanding of PS1 and .Net framework. Then the decoding of the objects collected from AD and the collection of DACL’s applied to the AD objects. The next step in the project is to create XML data structure used in the framework. Finally, import method used to insert the data into the OSSAMS database.

2.1 Development Environment Setup

2.1.1 Data Collection Targets

For the purposes of the initial research, the data collection target will be a Windows 2008 R2 domain controller built during the SEC505 course. The information collected will be from AD. The environment does not have MS Exchange or other MS components that would extend the schema. However, because of the dynamic nature of the framework, the PS1 scripts should not require modification should the AD schema be extended.

2.1.2 Workstation Configuration

The assessment workstation is configured with following Snap-in’s:

- Install RSAT on Win7 (64bit)
- .Net Framework v4

Other PS1 Development Tools used are:

- PowerGUI
- Notepad++
- PowerShell Community Extensions (PSCX)
- PowerShell Guy PowerTab

Cody Dumont, cody@melcara.com
2.2 Data Collection

The script has many modular tasks, such as connecting to AD, and formatting XML data. After the data is collected, the data is saved in the native XML object format used by PS1 to allow reimporting. The ability to reimport the collected data when needed will reduce the need to re-query AD for common tasks such as AD object collection.

2.2.1 AD Object Collection

The initial modular function is collection of all the AD objects. There are many types of AD objects; some examples are Users, Groups, Containers, and many more. The commands to connect to AD are relatively simple and easy to follow. Shown below is a snippet of code, each line will be explained in detail.

```powershell
$ad_dir_entry = New-Object DirectoryServices.DirectoryEntry($ad_ldap_svr,$ad_ldap_user,$ad_ldap_password)
$searcher.filter = "(objectclass=*)"
$adObj = $searcher.findall()
```

The first line of code creates a new object called “$ad_dir_entry”. The object located within the .Net namespace “DirectoryServices”. The DirectoryServices namespace provides easy access to AD from managed code. The namespace contains two component classes, DirectoryEntry and DirectorySearcher, which use the Active Directory Services Interfaces (ADSI) technology. ADSI is the set of interfaces that Microsoft provides as a flexible tool for working with a variety of network providers. ADSI gives the administrator the ability to locate and manage resources on a network with relative ease, regardless of the size of the network (MSDN, 2011). The first command is passed to the AD server IP address (or FQDN), username, and password.

Next, another object is created; this is the DirectorySearcher object. To create the DirectorySearcher object, the script must invoke ADSI and pass the DirectoryEntry object. A filter is applied to the DirectorySearcher, and the output is stored into the $adObj variable. From this point forward, the script does not need to query AD to get any AD object, but only call this variable. To save more time, this object is exported to a XML file for use by other scripts and functions yet to be developed.

Cody Dumont, cody@melcara.com
2.2.2 Security Identifiers

Active Directory uses a Security Identifier (SID) to identify each object uniquely in the LDAP database. There are well known SIDs for common groups such as the Administrators and Power Users groups. As the computer is not a member of the AD domain, by default the computer is unable to resolve the SID to an object name; therefore a manual process for this component is required. To address the SID resolution problem, an associative array is created using the predefined common SID, and extracting the SID and user name for the $adObj table captured earlier.

To create the associative array for the SID resolution, the script first calls a CSV file called “Well-Known-SID.csv”. The “Well-Known-SID.csv” SID data is stored, then a foreach loop is processed using the $adObj. The SID stored in AD is stored in hex, to address this issue MSDN had an example of the code to convert the SID to decimal. Additionally some objects within AD don’t have a SID, and use the Globally Unique Identifier (GUID) instead. This required a check for the SID to be present, if not the GUID is stored in the SID field.

As all items in PS1 are objects, each field properties or method is also an object. To prevent the unnecessary data passed from actual object to the newly created object, the use of the [string] variable type is required. The [string] variable type ensures only the data in the properties is entered into the new variable. The new entry for the SID table is then send to a CSV file location stored in the $sid_csv_file variable. As the entries may use a comma with the object name, the script uses the “|” pipe for value separator. The new SID entry is then stored in an associative array called $sid_array.

2.2.3 DACL Collection

The DACL collection proved to be the hardest obstacle to overcome. The “BSonPoSH” blog gives a great blog post on how to get started (Shell, 2008). The blog post shows how to query the DACL of a specific object. The blog post provides information about several different functions that can be completed using the “SecurityIdentifier” class. This class is located in “System.Security.Principal” hierarchy.

The DACL collection process begins with looping through the AD objects, and creating a new object using the “DirectoryServices.DirectoryEntry” class. From the
“DirectoryServices.DirectoryEntry” the ObjectSecurty property is selected. The ObjectSecurty property provides the security descriptor for the directory entry.

The ObjectSecurity property is stored into a new variable called “$acl”. The $acl is then used to extract the DACL using the “CommonObjectSecurity.GetAccessRules” methods. The GetAccessRules method provides a collection of the access rules associated with the specified security identifier. When calling the GetAccessRules method, there are three parameters that are required, which are includeExplicit, includeInherited, and targetType. The “targetType” can be one of two settings, “SecurityIdentifier” or “NTAccount”. To avoid naming errors, the script will collect the “SecurityIdentifier”.

The script then enters into a “Where-Object” loop and store several properties of the DACL using the [string] variable type. After the fields for the ACL entry are collected, the SID is searched for within the $sid_array previously collected. If there is a name, the SID is replaced with the name value of the SID array entry. Lastly, the data collected to create the DACL is then stored into an associative array for reporting and other analysis.

2.2.4 XML to Store Data

The script next stores the data in several XML formats. The first format is the PS1 native format called using the Export-Clixml. The Export-Clixml cmdlet creates a native XML format used by PS1 to recreate the object using the Import-Clixml cmdlet. This XML format is not the optimal format needed to import into the OSSAMS database.

To import data into the OSSAMS database, the script creates new XML object. Next the script creates a sample data format string of the XML structure. The associative array is then looped through and data is injected into the XML object using the formatted strings. Examples (Figures 1, 2, and 3) of the XML formatting can be found on the subsequent pages.

2.3 Functionality vs. Modularity

The script at this point achieves the goals that have been established. However, the script neither supported modularity nor resembled a useable framework for expansion. The next goal is to do the following:

Cody Dumont, cody@melcara.com
• Change the script to pull configuration variables from seed file or command line
• Dynamically create XML format
• Allow for the use of other cmdlet to gather DACLs
• Expand Functionality using plugin style components

3 Script Modularization

To modularize the script and begin the creation of the PS1 framework used in OSSAMS, the script will focus on the creating the XML structures dynamically, and creating the scalable method of calling additional cmdlets and imported data from other objects.

3.1 Dynamic XML Data Structure

There are several methods for interacting with XML data using PS1. As mentioned earlier, there is a native XML format used to reimport data using the Import-Clixml/Export-Clixml cmdlets. The next option is to use ConvertTo-XML cmdlet. This option is more descriptive than the Export-Clixml option, but the ConvertTo-XML cmdlet adds too much repetitive data. The third option is to create an XML object in a customized format, which is selected for use in the OSSAMS project.

3.1.1 PS1 Native XML Format

The native XML formats using the Export-Clixml and ConvertTo-XML cmdlets could be viable options in some cases. However for the OSSAMS project, the goal is to have the data normalized before being entered into the database. Therefore the PS1 scripts should remove unneeded data before inserting into the database. As part of the data analysis, this section will provide samples of the data found in the native formats and discuss why these options are not the best for importing into OSSAMS.

The purpose for the Export-Clixml cmdlet format is to reimport objects into a PS1 environment for analysis. As shown in the figure below, the namespace structure used is not easily understandable and the code required to analyze the data before importing would be rather difficult to create. However, this file is created to reimport the data if needed for other components with the OSSAMS framework.

Cody Dumont, cody@melcara.com
Auditing Windows Environments with PowerShell XML Output

The XML structure created from the ConvertTo-XML cmdlet is much easier to ready and comprehend. However, there is a lot of extra data also exported. Much of this extra data is important when used with .Net framework, however for the purposes of OSSAMS this extra data is not needed. During the initial script testing, this format was used as a guide for creating the OSSAMS XML format.

3.1.2 Customized XML Format

There are two customized XML format created for OSSAMS. One of the format that is the default dynamically creates XML from .Net objects. The other is a custom format created for the gathering of DACL information and is generated by the PS1 script. The custom format used for the DACL is based off the fields extracted from the AD object and DACL object, then imported into the $object_acl_array array.

The default format dynamically created from the .Net object is the corner stone of the scripts flexibility (Weltner, 2009). The function named “XML_Reformat” is the

Cody Dumont, cody@melcara.com
process where the data structure is extracted from the .Net object model, and then creates
dynamic field names for the OSSAMS XML structure. The function works by looping
through each object in the associative array and extracting the property name for each
object. If the property name is not previously identified, the property name is stored into
an array called $PropertyNames.

The next task within the function is to create the new XML structure. The XML
structure has a root element or level 1 node, and each object is the separate child element
of the root, the level 2 node. The level 2 node is the parent element that contains the
child elements that are created from the properties. The figure shown below has the root
element (level 1 node) of ACL, the child element (level 2 node) of ACE, and the
properties of acl_objectclass, ACL_Target_Name_CN, ActiveDirectoryRights,
ACL_Target_Name, User, AccessControlType, and DistinguishedName.

Figure 3 - XML Reformat structure

Now the new XML template is created and is inserted into a XML object. The
first step is to export the array $xml_template into a file using the “Out-File” cmdlet.
Next a new XML object is created called $xml. The file previously created is then
imported into the new XML object. The next function is to loop through each object
stored in the associative array $xmlObj, and for each property defined, the value is then
stored in the $xml object.

However, as each object in $xmlObj may not have the same properties, the result
is that many data fields defined are not used and the first level 2 node is the template
data. The next series of commands are used to extract all the unused fields from the level
2 nodes, and remove the template node. The method “$xml.$lvl1_node.RemoveChild” is
used to remove the level 2 node used for the template. The next command uses the
“Select-Xml -XPath” cmdlet to select all child nodes with the word “empty” in the field

Cody Dumont, cody@melcara.com
and removes the identified elements. This step alone reduces the data size by about 80% and creates a clean XML file for importing into OSSAMS database.

3.2 Data Structure

*NOTE: At the time of this writing the OSSAMS database architecture is being developed, therefore the data structure proposed is of an initial conceptual structure.*

The data is now formatted in a manner, which can be easily supported and imported into the database. Each level 2 node from the XML file will be converted into a server property. The level 1 node will denote the server property type, and the level 2 child nodes will be the property name and property value. For the DACLs, there is a need for more than two fields; therefore an additional field called “prop_permission” is added.

The database table will have the following fields:

- **prop_index**
  - The index of the table, also the primary key with auto increment.

- **prop_name**
  - The property name for all entry types other than DACL.
  - For DACL entries, this field will map to the “ActiveDirectoryRights” field.

- **prop_value**
  - The property value stored within the child node, for all entry types other than DACL.
  - For DACL entries, this field will map to the “User” field.

- **prop_type**
  - This is the type of entry and will be determined by the name of the root element or level 1 node.

- **prop_permission**
  - This field is for DACL entries only.
  - The value mapped to this field is AccessControlType.

- HostData_HostKey
4 Detailed Script Breakdown

The previous sections provided the conceptual review of the script’s functionality and this section will be a detailed block-by-block break down of the script.

4.1 Configuration Variables

The scripts configuration variables are configured from lines 3 to 41. There are four groupings of variable settings.

4.1.1 Command Line Arguments

The first section line 4 to line 9 collects arguments from the command line and sets the variable $myDir to the directory where the script is located. The interesting issue with this section of code is the “param” cmdlet must be the first command executed in the script, otherwise the command line arguments are not passed correctly (Goude, 2009). Many of the examples failed to mention this requirement is their code discussion. Additionally the $myDir variable must be set, as the Present Working Directory (PWD)
is not where the script is executed. This issue with PWD is commented on throughout many of the PS1 script blogs and configuration examples.

4.1.2 The Config File “-cfg”

The second block of code is from line 12 through 23, where the command line argument “-cfg” is tested. If “-cfg” is present, all other command line arguments are ignored, and the file referenced in the parameter is inserted into the array $cfg_data. The array $cfg_data is populated using the “import-csv -delimiter ”$" $cfg”. The interesting part of this command was the “-Delimiter” option. If the “-Delimiter” option is a “|” or “;” then the delimiter must be enclosed with quotes, however all other characters used as the delimiter must not be enclosed in quotes.

Next there is a series of “foreach” loops where the name of the object is tested for a defined variable and if found in the array $cfg_data, the data is imported into the appropriate variable. Note the use of the “break” command to stop processing the “foreach” loop after a match is located.

4.1.3 Passing Config Parameters via the Command Line

The third group of commands is from lines 24 to 33, where other command line arguments are evaluated. This group of commands are a series of “if, then, else” statements. The process is first checked to see if the argument variable is not null. If the parameter is not null, then the parameter variable is stored into the global variable. However, should the variable be null then the default data is stored into the global variable.

4.1.4 Global Variables

The fourth part of the configuration is the definition of the global variables. These variables are defined from lines 36 to 40. All variables are stored as null with exception of the $version variable. There are also two arrays defined using the “@()” as the data input.
4.2 Defined Functions

The next section starts at line 43 and continues to line 179, where the functions called from other parts of the scripts are stored. The script has two defined functions, namely HexSIDToDec and XML_Reformat.

4.2.1 HexSIDToDec

The “HexSIDToDec” function, beginning on line 46 to 74 was found on the Technet web site (Mueller, 2011). The syntax in the function appears to be fairly simple; however some of the logic is hard to follow if the formatting of the SID is unclear. The SID value is passed to the function in the form of an array of hex numbers. When the SID is passed the function, the SID becomes the first element in an array of arrays. The array of arrays can be confusing in this case because there is only one element in the array.

The first element of the array is prepended with “S-”, converted into a string, and then stored into the variable $strSID. The $strSID is then converted back into an array using the split method and stored into the $arrSID. Next the count of the arrays elements is stored in the $Max variable using the count method. The decimal formatted SID is then created using the array elements 0,1,8. The choice of using the first and eight elements is not clear.

On the Wikipedia page on security identifiers (Security Identifier, 2011) there is content that explains the hex to decimal SID conversion process. However, the description is limited to a machine account. At the bottom of the Wikipedia page there is a link to “selfadsi.org”, where there is a very detailed break down of the SID formatting. From the diagram shown below, we can graphically see the formatting of the SID (Föckeler, 2011). When we map the elements in the array to blocks below, there are eight blocks that make up the Revision, SubID, and Identifier Authority. As the array elements are mapped to the block lists, the ninth ($arrSID[8]) element is now clear, as the start of the SubAuthority.

Figure 5 - SID Format (Föckeler, 2011)
The next task is to test and see if the array has eleven elements in the array, as this would depict a “Well-Known-SID”. The figure shown above shows the SID for the “Everyone” group.

\[ S-1-1-0 = \text{Everyone} = 0x01 \ 0x01 \ 0x00 \ 0x00 \ 0x00 \ 0x00 \ 0x01 \ 0x00 \ 0x00 \ 0x00 \ 0x00 \]

Should the $Max be greater than 11, the variable $Temp1 is created using the following formula.

\[
[\text{Int64}] \text{arrSID}[12] + (256 * ([\text{Int64}] \text{arrSID}[13] + (256 * ([\text{Int64}] \text{arrSID}[14] + (256 * ([\text{Int64}] \text{arrSID}[15]))))))
\]

The value from $Temp1 is then pushed on the end of $DecSID, and should the value of $Max be equal to 15, the $DecSID is returned.

**Figure 6 - SID Second Example (Föckeler, 2011)**

The next few steps perform similar conversions previously described and using the $Temp2 and $Temp3 variables. If the $Max variable is less that 24, then the $DecSID is returned. If the $Max is greater than 24, then the $Temp4 variable is returned.

### 4.2.2 XML_Reformat

The XML_Reformat function is the first function created specifically for the script. The section begins on line 80 and ends on line 179. The lines 88 through 94 extract the parameters from the array passed when the function is called from within the script.

#### 4.2.2.1 Discover Properties

The next process is a “ForEach-Object” loop which discovers all the properties of the elements stored in the $xmlObj array. Within the “ForEach-Object” loop there are two test conditions, the first will test to see if the current object has the property “Properties.PropertyNames” and the second test will determine if the current object is an associative array. Objects pulled directly from AD or PS1 will have the property of “Properties.PropertyNames”, while the ACL array created by the script will be an associative array.

Cody Dumont, cody@melcara.com
The reason for bringing focus to this series of commands is the potential for customizing this section in the future. For other objects, for example file systems and registry keys, may require this code to be modified to normalize the data. If neither condition is true the script will report an error and exit.

4.2.2.2 Create XML Document

After the properties are discovered the XML file is created, and from line 120 through 171 is where the data from the input array is stored into the XML file. The first task is to create the XML object with the New-Object cmdlet. The XML template is described in detail in section 3.1.2. The next command opens the XML Object where data will be written. The first object created is then stored into a variable called “$xml_entry”. The variable called “$xml_entry” is cloned during the subsequent foreach loop. Note in this “foreach” loop the same tests are repeated from the loop that determines if the variable is an associative array or object with the specified property.

Next as the command to extract the syntax is not the same for the two testing methods, the variable “$PropertyNameType” is tested to see if the entry is either “Properties” or “Keys”. Then the appropriate command is used to store the text in the value setting properly into the new array. The final action taken by this sub function is to set the variables used to $null to prevent data from being corrupted.

4.2.2.3 XPATH & NULL Tasks

Starting on Line 164 and continuing to line 178 is a series of commands used to clean out data that is no longer needed. First the first XML node is removed. The second command uses the Select-Xml using the XPATH method to search for nodes that contain the string “empty” and delete them. Finally the following command sets variables to for reuse.

4.3 File Object Creation

The next series of commands, lines 181 to 199, create the working directory for the data files and creates the files used to store the data collected. The first command sets the variable $well_known_sid_file with file name where the well-known SIDS are stored, then changes directory to the location stored in the $myDir variable.

Cody Dumont, cody@melcara.com
Using the Get-Date cmdlet the date is stored in the $date variable. Then a directory name is created using the “-f” format operator. The command uses the properties of the $date variable are used to create a string with date and time attributes. The $DirectoryName is populated with year, month, day, hour, minute, and second. Then the directory is created, note the “> $null” usage, this sends the output from the “mkdir” command to $null. The output from the “mkdir” command returns a PS1 error, and it is irrelevant to script operations. Then the ACL CSV file is created, and PS1 environment changes directory to a newly created directory. The next two command sequences create two CSV files, one is the ACL CSV file, and the other is the SID CSV file.

4.4 AD Connection

This is first section of the script that is designed to operate with AD, beginning on line 201 and ending on line 243. This section will connect to AD, then collect all the objects and decode the GUID and SID. This is last preparatory phase of the script.

4.4.1 Load AD Object

To make the connection to Active Directory a new object must be created. The new object is a “DirectoryServices.DirectoryEntry” object. To create this object, the LDAP URL, username, and password must be submitted. Next another new object is created, using the “DirectoryServices.DirectorySearcher” class. This command uses the ADSI interface and is passed the previously created AD object. The searcher object requires a filter to be created. As the scripts intent is to capture all the objects the filter is set to the object class of “*”, meaning all objects. Finally an array called $adObj is created and stored with all AD objects.

4.4.2 Export AD Objects

This section begins on line 209 and ends on 213. This section is the first time the XML_Reformat function is called. The arguments passed are “AD” for the level 1 node, “OBJECT” for the level 2 node, “ad_data” for the XML files name and the $adObj array. The next two commands are temporary at this point, as they create export of the data in both XML formats using the Export-Clixml and convertTo-CML cmdlets. These last two
files are only used at this point for validation testing and will most likely be deleted when fully implemented into OSSAMS.

### 4.4.3 Decode SID for ACL’s

The next section is where the SIDs are evaluated and decoded. The section begins on line 214 and ends on line 243. The first task imports the well-known SID file and creates the arrays used to store the SID data. Following these tasks a “foreach” loop is started where each object stored in the $adObj is processed. The next four lines store AD related information into variables, then the $guid is populated with returned data from the HexSIDToDec function. As some objects in AD are not assigned a SID, the GUID is also used for ACL processing. Then the next “if, then, else” statement checks the status of the $sid variable and takes action accordingly.

Now the data has been collected, the data is stored into an associative array called $new_obj. Additionally the data which is stored in the $sid_csv_file is stored into a variable called $sid_csv_file_entry. This entry is redirected to the $sid_csv_file and into the $sid_array array. The remaining lines clear variables for reprocessing.

### 4.5 ACL Processing

The final section of code is the ACL Processing section and begins on line 245. As in the previous section, the first couple of lines are used to step through each object in the $adObj array and collect data for creating the ACL data files. The first test condition is used to determine the length of the “displayname”, and if the length is not equal to “0” then the $acl_target_name is set to the AD object’s property item called “displayname”.

#### 4.5.1 AD Object ACL Collection

The next task is to collect the ACL on the AD object. To start this process a new object must be created called $ad_object_entry, using the “DirectoryServices.DirectoryEntry” class. When using the “DirectoryServices.DirectoryEntry” class a variable called $acl_target is used along with username and password previously used. The $acl_target is the ADsPath of the AD object. The ADsPath is the hierarchical path to an object in AD (Codeidol.Com, 2009). The ADsPath is comprised of the LDAP URL and the full common name of the AD object. A sample of the an ADsPath is

Cody Dumont, cody@melcara.com
“LDAP://10.10.10.10/CN=Users,DC=sans505,DC=int”, for the “Users” container in AD. A new object is created using ADSI with the $ad_object_entry as the parameter. Next the ObjectSecurity property is stored in the $acl variable.

The GetAccessRules method of the $acl variable is called and sends variables for the parameters includeExplicit, includeInherited, and targetType. The “includeExplicit” parameter tells AD to include access rules explicitly set for the object. The “includeInherited” parameter tells AD to include inherited access rules. While the “targetType” specifies the security identifier type, the two options are:

- System.Security.Principal.NTAccount

The returned data is then passed to a “While-Object” loop where the IsInherited is checked. If IsInherited is false, the data ACL entry is stored in the $acl_xml_obj_entry array. The setting of “includeInherited” parameter, when calling the GetAccessRules method would allow the script to avoid the “While-Object” loop. In a future revision the “includeInherited” parameter will be set to null or $false to avoid the additional processing. The next two lines input the $acl_xml_obj_entry in to the $acl_xml_array and then sets the $acl_xml_obj_entry to $null. The setting of $acl_xml_obj_entry to null prevents old data from being used on the next iteration of the loop.

4.5.2 ACL Entry Creation

The next section, beginning on line 261, begins the same GetAccessRules method used previously. The reason for this is an error processing of the ACL properties, which will be corrected in a later version of the script.

The first step in this section is to get the ACL and then increment though all entries. The next two commands collects and stores the data in the $ActiveDirectoryRights and $AccessControlType variables.

Next the script extracts the SID for the current object, by creating a new “System.Security.Principal.SecurityIdentifier” object called $ID. The “AccountDomainSid” is then tested and if null the variable $DistinguishedName is set to “N/A”. The next test condition attempts to translate the $ID to an “NTAccount” name format. If that test fails then the well-known SIDs are evaluated. Note the modification

Cody Dumont, cody@melcara.com
of the $ErrorActionPreference setting. The reason for the $ErrorActionPreference
change is during the evaluation process there are a lot of errors which are false positives
that should be ignored. If the “AccountDomainSid” is not null, then the $sid_array
collected from AD is iterated though to look for a match. If a match is found the loop is
broken out of and the next entry is processed.

Now the data has been collected, an associative array called $new_obj is created
to store the ACE. The data is stored in a CSV file using the format method used
previously and then stored into an array for parsing. The next nine commands clear
variables for reuse. The last remaining commands are used to parse the data using the
“XML_Reformat” function and the Export-Clixml and convertTo-CML cmdlets.

5 Security Findings

This script allows the security professional to view all the objects in AD and
access controls assigned to each object. Some items to look for are hidden accounts in
containers that are not normal. For example there are many hidden containers used for
upgrades and other system functions. A crafty attacker or a malicious system
administrator could create a user account in these containers and they may not be seen by
the typical help desk person. Other concerns could be excessive permissions applied to
key objects in AD. This script also helps to show security professionals and system
administrators how many hidden objects are created by default.
6 References


7 Appendix A - Script Execution Flow Chart

Flow charts demonstrating the script processing are shown below.

Figure 7 - Primary Script Function Flow Chart
Figure 8 - Continuation of Primary Script

Input to CSV

Export to CSV

Function XML_Reformat

XML_Reformat

("ACL","ACE","acl_data",Subject_acl_array)

ConvertTo-XML

Export-Client
Figure 9 - Flow Chart for Function HexSIDToDec
Figure 10 - Flow Chart for Function XML_Reformat

MAIN PAGE 4.2.2 Function XML Reformat

- Format Input Variables
  - $xmlObj / forEach-Object

Else
  - $PropertyNames - $_.Properties PropertyNames

Else
  - $keys - $null

XML FILE PREP

4.2.2.1 Discover Properties

- New Object
  - $xml

Input
  - $xml Load template

Clone XML Node

Else
  - $xmlObj

4.2.2.2 Create XML Document

- $PropertyNames = $xml Properties PropertyNames

Else
  - $PropertyNames = $xml keys

Else
  - $PropertyNames = $xml Properties

Else
  - $ PropertyNameType = "Properties"

Else
  - $PropertyType $name Name = $xml Item($name)

Else
  - $PropertyType $name Name = $xml Item($name)

End ForEach Loop

4.2.2.3 XPath & NULL Tasks

- Remove the node template
  - XPath "/Text/empty?"?

Save XML Document

Else
  - $xmlParentNode / RemoveChildren($xml, $Node) - $null

Cody Dumont, cody@melcara.com
8 Appendix B - Script Output

The following text is a sample of the output for the script execution.

PS C:\PowerShell\get-AD-ACL.v03.3.c\20111120-105132> .\get-ad-objects-with-acl.v0.3.3c.ps1 -cfg cfg.0.3.3.txt
Getting Arguments from C:\PowerShell\get-AD-ACL.v03.3.c\cfg.0.3.3.txt
START THE ad_data XML_REFORMAT FUNCTION
FINISHED THE ad_data XML_REFORMAT FUNCTION
PARSING AD ACLS NOW
PARSED ACL for sans505
PARSED ACL for Users
PARSED ACL for Computers
PARSED ACL for Domain Controllers
PARSED ACL for System
PARSED ACL for LostAndFound
PARSED ACL for Infrastructure
PARSED ACL for ForeignSecurityPrincipals
PARSED ACL for Program Data
PARSED ACL for Microsoft
PARSED ACL for NTDS Quotas
PARSED ACL for Managed Service Accounts
PARSED ACL for WinsockServices
PARSED ACL for RscServices
PARSED ACL for FileLinks
PARSED ACL for VolumeTable
PARSED ACL for ObjectMoveTable
PARSED ACL for Default Domain Policy
PARSED ACL for AppCategories
PARSED ACL for Meetings
PARSED ACL for Policies
PARSED ACL for Default Domain Policy
PARSED ACL for User
PARSED ACL for Machine
PARSED ACL for Default Domain Controllers Policy
PARSED ACL for User
PARSED ACL for Machine
PARSED ACL for RAS and IAS Servers Access Check
PARSED ACL for File Replication Service
PARSED ACL for Dfs-Configuration
PARSED ACL for IP Security
PARSED ACL for ipsecPolicy{72385230-70FA-11D1-864C-14A300000000}
PARSED ACL for ipsecSAKMPolicy{72385231-70FA-11D1-864C-14A300000000}
PARSED ACL for ipsecFilter{72385232-70FA-11D1-864C-14A300000000}
PARSED ACL for ipsecNFA{59319BE2-5EE3-11D2-ACE8-0060B0ECCA17}
PARSED ACL for ipsecNFA{594272E2-071D-11D3-AD22-0060B0ECCA17}
PARSED ACL for ipsecNegotiationPolicy{72385233-70FA-11D1-864C-14A300000000}
PARSED ACL for ipsecNegotiationPolicy{72385234-70FA-11D1-864C-14A300000000}
PARSED ACL for ipsecFilter{72385235-70FA-11D1-864C-14A300000000}
PARSED ACL for ipsecPolicy{72385236-70FA-11D1-864C-14A300000000}
PARSED ACL for ipsecSAKMPolicy{72385237-70FA-11D1-864C-14A300000000}
PARSED ACL for ipsecNegotiationPolicy{59319C01-5EE3-11D2-ACE8-0060B0ECCA17}
PARSED ACL for ipsecNegotiationPolicy{59319C04-5EE3-11D2-ACE8-0060B0ECCA17}
PARSED ACL for ipsecPolicy{59319C07-70FA-11D1-864C-14A300000000}
PARSED ACL for ipsecNegotiationPolicy{59319C08-70FA-11D1-864C-14A300000000}
PARSED ACL for ipsecSAKMPolicy{59319C09-70FA-11D1-864C-14A300000000}
PARSED ACL for ipsecNegotiationPolicy{6A1F5C6F-72B7-11D2-ACFD-0060B0ECCA17}
PARSED ACL for AdminSDHolder

Cody Dumont, cody@melcara.com
Auditing Windows Environments with PowerShell XML Output

Cody Dumont, cody@melcara.com
Auditing Windows Environments with PowerShell XML Output

PARSED ACL for DNS Servers
PARSED ACL for RootDNSServers
PARSED ACL for @
PARSED ACL for a.root-servers.net
PARSED ACL for b.root-servers.net
PARSED ACL for c.root-servers.net
PARSED ACL for d.root-servers.net
PARSED ACL for e.root-servers.net
PARSED ACL for f.root-servers.net
PARSED ACL for g.root-servers.net
PARSED ACL for h.root-servers.net
PARSED ACL for i.root-servers.net
PARSED ACL for j.root-servers.net
PARSED ACL for k.root-servers.net
PARSED ACL for l.root-servers.net
PARSED ACL for m.root-servers.net
PARSED ACL for DFSR-GlobalSettings
PARSED ACL for Domain System Volume
PARSED ACL for Content
PARSED ACL for SYSVOL Share
PARSED ACL for Topology
PARSED ACL for WIN-7J8E90L988E
PARSED ACL for DFSR-LocalSettings
PARSED ACL for Domain System Volume
PARSED ACL for SYSVOL Subscription
PARSED ACL for BCKUPKEY_30b7a6ad-e26e-4166-a63a-551b7e22b986 Secret
PARSED ACL for BCKUPKEY_P Secret
PARSED ACL for BCKUPKEY_7a9808ff-f6af-4bd3-bda0-7169d2a837a7 Secret
PARSED ACL for BCKUPKEY_PREFERRED Secret
PARSED ACL for Joe
PARSED ACL for OU:For.Joe
PARSED ACL for Not:For.Joe
START THE acl_data.XML_REFORMAT FUNCTION
FINISHED THE acl_data.XML_REFORMAT FUNCTION
FINISHED - RUNNING THIS SCRIPT
PS C:\PowerShell\get-AD-ACL.v03.3.c\20111120-105132

Cody Dumont, cody@melcara.com
### Upcoming SANS Training

Click here to view a list of all SANS Courses

<table>
<thead>
<tr>
<th>SANS OnDemand</th>
<th>OnlineUS</th>
<th>Anytime</th>
<th>Self Paced</th>
</tr>
</thead>
<tbody>
<tr>
<td>SANS SelfStudy</td>
<td>Books &amp; MP3s Only</td>
<td>Anytime</td>
<td>Self Paced</td>
</tr>
</tbody>
</table>