Global Information Assurance Certification Paper

Securing Windows
GCNT Practical Assignment Version 2.1b

Using PERL for Network Share Enumeration and Analysis

By: Craig Robertson

OBJECTIVE

The objective of this practical is to give a thorough explanation of how File and Print Sharing for Microsoft Networks is implemented in both NT 4.0 and Windows 2000 systems. This paper will give an overview of the inner-workings of the Windows NT/2000 access control model and how it relates to File and Print Sharing, and I will further illustrate these concepts by creating a few Perl programs that administrators of Microsoft NT Domains can use to track user permissions to file and print shares. It is assumed that the reader is has an intermediate understanding of Perl and how it is used on Windows systems. I still consider myself a Perl novice, and I've tried to make the dialog as accessible as possible.

The Windows NT/2000 Acess Control Model

"The security systems in Windows 2000 are based on technologies originally developed for Windows NT. Both operating systems control access to resources in fundamentally the same way." (Windows 2000 Server Distributed Systems Guide, pg. 1) For this reason, the concepts covered in this section, and the Perl scripts included in this practical, apply to both Windows NT and Windows 2000. For the purposes of the Perl scripts contained in this practical, we are most interested in how the Windows NT/2000 access control model relates to file and print shares.

The access control model is made up of the following characteristics.

User-based authorization – any process that you, as a user, start, runs in your context. This means that if you are opening a document in Excel, you can only open a document to which you, the user, have access. You would not be able to open a document only accessible to another user, or the system account.

Inheritance - If an object, like an NTFS folder is created, the permissions of "child" objects can be controlled through inheritance passed down from the "parent" object. This means that the permissions which are set on that folder are inherited by any new files or folders created inside the "parent" folder. In Windows 2000, the permissions set on a "parent" object are inherited by existing objects as well as newly created ones.

Discretionary access to Securable Objects – Windows NT allows for an owner of an object to control permissions to that object. In the case of network shares, no owner is defined in the object. An object is shared from a server, and the user that creates the share (creation of permanent shared objects is a specific user right) defines the users and groups that have access. In Windows 2000, owners can allow or deny specific properties of certain objects, as well as, allow or deny the entire object. We see this difference in the way share permissions are defined. On a Windows NT share there are four permissions options: FULL ACCESS, CHANGE ACCESS, READ ACCESS, or NO

ACCESS. On Windows 2000, you are able to deny specific properties so that you can allow CHANGE ACCESS but DENY READ. You will see this first-hand in our Perl scripts.

Administrative privileges – This gives the ability to assign specific administrative rights, enabling an administrator to have more control over the rights a user has to the system. In Windows 2000 these privileges can be managed from a central location on a domain user Group Policy.

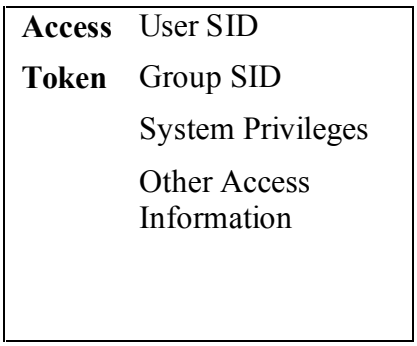
Auditing - NT and 2000 objects can be setup so that all attempted use is recorded.

How Access is Gained (by a Thread)

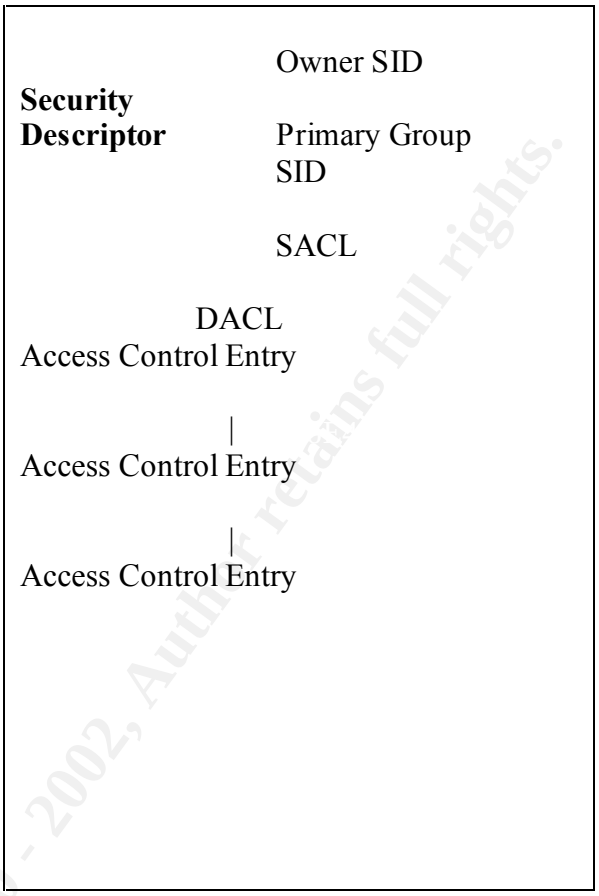
To gain access to an object, a thread's security context must be identified by the operating system. A thread doesn't have a SID (Security Identifier) it must use the SID of the user, or process, that called it.

Obtaining an Access Token

When a user logs on, that user gets an access token. That access token contains the User's SID, as well as, the SIDs of the groups to which the user is a member. There may also be privilege information or other access information for that user. Before a thread initiated by a user can have access to an object, the operating system takes the associated access token and compares it to the objects security descriptor. The security descriptor contains the DACL (the Discretionary Access Control List), the SACL (the System Access Control List), and in some cases (not in the case of network shares) the security descriptor will also contain the Owner SID and the Primary Group SID.



Figures 1 & 2 (from Windows 2000 Server Distributed Systems Guide, pg. 4)



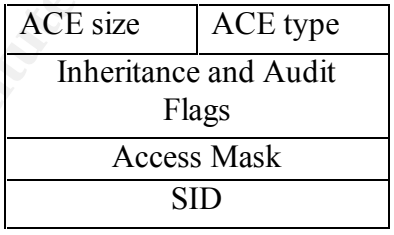
The operating system checks the user's token against each ACE, until it finds one that either allows access or denies it. For this reason, deny ACEs are listed first (although, Windows 2000 Server Distributed Systems Guide, pg. 4, mentions that there are instances where this is not the case).

A Closer Look at Access Control Entries

There are six types of ACEs supported by Windows 2000. Three of these types are generic ACE types and three are Object-Specific ACE types. The three generic are: Access Denied, used in a DACL to deny access; Access Allowed, used in a DACL to allow access; System-audit, which is used by the SACL to indicate that access to the object should be logged. The three object specific types are: Access Denied, object-specific; Access Allowed, object-specific; and System-audit, object-specific. The object-specific ACE types are fundamentally similar to the generic types, the difference is that with the object-specific rights, you have control over the types of objects that inherit those rights. This is so that when a certain type of object, like a computer, inherits rights from another object the parent object's rights are inherited. But in the case of a different type of object, like a user, those same rights are not inherited. It is important to note that these "object-specific" rights only apply to ACLs used in Active Directory, and as such, are not applicable to the environment discussed in this paper (specifically, Windows 2000 and Windows NT existing in an NT 4.0 single-domain).

All three generic ACE types have this data structure

Figure 3 (from Windows 2000 Server Distributed Systems Guide, pg. 35)

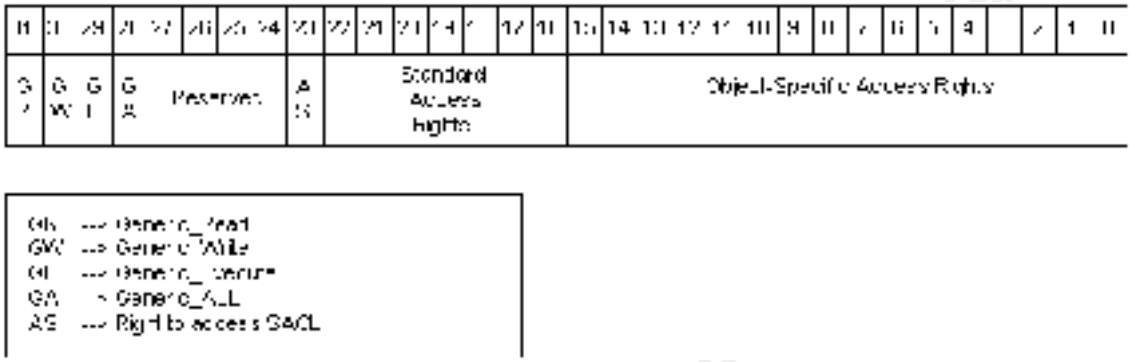


The ACE size is the amount of memory allocated to the ACE. The ACE type indicates what kind of an ACE it is (a deny ACE, an allow ACE, or an auditing ACE). The inheritance flags are used to determine how the ACE is inherited by child objects. Audit flags are specific to auditing ACEs and are used to determine if the object should be monitored for failed access or successful access, or both. The SID represents the user or group whose access is controlled or monitored by this particular ACE. Finally, the Access Mask is the 32-bit value that indicates the rights for the object. Specific bits correspond to specific rights. This access mask, combined with the ACE type, indicates the access that the user or group (determined by the SID) has to the object.

This access mask corresponds directly to the access mask that is dumped in getshares.pl. Unfortunately, I was never able to determine how the values dumped in the

program corresponded to a 32-bit access mask. Microsoft's documentation tells us that in an ACE, permissions are represented by one, or more, bits in the 32-bit access mask (Windows 2000 Server Distributed Systems Guide, pg. 9). The access mask is represented as follows:

Figure 4 (from Windows 2000 Server Distributed Systems Guide, pg. 9)



This was the extent of the information that I could find on the mask from Microsoft. (Nik Okuntseff's book, Windows NT Security, had some additional information that helped, but most of his examples were related to C, and I had difficulty drawing comparisons between the two languages.) Using Dave Roth's Win32::Perms module when we dump a Full Access mask we get the value 2032127. If we convert this to binary we get:

Full Access (2032127)=	11111	0000000111111111
Change Access (1245631) =	10011	0000000110111111
Read Access (1179817) =	10010	0000000010101001
W2K Change Access (1245462) =	10011	0000000100010110
W2K Read Access (1966313) =	11110	0000000011101001
W2K Change Access (2031958) =	11111	0000000101010110

The access that I've listed here was what I determined the access of these masks to be through trial and error. I would create a share and give it a group with Read Access, if that dumped a value of 1966313, I determined that this was a read access mask. I suspect that the Standard Access right bits are what set the rights, but without any more specific documentation, I had to go with my trial and error experiments.

Because I was unable to determine how the 32-bit access mask applied to the values dumped from Win32::Perms used in the getshares.pl script, access is determined by testing against the decimal masks that I'd seen and tested for in my home network. Admittedly, this is a weakness of these scripts and something that you should be aware of if you use these in your own network. If you run the getshares.pl script using a –p flag, you will see the shares and the permission masks that are dumped out with the shares. This is a good way to get comfortable with the access masks that these programs are reporting.

Perl Programs for System-wide Enumeration of Share Rights

OVERVIEW

Windows NT and 2000 servers have become dominant file and print servers in today's networks. Many credit the "usability" of Microsoft products. But for many system administrators, the easiness of Microsoft GUI tools to manage the network (User Manager, Server Manager, etc.) is often offset by the limiting nature of these same tools. Sure, for simple tasks it is a no-brainer, use the GUI. But when the task becomes more difficult, or bigger, you can quickly become drowned in an afternoon of mouse-clicks and (ugh) print screens.

This brings me to my example. As a new Security Analyst in a relatively small NT Single Domain, I was given a task, "Find out what access user "x" has to shares on our network." No problem, I thought. But soon I discovered that there was a problem. To find out the type of access this user had on our network, using Server Manager, I needed to check every shared directory on our network (a total of about 250 shares) to see if the user had permissions to those shares. But wait, that wouldn't totally do it, because most shares had permissions given to groups, not users. Not only that, but I then realized that if my user "x" was a member of several groups that had access to the same share, I had to calculate what their "true" access to that share was! This, to a former Novell NDS Administrator, was almost unthinkable.

Needless to say, that task didn't get completed. In the months to follow I looked on-line and I asked every NT Administrator I could if they knew of a way, or a utility, to solve this problem. There are different Resource Kit Utilities that can help, but these all seemed kind of clunky and weren't designed for the task at hand. And no one knew of any other way to solve this problem.

Then, I discovered the scripting language Perl. And soon after that I found Dave Roth's Win32::Perms module (www.roth.net). It was at this point that I realized it would be possible to script a solution to my problem. The Perl programs that I created for this practical are my solution to the task "Find out what access user "x" has to shares on our network".

DISCLAIMER

As much as I'd like to claim otherwise, I am not a programmer. I have completed introductory courses in Pascal, C++, and Java, but my real experience with programming has come from creating several Perl "hacks" related to security and network administration. At best, the included scripts should be considered "Betas" and you should use them at your own risk. In order to share this project with the SANS community, I had to develop this code at home, away from my corporate network.

Because of this, it has only been tested on my home network, a Single NT Domain with an NT 4 PDC and a WIN2K member server. These programs have been extensively tested on my home network, without any problems, and it is my feeling that they would work equally well in a larger domain. But, this has not been tested.

PREPARATION

I designed these programs with two things in mind, the first of which, was that I wanted to make this a solution that was free. The second, was to make it so that it was an extendable and customizable solution.

To make it extendable, I decided to dump to a database and use that database to manipulate the data. This would, in theory, make it possible to track a user's access, over time. One of my ideas was to take snapshots of your file and print shares to use for comparison. My hope is that over time, an administrator would get a feel of how access to network shares is given and taken away by junior admins (or users sharing folders on their desktops). Or even as a kind of crude IDS ("Hey, why does the mail room guy have Full Access to the Payroll share?"). I also wanted to make it possible to use several reporting interfaces to present the data recorded by these scripts.

To these ends, I used ODBC calls in the Perl code. By using Dave Roth's Win32::ODBC module, you can put the tables, used by these scripts, on any database in your organization that accepts ODBC connections. If you don't own a SQL database, you can always get a free one! On my home network, I used the mysql-3.23.38 for Windows NT/2000 database, a free SQL database distribution (www.mysql.com).

For these programs, I created a databases (I should probably put an "I am not a DBA" disclaimer in here too!) that I called PermissionsDB. I created two tables in this database, to which we will be dumping our data; tbl_group_listings and tbl_share_properties.

```
mysql> describe tbl_group_listings;
+-----------+-------------+------+-----+---------+-------+
| Field     | Type        | Null | Key | Default | Extra |
+-----------+-------------+------+-----+---------+-------+
| GroupName | varchar(25) | YES  |     | NULL    |       |
| UserName  | varchar(25) | YES  |     | NULL    |       |
+-----------+-------------+------+-----+---------+-------+
2 rows in set (0.00 sec)
```

In the above table, you can see that I created two fields, GroupName and UserName. Both of these fields are set to contain a variable length character string with a length up to 25 chars. You want to make sure that the 25 length is long enough to accommodate all of your UserNames and GroupNames.

```
mysql> describe tbl_share_properties;
+-------------+-------------+------+-----+---------+-------+
| Field       | Type        | Null | Key | Default | Extra |
+-------------+-------------+------+-----+---------+-------+
| ServerName  | varchar(25) | YES  |     | NULL    |       |
```

```
| ShareName   | varchar(25) | YES  |     | NULL    |       |
| Domain      | varchar(25) | YES  |     | NULL    |       |
| Account     | varchar(25) | YES  |     | NULL    |       |
| PermsMask   | int(11)     | YES  |     | NULL    |       |
| AccessType  | int(3)      | YES  |     | NULL    |       |
| CompareMask | char(3)     | YES  |     | NULL    |       |
+-------------+-------------+------+-----+---------+-------+
7 rows in set (0.00 sec)
```

In the above table, you can see that I created seven fields. Again I used a length of 25 for the ServerName, ShareName, Domain, Account fields. If you have a server, a share name, a domain over 25 chars in length, it would be necessary to increase these values. The Account field will contain both User and Group Accounts, so adjust the length if appropriate.

The PermsMask field will contain a long integer that is dumped inside of the script. The AccessType field will contain either a 1 or a 0, depending on whether we are looking at an allow or deny mask. And the CompareMask field will get a custom three character access mask that is also created by the script.

Once you get the database and the tables created, you'll want to make sure that you have an ODBC connection setup to this database from the machine that is running the program. You will need the ability to insert and query this database.

MY ENVIRONMENT

This Perl program was developed and tested using Activestate's Perl version 5.005 on my home network. This network consisted of an NT 4 server running as the PDC, as well as a WIN2K server running as a member server. Both were setup to use Microsoft's File and Print Sharing service, and were sharing both network shares and printers. It should be noted that these programs were developed on and tested from the NT 4 PDC. But, this was due to the limitations of my home network. The scipts would also be able to be run from a regular workstation (as long as that workstation has Perl installed with the needed modules) that doesn't participate in file or print sharing.

ENUMERATE GROUPS AND ACCOUNTS

The first thing that is necessary, is to get an up-to-date list of the users in the domain, and the groups to which those users belong. To do this, we use the getgroups.pl Perl script. This script uses two modules, Jens Helberg's Win32::Lanman module (http://manchester.pm.org/ActivePerlDocs/lib/site/lanman/lanman.html) and Dave Roth's Win32::ODBC module (http://ftp.roth.net/pub/ntperl/beta/odbc/). This paper will assume that you have all modules used in these programs installed on the machine from which the scripts are run.

At this point, if you are an administrator new to Perl, you might notice something that I found to be pretty incredible. With modules like the Win32::Lanman module, Perl gives you an interface to different function calls to the Win32 API! For me, this was a revelation. Finally, I had a way to get the information I wanted without using the limited functionality of Server Manager and User Manager.

Back to getgroups.pl---

To run getgroups.pl, you want to give it an argument of a domain controller. On my home network, the PDC is NT40PDC, so I type this command from the command line: perl getgroups.pl NT40PDC. This is the output:

```
E:\myperl>perl getgroups.pl NT40PDC
Connected: writing group and user info to database
```

This tells us that we successfully wrote some information to our database. If the program was unable to connect to the database to write the data we would've received the error: ERROR DID NOT CONNECT. This error indicates that the program was not able to connect the ODBC database. If you look at the source code for the getgroups.pl script, you'll notice that the database name is hard coded into the program as ("PermissionsDB") on line 25. If you've named your database something different, you'll have to change this line in the script to reflect that change.

Now, let's check the data that has been added to the Mysql database.

```
mysql> select * from tbl_group_listings;
+-----------------+---------------+
| GroupName       | UserName      |
+-----------------+---------------+
| Domain Admins   | Administrator |
| Domain Guests   | Guest         |
| Domain Users    | Administrator |
| Domain Users    | NT40_PDC$     |
| Domain Users    | IUSR_NT40_PDC |
| Domain Users    | 2000SERV$     |
| Domain Users    | Hunter        |
| Domain Users    | Jim           |
| Domain Users    | Rick          |
| Domain Users    | Kurt          |
| Gourmet Foods   | Jim           |
| Gourmet Foods   | Hunter        |
| Gourmet Foods   | Rick          |
| Munitions       | Hunter        |
| Nature          | Rick          |
| Nature          | Jim           |
| Science Fiction | Kurt          |
+-----------------+---------------+
17 rows in set (0.00 sec)
```

Now we have a table that contains all the global groups in our domain, and the users that belong to those groups. To get this same info from the GUI, we'd have to open User Manager, click on the global group and record the users in the group (I often did this with a print screen, or pen and paper), go to the next global group, etc. etc. On my small home network, this would not be a monumental task, but what about at work where we have 30 groups and 200 users! What about a domain that has close to Microsoft's maximum recommended SAM entries (40,000 entries of global groups, local groups, computer accounts and user accounts (Sirockman, p. 409))? In situations like these, the GUI is worthless.

By looking at the code for getgroups.pl we can work through what happens. The first lines declare the modules or extensions that we want this script to use (Win32::ODBC and Win32::Lanman). We give $PDC the value of the server that was passed into the program as a command line argument. Then we use the Lanman module to call the NetGroupEnum function. This function requires that we pass it the PDC (actually, any domain controller will do, we're just querying the SAM) and a reference to an array (this will become an array of hashes). This function will then return an array of hashes that includes all of the global groups. The next thing that we do is extract all of the groupnames from that hash and put them into a more manageable array. This is done by looping through the hashes and pushing their values into @globalGroups (see line# 22).

Next, we'll open a connection to our database. With the Win32::ODBC extension, you can see just how easy this is (line# 26). Now, for each global group, we need to identify all the users that are members of each group. We can do this by using Lanman to call the NetGroupGetUsers function. This takes three arguments; the domain controller to be queried, the group whose members we want, and a reference to an array to which an array of hashes will be dumped (line# 38). Now that we have a group name and a list of users that are members of that group, we can write the output to the database, accomplished by a simple SQL insert (line# 41). This is done for each user that is a member of that group. Then that loop is exited and it repeats the process for the next global group, until finished. When that's done, the tbl_group_listings table is now populated with information. Finally, we close the ODBC connection before the program exits.

GETTING THE SHARE INFO (GETSHARES.PL)

The next program is getshares.pl. This program enumerates all shares in the NT domain. It then extracts the ACE's for each of these shares. From these ACE's, the program calculates a custom permission mask, called the "compareMask". The program then creates a hash for each share. Eventually, this hash is populated with data from each permission ACE, as well as that custom "compareMask". Once this information has been obtained, it is written to our database.

Again, the first lines of the program are used to import the Perl mods this program will use. Just like in the getgroups.pl program, we will be using the Win32::Lanman and the Win32::ODBC, modules. But, the real work of the getshares.pl program is done by Dave Roth's Win32::Perms module(http://ftp.roth.net/perl/perms/).

The first step in gathering all of the share information in the domain is getting a list of all of the servers. First we need to know what domain we are in (line #17). Then we can enumerate the servers in that domain by calling Win32::Lanman's NetServerEnum function. If we look at the documentation included inside lanman.pm, we see that NetServerEnum is called this way:

```
=item NetServerEnum($server, $domain, $type, \@info)

Lists all servers of the specified type that are visible in the
specified domain.
```

```
$type can be any of the constants SV_TYPE_*. e.g.
SV_TYPE_SQLSERVER , SV_TYPE_TERMINALSERVER , SV_TYPE_NT .

These can also be combined using |. To get all SQL servers or
terminal servers user $type = SV_TYPE_SQLSERVER |
SV_TYPE_TERMINALSERVER
```

So, in the getshares.pl program, the function is called using the local server for the servername, this is the ‘’. Next, we pass it the domain we are in. Then we pass it the types of servers to enumerate, in this example, SV_TYPE_DOMAIN_CTRL for the PDC, SV_TYPE_DOMAIN_BAKCTRL, for BDC’s, and SV_TYPE_SERVER_NT for the member servers. You could also add SV_TYPE_WORKSTATION if you wanted to enumerate the shares on all the NT workstations (and 2000 Pro) in your domain. The final argument to the function is a reference to an array of hashes, @serverHash. The function will then populate that array with hashes of each server on the network.

Now that we have that array of hashes, we want to turn it into something a little easier to manage. So, we loop through each of the hashes and we extract the name of each server and add it to the array @tempServerList. Lastly, we take that list of servers and sort it into alphabetical order and then destroy @tempServerList.

We now have our list of servers, next we have to get the share information from each of those servers. We can do this by using the Lanman extension, yet again! (There are many, many valuable functions provided by this wonderful interface.) We want to start looping through our list of servers and then extract the share information for each of them. Getting the shares is easy, we just call the NetShareEnum function from Lanman, passing the server from which we want to extract the share information, and another array reference to which we want the hashes dumped.

Next, we loop through that array of hashes and extract the sharename from each hash and put it into a working hash (%serverHash). This hash is a hash of hashes and is structured this way:

```
%serverHash = (
      $server =>  {
            $share => (
                  {  shareName => "xxx" }
                  {  type => "xxx"      }
                    )
                )
            )
```

In addition to the shareName, it is also necessary to extract the share’s type. There are several share types; normal shares, IPC shares, printer shares, null shares, and hidden shares. Each of these have different properties and need to be processed in different ways, therefore it is necessary to know what type of share we are manipulating.

You’ll notice that on line #46, there is a test to see if the share type is not equal to null. If it is not, the share type is assigned to the type value in the %serverHash hash. If it is a null value, the null value is assigned to the type value in the %serverHash directly. It is necessary to assign the value this way because if you just assign the value of the dumped share type, you will end up assigning a reference to a null value, and this will

cause an error. *(When creating this program, I had some problems with exception errors with the data, to combat these errors I've done a lot of error checking like this. Again, because I am a novice programmer, I am sure there are other ways to go about this..)*

Armed with the share name and the share type, we can extract the detail of the share using Win32::Perms to dump the share attributes. First, we need to start looping through our %serverHash hash. We want to dump the share attributes for each share, but first the share type is tested to see if it is of the type –2147483645. This share type represents an Interprocess Communication (IPC) share. This is the share used by a client to communicate with a server. This is a share that is used by the server to receive Remote Procedure Calls (RPC's). As such, this share doesn't have conventional security permissions. In fact, it is a completely open communications channel not protected by any ACE's. Therefore, if the current share is an IPC share, we skip it (just know that every server has this share with it's default permissions—which is none!)

If the share is not an IPC share, $Perm, @Acl, $Ace are created to run in a local scope. The Win32::Perms new function is called and creates a new Win32::Perms object. If this object fails to be created, the program outputs an error message and moves onto the next share.

If the object creation is successful, the ACL's for the object are dumped. These are dumped using an "unless" statement. If the ACL's fail to be dumped, the type of the share is tested. If the share type = -2147483648, it is an Administrative share. Admin shares are the default shares that members of the Administration groups automatically get access too, typically the C$, D$ shares that represent the root of drive partitions. If it is an Admin share, a new hash is created. The structure of the hash is as follows:

```
%serverHash2 = (
      $server =>  {
            $share => (
                  Ace1 =>    (
                             {  ShareName => "<NAMEOFSHARE>" }
                             {  AccessType => "<TYPE>")
                             (  Account => "<ACCOUNT>" )
                        )
                    )
                )
            )
```

This hash is then populated with the corresponding info. "Account" gets the value "ADMIN$" to represent an administrative share. This complex hash structure (really a hash of a hash of a hash of a hash) has the static entry of Ace1. Because an administrative share doesn't have a typical ACL structure, we cheat a little and say that there is only one ACE, and that ACE is an Admin ACE. This will make more sense when we populate the same hash with shares with "real" ACL's, or multiple ACE's.

The next special share type is NULL. If there is no share ACL to be dumped, and the share type is NULL, you have a special type of access, Microsoft's notorious "Null Share". When a share is created, it is automatically created without an ACL. Contrary to what most of us would think, a share with null values (without an ACL) is actually a "wide-open" share. By default, the OS interprets null as full access. This is the explanation for why, when a share is created, the permissions are, by default, Everyone Full Access.

So, when there is a share without an ACL and with a type of 0, we can determine that the Everyone group has full access. The hash is then created with an entry for the Everyone group and given a custom CompareMask of 0xx (the compare mask will be better explained in the following paragraphs).

If the ACL is dumped successfully, then the next step is to extract the information for each ACE (for a description of the differences between ACL's and ACE's see section A Closer Look At ACE's). This is done by looping through @Acl and performing tests on the hash that $Acl represents. But, before this loop is started, $count is initialized to zero. This counter is used for building the hash structure.

In the @Acl loop, each $Ace is extracted. Several local variables are initialized. It is important to initialize all of these variables as variables local to this scope, so that they are properly destroyed when the scope is exited. Most of these local variables are self explanatory.

Again we don't want to run into the trouble of assigning a reference to a null value, so we need to test for null. The hash $Ace->{Account} is tested for a null value. If it is not a null value, we assign the hash value $Ace->{Account} to the local account variable. Then we do a similar test of the $Ace->{Mask} hash. Notice that in the case of $Ace->{Account} we tested for a null share by using *eq ""* , with $Ace->{Mask} we tested it with *==0*, this is because {Mask} is a number and {Account} is a string.

Now that the Ace has been successfully dumped, the next step is to analyze the permission mask. The first test is against the Ace 'type'. The 'type' indicates either an allow mask (type is equal to 0) or a deny mask (type is equal to 1). Once tested for mask type, the permission mask of the Ace is then evaluated and a corresponding compareMask is inserted into %serverHash2.

These configuration masks have been determined by dumping the Ace's, from a share, and comparing them with the share's known permissions. I found that the NT 4.0 shares were easy, 2032127 was equivalent to Full Access, 1245631 was equivalent to Change, and 1179817 was equivalent to Read. And these always seemed to stay constant. The 2000 shares were not so cut and dry. The NT permission masks applied to 2000 shares, but so did some others. 1966313 seemed to also indicate CHANGE access. And with Windows 2000 you could have shares that denied read but allowed change, and vice-versa (essentially, a share that denies read but allows change is a deny all). I've included the values that I have tested as accurate, others might have different results. If the permission mask is unknown, it is given a compareMask of ???. There is more explanation of the Permission Masks and my problems with it, in the Access Control Model Section. Please note that if you run the getshares.pl script with a –p option, the permission masks for each ACE will be dumped to the screen. You can then compare these numbers with the access rights displayed by Server Manager.

With the $compareMask variable assigned, we can complete the creation of %serverHash2. The structure of the completed hash is as follows:

```
%serverHash2 = (
      $server =>  {
            $share => (
                  $aceCount =>      (
```

```
                                    (  ServerName => "<SERVERNAME>"  )
                                    {  ShareName => "<NAMEOFSHARE>" }
                                    (  Domain => "<DOMAINOFACCOUNT>"  )
                                    (  Account => "<ACCOUNT>" )
                                    (  PermsMask => "<DUMPEDMASK>"  )
                                    (  AccessType => "<ALLOWORDENY>" )
                                    (  CompareMask => "<CUSTOMMASK>" )
                                       )
                              )
                         )
                    )
```

Look familiar? This hash structure matches the table *tbl_share_properties* in the *PermissionDB*. The next step will be to dump this data into that database.

The final step is to take the information that has been extracted and dump it to the database. Conveniently, all of our information is in a hash structure. To efficiently insert this data into our database, a SQL INSERT statement needs to be created. Because much of our data contains NULL values, it is necessary to be cautious in building the INSERT statement. It is only necessary to insert the data that is contained in the hash structure, attempting to insert NULL data with a SQL statement would cause an error. To fix this, the program takes each instance of the %serverHash2{$server}{$share}{$aceCount} values and checks them for null values, if the values are not null, it adds them to @valueString and it adds the corresponding column in the database to the @columnString. Once each of the hash values have been tested, and pushed into their arrays, the array is joined by commas and the appropriate SQL statement is created.

For example, this is what the hash structure would look like for a NULL share.

```
%serverHash2 = (
      NT40PDC =>  {
            TESTNULLSHARE => (
                  1 =>  (
                                    (  ServerName => "NT40PDC"  )
                                    {  ShareName => "TestNullShare" }
                                    (  Domain => ""  )
                                    (  Account => "Everyone" )
                                    (  PermsMask => ""  )
                                    (  AccessType => "" )
                                    (  CompareMask => "0xx" )
                                       )
                              )
                         )
                    )
```

The values that were not null would be pushed into two arrays:

```
      @columnString =
            ('ServerName','ShareName','Account','CompareMask')

      @valueString =
            ('NT40PDC','TestNullShare','NULL$','0xx' )
```

These arrays then get joined into a variable:

```
$columnString = ('ServerName,ShareName,Account,CompareMask')
$valueString = ('NT40PDC,TestNullShare,NULL$,0xx')
```

The final step is to use these variables in the SQL statement:

```
SQL query = "INSERT INTO tbl_share_properties ( $columnString )
VALUES ($valueString)"
```

Because of the way Perl works, the above query is equivalent to:

```
SQL query = "INSERT INTO tbl_share_properties
('ServerName','ShareName','Account','CompareMask') VALUES
('NT40PDC','TestNullShare','NULL$','0xx')"
```

This is done for each $ACE in the hash and then, that's it! The data has been written. Now let's go do something with it!

ENUMERATING A USER'S SHARE PERMISSIONS IN A DOMAIN (GETUSERPERMS.PL)

Both of the previous programs worked to gather information on the shares that exist in the Domain. But we have not found a solution to, "Find out what access user "x" has to shares on our network." Getuserperms.pl is the solution.

After getgroups.pl and getshares.pl are run, all the information that is needed for us to learn the access that user "x" has on the network, is found in those two tables. Getuserperms.pl extracts that information and applies logic to it, and, hopefully gives us the answer to our problem.

To extract information from the PermissionsDB, Dave Roth's Win32::ODBC module is, once again, used. When the program getuserperms.pl is called from the command line, it is called with an argument of the Username for whom a list of access is desired (our example user will be 'Jim'. When the array accountList (@accountList) is created, it will be filled with the groups to which the user belongs.

Immediately, the group Everyone is given to the user. This is because every user is a member of the everyone group by default. If you open the user's group listings in User Manager, you'll notice that the Everyone group doesn't exist. However, we know that there are many shares, such as null shares, that give access to the group everyone. Taking all of this into account, this quick hack is the easiest solution to the problem

After the Everyone group is added to @accountList, a connection is made to the database, this SQL query is run.

```
SELECT GroupName FROM tbl_group_listings WHERE UserName = 'Jim'
```

This query returns all of the groups to which the user ('Jim') belongs. Each of these groups is then pushed into @accountList.

The next step is to create another query. From the table, tbl_share_properties, we want to extract the name of the server, the share name, and the compare mask for each share to which Jim has access. To accomplish this, it is necessary to pull the data from each share that Jim has specific access to, as well as, all the shares to which Jim has access through his group memberships. This SQL statement is created by adding additional text to each variable in the accountList array. If the variable from the array is Domain Users, it then becomes "Account = 'Domain Users'. After each element of the array gets this makeover, the array elements are then joined by " OR ". This is then used to create the necessary SQL statement.

```
Jim is a member of Domain Users, Nature, and Gourmet Foods.  To query
all of the applicable Ace's in the share_properties table, a SQL
statement like this is needed:

SELECT ServerName,ShareName,CompareMask FROM tbl_share_properties WHERE
ACCOUNT = 'Domain Users' OR ACCOUNT = 'Nature' OR ACCOUNT = 'Gourmet
Foods'
```

The database is then queried using this SQL statement. The data returned by the query is then used to populate the %masks hash.

Now we have a list of all the shares to which the user has access. And we also have a list of all the different ACE's the user has to those shares. The trick is to then calculate what the true access is to that share for the user. This is the reason why CompareMask was created. Fortunately, the access to shares is cumulative (in these scripts we haven't included factoring in the access of the NTFS rights of the files or the folder, it is only used to determine share level access) with a deny flag that overrides all allowed values. To calculate that cumulative access, we need to compare two masks, calculate what the cumulative mask is between those two masks, and then take that cumulative mask and compare it to the next mask in the array. For example:

```
We have a user (Jim) that is a member of three groups:

Domain Users
Nature
Gourmet Foods

On the share \ContempLit, on the server \\NT40PDC, the following are
the groups access and the corresponding CompareMasks:

Domain Users = READ     (xx0)
Nature = CHANGE         (x0x)
Gourmet Foods = READ    (xx0)
```

The program first creates the default mask of (xxx). (This is a hack necessary because the first time we are in the loop on line# 21, it is necessary to pass two variables, so $tempMask needs to be defined before the loop is entered.) Inside of the loop, the $tempMask is passed to the subroutine CompareMasks along with the current CompareMask value from the hash (%masks) that is being looped through. The subroutine calculates the mask by comparing each character of the mask.

```
   TempValue  *  Domain Users  = CumulativeValue

      ( xxx ) * ( xx0 ) = ( xx0 )

xx0 is returned, and then that is compared to the next CompareMask


   TempValue  *  Nature = CumulativeValue

      ( xx0 ) * ( x0x ) = ( x00 )

x00 is returned, and then that is compared to the next CompareMask

   TempValue  *  Gourmet Foods = CumulativeValue

      ( x00 ) * ( xx0 ) = ( x00 )

This is the last CompareMask so ( x00 ) is the final value.  This value
is added to the %mask{$server}{$share} hash as the FinalMask hash
value.  Then this process continues for each share to which the user
has access.
```

The last part of the program takes the FinalMask values for each share and turns them into a more "human readable" format. To do this we simply have to send the FinalMask to the subroutine MakeMaskReadable and a human readable format is returned. The human readable format is calculated by multiple if comparisons. These tests compare which values are denies and which values are allowed, and then a recognizable value is returned (like "CHANGE ACCESS" or "Empty Mask. NO ACCESS", etc.)

Also, if any part of the mask contains a ? character, the share is automatically given the UNKNOWN ACCESS value. This is because I am not confident that I've found every type of access combination (particularly for Windows 2000 shares). So, if this value is returned, this share will need to be enumerated manually using the Server and User Managers

That is it, running this program gives us the answer to the problem , "Find out what access user "x" has to shares on our network."

OTHER POTENTIAL USES OF THESE SCRIPTS (OR SCRIPT'S SIMILAR TO THEM)

I didn't have time to create any more scripts that would be able to use this data, but there are a couple of possible programs that I can think of already.

1) A program that lists all the users and their access to a particular share. This wouldn't be too bad, we already have all the shares in a table with the different ACE values that they have. The challenge with this type of program would be getting the access outputted for the particular users. To do this you would have to return a list of all the users that have access, and then feed those into a program similar to the getuserperms.pl program. One of the differences would be that you

would only have to calculate the permissions on one share (but for multiple users).

2) One of the other possibilities, that I am excited about, is using the getshares.pl and the getgroups.pl to monitor the changes that occur on the network. This could work to supplement an intrusion detection system by keeping tabs on the changes to user rights and groups that occur on your network. By dumping a new table every week, or month, you could compare the permissions on your network today, to what the permissions were like six months ago. In an environment where there are multiple System Administrator's, each making changes to the network, unbeknownst to the other, a system that monitors changes to user accounts, share permissions and group permissions is a necessity. How many times has a user been added to a share "temporarily" and then forgotten? I know I'm guilty of it. (They have to make those sticky notes, stickier!)
3) There are probably many others, and the beauty of dumping the data to an ODBC accessible database is that information is a SQL statement away!

POTENTIAL PITFALLS

The major potential pitfall is new permission masks, or permission masks that are currently not included in the getshares.pl program. As I outlined before, I tested and added permission masks defined through trial and error. I did my best to research how the mask is determined, but I was never able to make the connection between the value of the permission mask which the Win32::Perms module returns and the 32-bit Windows permission mask that it represents. To list the permission masks that are being dumped by the getshares.pl script, add the –p parameter to the command line. Finally, it should be noted that I haven't included any tests for the access to shared printers with any of these scripts.

BIBLIOGRAPHY

Sirockman, Jason. MCSE Training Guide: Windows NT Server 4 Enterprise, Second Ed. New Riders Publishing, 1998.

Roth, Dave. Win32 Perl Scripting: The Administrator's Handbook. New Riders Publishing, 2001.

Roth, Dave. Windows NT Win32 Perl Programming: The Standard Extensions. Indianapolis: Macmillan Technical Publishing, 1998.

Okuntseff, Nik. Windows NT Security.. Lawrence, KS: R & D Books, 1997.

Windows 2000 Server Distributed Systems Guide, Microsoft Press, 1999, Chapter 12