iC2I: What Virtuals Rely on this iRule?

Don MacVittie, 2008-28-04

The last two articles and a little iRule learning I was doing got me to thinking that knowing which Virtual Servers will be impacted if you make a change to a given iRule is a good idea. We change iRules pretty regularly around here, and knowing what we're impacting is never a bad thing. So I sat down to figure out how we would do just that. For the simplest case - finding a list of Virtual Servers that directly use a given iRule - it is simple. A couple of calls to iControl and you're done. But there's a catch. When we got digging into it, we realized that an iRule associated with an Authentication Profile or a Persistence Profile could be running against any given Virtual Server.

The crux of what I said above is that a Virtual Server has a set of iRules assigned to it and a Profile assigned to it. If that Profile is Authentication or Persistence, then the profile could have an iRule assigned to it. Thus, we have a case where a Virtual could be affected from two different directions. Not a huge deal, but for the i2CI series, this turned into a much larger project than we had anticipated - because we wrote classes to stub ProfileAuth and ProfilePersistence from the API, and then had to add all of the code necessary to check these two for iRules that are relevant. Thanks to Lori for helping me find the small program that was screaming to get out of the first version of this article!

First up, changes we need to make to the BigIpVirtualServer class. We need several methods for this class to suit our needs. At the top level we need a routine that we'll call findVirtualServersUsingRule() that takes the name of a rule in a Java String as parameter and returns a list of Virtual Servers (or a list with one element - an empty string if none rely on a rule by that name). Next, we need to work on the component parts - We'll need a routine to get the rules assigned to a given Virtual Server, another to get the Auth Profiles (if any) assigned to a given Virtual Server and a third to get the Persistence profiles (if any) assigned to the given Virtual Server. We already have the getList() routine to give us a list of Virtual Servers on the BIG-IP, so we should be covered. The only thing left to do is to create the ProfileAuth and ProfilePersistence classes to stub the iControl APIs, and give them routines to find what rules a given profile relies upon.

But we'll look at the current BigIpVirtualServer class. Note that the code from previous i2CI articles is left in this class so that you have the complete compiled class in one place. Should this article series go on long enough we may not always be able to do that, but for now it works just fine.

```
package iC2i;

import java.util.Arrays;
import java.util.Vector;

import iControl.LocalLBProfileType;
import iControl.LocalLBVirtualServerBindingStub;
import iControl.LocalLBVirtualServerLocator;
import iControl.LocalLBVirtualServerVirtualServerProfileAttribute;
import iControl.LocalLBVirtualServerVirtualServerRule;


public class BigIpVirtualServer extends BigIpBase {
 LocalLBVirtualServerBindingStub stub = null;

 public BigIpVirtualServer(String username, String password, String addressOrName) throws Exception {
 super(username, password, addressOrName);

 // Get a connection to the BIG-IP for this instance (yeah, could get big if you're not careful)
 connect();
 }
```

```
private void connect() throws Exception {
stub = (LocalLBVirtualServerBindingStub) new LocalLBVirtualServerLocator().getLocalLBVirtualServerPo
stub.setTimeout(6000);
}

public String[] getList() throws java.lang.Exception {
return stub.get_list();
}

public String[] getDefaultPools(String virtualNames[]) throws java.lang.Exception {
return stub.get_default_pool_name(virtualNames);
}

public String getDefaultPool(String virtualName) throws java.lang.Exception{
String vNames[] = {virtualName};
String retval[] = getDefaultPools(vNames);
return retval[0];
}

public String[] getAuthProfiles(String virtualServer) throws java.lang.Exception {
String retVal[] = {""};
String virts[] = {virtualServer};
LocalLBVirtualServerVirtualServerProfileAttribute profiles[][] = null;

// Get the list of Profiles for this Virtual Server on the BIG-IP.
profiles = stub.get_profile(virts);

// Translate the list into something we can use as a Java object.
for(int i = 0; i < profiles[0].length; i++)
if(profiles[0][i].getProfile_type() == LocalLBProfileType.PROFILE_TYPE_AUTH)
retVal[i] = profiles[0][i].getProfile_name();

// Return the list.
return retVal;
}

public String[] getPersistenceProfiles(String virtualServer) throws java.lang.Exception {
String retVal[] = {""};
String virts[] = {virtualServer};
LocalLBVirtualServerVirtualServerProfileAttribute profiles[][] = null;

// Get the list of Profiles for this Virtual Server on the BIG-IP.
profiles = stub.get_profile(virts);

// Translate the list into something we can use as a Java object.
for(int i = 0; i < profiles[0].length; i++)
if(profiles[0][i].getProfile_type() == LocalLBProfileType.PROFILE_TYPE_PERSISTENCE)
retVal[i] = profiles[0][i].getProfile_name();

// Return the list.
return retVal;
}

public String[] getRules(String virtualName) throws Exception {
String retVal[] = {""};
String arrayVirtual[] = {virtualName};
LocalLBVirtualServerVirtualServerRule rules[][] = null;
```

```
// Get the list of Rules for this Virtual Server on the BIG-IP.
rules = stub.get_rule(arrayVirtual);

for(int i = 0; i < rules[0].length; i++)
retVal[i] = rules[0][i].getRule_name();

// Send the caller a list of rules - or an empty list.
return retVal;
}

public String[] findVirtualServersUsingRule(String ruleName) throws Exception {
String retVal[] = {""};
Vector rules = new Vector();;
int j = 0;
// create objects representing auth and persistence profile interfaces on the BIG-IP.
BigIpProfileAuth authProfile = new BigIpProfileAuth(username_, password_, address_);
BigIpProfilePersistence persistProfile = new BigIpProfilePersistence(username_, password_, address_)

// Create a temporary holding spot for the rules used by profiles.
String profileRules[] = null;

// First, get the list of Virtuals
String[] servers = getList();

// Take the list and ask it which ones rely upon our rule.
for(int i = 0; i < servers.length; i++) {
// Get the iRules for this server.
// Since we're returning an array, but we want a Vector for easier use later, convert.
String vipRules[] = getRules(servers[i]);
if(!vipRules[0].isEmpty())
rules.addAll(Arrays.asList(vipRules));

// Now get the lists of auth and persistence profiles for this virtual server.
// Again we'll take advantage of the "single trip across the network" functionality of
// sending arrays, and just make one call for each type.
String authProfiles[] = getAuthProfiles(servers[i]);
String persistProfiles[] = getPersistenceProfiles(servers[i]);

// And get the list of rules for each Auth Profile.
profileRules = authProfile.getRuleNames(authProfiles);
if(!profileRules[0].isEmpty())
rules.addAll(Arrays.asList(profileRules));

// And the list for each Persistence Profile.
profileRules = persistProfile.getRuleNames(persistProfiles);
if(!profileRules[0].isEmpty())
rules.addAll(Arrays.asList(profileRules));

// The rules Vector now has all the rules this Virtual relies on, so ask if it has ours...
if(rules.contains(ruleName))
retVal[j++] = servers[i];

// Now clear the vector for the next iteration so we don't get false positives.
rules.clear();

}
```

```
 return retVal;
 }
}
```

Note that in findVirtualServersUsingRule() we get the list of servers and then find all rules each Virtual Server in the list is dependent upon in a loop. The only negative about this implementation is that the entire loop will execute even if we could have just added the Virtual Server after the call to getRules() - who cares if one of the profiles uses the rule we're looking for if we know that the Virtual uses it directly - but for our sample purposes this is less complex, and the class is getting big for a sample to begin with.

The routines getAuthProfiles() and getPersistenceProfiles() are the same except for which profile type we're looking for. They could be implemented as a single routine, and a future iC2I may do just that - when we query a Virtual for all profiles it will be easier to use a more generic version of the routine than to write one for each major profile type.

All of these routines return an array of strings with the first element empty if there are no matches. This eliminates the need to worry about nulls, but we could have implemented with null just as well.

The guiding bit for findVirtualServersUsingRule() is to build a list of all of the rules the server being examined relies upon, then ask the list if our rule is in it, then dump the list and start over for the next Virtual Server. Elegant? Not really, but pretty effective.

Next up are the two profile classes, which are basically the same but we implemented as independent classes because then we're set for the future when we do more in this series with Profiles.

```
public class BigIpProfileAuth extends BigIpBase {

 private LocalLBProfileAuthBindingStub stub;
 public BigIpProfileAuth(String username, String password, String address) throws Exception {
 super(username, password, address);
 connect();
 }

 public void connect() throws Exception {
 // Get a connection to the VirtualServer services on the bigIP and ask it for the list of rules for
 stub = (LocalLBProfileAuthBindingStub) new LocalLBProfileAuthLocator().getLocalLBProfileAuthPort(new
 }

 /**
 * getRuleNames - get the list of rules the indicated profile depends upon.
 * @param authProfileNames[] an array of Authentication Profile names
 * @return array of strings with all rules these profiles use in it.
 * @throws java.lang.Exception there are several exceptions thrown by the remote routines, we lump th
 *
 */
 public String[] getRuleNames(String authProfileNames[]) throws java.lang.Exception {
 String ret[] = {""};
 LocalLBProfileString rules[] = null;

 rules = stub.get_rule_name(authProfileNames);

 // We now have a list of ProfileStrings, which contain two values, get the string out of it.
 for(int i = 0; i < rules.length; i++)
 ret[i] = rules[i].toString();

 return ret;
```

```
 }
}
```

```
public class BigIpProfilePersistence extends BigIpBase {

 private LocalLBProfilePersistenceBindingStub stub = null;

 public BigIpProfilePersistence(String username, String password, String address) throws Exception {
 super(username, password, address);
 connect();
 }

 public void connect() throws Exception {
 // Get a connection to the Persistence Profile services on the bigIP and ask it for the list of rule
 stub = (LocalLBProfilePersistenceBindingStub) new LocalLBProfilePersistenceLocator().getLocalLBProfi

 }

 public String[] getRuleNames(String persistenceProfileNames[]) throws java.lang.Exception {
 String ret[] = {""};
 LocalLBProfileString rules[] = null;


 rules = stub.get_rule(persistenceProfileNames);

 // We now have a list of ProfileStrings, which contain two values, get the string out of it.
 for(int i = 0; i < rules.length; i++)
 ret[i] = rules[i].toString();

 return ret;
 }
}
```

It should be mentioned once again that leaving all of these routines as just throwing Exception and not the actual detailed Exception that they might throw is not the best solution by a long shot, but again keeps our code simpler for example purposes. You should consider putting the actual exceptions that can be generated in a routine into the throws statement.

There really isn't much to change in the main() routine, so we'll just show you the new lines. Insert them right after the last article, or create a new project and add them to that project's main() routine - inside a try{}catch, of course.

```
 BigIpVirtualServer bipVip = new BigIpVirtualServer(USERNAME, PASSWORD, IP_OR_DNS_NAME);
 String svrs[] = bipVip.findVirtualServersUsingRule(IRULE_NAME);
 for(int i = 0; i < svrs.length; i++)
 System.out.println("Virtual " + svrs[i] + " relies upon rule " + IRULE_NAME);
```

And as usual, anything in all capital letters in this last snippet needs to be replaced - either with command line args[] or with hardcoded values.

That's it for this week, this version will just print out the list of Virtuals, but you could of course use that list for other things. Perhaps we will in a future installment.

F5 Networks, Inc. | 401 Elliot Avenue West, Seattle, WA 98119 | 888-882-4447 | f5.com

F5 Networks, Inc.
Corporate Headquarters
info@f5.com

F5 Networks
Asia-Pacific
apacinfo@f5.com

F5 Networks Ltd.
Europe/Middle-East/Africa
emeainfo@f5.com

F5 Networks
Japan K.K.
f5j-info@f5.com