Technical Disclosure Commons

Defensive Publications Series

May 2020

Mechanism for Identifying Export Rules for a Given Subnet from an Export Rule String

Kyle Seipp
Pure Storage, Inc.

Jesse Salomon
Pure Storage, Inc.

PURESTORAGE

PURE STORAGE DEFENSIVE PUBLICATION

Mechanism for Identifying Export Rules for a Given Subnet from an Export Rule String

Kyle Seipp

Jesse Salomon

Mechanism for Identifying Export Rules for a Given Subnet from an Export Rule String

April 20, 2020

1 Background

Customers want to know which IP addresses on which systems have misconfigurations. However, it is difficult for them to determine which addresses and subnets have this issue because the export rules explain what the current rules are, but not how they apply to a given IP or subnet. We have developed an algorithm to help determine the rules for their chosen IPs and subnets.

For example, the customer would give us some input rule subject like "1.2.3.4" or "1.2.3.0/28" as well as a Export Rule String like "1.2.3.4 foo 1.2.3.5 bar 1.2.3.4/30 bat". And we want to be able to output what the export rules are for the inputed NetObj and where they come from. In the first example we would want to output "1.2.3.4" has rule "foo". In the second example we would want to output

Input NetObject String	Breakdown	Inherited From	Rules
1.2.3.0/28	1.2.3.0/30	-	Deny
	1.2.3.4	1.2.3.4	foo
	1.2.3.5	1.2.3.5	bar
	1.2.3.6/31	1.2.3.4/30	bat
	1.2.3.8/29	-	Deny

1.1 Key Definitions

- A NetworkObject or NetObj is an IP address or subnet. It can be ipv4 or ipv6 based
- A rule subject is a NetObj in an export rule string that has some rules associated with it.

2 Brief Summary of NFS Export Rules

An NFS export rule is a list of settings for a given Filesystem and which IPs and/or subnets these settings should apply to. In general, an export rule is

made up of alternating a rule subject, and then some set of rules that apply to it. These rules take precedence in the order of specific IPs from left to right, then subnets from left to right, and then any global wildcard rules. However, if any rule subject is repeated, use its rightmost instance. This format is well-known, public and documented as part of the NFS spec. An example rule string is described in the background section above.

Typically one parses these rules into a struct of a 3-tuple made up of 2 lists and a string. The first list represents the IP rule strings. It is a potentially empty list of 2-tuples that are the IP and the rules associated with that IP. The second list is the same, but for subnets. The string is the list of rules for the wildcard subnet. This string is potentially empty if there are no wildcard rules. An example of this data structure would look something like this

```
struct ExportRuleDataStruct {
list<tuple<str, str>> ip_rule_list;
list<tuple<str, str>> subnet_rule_list;
str rule_string_for_wildcards;
}
```

```
([("ip1","rule string for ip1"),("ip2","rule string for ip2")],
 [("subnet1","subnet1 rules"),("subnet2","subnet2 rules")],
 "rule string for wildcard subnet"
)
```

Note that transforming the export rules into such a data structure is well known and standard via the NFS spec. Thus, we will discuss operating on such a data structure interchangeably with operating on an explicit export rule string. Further, note that as part of creating such a data structure, we automatically collapse repeated subjects to their rightmost instance and preserve the order of the IPs and subnets from left to right.

3 Naive Approach

Suppose one wished to solve the problem as discussed in the Background section without using the algorithm that will be discussed below. They would have as input a NetObj and an export rule string. They can trivially convert that export rule string into an export rule data structure as described above. If the input rule subject is an IP and explicitly listed in the data structure, it can be found in O(number of rules) time by explicitly searching through the data structure. If is a subnet, then it takes at least $O(rules^2)$. The reason for this is that we must check each IP and subnet to the left to see if it is a subset. We must do this recursively to ensure that any of *those* sub-subnets have no sub-subnets or IPs. If the input rule subject is a strict subset of a subnet in the data structure, we can also find it in O(number of rules) time. However, if the input subject is a superset of subnets in the data structure, we must look for every subset. We must then join them together and check that this makes up the entire breadth of the input subject. This joining is not entirely trivial because some subnets

might be fully covered by explicit IPs. It is the process of joining quickly and easily that is the main subject of this patent. To naively check that all IPs and subnets that are part of the input subject are represented, we must explicitly look at every IP in the region. In the worst case, this can be 2^{32} IPv4 addresses, which is prohibitively expensive and even worse for IPv6.

4 Algorithm Description

4.1 Converting from Rule String to Data Structure

As mentioned in Section 2 above, we consider this process a given from the NFS Spec. If the structure is empty, we can short circuit the rest of the algorithm and respond with a blanket deny. We can convert all of the IPs in the structure into trivial subnets. For example an ip "1.2.3.4" would be converted into "1.2.3.4/32". We can also convert any wildcard rules into rules associated with the maximal subnet – "0.0.0.0/0". This means that the entire data structure is made up of subnets. Note that we will preserve ordering. Therefore, the leftmost elements will be trivial subnets if there are any and the rightmost subnet will be 0.0.0.0/0 if there is a wildcard entry.

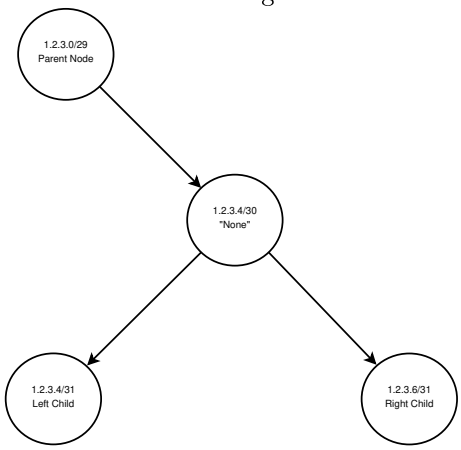
4.2 Building a Network Tree

4.2.1 Defining a Network Tree

We are going to build a data structure we call a Network Tree. This structure is a binary tree made up of nodes. A node of the Network Tree is a struct made up of five parts, a subnet, a rule source, and pointers to its parent and both children. The subnet is the name of the node and represents what part of the NetObj space the node represents. The rule source represents which set of export rules apply to this node. Note that "None" is a valid value for rule source. The pointers to parents and children represent how the nodes attach to one another. Note that a node is either a leaf node or it isn't. A leaf node has no children. A non-leaf node must have both of its children. A Tree is defined by a root node, and its descendants. The parent of the root node is None.

The subnets of nodes and the relationships between nodes is deterministic and depends entirely on the subnet. We will describe how this works using IPv4, but this will work in exactly the same manner with IPv6. A given node has a subnet property. For example, 1.2.3.4/30. This subnet can be partioned into two halves – 1.2.3.4/31 and 1.2.3.6/31. Thus, those two nodes are the two children of the 1.2.3.4/30 node. Since 1.2.3.4/30 and 1.2.3.0/30 completely partition 1.2.3.0/29, they are the children of 1.2.3.0/29. Thus, 1.2.3.0/29 is the parent of 1.2.3.4/30. Note that a parent will always have a netmask that is one smaller and that a child will always have a netmask that is one larger. Note that some subnets are of size 1 and correspond to exactly one IP address, like 1.2.3.0/32. These nodes will never have children. Additionally 0.0.0.0/0 has no parent because it contains the entire NetObj space.

Figure 1: Network Tree Node Example



4.2.2 Adding Nodes

Suppose we have an existing Network Tree as in the example from Figure 1. Note that in the figure, only one node is shown because the parent and children nodes don't have a rule source. They are shown to explain what the parent and child would be. How would we add a child to the 1.2.3.4/30, say 1.2.3.4/31. We see that the current node has the new node as a direct child. So, we create a new node with subnet 1.2.3.4/31 and set the child point from the parent and the parent pointer in the child to point to one another. We can set its rule source. We also must create the **other** child node of 1.2.3.6/31. Now we are done.

What if we want to add a descendant that is not a direct child? We create both children of the current node. Then we determine which of those is an ancestor of the target node. Then, we create that nodes children. Continue in this manner until we create the child that we intended as a leaf node. It is fully connected to the root node that we started with.

What if we want to add an ancestor? We create the parent node of the root node of the Network Tree and set up the pointers. Then, we set the parent as the root node of the tree. Then we set up the other direct child of the parent node. This is the node that is the sibling of the original root node. This keeps all nodes having either 0 or 2 children.

What if we want to add a node that is not a descendant of the root node, and is not an ancestor? This means that this node must be "adjacent" in some sense, though perhaps not a sibling. Create parent nodes (appropriately as described above) until one of them is an ancestor of the target node. Then, we can follow the rules for targets that are descendants of the root node as described above.

4.2.3 Algorithm Steps

1. We are going to build a NetworkTree from the elements of the Export Rule Data Struct. To do so, we are going to loop over the NetObjs in the struct starting with the first subnet.

2. Create the node representing the element and set the rule source to the element. This is the root of the tree.

3. Consider the next NetObj in the Data Struct. Call it the current element.

4. Start from the root and add the current element to the tree. The new element must be either a descendant of the root node, an ancestor of the current node, adjacent to the current node, or must be the current node. In section 4.2.2 above we have already described how to add nodes to the tree in all of these cases. If the current element is equal to the root node and the root node has None as its rule source, set the rule source to the current element. If the rule source is already set, then we are done with this element and we can return to step 3.

5. If the current element is not equal to the root node, we want to create the new node (and the connecting nodes) as discussed above in section 4.2.2. The connecting nodes should have the rule sources left as None. The new target node should have its rule source set to that of the current element. As discussed above, we also must create the children of the new target node if they have not already been created. When you set the rule source for any node as not None, you then look to see if we've already defined children nodes. If we have, then check those children - each one which currently has rule source as None will set its rule source recursively (thus itself also checking for existing children and such). If we haven't, then stop and don't bother creating the children. Notably, if we find a child with a rule source which is already set, then we don't have to check its descendants - any that exist will guaranteed have the correct not-None rule sources. We are now done with the current element and can return to step 3 to get a new element.

6. When we have completed every subnet in the Data Structure (including the wild card entry) we are done building the NetworkTree.

Note that this means that all nodes have either 0 or 2 children and that all nodes with 0 children (leaves) have a non-empty rule source.

4.3 Using the Network Tree

Now that we have built this Network Tree, we can use it along with the original Data Structure, and the input NetObj to build the table that the customer wants. There are a few cases.

1. If the input NetObj is a specific IP address, treat it as if it were the trivial subnet.

2. Suppose the input NetObj is a subnet that is a descendant of the root node. We start at the top of the Network Tree and move from node to child based on which of the two children will contain the input NetObj. If we encounter a leaf node or a node with the same subnet as the input NetObj, consider the entire subtree with this this node as the root. Look through all this subtree for their rule sources and look up the rule sources of the leaf nodes and put those into the table.

3. Suppose the input is an ancestor of the root node. Keep track of the current root node. Much like step 3 in section 4.2, we want to create parent nodes (and the empty sibling nodes) to the Network Tree's root until we reach the specified ancestor. The result will look something like this. Now, we can mark all of those sibling nodes to the table as having a rule source of "-". When we put these into the table, they will output "Deny" to represent the fact that the export rule string should deny these IPs access. Then we can traverse the current tree to get the existing rules. Note that we do not want to save these temprorary expansions to the Network Tree in order to save space. They are trivial to construct, so they are not valuable for performance. To revert to the original tree, we can simply use the original root node that we saved at the beginning of the step. The other nodes have nothing in memory pointing to them, so they are discarded. This is implicit based on implementation details.

4. Suppose the input is adjacent to the root node. This means that there are no rules in the export rule string that describe it. Thus, the entire table should be "Deny".

5 Advantages Over the Naive Approach

This approach trades a trivial amount of memory in order to make this problem much faster in the normal use-cases and tractable in the more expensive cases. Additionally, because we can cache the Network Tree, subsequent runs with the same set of export rules can be made even more cheaply. However, under worst case scenarios, this approach will still not perform well.

First, we will compare worst cases for both approaches. The worst case for the naive approach is an input NetObj of 0.0.0.0/0 and the entire rule string is made up of explicit IPs. This will take 2^{32} checks of each explicit IP that can be in the NetObj against a 4096 character export string. How does this case fair for our improved approach? Each individual rule in the string will take about 10 characters at the shortest, so it will result in about 400 leaf nodes in the Network Tree. If they are maximally spread out, this means we need to traverse the entire tree from root to leaf, 400 times with a depth of 32. This works out to $400 * 32 = 2^2 5^2 2^5 = 2^7 5^2 < 2^{12}$

The worst case for our improved approach is when we have to build a large Network Tree. This occurs when we have a large number of explicit rules, especially with spread out explicit IP addresses because it means we have more intermediate nodes. Additionally, the worst case is the largest input NetObj because it requires either having a large tree or temporarily increasing its size during execution. Note that this is exactly the same as the case above.

The best case scenarios for both algorithms is the case where the customer provides empty rule strings or trivial input NetObjs. In these cases both algorithms will behave similarly by short circuiting to the right answer.

What happens in the most common medium cases? In the naive approach we have some input NetObj that is not completely covered by explicit IP addresses in the export rule string and some of the subnets likely overlap with either each other or the explicit IPs. I claim that this is common because the use case for this system is to diagnose poorly set up export rules. Thus, once we have exhausted the explicit IPs, the rest must be done by exhaustive lookup. This scales with $O(n)$ where n is the size of the subnet. Note that the size of the subnet is $2^{\text{value of the subnet size}}$. How does this work in the new approach? Well we build up a sparse NetworkTree and we are able to find a specific leaf node in $O(\log(n))$ time. The majority of the leaf nodes are not going to be size-1 subnets. If they are, then the whole process will take $O(n)$ time which is no faster than the naive approach. But if some of the leaf nodes are caused by having non-trivial subnets as leaf nodes, then we can run significantly faster.

6 Misc Notes

- This system works regardless of IP version because IPs and subnets always have the same subsystem properties. The only difference would be the upper and lower bounds on the number of rules and their sizes.